

Наука о данных в R для программы Цифровых гуманитарных исследований

Г. А. Мороз, И. С. Поздняков

Оглавление

Глава 1

О курсе

Материалы для курса Наука о данных для магистерской программы Цифровых гуманитарных исследований 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
- датаждурналистика
- ...

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

- psychologists (?)
- immunologists (?)
- business (?)
- public policy (?)
- fraud detection (?)
- ...

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 (?), 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 4.0.3 (2020-10-10)"
```

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.

2.3 Полезные ссылки

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

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/>

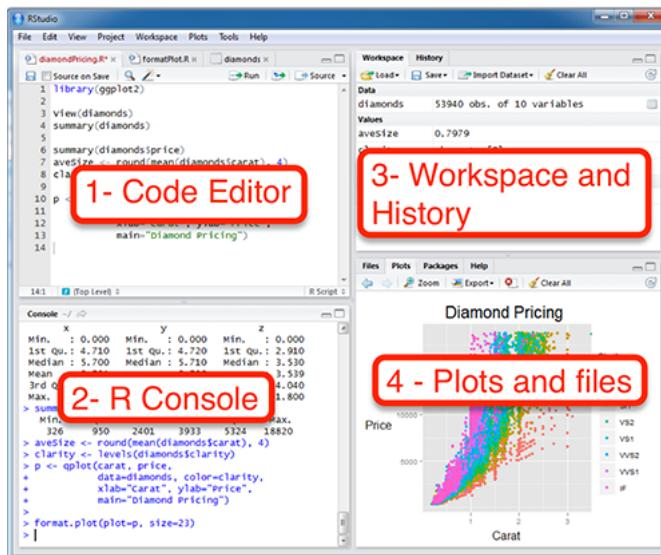
⁵<https://rstudio.cloud/>

⁶<https://www.rollapp.com/app/rstudio>

- книга (?)⁷ является достаточно сильной альтернативой всему курсу
- stackoverflow⁸ — сервис, где достаточно быстро отвечают на любые вопросы (не обязательно по R)
- RStudio community⁹ — быстро отвечают на вопросы, связанные с R
- русский stackoverflow¹⁰
- R-bloggers¹¹ — 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, data-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.



Существуют разные типы пользователей: одни любят работать в консоли (на картин-

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/rlang_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

ке 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. Подсказку по порядку выполнения операций в R можно получить с помощью следующей команды:

```
?Syntax
```

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

How to actually learn any new programming concept



Essential

Changing Stuff and
Seeing What Happens

O RLY?

@ThePracticalDev

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
```

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

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

Вот, например, функция для корня:

```
sqrt(16)
```

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

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

А вот так выглядит функция логарифма:

```
log(8)
```

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

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

Здесь не хватает основания логарифма!

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

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

$$\log_2 8 = 3$$

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

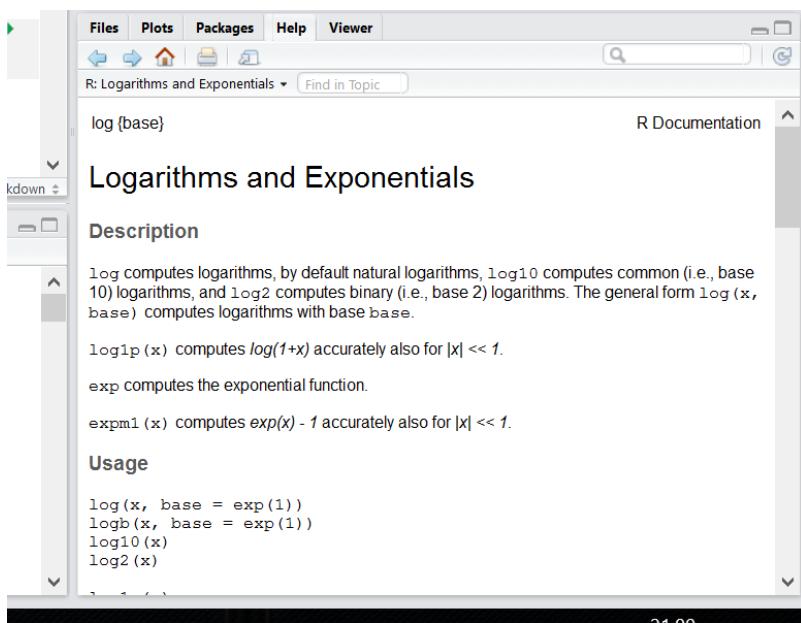
$$2^3 = 8$$

Только наша функция считает все как-то не так.

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

```
?log
```

Справа внизу в RStudio появится вот такое окно:



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

```
a <- 2
a
```

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

Справа от <- находится значение, которое вы хотите сохранить, или же какое-то выражение, результат которого вы хотите сохранить в эту переменную¹⁸:

```
a <- log(9, 3)
```

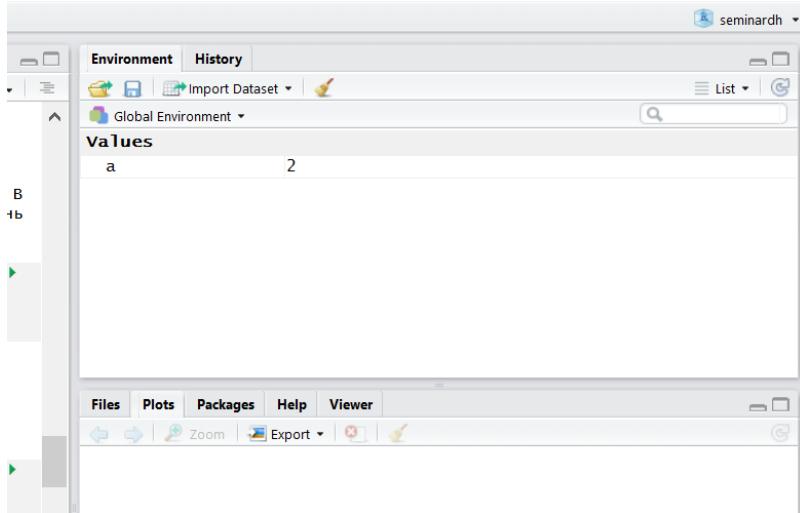
Слева от <- находится название будущей переменной. Название переменных может быть самым разным. Есть несколько ограничений для синтаксически валидных имен переменных: они должны включать в себя буквы, цифры, . или _, начинаться на букву (или точку, за которой не будет следовать цифра), не должны совпадать с коротким списком зарезервированных слов¹⁹. Короче говоря, название не должно включать в себя пробелы и большинство других знаков.

Нельзя: - new variable - _new_variable - .1var - v-r

Можно: - new_variable - .new.variable - var_2

Обязательно делайте названия переменных осмысленными! Страйтесь делать при этом их понятными и короткими, это сохранит вам очень много времени, когда вы (или кто-то еще) будете пытаться разобраться в написанном ранее коде. Если название все-таки получается длинным и состоящим из нескольких слов, то лучше всего использовать нижнее подчеркивание в качестве разделителя: some_variable²⁰.

После присвоения переменная появляется во вкладке Environment в RStudio:



Можно использовать переменные в функциях и просто вычислениях:

¹⁸Есть еще оператор ->, который позволяет присваивать значения слева направо, но так делать не рекомендуется, хотя это бывает довольно удобным.

¹⁹<https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html>

²⁰Еще иногда используются большие буквы SomeVariable, но это плохо читается, а иногда — точка, но это тоже не рекомендуется.

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

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

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

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

2.6 Логические операторы

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.

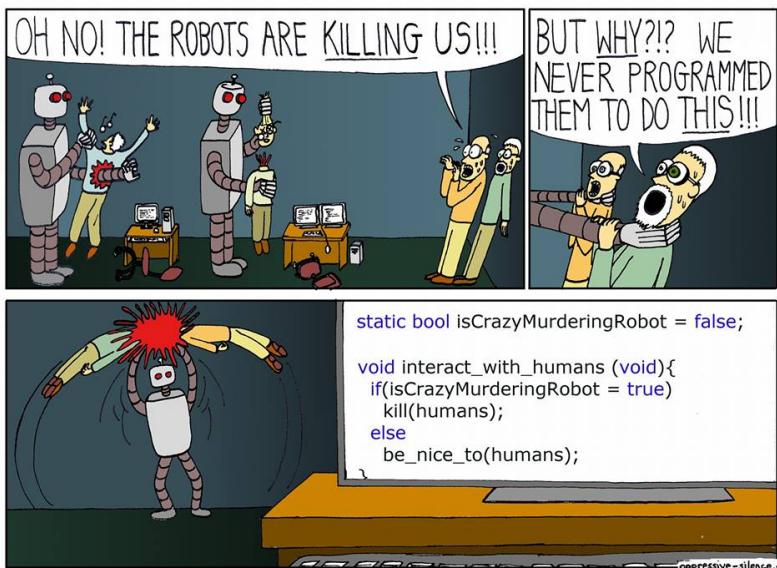
```
a = b # , !
a
```

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

```
b
```

```
## [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:

```

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
```

Этим мы будем пользоваться в дальнейшем регулярно! Именно на таких простых логических операциях построено большинство операций с данными.

2.7 Типы данных

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

```
class(a)
```

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

На самом деле, в R три типа numeric: integer (целые), double (дробные), complex (комплексные числа)²¹. R сам будет конвертировать числа в нужный тип numeric при необходимости, поэтому этим можно не заморачиваться.

Если же все-таки нужно задать конкретный тип числа эксплицитно, то можно воспользоваться функциями `as.integer()`, `as.double()` и `as.complex()`. Кроме того, при создании числа можно поставить в конце L, чтобы обозначить число как integer:

```
is.integer(5)
```

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

```
is.integer(5L)
```

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

Про double есть еще один маленький секрет. Дело в том, что дробные числа хранятся в R как числа с плавающей запятой двойной точности²². Дробные числа в компьютере могут быть записаны только с определенной степенью точности, поэтому иногда встречаются вот такие вот ситуации:

²¹Комплексные числа в R пишутся так: `complexnumber <- 2+2i.i` здесь - это та самая мнимая единица, которая является квадратным корнем из -1.

²²<https://ru.wikipedia.org/wiki/>

```
sqrt(2)^2 == 2
```

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

Это довольно стандартная ситуация, характерная не только для R. Чтобы ее избежать, можно воспользоваться функцией `all.equal()`:

```
all.equal(sqrt(2)^2, 2)
```

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

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

1. **character**: строки символов. Они должны выделяться кавычками.

```
s <- '          !'
s
```

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

```
class(s)
```

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

Можно использовать как ", так и ' (что удобно, когда строчка внутри уже содержит какие-то кавычки).

```
"Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn"
```

```
## [1] "Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn"
```

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
```

Мы уже встречались с логическими значениями при сравнении двух числовых переменных. Теперь вы можете догадаться, что результаты сравнения, например, числовых или строковых переменных, можно тоже сохранять в переменные!

```
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, что 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 логическое НЕ (!). ! превращает TRUE в FALSE, а FALSE в TRUE:

```
t1
```

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

```
!t1
```

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

```
!!t1 # !
```

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

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

```
t1 & t2
```

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

```
t1 & f1
```

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

А еще логическое 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.

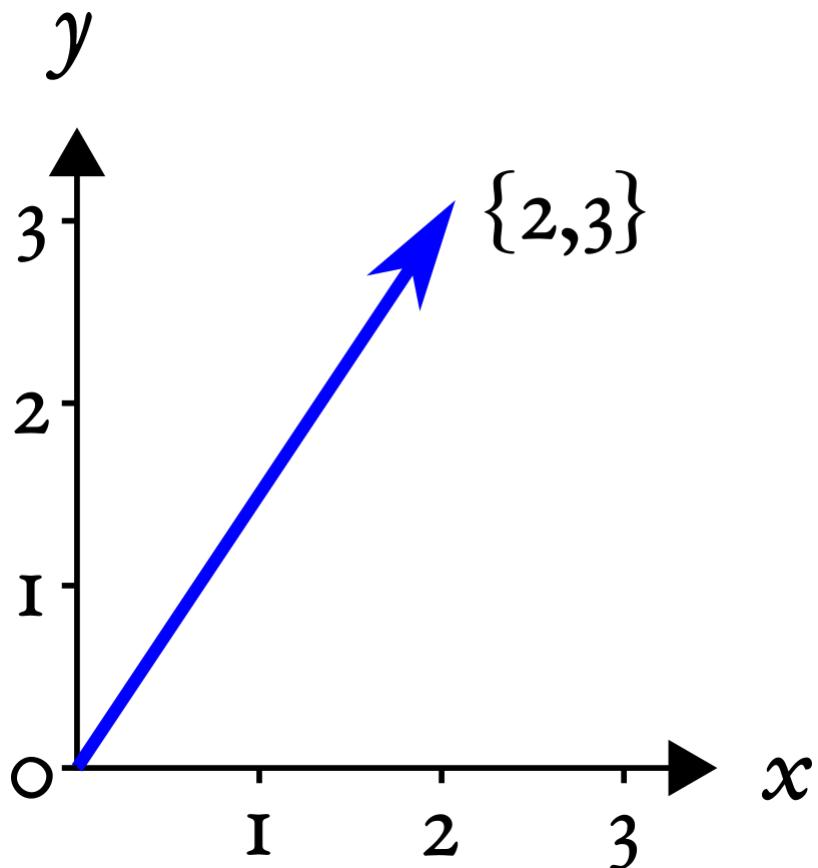
Итак, мы только что разобрались с самой занудной (хотя и важной) частью - с основными типами данных в R и как с ними работать²³. Пора переходить к чему-то более интересному и специальному для R. Вперед к ВЕКТОРАМ!

2.8 Вектор

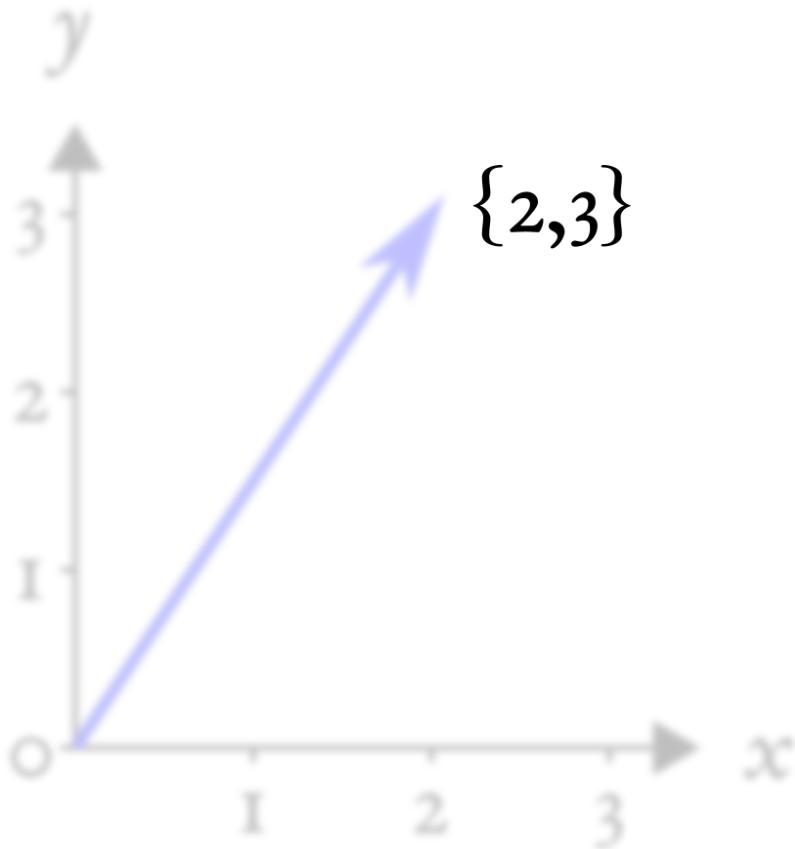
Если у вас не было линейной алгебры (или у вас с ней было все плохо), то просто запомните, что **вектор** (или `atomic vector` или `atomic`) — это набор (столбик) чисел в определенном порядке.

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}:

²³Кроме описанных пяти типов данных (`integer`, `double`, `complex`, `character` и `logical`) есть еще и шестой — это `raw`, сырья последовательность байтов, но нам она не понадобится.



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



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

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): 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!"
```

2.8.1 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 пройдет 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 много не знаем, сейчас важно запомнить, что логические данные — TRUE и FALSE — превращаются в 0 и 1 соответственно, а 0 и 1 в строчки "0" и "1".

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

```
as.numeric(c(T, F, F))
## [1] 0 0 0

as.character(as.numeric(c(T, F, F)))
## [1] "0" "0" "0"
```

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

```
as.numeric(c("1", "2", ""))
## Warning: NAs introduced by coercion
## [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!

2.8.2 Векторизация

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

```
n <- 1:4
m <- 4:1
n + m
## [1] 5 5 5 5

n - m
## [1] -3 -1  1  3
```

```
n * m
## [1] 4 6 6 4

n / m
## [1] 0.2500000 0.6666667 1.5000000 4.0000000

n ^ m + m * (n - m)
## [1] -11    5   11    7
```

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

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

```
n %*% m
##      [,1]
## [1,] 20
```

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

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

```
sqrt(1:10)
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
## [9] 3.000000 3.162278
```

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

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

2.8.3 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): 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. здесь²⁴. А вот ресайклинг кратных по длине векторов — это очень удобная штука, которая используется очень часто.

²⁴<https://stackoverflow.com/questions/6555651/under-what-circumstances-does-r-recycle>

2.8.4 Индексирование векторов

Итак, мы подошли к одному из самых сложных моментов. И одному из основных. От того, как хорошо вы научились 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 привыкли 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:

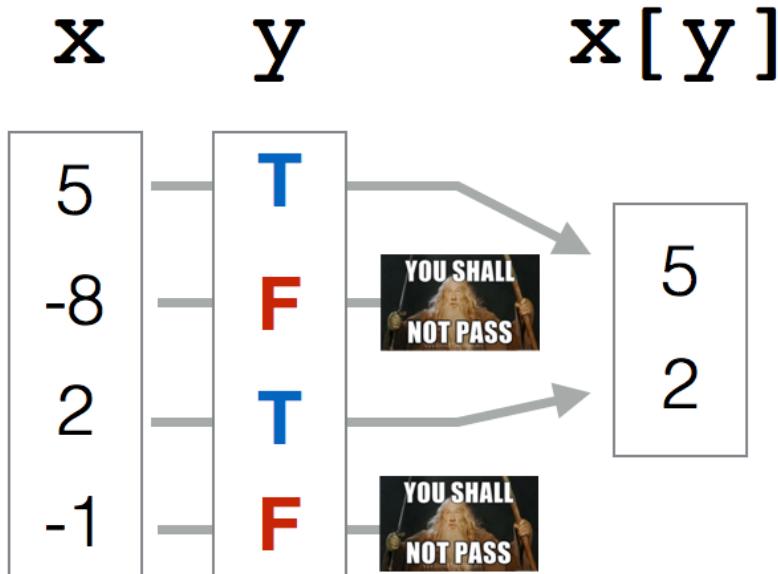
```
n[-1]  
## [1] 2 20 4 5 6 7 8 9 10  
  
n[c(-4, -5)]  
## [1] 1 2 20 6 7 8 9 10
```

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 ресайклнга!

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

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

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

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

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

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

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

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

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

Теперь посчитаем среднее вектора n:

```
mean(n)
```

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

А как вытащить все значения, которые больше среднего?

Сначала получим логический вектор — какие значения больше среднего:

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

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

А теперь используем его для индексирования вектора n:

```
n[larger]
```

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

Можно все это сделать в одну строчку:

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

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

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

2.8.5 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 вычленить из вектора n все значения кроме NA нам нужно, чтобы было наоборот: TRUE, если это не NA, FALSE, если это NA. Здесь нам понадобится логический оператор НЕ ! (мы его уже встречали), который инвертирует логические значения:

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

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

Ура, мы можем считать среднее!

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

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

Теперь Вы понимаете, зачем нужно отрицание (!)

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

```
?mean()
```

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

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

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

NA может появляться в векторах других типов тоже. На самом деле, NA - это специальное значение в логических векторах, тогда как в векторах других типов NA появляется как NA_integer_, NA_real_, NA_complex_ или NA_character_, но R обычно сам все переводит в нужный формат и показывает как просто NA.

Кроме NA есть еще NaN — это разные вещи. NaN расшифровывается как Not a Number и получается в результате таких операций как 0/0.

2.8.6 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-тutorиалы!

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.

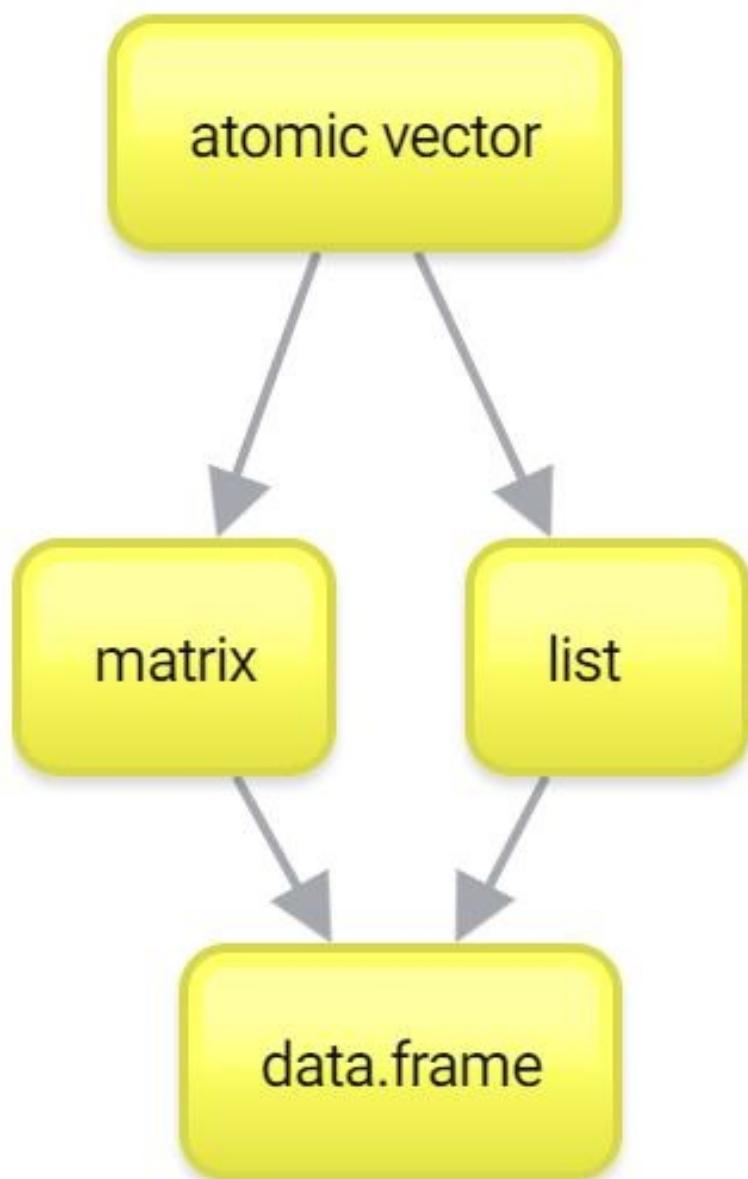
Does anyone ever get good at R or do they just get good at googling how to do things in R

— https://twitter.com/mousquemere/status/1125522375141883907?ref_src=twsrc%5Etfw May 6, 2019

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



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*.



2.9 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
```

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 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 здесь²⁵ или на стр. 99–101 книги “R in a Nutshell” (?).

²⁵<https://perso.esiee.fr/~courivad/R/06-objects.html>

2.10 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]]
## [1] "      "
##
## [[3]]
## [1] TRUE
```

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

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

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

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

```
str(complex_list)
```

```
## List of 3
```

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

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

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

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

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

```
named_list$age
```

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

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

```
named_list[1]
```

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

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

```
class(named_list)
```

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

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

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

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

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

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

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

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

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

— Hadley Wickham (@ href="https://twitter.com/hadleywickham/status/643381054758363136?ref_src=twsrctfw" ref_src="twsrctfw">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 dataфрейм.

2.11 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 dataфрейм.

```
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, чтобы 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$errorsR[df$age < mean(df$age)]
```

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

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

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

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

Датасеты удобно просматривать в RStudio. Для этого нужно написать команду `View(df)` или же просто нажать на название 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 которых 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 функции очень скоро перейдем), указав его имя и разрешение, то наткнемся на такую ошибку:

```
read.csv("heroes_information.csv")  
  
## Warning in file(file, "rt"): cannot open file 'heroes_information.csv': No such  
## file or directory  
  
## Error in file(file, "rt"): cannot open the connection
```

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

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

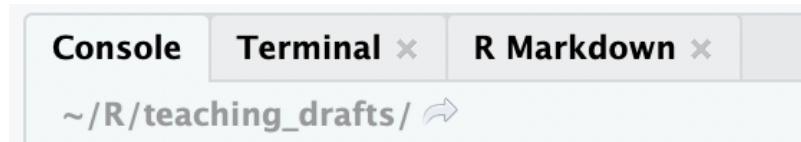
файлы. Для этого есть несколько способов.

- 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”):



- Гора идет к Магомету: изменение рабочей папки.

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

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

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

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

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

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

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

- Магомет использует кнопочный интерфейс: Import Dataset.

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

- Гора находит Магомета в интернете.

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

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

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

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

При закрытии проекта все переменные по умолчанию тоже будут сохраняться, а при открытии — восстанавливаются. 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 справится, то все остальное - ерунда. Мы уже разобрались с первой частью этого процесса - нахождением файла с данными, осталось научиться их читать.

Здесь стоит сделать небольшую ремарку. Довольно часто данные представляют собой табличку. Или же их можно свести к табличке. Такая табличка, как мы уже выяснили, удобно презентируется 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 именно последний случай. Таблица может быть представлена как текст: отдельные строчки в файле будут разделять разные строчки таблицы, а какой-нибудь знак-разделитель отделять колонки друг от друга.

Для чтения данных из текстового файла есть довольно удобная функция `read.table()`. Почитайте хэлп по ней и ужаснитесь: столько разных параметров на входе! Но там же вы увидете функции `read.csv()`, `read.csv2()` и некоторые другие — по сути, это тот же `read.table()`, но с другими параметрами по умолчанию, соответствующие формату файла, который мы загружаем. В данном случае используется формат

³<https://r4ds.had.co.nz/workflow-projects.html>

.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 разделения дробной части числа, а не точку, как это делается в большинстве стран мира. Поэтому есть альтернативный вариант формата .csv, где значения разделены точкой с запятой (;), а дробные значения - запятой (,). В этом и различие функций `read.csv()` и `read.csv2()` — первая функция предназначена для “международного” формата, вторая - для (условно) “Российского”. Оба варианта формата имеют расширение .csv, поэтому заранее понять какой именно будет вариант довольно сложно, приходится либо пробовать оба, либо заранее открывать файл в текстовом редакторе.

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

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

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

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

Параметр `stringsAsFactors =` задает то, как будут прочитаны строковые значения - как уже знакомые нам строки или как факторы. По сути, факторы - это примерно то же самое, что и `character`, но закодированные числами. Когда-то это было придумано для экономии используемых времени и памяти, сейчас же обычно становится просто лишней морокой. Но некоторые функции требуют именно `character`, некоторые `factor`, в большинстве случаев это без разницы. Но иногда непонимание может

привести к дурацким ошибкам. В данном случае мы просто пока обойдемся без факторов. Если у вас версия R выше 4.0, то `stringsAsFactors = FALSE` по умолчанию.

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

3.2 Проверка импортированных данных

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

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

```
str(heroes)
```

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

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

На что нужно обращать внимание?

1. Прочитаны ли пропущенные значения как NA. По умолчанию пропущенные значения обозначаются пропущенной строчкой или "NA", но встречаются самые разнообразные варианты. Возможные варианты кодирования пропущенных значений можно задать в параметре `na.strings` = функции `read.table()` и ее вариантов. В нашем датасете как раз такая ситуация, где нужно самостоятельно задавать, какие значения будут прочитаны как NA. Попытайтесь самостоятельно догадаться, как именно.
2. Прочитаны ли те столбики, которые должны быть числовыми, как `int` или `num`. Если в колонке содержатся числа, а написано `chr (= "character")` или `Factor`

(в случае если `stringsAsFactors = TRUE`), то, скорее всего, одна из строчек содержит в себе нечисловые знаки, которые не были прочитаны как `NA`.

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

⁴<https://www.artlebedev.ru/decoder/>

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 попробуете открыть .xlsx файл в Блокноте, то увидите кракозябры. Это делает работу с этим файлами гораздо менее удобной, поэтому стоит избегать экселеевых 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 попробуете 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]

##
## -- 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>    <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 соответствующе: `fread()` и `fwrite()`, где `f` означает fast⁵.

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

```
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  Ichthyosapien 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
## 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!) экспортовать 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 чте-

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

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

ния: `vroom` и `data.table` обгоняют всех, затем идет `readr`, и только после него - базовые функции R.

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) "
```



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

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

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



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

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

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

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){
  "
}
} else {
  "
}
```

```
## Warning in if (number > 0) {: the condition has length > 1 and only the first
## element will be used
```

²Аналогично, достаточно просто значения `FALSE`.

```
## Warning in if (number < 0) {: the condition has length > 1 and only the first
## element will be used
## [1] "
```

R выдаст 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 у `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 встречае-

те 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 строчки зависят от расчетов предыдущих строчек, 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] "           "
## [5] "           "
```

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] "           "
## [5] "           "
```


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 результата 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. Напри-

мер, 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 Проверка на адекватность

Лучший способ не бояться ошибок и предупреждений — научиться прописывать их самостоятельно в собственных функциях. Это позволит понять, что за текстом предупреждений и ошибок, которые у вас возникают, стоит забота разработчиков о пользователях, которые хотят максимально обезопасить нас от наших непродуманных действий.

Хорошо написанные функции не только выдают правильный результат на все возможные адекватные данные на входе, но и не дают получить правдоподобные результаты при неадекватных входных данных. Как вы уже знаете, если на входе у вас имеются пропущенные значения, то многие функции будут в ответ тоже выдавать пропущенные значения. И это вполне осознанное решение, которое позволяет избегать ситуаций вроде той, когда около одной пятой научных статей по генетике содержало ошибки в приложенных данных¹ и замечать пропущенные значения на ранней стадии. Кроме того, можно проводить проверки на адекватность входящих данных (`sanity check`).

Разберем это на примере самодельной функции `imt()`, которая выдает индекс массы тела, если на входе задать вес (аргумент `weight =`) в килограммах и рост (аргумент `height =`) в метрах.

```
imt <- function(weight, height) weight / height ^ 2
```

Проверим, что функция работает верно:

```
w <- c(60, 80, 120)
h <- c(1.6, 1.7, 1.8)
imt(weight = w, height = h)
```

```
## [1] 23.43750 27.68166 37.03704
```

Очень легко перепутать и написать рост в сантиметрах. Было бы здорово предупредить об этом пользователю, показав ему предупреждающее сообщение, если рост больше, чем, например, 3. Это можно сделать с помощью функции `warning()`

¹<https://genomeweb.biomedcentral.com/articles/10.1186/s13059-016-1044-7>

```

imt <- function(weight, height) {
  if (height > 3) warning("           height      3:      ,
                           weight / height ^ 2
}
imt(78, 167)

## Warning in imt(78, 167):           height      3:      ,
## [1] 0.002796802

```

В некоторых случаях ответ будет совершенно точно некорректным, хотя функция все посчитает и выдаст ответ, как будто так и надо. Например, если какой-то из аргументов функции `imt()` будет меньше или равен 0. В этом случае нужно прописать проверку на это условие, и если это действительно так, то выдать пользователю ошибку.

```

imt <- function(weight, height) {
  if (any(weight <= 0 | height <= 0)) stop("           ")
  if (height > 3) warning("           height      3:      ,
                           weight / height ^ 2
}
imt(-78, 167)

## Error in imt(-78, 167):

```

Когда вы попробуете самостоятельно прописывать предупреждения и ошибки в функциях, то быстро поймете, что ошибки - это вовсе не обязательно результат того, что где-то что-то сломалось и нужно паниковать. Совсем даже наоборот, прописанная ошибка - чья-то забота о пользователях, которых пытаются максимально проинформировать о том, что и почему пошло не так.

Это естественно в начале работы с R (и вообще с программированием) избегать ошибок, конечно, в самом начале обучения большая часть из них остается непонятной. Но постарайтесь понять текст ошибки, вспомнить в каких случаях у вас возникала похожая ошибка. Очень часто этого оказывается достаточно чтобы понять причину ошибки даже если вы только-только начали изучать R.

Ну а в дальнейшем я советую ознакомиться со средствами отладки кода в R² для того, чтобы научиться справляться с ошибками в своем коде на более продвинутом уровне.

5.3 Когда и зачем создавать функции?

Когда стоит создавать функции? Существует “правило трех”³ — если у вас есть три куска очень похожего кода, то самое время превратить код в функцию. Это очень услов-

²<https://adv-r.hadley.nz/debugging.html>

³[https://en.wikipedia.org/wiki/Rule_of_three_\(computer_programming\)](https://en.wikipedia.org/wiki/Rule_of_three_(computer_programming))

ное 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.

5.4 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:

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

## [[1]]
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x5fe2f0338d20>
## <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 функций⁴, использовать функции в качестве аргументов функций, сохранять функции как переменные. Пожалуй, самое важное из

⁴Функция, которая создает другие функции, называется фабрикой функций.

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.

5.5 Семейство функций `apply()`

5.5.1 Применение `apply()` для матриц

Семейство? Да, их целое множество: `apply()`, `lapply()`, `sapply()`, `vapply()`, `tapply()`, `mapply()`, `taapply()`... 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]
## [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.

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:

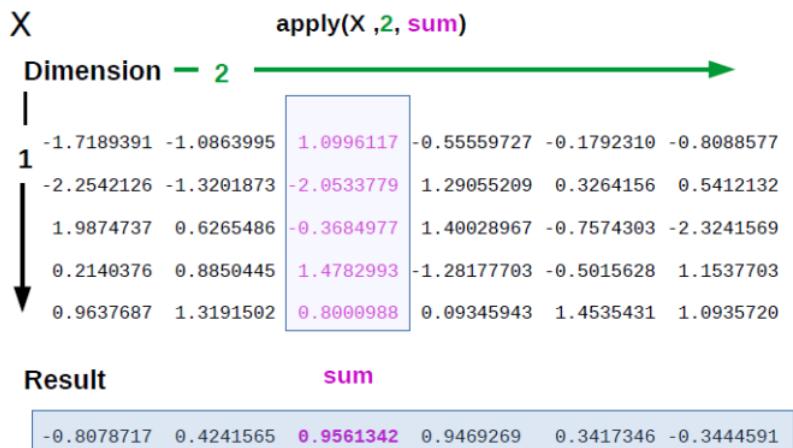


Рис. 5.1: apply

```
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
```

5.5.2 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
```

5.5.3 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 невекторизованной функции с помощью `Vectorize()`. Эта функция просто обворачивает функцию одним из вариантов `apply()`.

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

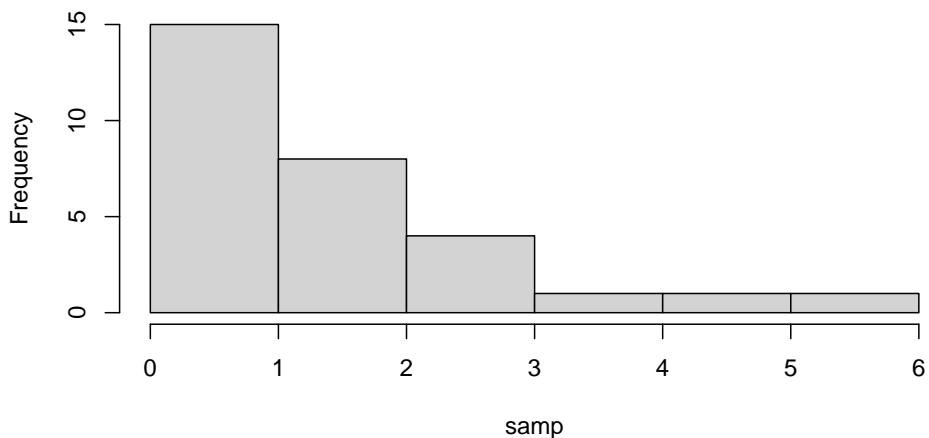
```
heroes <- read.csv("data/heroes_information.csv",
                     na.strings = c("-", "-99"))
sapply(heroes, class)
```

```
##           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:

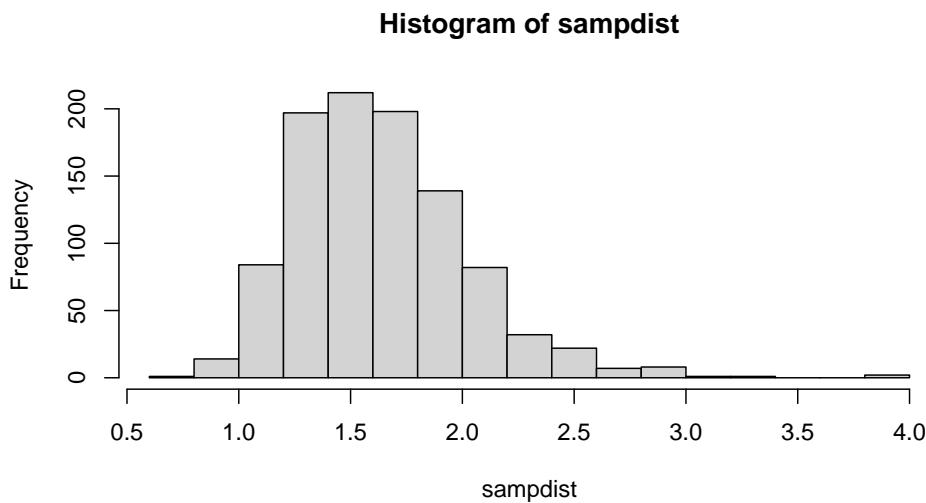
```
samp <- rlnorm(30)
hist(samp)
```

Histogram of 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()` чуть поближе, то рекомендую вот этот туториал⁵.

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`.

```
#Введение в tidyverse {#tidy_intro}
```

5.6 Вселенная tidyverse

tidyverse — это не один, а целое множество пакетов. Есть ключевые пакеты (ядро тайди-верса), а есть побочные — в основном для работы со специфическими видами данных.

*tidyverse*⁶ — это набор пакетов:

- *ggplot2*, для визуализации
- *tibble*, для работы с тиблами, продвинутый вариант датафрейма
- *tidyR*, для формата tidy data
- *readr*, для чтения файлов в R

⁵<https://www.datacamp.com/community/tutorials/r-tutorial-apply-family>

⁶<https://www.tidyverse.org>

- *purrr*, для функционального программирования (замена семейства функций `*apply()`)
- *dplyr*, для преобразования данных
- *stringr*, для работы со строковыми переменными
- *forcats*, для работы с переменными-факторами

Полезно также знать о следующих пакетах, не включенных в ядро, но также считающихся частью тайдиверса:

- *vroom*, для быстрой загрузки табличных данных
- *readxl*, для чтения .xls и .xlsx
- *jsonlite*, для работы с JSON
- *xml*, для работы с XML
- *DBI*, для работы с базами данных
- *rvest*, для веб-скраппинга
- *lubridate*, для работы с временем
- *tidytext*, для работы с текстами и корпусами
- *glue*, для продвинутого объединения строк
- *magrittr*, с несколькими вариантами pipe оператора
- *tidymodels*, для моделирования и машинного обучения⁷
- *dtplyr*, для ускорения dplyr за счет перевод синтаксиса на `data.table`

И это еще не все пакеты tidyverse! Есть еще много других небольших пакетов, которые тоже считаются частью tidyverse. Кроме официальных пакетов tidyverse есть множество пакетов, которые пытаются соответствовать принципам tidyverse и дополняют его.

Все пакеты tidyverse объединены tidy философией и взаимосовместимым синтаксисом. Это означает, что, во многих случаях даже не нужно думать о том, из какого именно пакета тайдиверса пришла функция. Можно просто установить и загрузить пакет tidyverse.

```
install.packages("tidyverse")
```

Пакет tidyverse — это такой пакет с пакетами⁸.

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

```
## -- Attaching packages -----
## v ggplot2 3.3.2     v purrr    0.3.4
## v tibble   3.0.4     v dplyr    1.0.2
## v tidyr    1.1.2     v stringr  1.4.0
## v readr    1.4.0     v forcats 0.5.0
```

⁷Как и пакет tidyverse, tidymodels — это пакет с несколькими пакетами.

⁸https://cs11.pikabu.ru/post_img/big/2019/03/12/11/1552415351186680692.jpg

```
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Подключение пакета `tidyverse` автоматически приводит к подключению ядра `tidyverse`, остальные же пакеты нужно подключать дополнительно при необходимости.

5.7 Загрузка данных с помощью `readr`

Стандартной функцией для чтения .csv файлов в R является функция `read.csv()`, но мы будем использовать функцию `read_csv()` из пакета `readr`. Синтаксис функции `read_csv()` очень похож на `read.csv()`: первым аргументом является путь к файлу (в том числе можно использовать URL), некоторые остальные параметры тоже совпадают.

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

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

Подробнее про импорт данных, в том числе в `tidyverse`, смотри в @ref(real_data).

`#tibble`

Когда мы загрузили данные с помощью `read_csv()`, то мы получили `tibble`, а не `data.frame`:

```
class(heroes)

## [1] "spec_tbl_df" "tbl_df"        "tbl"          "data.frame"
```

Тиблл (`tibble`) - это такой “усовершенствованный” `data.frame`. Почти⁹ все, что работает с `data.frame`, работает и с тибллами. Однако у тибллов есть свои дополнительные фишки. Самая очевидная из них - более аккуратный вывод в консоль:

```
heroes

## # A tibble: 734 x 11
##       X1 name  Gender `Eye color` Race  `Hair color` Height Publisher
##   <dbl> <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>
```

Выводятся только первые 10 строк, если какие-то колонки не влезают на экран, то они просто перечислены внизу. Ну а тип данных написан прямо под названием колонки.

Функции различных пакетов tidyverse сами конвертируют в тиблл при необходимости. Если же нужно это сделать самостоятельно, то можно это сделать так:

```
heroes_df <- as.data.frame(heroes) #
class(heroes_df)

## [1] "data.frame"

as_tibble(heroes_df) #

## # A tibble: 734 x 11
##       X1 name  Gender `Eye color` Race  `Hair color` Height Publisher
##   <dbl> <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~
```

⁹<https://www.jumpingrivers.com/blog/the-trouble-with-tibbles/>

```

## 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>

```

В дальнейшем мы будем работать только с tidyverse, а это значит, что только с тиблами, а не обычными датафреймами. Тем не менее, тиблы и датафреймы будут в дальнейшем использоваться как синонимы.

Можно создавать тиблы вручную с помощью функции `tibble()`, которая работает аналогично функции `data.frame()`:

```

tibble(
  a = 1:3,
  b = letters[1:3]
)

```

```

## # A tibble: 3 x 2
##       a     b
##   <int> <chr>
## 1     1     a
## 2     2     b
## 3     3     c

```

5.8 magrittr::%>%

Оператор `%>%` называется “пайпом” (pipe), т.е. “трубой”. Он означает, что следующая функция (справа от пайпа) принимает на вход в качестве первого аргумента результат выполнения предыдущей функции (той, что слева). Фактически, это примерно то же самое, что и вставлять результат выполнения функции в качестве первого аргумента в другую функцию. Просто выглядит это красивее и читабельнее. Как будто данные пропускаются через трубы функций или конвейерную ленту на заводе, если хотите. А то, что первый параметр функции - это почти всегда данные, работает нам здесь на руку. Этот оператор взят из пакета `magrittr`¹⁰. Возможно, даже если вы не захотите пользоваться tidyverse, использование пайпов Вам понравится.

Важно понимать, что пайп не дает какой-то дополнительной функциональности или дополнительной скорости работы¹¹. Он создан исключительно для читабельности и комфорта.

С помощью пайпов вот эту команду...

¹⁰Если быть точным, то оператор `%>%` был импортирован во все основные пакеты tidyverse, а сам пакет `magrittr` не входит в базовый набор tidyverse. Тем не менее, в самом `magrittr` есть еще несколько интересных операторов.

¹¹Даже наоборот, использование пайпов незначительно снижает скорость выполнения команды.

```
sum(sqrt(abs(sin(1:22))))
```

```
## [1] 16.72656
```

...можно переписать вот так:

```
1:22 %>%
  sin() %>%
  abs() %>%
  sqrt() %>%
  sum()
```

```
## [1] 16.72656
```

В очень редких случаях результат выполнения функции нужно вставить не на первую позицию (или же мы хотим использовать его несколько раз). В этих случаях можно использовать `..`, чтобы обозначить, куда мы хотим вставить результат выполнения выражения слева от `%>%`.

```
"           ! " %>%
  c("—", .., "—")
```

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

5.9 Главные пакеты tidyverse: dplyr и tidyverse

`dplyr`¹² — это самая основа всего `tidyverse`. Этот пакет предоставляет основные функции для манипуляции с тиблами. Пакет `dplyr` является наследником и более усовершенствованной версией `plyr`, так что если увидите использование пакета `plyr`, то, скорее всего, скрипт был написан очень давно.

Пакет `tidyverse` дополняет `dplyr`, предоставляя полезные функции для тайдификации тиблов. Тайдификация (“аккуратизация”) данных означает приведение табличных данных к такому формату, в котором:

- Каждая переменная имеет собственный столбец
- Каждый наблюдение имеет собственную строку
- Каждое значение имеет свою собственную ячейку

Впрочем, многие функции `dplyr` часто используются при тайдификации, так же как и многие функции `tidyverse` имеют применение вне тайдификации. В общем, функционал этих двух пакетов несколько смешался, поэтому мы будем рассматривать их вместе. А чтобы представлять, какая функция относится к какому пакету (хотя запомнить это необязательно), я буду использовать запись с двумя двоеточиями `::`, которая

¹² Есть споры о том, как это правильно читать¹³. Используемые варианты: `диплаер`, `диплер`, `диплр`.

обычно используется для использования функции без подгрузки всего пакета, при первом упоминании функции.

Пакет `tidyverse` — это более усовершенствованная версия пакета `reshape2`, который в свою очередь является усовершенствованной версией `reshape`. По аналогии с `plyr`, если вы видите использование этих пакетов, то это указывает на то, что перед вами морально устаревший код.

Код с использованием `dplyr` и `tidyverse` сильно непохож на то, что мы видели раньше. Большинство функций `dplyr` и `tidyverse` работают с целым тиблом сразу, принимая его в качестве первого аргумента и возвращая измененный тибл. Это позволяет превратить весь код в последовательный набор применяемых функций, соединенный пайпами. На практике это выглядит очень элегантно, и вы в этом скоро убедитесь.

5.10 Работа с колонками тиббла

5.10.1 Выбор колонок: `dplyr::select()`

Функция `dplyr::select()` позволяет выбирать колонки по номеру или имени (кавычки не нужны).

```
heroes %>%
  select(1,5)
```

```
## # A tibble: 734 x 2
##       X1 Race
##   <dbl> <chr>
## 1     0 Human
## 2     1 Icthyo Sapien
## 3     2 Ungaran
## 4     3 Human / Radiation
## 5     4 Cosmic Entity
## 6     5 Human
## 7     6 <NA>
## 8     7 Human
## 9     8 <NA>
## 10    9 Human
## # ... with 724 more rows
```

```
heroes %>%
  select(name, Race, Publisher, `Hair color`)
```

```
## # A tibble: 734 x 4
##       name      Race      Publisher      `Hair color`
##   <chr>      <chr>      <chr>          <chr>
## 1 A-Bomb    Human    Marvel Comics    No Hair
```

```

##  2 Abe Sapien    Icthyo Sapien    Dark Horse Comics No Hair
##  3 Abin Sur     Ungaran        DC Comics      No Hair
##  4 Abomination  Human / Radiation Marvel Comics No Hair
##  5 Abraxas      Cosmic Entity   Marvel Comics Black
##  6 Absorbing Man Human          Marvel Comics No Hair
##  7 Adam Monroe   <NA>           NBC - Heroes Blond
##  8 Adam Strange  Human          DC Comics      Blond
##  9 Agent 13      <NA>           Marvel Comics Blond
## 10 Agent Bob    Human          Marvel Comics Brown
## # ... with 724 more rows

```

Обратите внимание, если в названии колонки присутствует пробел или, например, колонка начинается с цифры или точки и цифры, то это синтаксически невалидное имя (??). Это не значит, что такие названия колонок недопустимы. Но такие названия колонок нужно обособлять ' грависом (правый штрих, на клавиатуре находится там же где и буква ё и ~).

Еще обратите внимание на то, что функции tidyverse не изменяют сами изначальные тибллы/датафреймы. Это означает, что если вы хотите полученный результат сохранить, то нужно добавить присвоение:

```

heroes_some_cols <- heroes %>%
  select(name, Race, Publisher, `Hair color`)
heroes_some_cols

```

```

## # A tibble: 734 x 4
##   name      Race      Publisher `Hair color`
##   <chr>     <chr>     <chr>       <chr>
## 1 A-Bomb   Human    Marvel Comics No Hair
## 2 Abe Sapien Icthyo Sapien Dark Horse Comics No Hair
## 3 Abin Sur  Ungaran  DC Comics      No Hair
## 4 Abomination Human / Radiation Marvel Comics No Hair
## 5 Abraxas   Cosmic Entity Marvel Comics Black
## 6 Absorbing Man Human    Marvel Comics No Hair
## 7 Adam Monroe <NA>      NBC - Heroes Blond
## 8 Adam Strange Human   DC Comics      Blond
## 9 Agent 13   <NA>      Marvel Comics Blond
## 10 Agent Bob Human   Marvel Comics Brown
## # ... with 724 more rows

```

5.10.2 Мини-язык tidyselect для выбора колонок

Для выбора столбцов (не только в `select()`, но и для других функций tidyverse) используется специальный мини-язык `tidyselect` из одноименного пакета¹⁴. `tidyselect` да-

¹⁴Как и в случае с `magrittr`, пакет `tidyselect` не содержится в базовом tidyverse, но функции импортируются основными пакетами tidyverse.

ет очень широкие возможности для выбора колонок.

Можно использовать оператор : для выбора нескольких соседних колонок (по аналогии с созданием числового вектора с шагом 1).

```
heroes %>%
  select(name:Publisher)

## # A tibble: 734 x 7
##   name      Gender `Eye color` Race      `Hair color` Height Publisher
##   <chr>     <chr>    <chr>    <chr>    <chr>       <dbl> <chr>
## 1 A-Bomb    Male    yellow   Human    No Hair      203 Marvel Comics
## 2 Abe Sapien Male    blue    Ichtyo Sapien No Hair      191 Dark Horse C-
## 3 Abin Sur   Male    blue    Ungaran  No Hair      185 DC Comics
## 4 Abomination Male   green   Human / Radi~ No Hair      203 Marvel Comics
## 5 Abraxas    Male    blue    Cosmic Entity Black        NA Marvel Comics
## 6 Absorbing~ Male   blue    Human    No Hair      193 Marvel Comics
## 7 Adam Monr~ Male   blue    <NA>      Blond        NA NBC - Heroes
## 8 Adam Stra~ Male   blue    Human    Blond        185 DC Comics
## 9 Agent 13   Female  blue    <NA>      Blond        173 Marvel Comics
## 10 Agent Bob  Male   brown   Human    Brown       178 Marvel Comics
## # ... with 724 more rows

heroes %>%
  select(name:`Eye color`, Publisher:Weight)

## # A tibble: 734 x 7
##   name      Gender `Eye color` Publisher      `Skin color` Alignment Weight
##   <chr>     <chr>    <chr>    <chr>    <chr>       <chr>       <dbl>
## 1 A-Bomb    Male    yellow   Marvel Comics <NA>        good        441
## 2 Abe Sapien Male    blue    Dark Horse Com~ blue        good        65
## 3 Abin Sur   Male    blue    DC Comics     red        good        90
## 4 Abomination Male   green   Marvel Comics <NA>        bad         441
## 5 Abraxas    Male    blue    Marvel Comics <NA>        bad         NA
## 6 Absorbing M~ Male   blue    Marvel Comics <NA>        bad        122
## 7 Adam Monroe Male   blue    NBC - Heroes <NA>        good        NA
## 8 Adam Strange Male   blue    DC Comics     <NA>        good        88
## 9 Agent 13   Female  blue    Marvel Comics <NA>        good        61
## 10 Agent Bob  Male   brown   Marvel Comics <NA>        good        81
## # ... with 724 more rows
```

Используя ! можно вырезать ненужные колонки.

```
heroes %>%
  select(!X1)
```

```

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

heroes %>%
  select(!(Gender:Height))

## # A tibble: 734 x 6
##   X1 name          Publisher `Skin color` Alignment Weight
##   <dbl> <chr>        <chr>     <chr>     <chr>      <dbl>
## 1 0 A-Bomb        Marvel Comics <NA>      good      441
## 2 1 Abe Sapien    Dark Horse Comics blue    good      65
## 3 2 Abin Sur     DC Comics      red      good      90
## 4 3 Abomination  Marvel Comics <NA>      bad      441
## 5 4 Abraxas       Marvel Comics <NA>      bad      NA
## 6 5 Absorbing Man Marvel Comics <NA>      bad      122
## 7 6 Adam Monroe   NBC - Heroes <NA>      good      NA
## 8 7 Adam Strange  DC Comics     <NA>      good      88
## 9 8 Agent 13      Marvel Comics <NA>      good      61
## 10 9 Agent Bob    Marvel Comics <NA>      good      81
## # ... with 724 more rows

```

Другие известные нам логические операторы (`&` и `|`) тоже работают в `tidyselect`.

В дополнение к логическим операторам `и` и `:`, в `tidyselect` есть набор вспомогательных функций, работающих исключительно в контексте выбора колонок с помощью `tidyselect`.

Вспомогательная функция `last_col()` позволит обратиться к последней колонке тибла:

```

heroes %>%
  select(name:last_col())

## # A tibble: 734 x 10

```

```

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

```

А функция `everything()` позволяет выбрать все колонки.

```

heroes %>%
  select(everything())

```



```

## # A tibble: 734 x 11
##   X1 name  Gender `Eye color` Race  `Hair color` Height Publisher
##   <dbl> <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>

```

При этом `everything()` не будет дублировать выбранные колонки, поэтому можно использовать `everything()` для перестановки колонок в тиббле:

```

heroes %>%
  select(name, Publisher, everything())

```



```

## # A tibble: 734 x 11
##   name  Publisher X1 Gender `Eye color` Race  `Hair color` Height
##   <chr> <chr>    <dbl> <chr>  <chr> <chr>       <dbl>
## 1 A-Bo~ Marvel C~      0 Male    yellow    Human No Hair      203
## 2 Abe ~ Dark Hor~      1 Male    blue     Icth~ No Hair      191

```

```

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

```

Впрочем, для перестановки колонок удобнее использовать специальную функцию `relocate()` (@ref(tidy_relocate)) Можно даже выбирать колонки по паттернам в названиях. Например, с помощью `ends_with()` можно выбрать все колонки, заканчивающиеся одинаковым суффиксом:

```

heroes %>%
  select(ends_with("color"))

## # A tibble: 734 x 3
##   `Eye color` `Hair color` `Skin color`
##   <chr>        <chr>        <chr>
## 1 yellow      No Hair     <NA>
## 2 blue        No Hair     blue
## 3 blue        No Hair     red
## 4 green       No Hair     <NA>
## 5 blue        Black       <NA>
## 6 blue        No Hair     <NA>
## 7 blue        Blond       <NA>
## 8 blue        Blond       <NA>
## 9 blue        Blond       <NA>
## 10 brown       Brown      <NA>
## # ... with 724 more rows

```

Аналогично, с помощью функции `starts_with()` можно найти колонки с одинаковым префиксом, с помощью `contains()` — все колонки с выбранным паттерном в любой части названия колонки¹⁵.

```

heroes %>%
  select(starts_with("Eye") & ends_with("color"))

## # A tibble: 734 x 1
##   `Eye color`

```

¹⁵ Выбранный паттерн будет найден посимвольно, если же вы хотите искать по регулярным выражениям, то вместо `contains()` нужно использовать `matches()`.

```

##      <chr>
## 1 yellow
## 2 blue
## 3 blue
## 4 green
## 5 blue
## 6 blue
## 7 blue
## 8 blue
## 9 blue
## 10 brown
## # ... with 724 more rows

```

```

heroes %>%
  select(contains("eight"))

```

```

## # A tibble: 734 x 2
##       Height Weight
##      <dbl>   <dbl>
## 1     203     441
## 2     191      65
## 3     185      90
## 4     203     441
## 5       NA      NA
## 6     193     122
## 7       NA      NA
## 8     185      88
## 9     173      61
## 10    178      81
## # ... with 724 more rows

```

Ну и наконец, можно выбирать по содержимому колонок с помощью `where()`. Это напоминает применение `sapply()(@ref(apply_other))` на датафрейме для индексирования колонок: в качестве аргумента для `where` принимается функция, которая применяется для каждой из колонок, после чего выбираются только те колонки, для которых было получено TRUE.

```

heroes %>%
  select(where(is.numeric))

```

```

## # A tibble: 734 x 3
##       X1 Height Weight
##      <dbl>   <dbl>   <dbl>
## 1     0     203     441
## 2     1     191      65

```

```
##  3    2    185    90
##  4    3    203   441
##  5    4     NA     NA
##  6    5    193   122
##  7    6     NA     NA
##  8    7    185    88
##  9    8    173    61
## 10   9    178    81
## # ... with 724 more rows
```

Функция `where()` дает невиданную мощь. Например, можно выбрать все колонки без `NA`:

```
heroes %>%
  select(where(function(x) !any(is.na(x))))
```

```
## # A tibble: 734 x 3
##       X1 name      Publisher
##   <dbl> <chr>     <chr>
## 1     0 A-Bomb    Marvel Comics
## 2     1 Abe Sapien Dark Horse Comics
## 3     2 Abin Sur   DC Comics
## 4     3 Abomination Marvel Comics
## 5     4 Abraxas    Marvel Comics
## 6     5 Absorbing Man Marvel Comics
## 7     6 Adam Monroe NBC - Heroes
## 8     7 Adam Strange DC Comics
## 9     8 Agent 13   Marvel Comics
## 10    9 Agent Bob  Marvel Comics
## # ... with 724 more rows
```

###Переименование колонок: `dplyr::rename()`

Внутри `select()` можно не только выбирать колонки, но и переименовывать их:

```
heroes %>%
  select(id = X1)
```

```
## # A tibble: 734 x 1
##       id
##   <dbl>
## 1     0
## 2     1
## 3     2
## 4     3
## 5     4
```

```
## 6    5
## 7    6
## 8    7
## 9    8
## 10   9
## # ... with 724 more rows
```

Однако удобнее для этого использовать специальную функцию `dplyr::rename()`. Синтаксис у нее такой же, как и у `select()`, но `rename()` не выбрасывает колонки, которые не были упомянуты.

```
heroes %>%
  rename(id = X1)

## # A tibble: 734 x 11
##       id name  Gender `Eye color` Race `Hair color` Height Publisher
##     <dbl> <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>
```

Для массового переименования колонок можно использовать функцию `rename_with()`. Эта функция так же использует tidyselect синтаксис для выбора колонок (по умолчанию выбираются все колонки) и применяет функцию в качестве аргумента, которая изменяет

```
heroes %>%
  rename_with(make.names)

## # A tibble: 734 x 11
##       X1 name  Gender Eye.color Race  Hair.color Height Publisher Skin.color
##     <dbl> <chr> <chr>    <chr> <chr>      <dbl> <chr>      <chr>
## 1     0 A-Bo~ Male    yellow Human No Hair      203 Marvel C~ <NA>
## 2     1 Abe ~ Male    blue  Icth~ No Hair      191 Dark Hor~ blue
## 3     2 Abin~ Male    blue  Unga~ No Hair      185 DC Comics red
## 4     3 Abom~ Male    green Huma~ No Hair      203 Marvel C~ <NA>
## 5     4 Abra~ Male    blue  Cosm~ Black        NA Marvel C~ <NA>
```

```

## 6      5 Absor~ Male   blue      Human No Hair      193 Marvel C~ <NA>
## 7      6 Adam~ Male   blue      <NA> Blond        NA NBC - He~ <NA>
## 8      7 Adam~ Male   blue      Human Blond       185 DC Comics <NA>
## 9      8 Agen~ Female blue      <NA> Blond       173 Marvel C~ <NA>
## 10     9 Agen~ Male   brown     Human Brown       178 Marvel C~ <NA>
## # ... with 724 more rows, and 2 more variables: Alignment <chr>, Weight <dbl>

###Перестановка колонок:dplyr::relocate() {#tidy_relocate}

```

Для изменения порядка колонок можно использовать функцию `relocate()`. Она тоже работает похожим образом на `select()` и `rename()`¹⁶. Как и `rename()`, функция `relocate()` не выкидывает неиспользованные колонки:

```

heroes %>%
  relocate(Publisher)

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

```

При этом `relocate()` имеет дополнительные параметры `.after =` и `.before =`, которые позволяют выбирать, куда поместить выбранные колонки.

```

heroes %>%
  relocate(Publisher, .after = name)

## # A tibble: 734 x 11
##   X1 name Publisher Gender `Eye color` Race `Hair color` Height
##   <dbl> <chr> <chr> <chr>    <chr> <chr>    <dbl>
## 1 0 A-Bo~ Marvel C~ Male   yellow   Human No Hair    203
## 2 1 Abe ~ Dark Hor~ Male   blue    Icth~ No Hair    191
## 3 2 Abin~ DC Comics Male   blue    Unga~ No Hair    185
## 4 3 Abom~ Marvel C~ Male   green   Huma~ No Hair    203

```

¹⁶ `relocate()` не позволяет переименовывать колонки в отличие от `select()` и `rename()`

```

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

```

`relocate()` очень хорошо работает в сочетании с выбором колонок с помощью `tidyselect`. Например, можно передвинуть в одно место все колонки с одним типом данных:

```

heroes %>%
  relocate(Publisher, where(is.numeric), .after = name)

```

```

## # A tibble: 734 x 11
##   name Publisher   X1 Height Weight Gender `Eye color` `Race` `Hair color`
##   <chr> <chr>     <dbl>  <dbl>  <dbl> <chr>   <chr>   <chr>
## 1 A-Bo~ Marvel      0     203    441  Male   yellow  Human  No Hair
## 2 Abe ~ Dark Hor~   1     191     65  Male   blue   Icth~  No Hair
## 3 Abin~ DC Comics    2     185     90  Male   blue   Unga~  No Hair
## 4 Abom~ Marvel      3     203    441  Male   green  Huma~  No Hair
## 5 Abra~ Marvel      4     NA     NA  Male   blue   Cosm~ Black
## 6 Abso~ Marvel      5     193    122  Male   blue   Human  No Hair
## 7 Adam~ NBC - He~    6     NA     NA  Male   blue   <NA>  Blond
## 8 Adam~ DC Comics    7     185     88  Male   blue   Human  Blond
## 9 Agen~ Marvel      8     173     61  Female blue   <NA>  Blond
## 10 Agen~ Marvel     9     178     81  Male   brown  Human Brown
## # ... with 724 more rows, and 2 more variables: `Skin color` <chr>,
## #   Alignment <chr>

```

Последняя важная функция для выбора колонок — `pull()`. Эта функция делает то же самое, что и индексирование с помощью `$`, т.е. вытаскивает из тиббла вектор с выбранным названием. Это лучше вписывается в логику tidyverse, поскольку позволяет извлечь колонку из тиббла с использованием пайпа:

```

heroes %>%
  select(Height) %>%
  pull() %>%
  head()

```

```

## [1] 203 191 185 203 NA 193

```

```
heroes %>%
  pull(Height) %>%
  head()
```

```
## [1] 203 191 185 203 NA 193
```

У функции `pull()` есть аргумент `name =`, который позволяет создать проименованный вектор:

```
heroes %>%
  pull(Height, name) %>%
  head()
```

	A-Bomb	Abe Sapien	Abin Sur	Abomination	Abraxas
##	203	191	185	203	NA
## Absorbing Man					
##	193				

В отличие от базового R, tidyverse никогда не сокращает имплицитно результат вычислений до вектора, поэтому функция `pull()` - это основной способ извлечения колонки из тибла как вектора.

5.11 Работа со строками тибла

5.11.1 Выбор строк по номеру: `dplyr::slice()`

Начнем с выбора строк. Функция `dplyr::slice()` выбирает строчки по их числовому индексу.

```
heroes %>%
  slice(1:3)
```

```
## # A tibble: 3 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
## # ... with 3 more variables: `Skin color` <chr>, Alignment <chr>, Weight <dbl>
```

5.11.2 Выбор строк по условию: `dplyr::filter()`

Функция `dplyr::filter()` делает то же самое, что и `slice()`, но уже по условию. Причем для условий нужно использовать не векторы из тибла, а название колонок (без кавычек) как будто бы они были переменными в окружении.

```
heroes %>%
  filter(Publisher == "DC Comics")

## # A tibble: 215 x 11
##   X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1 2 Abin~ Male blue Unga~ No Hair     185 DC Comics
## 2 7 Adam~ Male blue Human Blond      185 DC Comics
## 3 13 Alan~ Male blue <NA> Blond      180 DC Comics
## 4 16 Alfr~ Male blue Human Black     178 DC Comics
## 5 19 Amazo Male red Andr~ <NA>       257 DC Comics
## 6 27 Anim~ Male blue Human Blond      183 DC Comics
## 7 31 Anti~ Male yellow God ~ No Hair   61 DC Comics
## 8 35 Aqua~ Male blue <NA> Blond      NA DC Comics
## 9 36 Aqua~ Male blue Atla~ Black      178 DC Comics
## 10 37 Aqua~ Male blue Atla~ Blond     185 DC Comics
## # ... with 205 more rows, and 3 more variables: `Skin color` <chr>,
## #   Alignment <chr>, Weight <dbl>
```

5.11.3 Семейство функций slice()

У функции `slice()` есть множество родственников, которые объединяют функционал обычного `slice()` и `filter()`. Например, с помощью функций `dplyr:::slice_max()` и `dplyr:::slice_min()` можно выбрать заданное количество строк, содержащих наибольшие или наименьшие значения по колонке соответственно:

```
heroes %>%
  slice_max(Weight, n = 3)

## # A tibble: 3 x 11
##   X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1 575 Sasq~ Male red <NA> Orange     305 Marvel C-
## 2 373 Jugg~ Male blue Human Red        287 Marvel C-
## 3 203 Dark~ Male red New ~ No Hair    267 DC Comics
## # ... with 3 more variables: `Skin color` <chr>, Alignment <chr>, Weight <dbl>
```

```
heroes %>%
  slice_min(Weight, n = 3)

## # A tibble: 3 x 11
##   X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
```

```
## 1 346 Iron~ Male blue <NA> No Hair NA Marvel C~
## 2 302 Groot Male yellow Flor~ <NA> 701 Marvel C~
## 3 350 Jack~ Male blue Human Brown 71 Dark Hor~
## # ... with 3 more variables: `Skin color` <chr>, Alignment <chr>, Weight <dbl>
```

Функция `slice_sample()` позволяет выбирать заданное количество случайных строчек:

```
heroes %>%
  slice_sample(n = 3)
```

```
## # A tibble: 3 x 11
##   X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1 274 Gamo~ Female yellow Zen~ Black 183 Marvel C~
## 2 68 Batm~ Male blue Human black 188 DC Comics
## 3 76 Beet~ Male <NA> <NA> <NA> NA Marvel C~
## # ... with 3 more variables: `Skin color` <chr>, Alignment <chr>, Weight <dbl>
```

Или же долю строчек:

```
heroes %>%
  slice_sample(prop = .01)
```

```
## # A tibble: 7 x 11
##   X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1 15 Alex~ Male <NA> <NA> <NA> NA "NBC - H~
## 2 126 Boba~ Male brown Huma~ Black 183 "George ~
## 3 119 Bloo~ Female blue Human Brown 218 "Marvel ~
## 4 324 Herc~ Male blue Demi~ Brown 196 "Marvel ~
## 5 490 Nigh~ Male yellow <NA> Indigo 175 "Marvel ~
## 6 164 Catw~ Female green Human Black 175 "DC Comi~
## 7 86 Bion~ Female blue Cybo~ Black NA ""
## # ... with 3 more variables: `Skin color` <chr>, Alignment <chr>, Weight <dbl>
```

Если поставить значение параметра `prop =` равным 1, то таким образом можно перемешать порядок строчек в тibble:

```
heroes %>%
  slice_sample(prop = 1)
```

```
## # A tibble: 734 x 11
##   X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
```

```

## 1 240 Emma~ Female blue <NA> Blond 178 Marvel C-
## 2 374 Junk~ Male <NA> Muta~ <NA> NA Marvel C-
## 3 661 Thor Male blue Asga~ Blond 198 Marvel C-
## 4 235 Elas~ Female brown Human Brown 168 Dark Hor-
## 5 310 Havok Male blue Muta~ Blond 183 Marvel C-
## 6 273 Gamb~ Male red Muta~ Brown 185 Marvel C-
## 7 597 Silk Female brown Human Black NA Marvel C-
## 8 480 Myst~ Female yellow (wi~ Muta~ Red / Orange 178 Marvel C-
## 9 277 Gene~ Male black Kryp~ Black NA DC Comics
## 10 42 Ares Male brown <NA> Brown 185 Marvel C-
## # ... with 724 more rows, and 3 more variables: `Skin color` <chr>,
## # Alignment <chr>, Weight <dbl>

```

5.11.4 Удаление строчек с NA: tidyverse::drop_na()

Если нужно выбрать только строчки без пропущенных значений, то можно воспользоваться удобной функцией `tidyverse::drop_na()`.

```

heroes %>%
  drop_na()

```

```

## # A tibble: 50 x 11
##       X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1     1 Abe ~ Male blue Icth~ No Hair 191 Dark Hor-
## 2     2 Abin~ Male blue Unga~ No Hair 185 DC Comics
## 3    34 Apoc~ Male red Muta~ Black 213 Marvel C-
## 4    39 Arch~ Male blue Muta~ Blond 183 Marvel C-
## 5    41 Ardi~ Female white Alien Orange 193 Marvel C-
## 6    56 Azaz~ Male yellow Neya~ Black 183 Marvel C-
## 7    74 Beast Male blue Muta~ Blue 180 Marvel C-
## 8    75 Beas~ Male green Human Green 173 DC Comics
## 9    92 Biza~ Male black Biza~ Black 191 DC Comics
## 10   108 Blac~ Male red Demon White 191 Marvel C-
## # ... with 40 more rows, and 3 more variables: `Skin color` <chr>,
## # Alignment <chr>, Weight <dbl>

```

Можно выбрать колонки, наличие NA в которых будет приводить к удалению соответствующих строчек (не затрагивая другие строчки, в которых есть NA в остальных столбцах).

```

heroes %>%
  drop_na(Weight)

```

```

## # A tibble: 495 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    5 Abso~ Male   blue   Human No Hair     193 Marvel C~
## 6    7 Adam~ Male   blue   Human Blond      185 DC Comics
## 7    8 Agen~ Female blue   <NA> Blond      173 Marvel C~
## 8    9 Agen~ Male   brown   Human Brown     178 Marvel C~
## 9   10 Agen~ Male   <NA>   <NA> <NA>      191 Marvel C~
## 10   11 Air~~ Male   blue   <NA> White      188 Marvel C~
## # ... with 485 more rows, and 3 more variables: `Skin color` <chr>,
## #   Alignment <chr>, Weight <dbl>
```

Для выбора колонок в `drop_na()` используется `tidyselect`, с которым мы недавно познакомились (??).

5.11.5 Сортировка строк: `dplyr::arrange()`

Функция `dplyr::arrange()` сортирует строчки от меньшего к большему (или по алфавиту - для текстовых значений) по выбранной колонке.

```
heroes %>%
  arrange(Weight)
```

```
## # A tibble: 734 x 11
##      X1 name Gender `Eye color` Race `Hair color` Height Publisher
##      <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1    346 Iron~ Male   blue   <NA> No Hair      NA Marvel C~
## 2    302 Groot Male   yellow Flor~ <NA>      701 Marvel C~
## 3    350 Jack~ Male   blue   Human Brown     71 Dark Hor~
## 4    272 Gala~ Male   black  Cosm~ Black     876 Marvel C~
## 5    731 Yoda~ Male   brown  Yoda~ White     66 George L~
## 6    255 Fin ~ Male   red   Kaka~ No Hair    975 Marvel C~
## 7    330 Howa~ Male   brown  <NA> Yellow     79 Marvel C~
## 8    396 Kryp~ Male   blue   Kryp~ White     64 DC Comics
## 9    568 Rock~ Male   brown  Anim~ Brown    122 Marvel C~
## 10   208 Dash~ Male   blue   Human Blond    122 Dark Hor~
## # ... with 724 more rows, and 3 more variables: `Skin color` <chr>,
## #   Alignment <chr>, Weight <dbl>
```

Чтобы отсортировать в обратном порядке, воспользуйтесь функцией `desc()`.

```
heroes %>%
  arrange(desc(Weight))
```

```
## # A tibble: 734 x 11
##       X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr>    <chr> <chr>    <dbl> <chr>
## 1 575 Sasq~ Male  red      <NA> Orange     305  Marvel C~
## 2 373 Jugg~ Male  blue     Human Red      287  Marvel C~
## 3 203 Dark~ Male  red      New ~ No Hair  267  DC Comics
## 4 283 Giga~ Female green  <NA> Red        62.5 DC Comics
## 5 331 Hulk  Male  green   Huma~ Green     244  Marvel C~
## 6 549 Red ~ Male  yellow  Huma~ Black     213  Marvel C~
## 7 119 Bloo~ Female blue   Human Brown    218  Marvel C~
## 8 718 Wolf~ Female green  <NA> Auburn    366  Marvel C~
## 9 657 Than~ Male  red      Eter~ No Hair  201  Marvel C~
## 10 0 A-Bo~ Male  yellow  Human No Hair  203  Marvel C~
## # ... with 724 more rows, and 3 more variables: `Skin color` <chr>,
## #   Alignment <chr>, Weight <dbl>
```

Можно сортировать по нескольким колонкам сразу. В таких случаях удобно в качестве первой переменной выбирать переменную, обозначающую принадлежность к группе, а в качестве второй — континуальную числовую переменную:

```
heroes %>%
  arrange(Gender, desc(Weight))
```

```
## # A tibble: 734 x 11
##       X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr>    <chr> <chr>    <dbl> <chr>
## 1 283 Giga~ Female green  <NA> Red        62.5 DC Comics
## 2 119 Bloo~ Female blue   Human Brown    218  Marvel C~
## 3 718 Wolf~ Female green  <NA> Auburn    366  Marvel C~
## 4 591 She~~ Female green  Human Green    201  Marvel C~
## 5 320 Hela  Female green  Asga~ Black     213  Marvel C~
## 6 686 Valk~ Female blue   <NA> Blond     191  Marvel C~
## 7 596 Sif   Female blue   Asga~ Black     188  Marvel C~
## 8 271 Frig~ Female blue   <NA> White     180  Marvel C~
## 9 667 Thun~ Female green  <NA> Red       218  Marvel C~
## 10 592 She~~ Female blue  Huma~ No Hair  183  Marvel C~
## # ... with 724 more rows, and 3 more variables: `Skin color` <chr>,
## #   Alignment <chr>, Weight <dbl>
```

5.12 Создание колонок: `dplyr::mutate()` и `dplyr::transmute()`

Функция `dplyr::mutate()` позволяет создавать новые колонки в тиббле.

```
heroes %>%
  mutate(imt = Weight/(Height/100)^2) %>%
  select(name, imt) %>%
  arrange(desc(imt))

## # A tibble: 734 x 2
##       name      imt
##   <chr>     <dbl>
## 1 Utgard-Loki 2510.
## 2 Giganta     1613.
## 3 Red Hulk    139.
## 4 Darkseid    115.
## 5 Machine Man 114.
## 6 Thanos      110.
## 7 Destroyer    108.
## 8 A-Bomb      107.
## 9 Abomination 107.
## 10 Hulk        106.
## # ... with 724 more rows
```

`dplyr::transmute()` - это аналог `mutate()`, который не только создает новые колонки, но и сразу же выкидывает все старые:

```
heroes %>%
  transmute(imt = Weight/(Height/100)^2)

## # A tibble: 734 x 1
##       imt
##   <dbl>
## 1 107.
## 2 17.8
## 3 26.3
## 4 107.
## 5 NA
## 6 32.8
## 7 NA
## 8 25.7
## 9 20.4
## 10 25.6
## # ... with 724 more rows
```

Внутри `mutate()` и `transmute()` мы можем использовать либо векторизованные

операции (длина новой колонки должна равняться длине датафрейма), либо операции, которые возвращают одно значение. В последнем случае значение будет одинаковым на всю колонку, т.е. будет работать правило ресайклнга (??):

```
heroes %>%
  transmute(name, weight_mean = mean(Weight, na.rm = TRUE))

## # A tibble: 734 x 2
##   name      weight_mean
##   <chr>        <dbl>
## 1 A-Bomb     112.
## 2 Abe Sapien 112.
## 3 Abin Sur   112.
## 4 Abomination 112.
## 5 Abraxas    112.
## 6 Absorbing Man 112.
## 7 Adam Monroe 112.
## 8 Adam Strange 112.
## 9 Agent 13    112.
## 10 Agent Bob   112.
## # ... with 724 more rows
```

Однако в функциях `mutate()` и `transmute()` правило ресайклнга не будет работать в остальных случаях: если полученный вектор будет не равен 1 или длине датафрейма, то мы получим ошибку.

```
heroes %>%
  mutate(one_and_two = 1:2)

## Error: Problem with `mutate()` input `one_and_two`.
## x Input `one_and_two` can't be recycled to size 734.
## i Input `one_and_two` is `1:2`.
## i Input `one_and_two` must be size 734 or 1, not 2.
```

Это не баг, а фича: авторы пакета `dplyr` считают, что ресайклнг кратных друг другу векторов — это слишком удобное место для выстрелов себе в ногу. Поэтому в таких случаях разработчики `dplyr` рекомендуют использовать функцию `rep()`, знакомую нам уже очень давно (??).

```
heroes %>%
  mutate(one_and_two = rep(1:2, length.out = nrow(.)))

## # A tibble: 734 x 12
##   name  Gender `Eye color` Race  `Hair color` Height Publisher
##   <dbl> <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 4 more variables: `Skin color` <chr>,
## #   Alignment <chr>, Weight <dbl>, one_and_two <int>

```

5.13 Агрегация данных в тиббле

5.13.1 Подытоживание: `summarise()`

Агрегация по группам - это очень часто возникающая задача, например, это может использоваться для усреднения данных по испытуемым или условиям. Сделать агрегацию в датафрейме удобной Хэдли Уикхэм пытался еще в предшественнике `dplyr`, пакете `plyr`. `dplyr` позволяет делать агрегацию очень симпатичным и понятным способом. Агрегация в `dplyr` состоит из двух этапов: группировки (`group_by()`) и подытоживания (`summarise()`). Начнем с последнего.

Функция `dplyr::summarise()`¹⁷ позволяет агрегировать данные в тиббле. Работает она очень похоже на `mutate()`, но если внутри `mutate()` используются векторизованные функции, возвращающие вектор такой же длины, что и колонки, использовавшиеся для расчетов, то в `summarise()` используются функции, которые возвращают вектор длиной 1. Например, `min()`, `mean()`, `max()` и т.д. Можно создавать несколько колонок через запятую (это работает и для `mutate()`).

```

heroes %>%
  mutate(imt = Weight/(Height/100)^2) %>%
  summarise(min(imt, na.rm = TRUE),
            max(imt, na.rm = TRUE))

## # A tibble: 1 x 2
##   `min(imt, na.rm = TRUE)` `max(imt, na.rm = TRUE)`
##                 <dbl>                <dbl>
## 1             0.0814            2510.

```

В `dplyr` есть дополнительные суммирующие функции для более удобного индексирования в стиле tidyverse. Например, функции `dplyr::nth()`, `dplyr::first()` и

¹⁷У функции `dplyr::summarise()` есть синоним `dplyr::summarize()`, которая делает абсолютно то же самое. Просто потому что в американском английском и британском английском это слово пишется по-разному.

`dplyr::last()`, которые позволяют вытаскивать значения из вектора по индексу (что-то вроде `slice()`, но для векторов)

```
heroes %>%
  mutate(imt = Weight/(Height/100)^2) %>%
  arrange(imt) %>%
  summarise(first = first(imt),
            tenth = nth(imt, 10),
            last = last(imt))

## # A tibble: 1 x 3
##   first tenth last
##     <dbl> <dbl> <dbl>
## 1 0.0814  16.7  NA
```

В отличие от `mutate()`, функции внутри `summarise()` вполне позволяют функциям внутри возвращать вектор из нескольких значений, создавая тиббл такой же длины, как и получившийся вектор.

```
heroes %>%
  mutate(imt = Weight/(Height/100)^2) %>%
  summarise(imt_range = range(imt, na.rm = TRUE)) #      range()

## # A tibble: 2 x 1
##   imt_range
##       <dbl>
## 1 0.0814
## 2 2510.
```

5.13.2 Группировка: `group_by()`

`dplyr::group_by()` - это функция для группировки данных в тиббле по дискретной переменной для дальнейшей агрегации с помощью `summarise()`. После применения `group_by()` тиббл будет выглядеть так же, но у него появится атрибут `groups`¹⁸:

```
heroes %>%
  group_by(Gender)

## # A tibble: 734 x 11
## # Groups:   Gender [3]
##       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~
```

¹⁸Снять группировку можно с помощью функции `ungroup()`.

```

## 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>

```

Если после этого применить на тibble функию `summarise()`, то мы получим не тibble длиной один, а тibble со значением для каждой из групп.

```

heroes %>%
  mutate(imt = Weight/(Height/100)^2) %>%
  group_by(Gender) %>%
  summarise(min(imt, na.rm = TRUE),
            max(imt, na.rm = TRUE))

## `summarise()` ungrouping output (override with `.`groups` argument)

## # A tibble: 3 x 3
##   Gender `min(imt, na.rm = TRUE)` `max(imt, na.rm = TRUE)`
##   <chr>          <dbl>           <dbl>
## 1 Female         15.5            1613.
## 2 Male           0.0814         2510.
## 3 <NA>           16.3            114.

```

Схематически это выглядит вот так:



5.13.3 Подсчет строк: `dplyr::n()`, `dplyr::count()`

Для подсчет количества значений можно воспользоваться функцией `n()`.

```

heroes %>%
  group_by(Gender) %>%

```

```
summarise(n = n())

## `summarise()` ungrouping output (override with `groups` argument)

## # A tibble: 3 x 2
##   Gender     n
##   <chr>    <int>
## 1 Female    200
## 2 Male      505
## 3 <NA>       29
```

Функция `n()` вместе с `group_by()` внутри `filter()` позволяет удобным образом “отрезать” от тиббла редкие группы...

```
heroes %>%
  group_by(Race) %>%
  filter(n() > 10) %>%
  select(name, Race)

## # A tibble: 611 x 2
## # Groups:   Race [6]
##   name        Race
##   <chr>      <chr>
## 1 A-Bomb     Human
## 2 Abomination Human / Radiation
## 3 Absorbing Man Human
## 4 Adam Monroe <NA>
## 5 Adam Strange Human
## 6 Agent 13    <NA>
## 7 Agent Bob   Human
## 8 Agent Zero  <NA>
## 9 Air-Walker   <NA>
## 10 Ajax        Cyborg
## # ... with 601 more rows
```

или же наоборот, выделить только маленькие группы:

```
heroes %>%
  group_by(Race) %>%
  filter(n() == 1) %>%
  select(name, Race)
```

```
## # A tibble: 34 x 2
## # Groups:   Race [34]
##   name        Race
```

```

## <chr>      <chr>
## 1 Abe Sapien  Icthyo Sapien
## 2 Abin Sur   Ungaran
## 3 Alien       Xenomorph XX121
## 4 Azazel     Neyaphem
## 5 Bizarro    Bizarro
## 6 Boba Fett   Human / Clone
## 7 Darth Maul  Dathomirian Zabrak
## 8 Fin Fang Foom Kakarantharaian
## 9 Gamora     Zen-Whoberian
## 10 Gladiator  Strontian
## # ... with 24 more rows

```

Таблицу частот можно создать без `group_by()` и `summarise(n = n())`. Функция `count()` заменяет эту конструкцию:

```

heroes %>%
  count(Gender)

```

```

## # A tibble: 3 x 2
##   Gender     n
##   <chr>   <int>
## 1 Female    200
## 2 Male      505
## 3 <NA>      29

```

Эту таблицу частот удобно сразу проранжировать, указав в параметре `sort =` значение `TRUE`.

```

heroes %>%
  count(Gender, sort = TRUE)

```

```

## # A tibble: 3 x 2
##   Gender     n
##   <chr>   <int>
## 1 Male      505
## 2 Female    200
## 3 <NA>      29

```

Функция `count()`, несмотря на свою простоту, является одной из наиболее используемых в tidyverse.

5.13.4 Уникальные значения: `dplyr::distinct()`

`dplyr::distinct()` - это более быстрый аналог `unique()`, позволяет извлекать уникальные значения для одной или нескольких колонок.

```
heroes %>%
  distinct(Gender)
```

```
## # A tibble: 3 x 1
##   Gender
##   <chr>
## 1 Male
## 2 Female
## 3 <NA>
```

```
heroes %>%
  distinct(Gender, Race)
```

```
## # A tibble: 81 x 2
##   Gender Race
##   <chr>  <chr>
## 1 Male   Human
## 2 Male   Icthyo Sapien
## 3 Male   Ungaran
## 4 Male   Human / Radiation
## 5 Male   Cosmic Entity
## 6 Male   <NA>
## 7 Female <NA>
## 8 Male   Cyborg
## 9 Male   Xenomorph XX121
## 10 Male  Android
## # ... with 71 more rows
```

Иногда нужно агрегировать данные, но при этом сохранить исходную структуру тиббла. Например, нужно посчитать размер групп или посчитать средние значения по группе для последующего сравнения с индивидуальными значениями.

5.13.5 Создание колонок с группировкой

В tidyverse это можно сделать с помощью сочетания `group_by()` и `mutate()` (вместо `summarise()`):

```
heroes %>%
  group_by(Race) %>%
  mutate(Race_n = n()) %>%
  select(Race, name, Gender, Race_n)
```

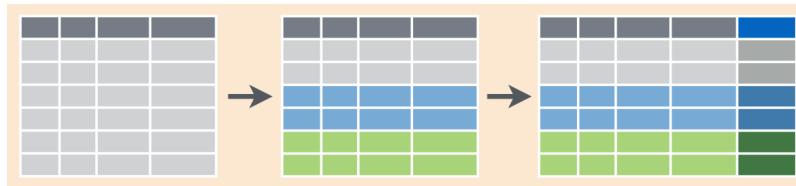
```
## # A tibble: 734 x 4
## # Groups:   Race [62]
```

```

##   Race           name      Gender Race_n
##   <chr>         <chr>     <chr>    <int>
## 1 Human          A-Bomb    Male     208
## 2 Icthyo Sapien Abe Sapien Male      1
## 3 Ungaran       Abin Sur   Male     1
## 4 Human / Radiation Abomination Male    11
## 5 Cosmic Entity Abraxas   Male     4
## 6 Human          Absorbing Man Male    208
## 7 <NA>           Adam Monroe Male   304
## 8 Human          Adam Strange Male   208
## 9 <NA>           Agent 13   Female  304
## 10 Human          Agent Bob   Male    208
## # ... with 724 more rows

```

Результаты агрегации были записаны в отдельную колонку, при этом значения этой колонки внутри одной группы повторяются:



5.14 Трансформация нескольких колонок: `dplyr::across()`

Допустим, вы хотите посчитать среднюю массу и рост, группируя по полу супергероев. Можно посчитать это внутри одного `summarise()`, используя запятую:

```

heroes %>%
  group_by(Gender) %>%
  summarise(height = mean(Height, na.rm = TRUE),
            weight = mean(Weight, na.rm = TRUE))

```

```

## `summarise()` ungrouping output (override with ` `.groups` argument)

## # A tibble: 3 x 3
##   Gender height weight
##   <chr>    <dbl>  <dbl>
## 1 Female    175.   78.8
## 2 Male      192.   126.
## 3 <NA>     177.   129.

```

Если таких колонок будет много, то это уже станет сильно неудобным, нам придется много копировать код, а это чревато ошибками и очень скучно.

Поэтому в `dplyr` есть функция для операций над несколькими колонками сразу:

`dplyr::across()`¹⁹. Эта функция работает похожим образом на функции семейства `apply()` и использует `tidyselect` для выбора колонок.

Таким образом, конструкции с функцией `across()` можно разбить на три части:

1. Выбор колонок с помощью `tidyselect`. Здесь работают все те приемы, которые мы изучили при выборе колонок `(??)`.
2. Собственно применение функции `across()`. Первый аргумент `.col` — колонки, выбранные на первом этапе с помощью `tidyselect`, по умолчанию это `everything()`, т.е. все колонки. Второй аргумент `.fns` — это функция или целый список из функций, которые будут применены к выбранным колонкам. Если функции требуют дополнительных аргументов, то они могут быть перечислены внутри `across()`.
3. Использование `summarise()` или другой функции `dplyr`. В этом случае в качестве аргумента для функции используется результат работы функции `across()`.

Вот такой вот бутерброд выходит. Давайте посмотрим, как это работает на практике и посчитаем среднее значение по колонкам `Height` и `Weight`.

```
heroes %>%
  group_by(Gender) %>%
  summarise(across(c(Height,Weight), mean))

## `summarise()` ungrouping output (override with `.`groups` argument)

## # A tibble: 3 x 3
##   Gender Height Weight
##   <chr>    <dbl>   <dbl>
## 1 Female     NA     NA
## 2 Male       NA     NA
## 3 <NA>      NA     NA
```

Здесь мы столкнулись с уже известной нам проблемой: функция `mean()` при столкновении хотя бы с одним `NA` будет возвращать `NA`, если мы не изменим параметр `na.rm = TRUE`. Как и в случае с функциями семейства `apply()` (`@ref{apply_f}`), дополнительные параметры для функции можно перечислить через запятую после самой функции:

```
heroes %>%
  group_by(Gender) %>%
  summarise(across(c(Height, Weight), mean, na.rm = TRUE))
```

¹⁹Функция `across()` появилась в пакете `dplyr` относительно недавно, до этого для работы с множественными колонками в `tidyverse` использовались многочисленные функции `*_at()`, `*_if()`, `*_all()`, например, `summarise_at()`, `summarise_if()`, `summarize_all()`. Эти функции до сих пор присутствуют в `dplyr`, но считаются устаревшими. Другая альтернатива - использование пакета `purrr` `(??)` или семейства функций `apply()` (`@ref{apply_f}`).

```
## `summarise()` ungrouping output (override with `groups` argument)

## # A tibble: 3 x 3
##   Gender Height Weight
##   <chr>    <dbl>   <dbl>
## 1 Female    175.   78.8
## 2 Male      192.   126.
## 3 <NA>     177.   129.
```

До этого мы просто использовали выбор колонок по их названию. Но именно внутри `across()` использование `tidyselect` раскрывается как удивительно элегантный и мощный инструмент. Например, можно посчитать среднее для всех numeric колонок:

```
heroes %>%
  drop_na(Height, Weight) %>%
  group_by(Gender) %>%
  summarise(across(where(is.numeric), mean, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `groups` argument)

## # A tibble: 3 x 4
##   Gender   X1 Height Weight
##   <chr>    <dbl>   <dbl>   <dbl>
## 1 Female    394.   174.   78.3
## 2 Male      369.   193.   126.
## 3 <NA>     375.   182.   129.
```

Или длину строк для строковых колонок. Для этого нам понадобится вспомнить, как создавать анонимные функции (`@ref(anon_f)`).

```
heroes %>%
  group_by(Gender) %>%
  summarise(across(where(is.character),
                  function(x) mean(nchar(x), na.rm = TRUE)))
```

```
## `summarise()` ungrouping output (override with `groups` argument)

## # A tibble: 3 x 8
##   Gender name `Eye color` `Race` `Hair color` Publisher `Skin color` Alignment
##   <chr>    <dbl>        <dbl>       <dbl>        <dbl>       <dbl>        <dbl>
## 1 Female    9.04        4.68       6.42        5.05       11.5       4.57       3.88
## 2 Male      9.05        4.53       6.75        5.48       11.4       5.02       3.78
## 3 <NA>     9.48        5.16      10.1        6.44       11.9        4          3.96
```

Или же даже посчитать и то, и другое внутри одного `summarise()`!

```
heroes %>%
  group_by(Gender) %>%
  summarise(across(where(is.numeric), mean, na.rm = TRUE),
            across(where(is.character),
                   function(x) mean(nchar(x), na.rm = TRUE)))
```

`summarise()` ungrouping output (override with `groups` argument)

A tibble: 3 x 11

	Gender	X1	Height	Weight	name	Eye color	Race	Hair color	Publisher	
## 1	Female	395.	175.	78.8	9.04	4.68	6.42	5.05	11.5	<dbl>
## 2	Male	357.	192.	126.	9.05	4.53	6.75	5.48	11.4	<dbl>
## 3	<NA>	329	177.	129.	9.48	5.16	10.1	6.44	11.9	<dbl>

... with 2 more variables: `Skin color` <dbl>, Alignment <dbl>

Внутри одного `across()` можно применить не одну функцию к каждой из выбранных колонок, а сразу несколько функций для каждой из колонок. Для этого нам нужно использовать список функций (желательно - проименованный).

```
heroes %>%
  group_by(Gender) %>%
  summarise(across(c(Height, Weight),
                  list(minimum = min,
                       average = mean,
                       maximum = max),
                  na.rm = TRUE))
```

`summarise()` ungrouping output (override with `groups` argument)

A tibble: 3 x 7

	Gender	Height_minimum	Height_average	Height_maximum	Weight_minimum	
## 1	Female	62.5	175.	366	41	<dbl>
## 2	Male	15.2	192.	975	2	<dbl>
## 3	<NA>	108	177.	198	39	<dbl>

... with 2 more variables: Weight_average <dbl>, Weight_maximum <dbl>

Бот нам и понадобился список функций (@ref(functions_objects))!

```
heroes %>%
  group_by(Gender) %>%
  summarise(across(c(Height, Weight),
                  list(min = function(x) min(x, na.rm = TRUE),
                       mean = function(x) mean(x, na.rm = TRUE),
```

```

        max = function(x) max(x, na.rm = TRUE),
        na_n = function(x, ...) sum(is.na(x)))
    )
}

## `summarise()` ungrouping output (override with `groups` argument)

## # A tibble: 3 x 9
##   Gender Height_min Height_mean Height_max Height_na_n Weight_min Weight_mean
##   <chr>     <dbl>      <dbl>      <dbl>      <int>      <dbl>      <dbl>
## 1 Female     62.5      175.      366       56       41      78.8
## 2 Male       15.2      192.      975      147        2     126.
## 3 <NA>        108      177.      198       14       39     129.
## # ... with 2 more variables: Weight_max <dbl>, Weight_na_n <int>

```

Хотя основное применение функции `across()` — это массовое подытоживание с помощью `summarise()`, `across()` можно использовать и с другими функциями `dplyr`. Например, можно делать массовые операции с колонками с помощью `mutate()`:

```

heroes %>%
  mutate(across(where(is.character), as.factor))

## # A tibble: 734 x 11
##   X1 name  Gender `Eye color` Race  `Hair color` Height Publisher
##   <dbl> <fct> <fct>   <fct> <fct>   <dbl> <fct>
## 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` <fct>,
## #   Alignment <fct>, Weight <dbl>

```

Менее очевидный способ применения `across()` — использование `across()` внутри `count()` вместе с функцией `n_distinct()`, которая считает количество уникальных значений в векторе. Это позволяет посмотреть таблицу частот для группирующих переменных:

```

heroes %>%
  select(where(function(x) n_distinct(x) <= 6))

```

```

## # A tibble: 734 x 2
##   Gender Alignment
##   <chr>  <chr>
## 1 Male   good
## 2 Male   good
## 3 Male   good
## 4 Male   bad
## 5 Male   bad
## 6 Male   bad
## 7 Male   good
## 8 Male   good
## 9 Female good
## 10 Male  good
## # ... with 724 more rows

heroes %>%
  count(across(where(function(x) n_distinct(x) <= 6)))

## # A tibble: 11 x 3
##   Gender Alignment     n
##   <chr>  <chr>    <int>
## 1 Female bad        35
## 2 Female good       161
## 3 Female neutral    4
## 4 Male   bad        165
## 5 Male   good       316
## 6 Male   neutral    18
## 7 Male   <NA>        6
## 8 <NA>   bad         7
## 9 <NA>   good        19
## 10 <NA>  neutral      2
## 11 <NA>  <NA>        1

```

5.15 Объединение нескольких датафреймов

5.15.1 Соединение структурно схожих датафреймов: bind_rows(), bind_cols()

Для начала создадим следующие тибблы и сохраним их как dc, marvel и other_publishers:

```

dc <- heroes %>%
  filter(Publisher == "DC Comics") %>%
  group_by(Gender)

```

```

  summarise(weight_mean = mean(Weight, na.rm = TRUE))

## `summarise()` ungrouping output (override with `groups` argument)

dc

## # A tibble: 3 x 2
##   Gender weight_mean
##   <chr>      <dbl>
## 1 Female     76.8
## 2 Male       113.
## 3 <NA>       NaN

marvel <- heroes %>%
  filter(Publisher == "Marvel Comics") %>%
  group_by(Gender) %>%
  summarise(weight_mean = mean(Weight, na.rm = TRUE))

## `summarise()` ungrouping output (override with `groups` argument)

marvel

## # A tibble: 3 x 2
##   Gender weight_mean
##   <chr>      <dbl>
## 1 Female     80.1
## 2 Male       134.
## 3 <NA>       129.

other_publishers <- heroes %>%
  filter(!(Publisher %in% c("DC Comics", "Marvel Comics"))) %>%
  group_by(Gender) %>%
  summarise(weight_mean = mean(Weight, na.rm = TRUE))

## `summarise()` ungrouping output (override with `groups` argument)

other_publishers

## # A tibble: 3 x 2
##   Gender weight_mean
##   <chr>      <dbl>
## 1 Female     70.8

```

```
## 2 Male      111.
## 3 <NA>      NaN
```

Несколько тибллов можно объединить вертикально с помощью функции `bind_rows()`. Для корректного объединения тибллы должны иметь одинаковые названия колонок.

```
bind_rows(dc, marvel)
```

```
## # A tibble: 6 x 2
##   Gender weight_mean
##   <chr>     <dbl>
## 1 Female    76.8
## 2 Male      113.
## 3 <NA>      NaN
## 4 Female    80.1
## 5 Male      134.
## 6 <NA>      129.
```

Чтобы соединить тибллы горизонтально, воспользуйтесь функцией `bind_cols()`.

```
bind_cols(dc, marvel)
```

```
## New names:
## * Gender -> Gender...1
## * weight_mean -> weight_mean...2
## * Gender -> Gender...3
## * weight_mean -> weight_mean...4

## # A tibble: 3 x 4
##   Gender...1 weight_mean...2 Gender...3 weight_mean...4
##   <chr>          <dbl> <chr>          <dbl>
## 1 Female        76.8 Female        80.1
## 2 Male          113. Male          134.
## 3 <NA>          NaN  <NA>          129.
```

Функции `bind_rows()` и `bind_cols()` могут работать не только с двумя, но сразу с несколькими датафреймами.

```
bind_rows(dc, marvel, other_publishers)
```

```
## # A tibble: 9 x 2
##   Gender weight_mean
##   <chr>     <dbl>
## 1 Female    76.8
## 2 Male      113.
## 3 <NA>      NaN
```

```
## 4 Female      80.1
## 5 Male        134.
## 6 <NA>        129.
## 7 Female      70.8
## 8 Male        111.
## 9 <NA>        NaN
```

На входе в функции `bind_rows()` и `bind_cold()` можно подавать как сами датафреймы или тибллы через запятую, так и список из датафреймов/тибллов.

```
heroes_list_of_df <- list(DC = dc,
                           Marvel = marvel,
                           Other = other_publishers)
bind_rows(heroes_list_of_df)
```

```
## # A tibble: 9 x 2
##   Gender weight_mean
##   <chr>     <dbl>
## 1 Female     76.8
## 2 Male       113.
## 3 <NA>        NaN
## 4 Female     80.1
## 5 Male       134.
## 6 <NA>        129.
## 7 Female     70.8
## 8 Male       111.
## 9 <NA>        NaN
```

Чтобы не потерять, из какого датафрейма какие данные, можно указать любое строковое значение (название будущей колонки) для необязательного аргумента `.id =`.

```
bind_rows(heroes_list_of_df, .id = "Publisher")
```

```
## # A tibble: 9 x 3
##   Publisher Gender weight_mean
##   <chr>     <chr>     <dbl>
## 1 DC        Female     76.8
## 2 DC        Male      113.
## 3 DC        <NA>       NaN
## 4 Marvel    Female     80.1
## 5 Marvel    Male      134.
## 6 Marvel    <NA>       129.
## 7 Other     Female     70.8
## 8 Other     Male      111.
## 9 Other     <NA>       NaN
```

`bind_rows()` обычно используется, когда ваши данные находятся в разных файлах с одинаковой структурой. Тогда вы можете прочитать все таблицы в папке, сохранить их в качестве списка из датафреймов и объединить в один датафрейм с помощью `bind_rows()`.

5.15.2 Реляционные данные: `*_join()`

В реальности иногда возникает ситуация, когда нужно соединить две таблички, у которых есть общий столбец (или несколько столбцов), но все остальные столбцы различаются. Табличек может быть и больше, это может быть целая сеть таблиц, некоторые из которых содержат основные данные, а некоторые - дополнительные, которые необходимо на определенном этапе “включить” в анализ. Например, таблица с расшифровкой аббревиатур или сокращений вроде коротких названий стран или таблица телефонных кодов разных стран. Совокупность нескольких связанных друг с другом таблиц называют реляционными данными.

В случае с реляционными данными простых `bind_rows()` и `bind_cols()` становится недостаточно.

Эти две таблички нужно объединить (*join*). Эта задача обычно возникает не очень часто, обычно это происходит один-два раза в одном проекте, когда нужно дополнить имеющиеся данные дополнительной информацией извне или объединить два набора данных, обрабатывавшихся в разных программах. Всякий раз, когда такая задача возникает, это доставляет много боли. `dplyr` предлагает интуитивно понятный инструмент для объединения реляционных данных - семейство функций `*_join()`.

Возьмем для примера два тibble `band_members` и `band_instruments`, встроенных в `dplyr` специально для демонстрации работы функций `*_join()`.

```
band_members
```

```
## # A tibble: 3 x 2
##   name   band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name   plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

У этих двух тибллов есть колонка с одинаковым названием, которая по своему смыслу соединяет данные обоих тибллов. Такая колонка называется **ключом**. Ключ должен однозначно идентифицировать наблюдения²⁰.

Давайте попробуем поссоединять `band_members` и `band_instruments` разными вариантами `*_join()` и посмотрим, что у нас получится. Все эти функции имеют на входе два обязательных аргумента (`x` = и `y` =) в которые мы должны подставить два датафрейма/тиблла которые мы хотим объединить. Главное различие между этими функциями заключается в том, что они будут делать, если уникальные значения в ключах `x` и `y` не соответствуют друг другу.

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::left_join(a, b, by = "x1")
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::right_join(a, b, by = "x1")
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::inner_join(a, b, by = "x1")
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::full_join(a, b, by = "x1")
Join data. Retain all values, all rows.

- `left_join()`:

```
band_members %>%
  left_join(band_instruments)
```

```
## Joining, by = "name"
## # A tibble: 3 x 3
##   name  band  plays
```

²⁰Если ключи будут неуникальными, то функции `*_join()` не будут выдавать ошибку. Вместо этого они добавят в итоговую таблицу все возможные пересечения повторяющихся ключей. С этим нужно быть очень осторожным, поэтому рекомендуется, во-первых, проверять уникальность ключей на входе и, во-вторых, проверять тибл на выходе. Ну или использовать эту особенность работы функции `*_join()` себе во благо.

```
## <chr> <chr> <chr>
## 1 Mick Stones <NA>
## 2 John Beatles guitar
## 3 Paul Beatles bass
```

`left_join()` - это самая простая для понимания и самая используемая функция из семейства `*_join()`. Она как бы “дополняет” информацию из первого тиббла вторым тибблом. В этом случае сохраняются все уникальные наблюдения в `x`, но отбрасываются лишние наблюдения в тиббле `y`. Тем значениям, которым не нашлось соответствия в `y`, в колонках, взятых их `y`, ставятся значения `NA`.

Вы можете сами задать колонки-ключи параметром `by =`, по умолчанию это все колонки с одинаковыми названиями в двух тибблах.

```
band_members %>%
  left_join(band_instruments, by = "name")
```

```
## # A tibble: 3 x 3
##   name band   plays
##   <chr> <chr> <chr>
## 1 Mick Stones <NA>
## 2 John Beatles guitar
## 3 Paul Beatles bass
```

Часто случается, что колонки-ключи называются по-разному в двух тибблах. Их необязательно переименовывать, можно поставить соответствие вручную используя преименованный вектор:

```
band_members %>%
  left_join(band_instruments2, by = c("name" = "artist"))
```

```
## # A tibble: 3 x 3
##   name band   plays
##   <chr> <chr> <chr>
## 1 Mick Stones <NA>
## 2 John Beatles guitar
## 3 Paul Beatles bass
```

· `right_join()`:

```
band_members %>%
  right_join(band_instruments)
```

```
## Joining, by = "name"
## # A tibble: 3 x 3
```

```
##   name  band   plays
##   <chr> <chr>   <chr>
## 1 John Beatles guitar
## 2 Paul Beatles bass
## 3 Keith <NA>     guitar
```

`right_join()` отбрасывает строчки в `x`, которых не было в `y`, но сохраняет соответствующие строчки `y` — `left_join()` наоборот.

- `full_join()`:

```
band_members %>%
  full_join(band_instruments)
```

```
## Joining, by = "name"
## # A tibble: 4 x 3
##   name  band   plays
##   <chr> <chr>   <chr>
## 1 Mick Stones <NA>
## 2 John Beatles guitar
## 3 Paul Beatles bass
## 4 Keith <NA>     guitar
```

Функция `full_join()` сохраняет все строчки из `x` и `y`. Пожалуй, наиболее используемая функция после `left_join()` — благодаря `full_join()` вы точно ничего не потеряете при объединении.

- `inner_join()`:

```
band_members %>%
  inner_join(band_instruments)
```

```
## Joining, by = "name"
## # A tibble: 2 x 3
##   name  band   plays
##   <chr> <chr>   <chr>
## 1 John Beatles guitar
## 2 Paul Beatles bass
```

Функция `full_join()` сохраняет только строчки, которые присутствуют и в `x`, и в `y`.

- `semi_join()`:

```
band_members %>%
  semi_join(band_instruments)
```

```

## Joining, by = "name"

## # A tibble: 2 x 2
##   name   band
##   <chr> <chr>
## 1 John  Beatles
## 2 Paul  Beatles
· anti_join():

band_members %>%
  anti_join(band_instruments)

```

```

## Joining, by = "name"

## # A tibble: 1 x 2
##   name   band
##   <chr> <chr>
## 1 Mick  Stones

```

Функции `semi_join()` и `anti_join()` не присоединяют второй датафрейм/тиббл (у) к первому. Вместо этого они используются как некоторый словарь-фильтр для отделения только тех значений в x, которые есть в у (`semi_join()`) или, наоборот, которых нет в у (`anti_join()`).

5.16 Tidydata:tidyr::pivot_longer(),tidyr::pivot_wider()

Принцип tidy data предполагает, что каждая строчка содержит в себе одно измерение, а каждая колонка - одну характеристику. Тем не менее, это не говорит однозначно о том, как именно хранить повторные измерения. Их можно хранить как одну колонку для каждого измерения (широкий формат) и как две колонки: одна колонка - для идентификатора измерения, другая колонка - для записи самого измерения.

Это лучше понять на примере. Например, вес до и после прохождения курса. Как это лучше записать - как два числовых столбца (один испытуемый - одна строка) или же создать отдельную “группирующую” колонку, в которой будет написано время измерения, а в другой - измеренные значения (одно измерение - одна строка)?

- Широкий формат:

Студент	До курса по R	После курса по R
Маша	70	63
Рома	80	74
Антонина	86	71

- Длинный”формат:

Студент	Время измерения	Масса (кг)
Маша	До курса по R	70
Рома	До курса по R	80
Антонина	До курса по R	86
Маша	После курса по R	63
Рома	После курса по R	74
Антонина	После курса по R	71

На самом деле, оба варианта приемлемы, оба варианта возможны в реальных данных, а разные функции и статистические пакеты могут требовать от вас как длинный, так и широкий форматы.

Таким образом, нам нужно научиться переводить из широкого формата в длинный и наоборот.

- `tidy::pivot_longer()`: из широкого в длинный формат
- `tidy::pivot_wider()`: из длинного в широкий формат

```
new_diet <- tibble(
  student = c(" ", " ", " "),
  before_r_course = c(70, 80, 86),
  after_r_course = c(63, 74, 71)
)
new_diet

## # A tibble: 3 x 3
##   student  before_r_course after_r_course
##   <chr>        <dbl>         <dbl>
## 1          70            63
## 2          80            74
## 3          86            71
```

Тибл new_diet - это пример широкого формата данных.

Превратим тибл new_diet длинный:

```
new_diet %>%
  pivot_longer(cols = before_r_course:after_r_course,
               names_to = "measurement_time",
               values_to = "weight_kg")

## # A tibble: 6 x 3
##   student measurement_time weight_kg
##   <chr>     <chr>           <dbl>
## 1          before_r_course    70
## 2          after_r_course     63
## 3          before_r_course    80
## 4          after_r_course     74
## 5          before_r_course    86
## 6          after_r_course     71
```

```
## 2      after_r_course      63
## 3      before_r_course     80
## 4      after_r_course      74
## 5      before_r_course     86
## 6      after_r_course      71
```

А теперь обратно в короткий:

```
new_diet %>%
  pivot_longer(cols = before_r_course:after_r_course,
               names_to = "measurement_time",
               values_to = "weight_kg") %>%
  pivot_wider(names_from = "measurement_time",
              values_from = "weight_kg")

## # A tibble: 3 x 3
##   student before_r_course after_r_course
##   <chr>        <dbl>        <dbl>
## 1 1            70          63
## 2 2            80          74
## 3 3            86          71
```


Глава 6

Визуализация данных

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

6.1 Зачем визуализировать данные?

6.1.1 КвартетAnscombe

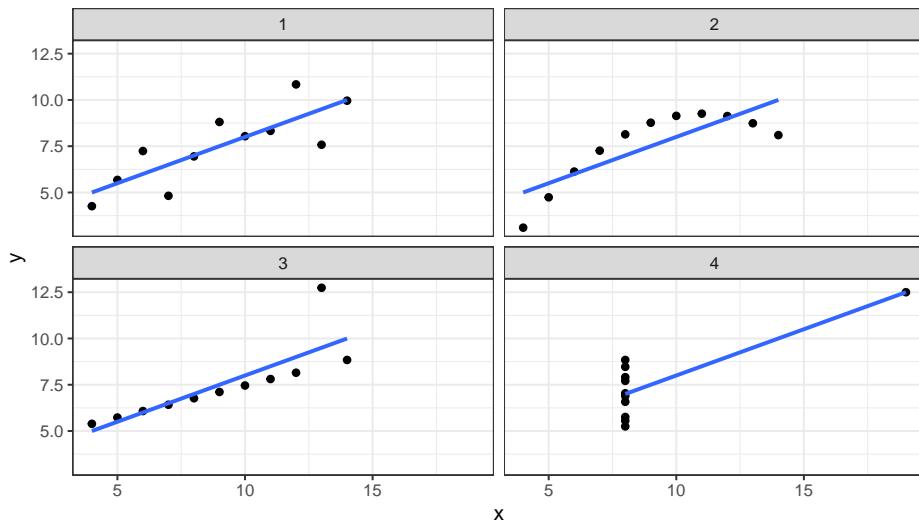
В работе Anscombe, F. J. (1973). "Graphs in Statistical Analysis" представлен следующий датасет:

```
quartet <- read_csv("https://raw.githubusercontent.com/agricolamz/2020-2021-ds4dh/master/data/anscombes_quartet.csv")
quartet
```

```
## # A tibble: 44 x 4
##       id dataset     x     y
##   <dbl>   <dbl> <dbl> <dbl>
## 1     1      1    10  8.04
## 2     1      1    10  9.14
## 3     1      1    10  7.46
## 4     1      1     8  6.58
## 5     2      2     8  6.95
## 6     2      2     8  8.14
## 7     2      2     8  6.77
## 8     2      2     8  5.76
## 9     3      1    13  7.58
## 10    3      1    13  8.74
## # ... with 34 more rows
```

```
quartet %>%
  group_by(dataset) %>%
  summarise(mean_X = mean(x),
            mean_Y = mean(y),
            sd_X = sd(x),
            sd_Y = sd(y),
            cor = cor(x, y),
            n_obs = n()) %>%
  select(-dataset) %>%
  round(2)
```

```
## # A tibble: 4 x 6
##   mean_X mean_Y  sd_X  sd_Y    cor n_obs
##     <dbl>   <dbl> <dbl> <dbl>  <dbl>   <dbl>
## 1      9     7.5  3.32  2.03  0.82     11
## 2      9     7.5  3.32  2.03  0.82     11
## 3      9     7.5  3.32  2.03  0.82     11
## 4      9     7.5  3.32  2.03  0.82     11
```



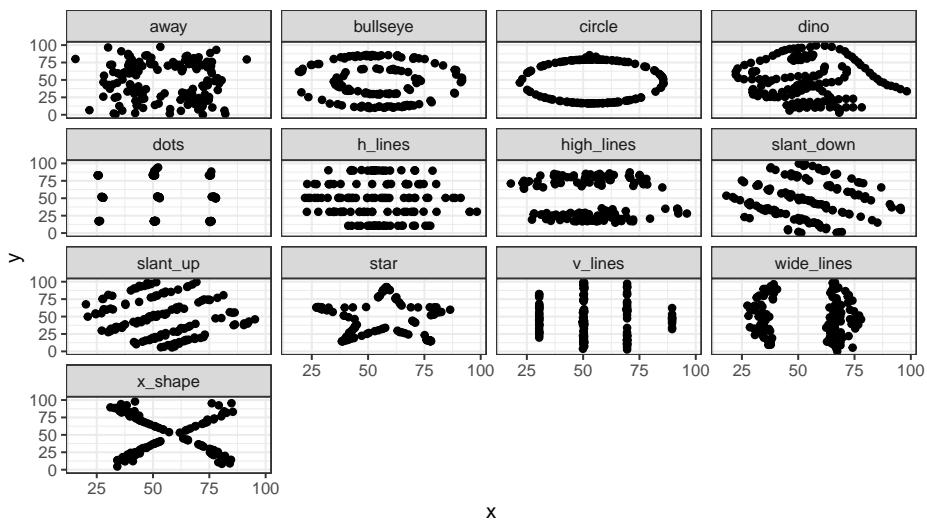
6.1.2 Датазаурус

В работе Matejka and Fitzmaurice (2017) “Same Stats, Different Graphs”¹ были представлены следующие датасеты:

¹<https://www.autodeskresearch.com/sites/default/files/SameStats-DifferentGraphs.pdf>

```
datasaurus <- read_csv("https://raw.githubusercontent.com/agricolamz/2020-2021-ds4dh/master/data/datasaurus")
```

```
## # A tibble: 1,846 x 3
##   dataset      x     y
##   <chr>    <dbl> <dbl>
## 1 dino      55.4  97.2
## 2 dino      51.5  96.0
## 3 dino      46.2  94.5
## 4 dino      42.8  91.4
## 5 dino      40.8  88.3
## 6 dino      38.7  84.9
## 7 dino      35.6  79.9
## 8 dino      33.1  77.6
## 9 dino      29.0  74.5
## 10 dino     26.2  71.4
## # ... with 1,836 more rows
```



```
datasaurus %>%
  group_by(dataset) %>%
  summarise(mean_X = mean(x),
            mean_Y = mean(y),
            sd_X = sd(x),
            sd_Y = sd(y),
            cor = cor(x, y),
            n_obs = n()) %>%
  select(-dataset) %>%
```

```
round(1)
```

```
## `summarise()` ungrouping output (override with `groups` argument)

## # A tibble: 13 x 6
##   mean_X  mean_Y  sd_X  sd_Y  cor n_obs
##   <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 54.3    47.8   16.8   26.9 -0.1  142
## 2 54.3    47.8   16.8   26.9 -0.1  142
## 3 54.3    47.8   16.8   26.9 -0.1  142
## 4 54.3    47.8   16.8   26.9 -0.1  142
## 5 54.3    47.8   16.8   26.9 -0.1  142
## 6 54.3    47.8   16.8   26.9 -0.1  142
## 7 54.3    47.8   16.8   26.9 -0.1  142
## 8 54.3    47.8   16.8   26.9 -0.1  142
## 9 54.3    47.8   16.8   26.9 -0.1  142
## 10 54.3   47.8   16.8   26.9 -0.1  142
## 11 54.3   47.8   16.8   26.9 -0.1  142
## 12 54.3   47.8   16.8   26.9 -0.1  142
## 13 54.3   47.8   16.8   26.9 -0.1  142
```

6.2 Основы ggplot2

Пакет `ggplot2` – современный стандарт для создания графиков в R. Для этого пакета пишут массу расширений². В сжатом виде информация про `ggplot2` содержиться здесь³.

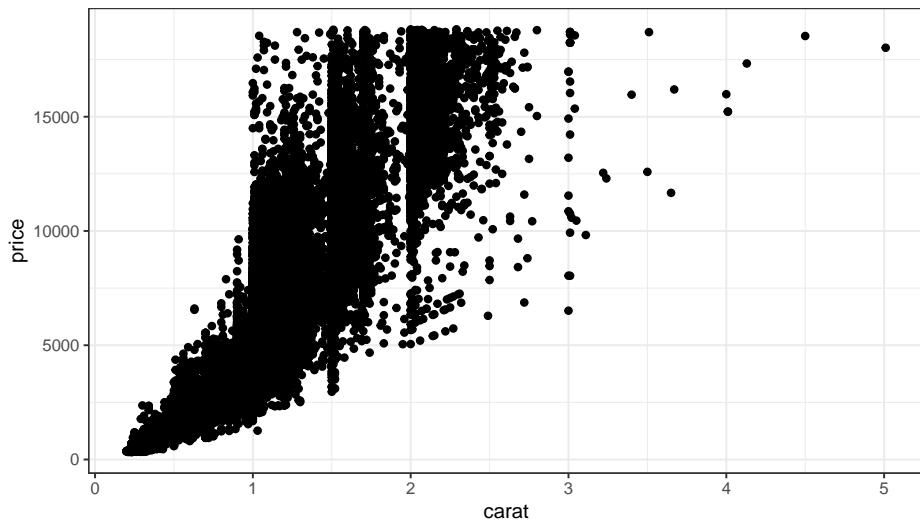
6.2.1 Диаграмма рассеяния (Scaterplot)

- `ggplot2`

```
ggplot(data = diamonds, aes(carat, price)) +
  geom_point()
```

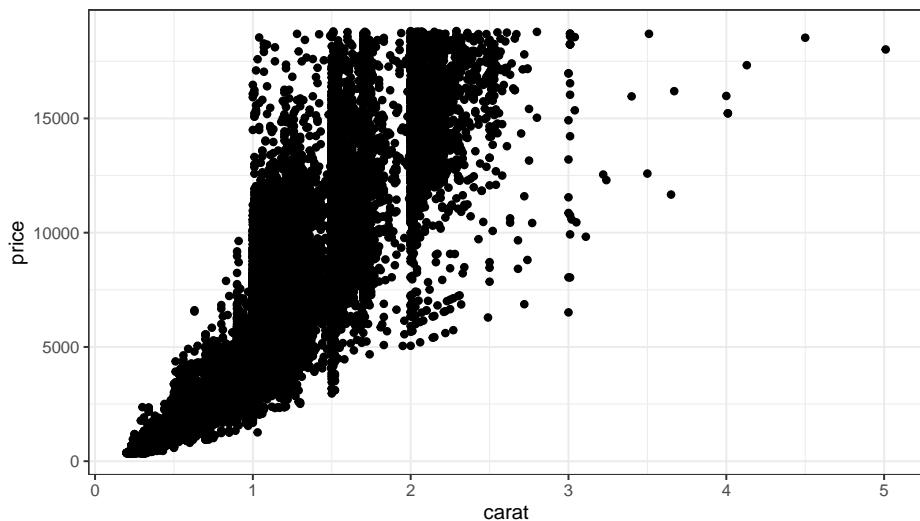
²<http://www.ggplot2-exts.org/gallery/>

³<https://github.com/rstudio/cheatsheets/raw/master/data-visualization-2.1.pdf>



```
• dplyr, ggplot2
```

```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point()
```

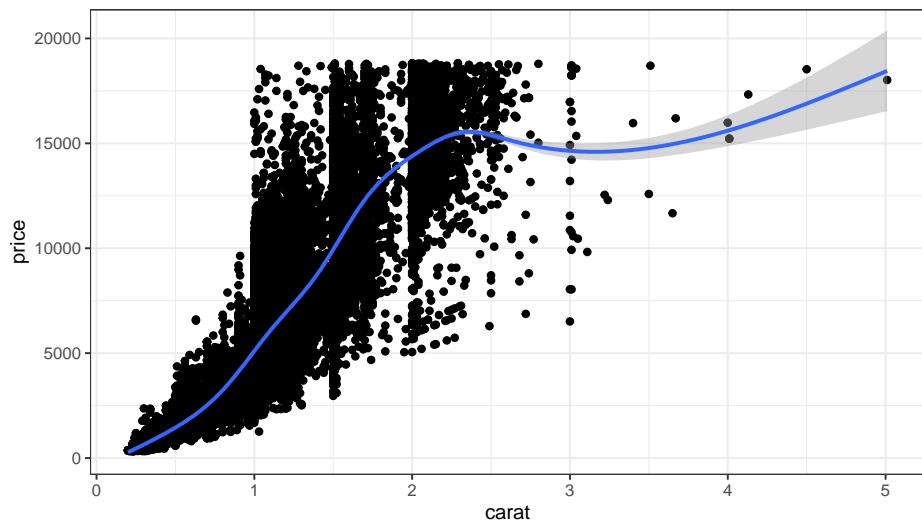


6.2.2 Слои

```
diamonds %>%
  ggplot(aes(carat, price)) +
```

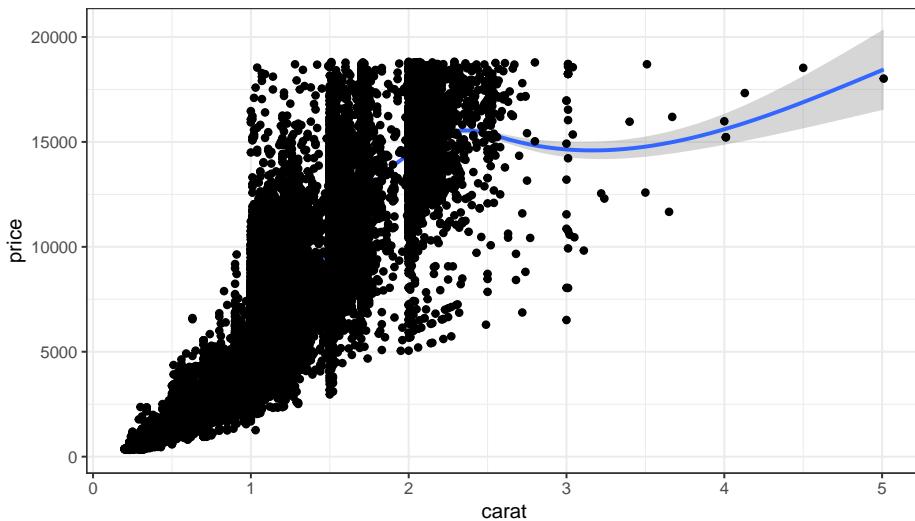
```
geom_point()+
geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



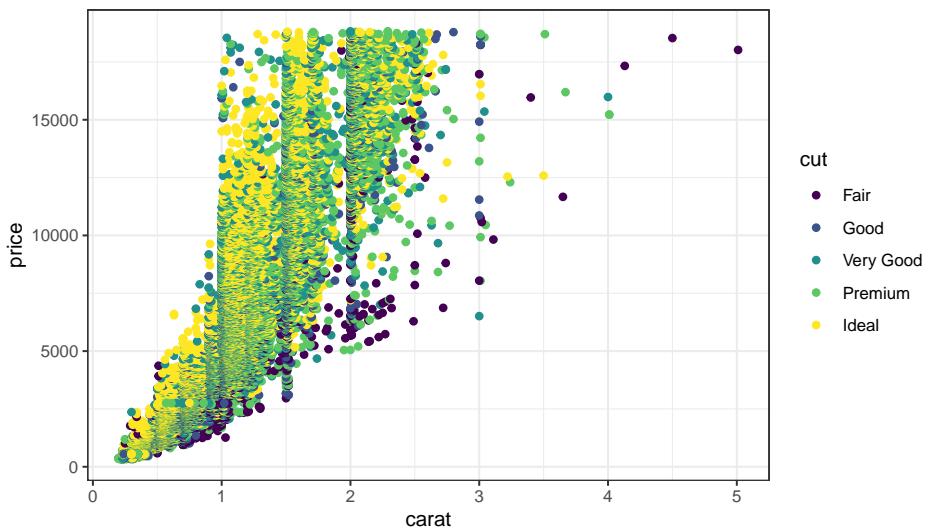
```
diamonds %>%
  ggplot(aes(carat, price))+
  geom_smooth()+
  geom_point()
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

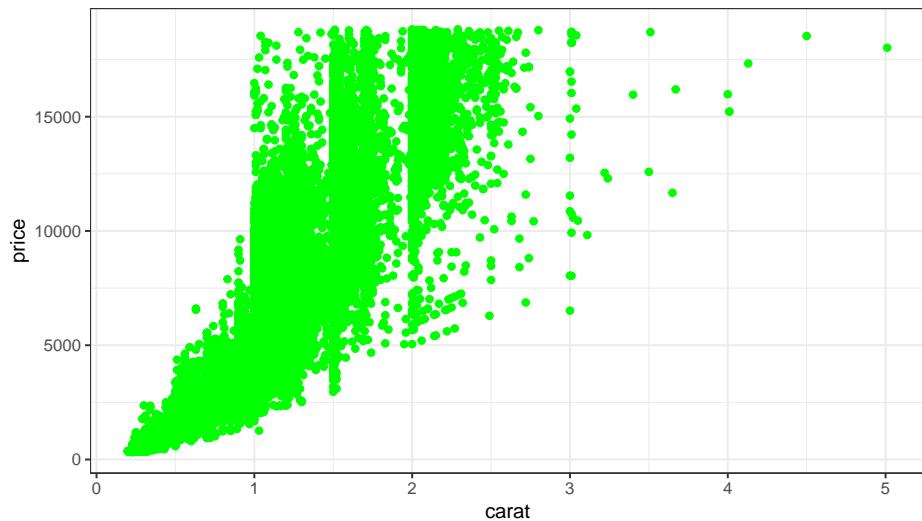


6.2.3 aes()

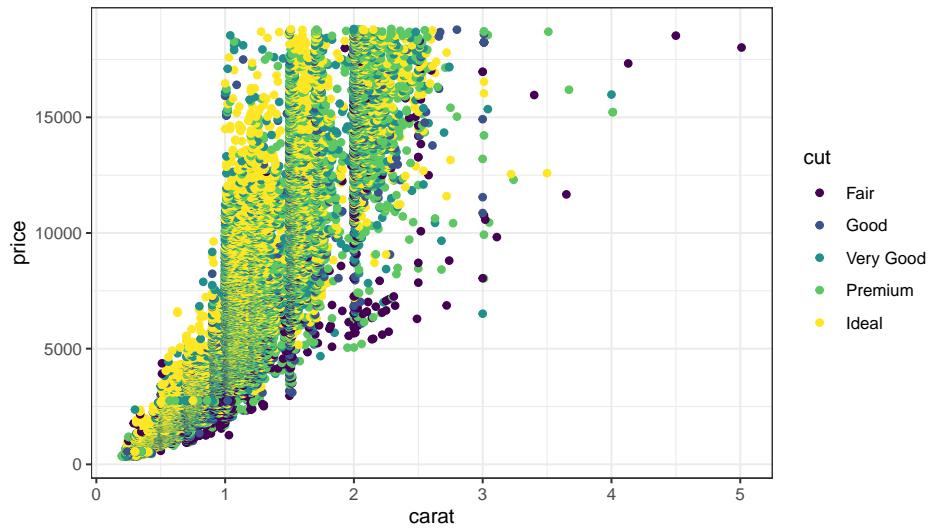
```
diamonds %>%
  ggplot(aes(carat, price, color = cut)) +
  geom_point()
```



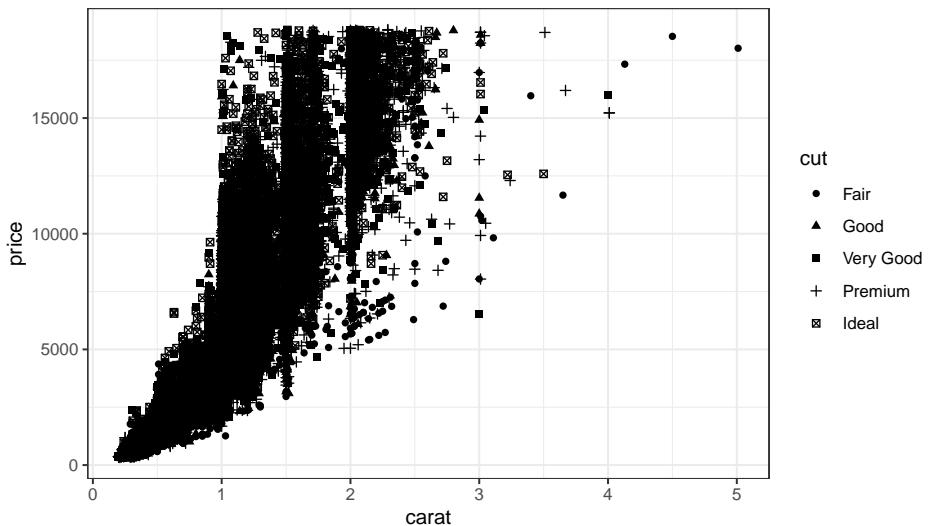
```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point(color = "green")
```



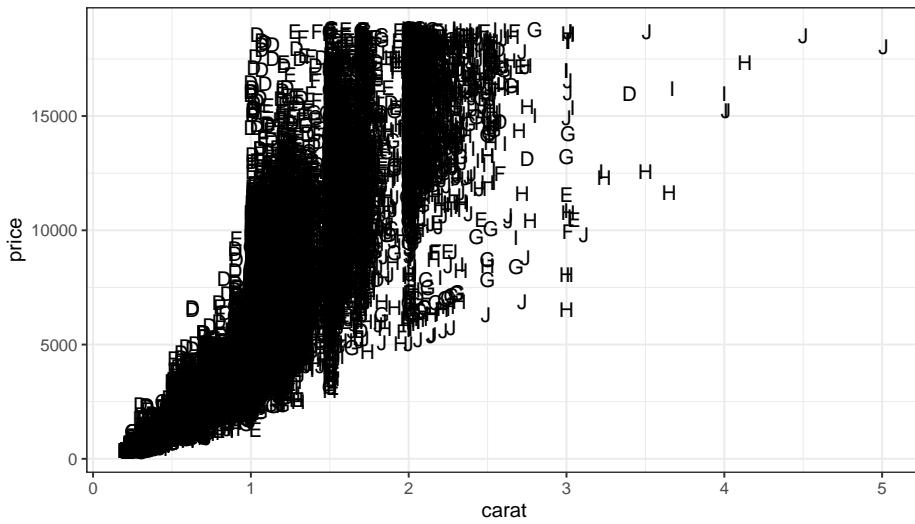
```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point(aes(color = cut))
```



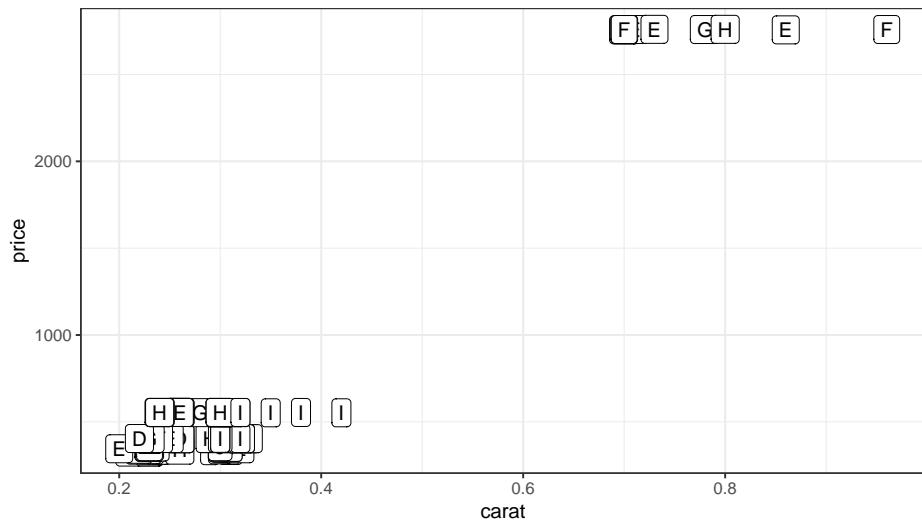
```
diamonds %>%
  ggplot(aes(carat, price, shape = cut)) +
  geom_point()
```



```
diamonds %>%
  ggplot(aes(carat, price, label = color)) +
  geom_text()
```

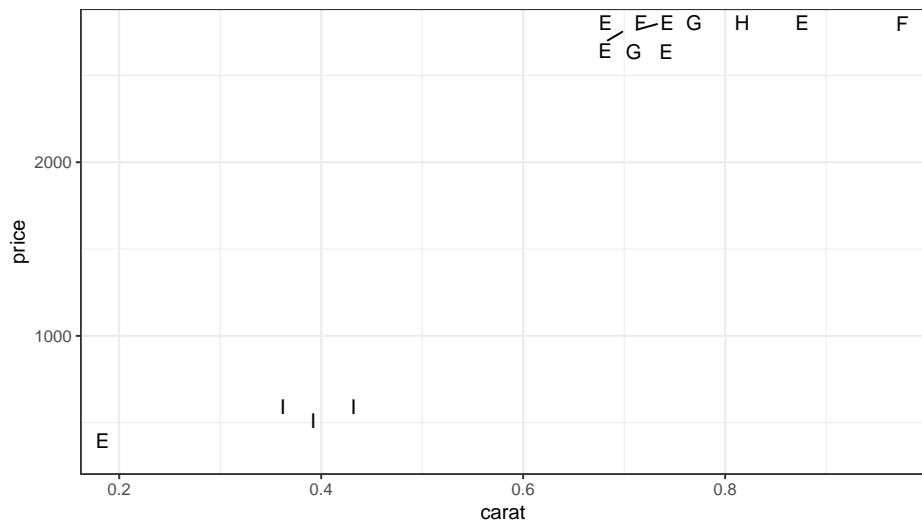


```
diamonds %>%
  slice(1:100) %>%
  ggplot(aes(carat, price, label = color)) +
  geom_label()
```



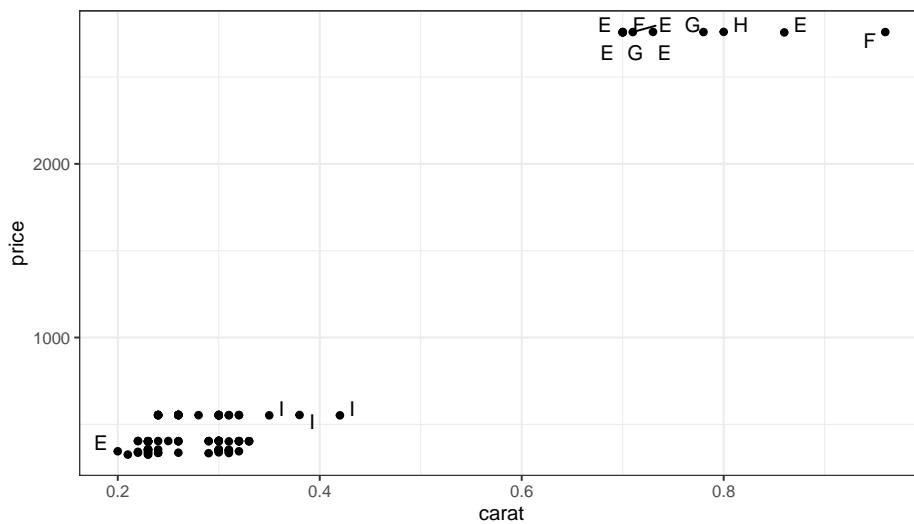
Иногда аннотации налазают друг на друга:

```
library(ggrepel)
diamonds %>%
  slice(1:100) %>%
  ggplot(aes(carat, price, label = color)) +
  geom_text_repel()
```

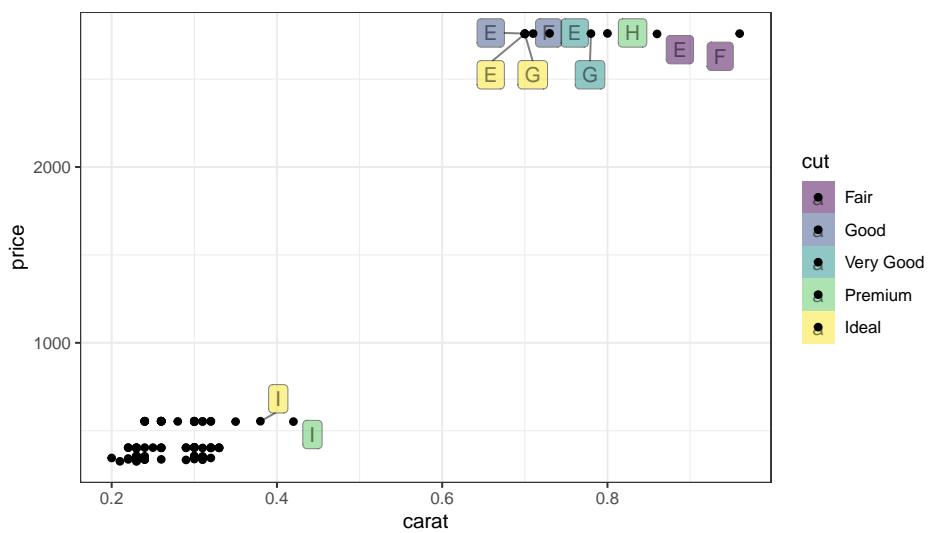


```
diamonds %>%
  slice(1:100) %>%
```

```
ggplot(aes(carat, price, label = color))+
  geom_text_repel()+
  geom_point()
```

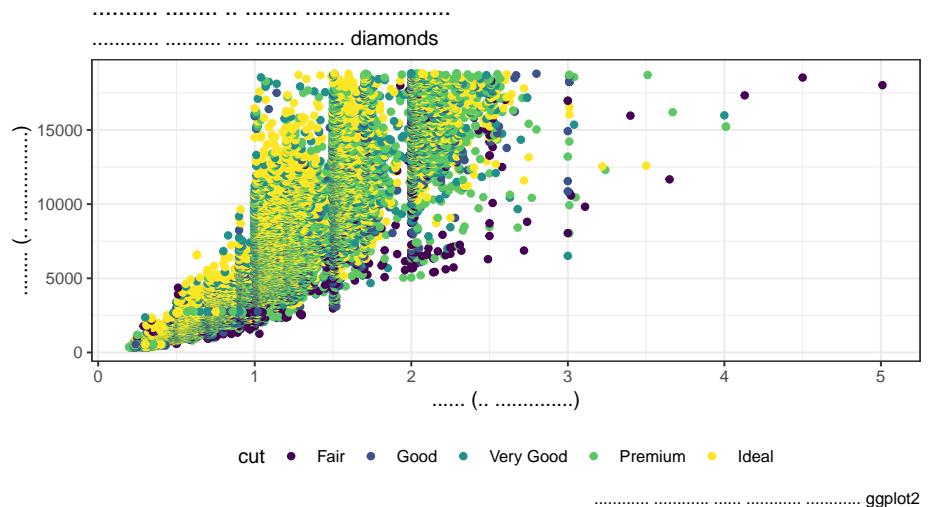


```
diamonds %>%
  slice(1:100) %>%
  ggplot(aes(carat, price, label = color, fill = cut))# fill
  geom_label_repel(alpha = 0.5)+ # alpha
  geom_point()
```

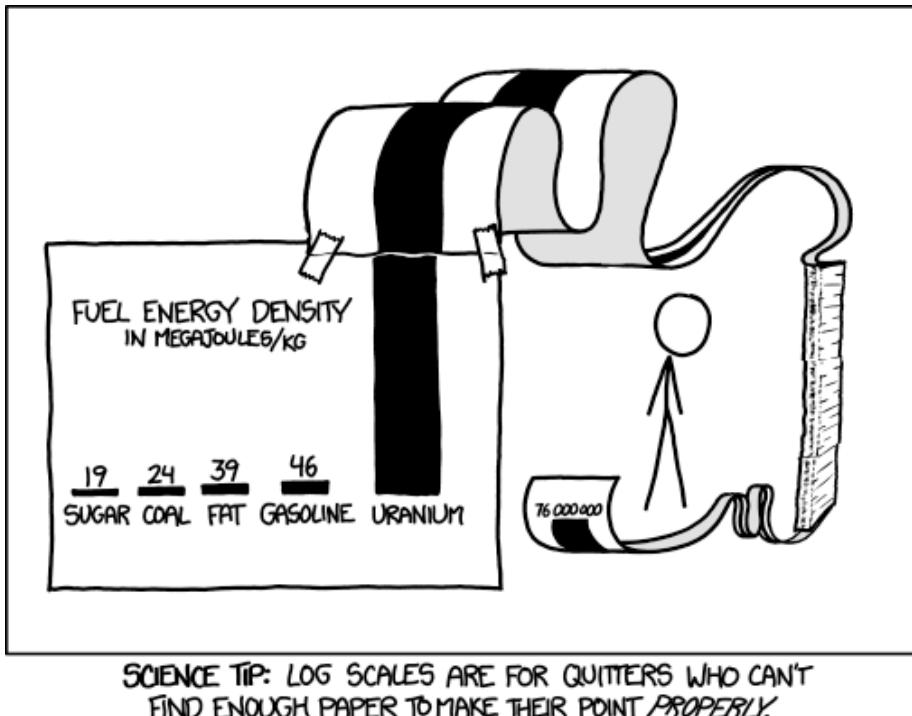


6.2.4 Оформление

```
diamonds %>%
  ggplot(aes(carat, price, color = cut)) +
  geom_point() +
  labs(x = " ( )",
       y = " ( )",
       title = "",
       subtitle = " diamonds",
       caption = " ggplot2") +
  theme(legend.position = "bottom") # theme()
```



6.2.5 Логарифмические шкалы



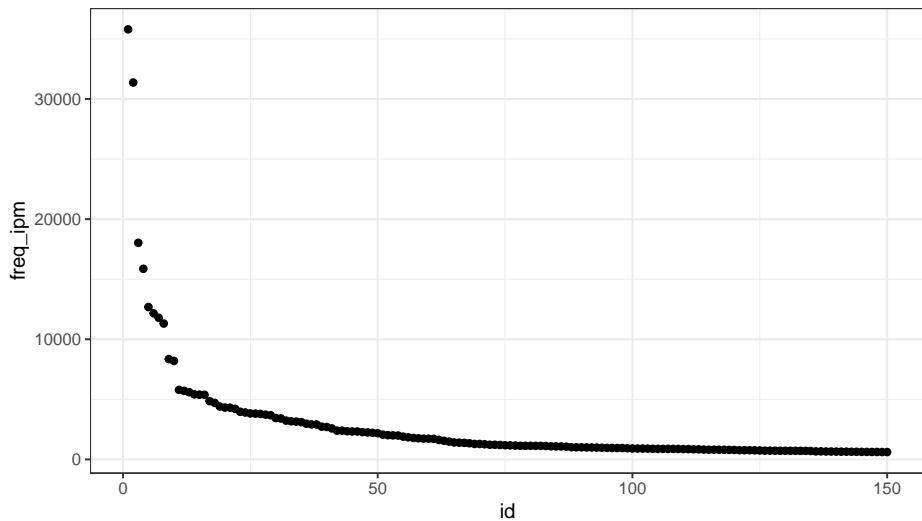
SCIENCE TIP: LOG SCALES ARE FOR QUITTERS WHO CAN'T FIND ENOUGH PAPER TO MAKE THEIR POINT PROPERLY.

Рассмотрим словарь [Ляшевской, Шарова 2011]

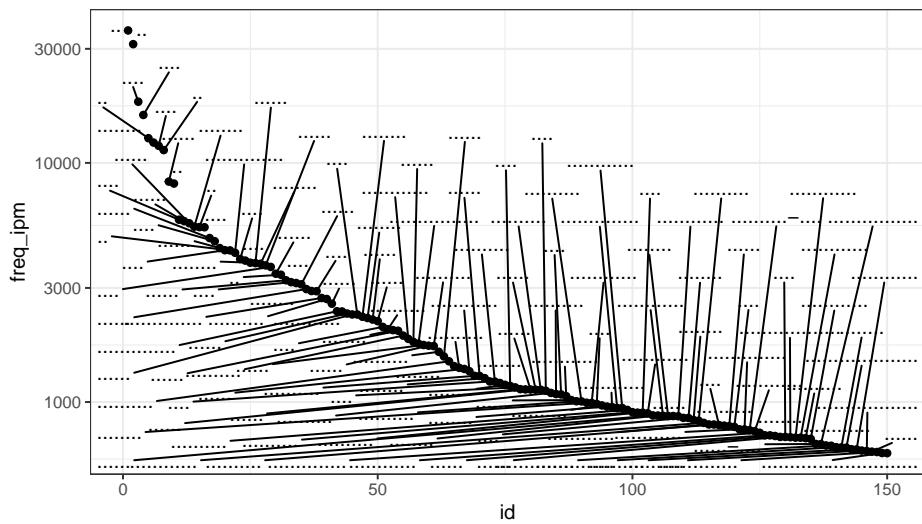
```
freqdict <- read_tsv("https://github.com/agricolamz/2020-2021-ds4dh/raw/master/data/freq_dict_2021-01-01.csv")
```

```
## 
## -- Column specification --
## cols(
##   lemma = col_character(),
##   pos = col_character(),
##   freq_ipm = col_double()
## )
```

```
freqdict %>%
  arrange(desc(freq_ipm)) %>%
  mutate(id = 1:n()) %>%
  slice(1:150) %>%
  ggplot(aes(id, freq_ipm)) +
  geom_point()
```



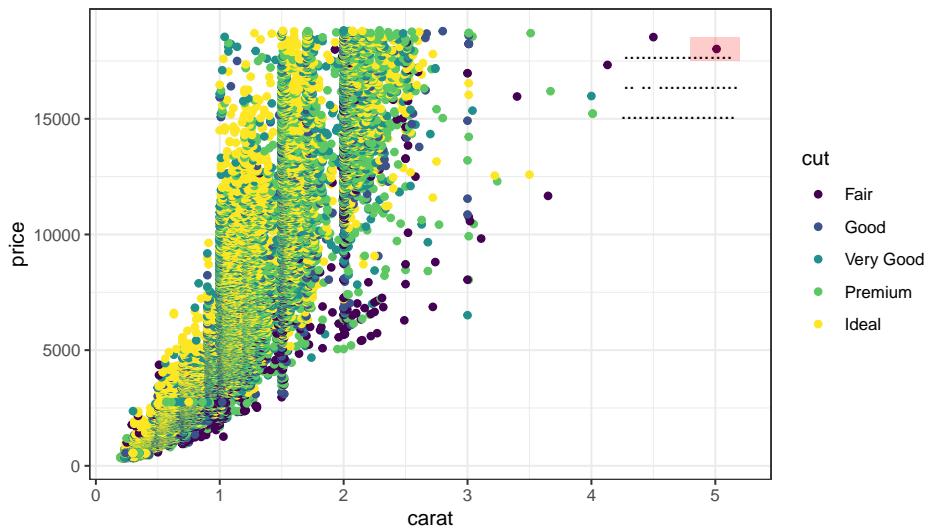
```
freqdict %>%
  arrange(desc(freq_ipm)) %>%
  mutate(id = 1:n()) %>%
  slice(1:150) %>%
  ggplot(aes(id, freq_ipm, label = lemma))+
  geom_point()+
  geom_text_repel()+
  scale_y_log10()
```



6.2.6 `annotate()`

Функция `annotate` добавляет `geom` к графику.

```
diamonds %>%
  ggplot(aes(carat, price, color = cut)) +
  geom_point() +
  annotate(geom = "rect", xmin = 4.8, xmax = 5.2,
          ymin = 17500, ymax = 18500, fill = "red", alpha = 0.2) +
  annotate(geom = "text", x = 4.7, y = 16600,
          label = "... \n ...")
```



Скачайте вот этот датасет⁴ и постройте диаграмму рассеяния.

6.3 Столбчатые диаграммы (barplots)

Одна и та же информация может быть представлена в агрегированном и не агрегированном варианте:

```
misspelling <- read_csv("https://raw.githubusercontent.com/agricolamz/2020-2021-ds4dh/master/data/misspellings.csv")
```

```
## 
## -- Column specification --
## cols(
##   correct = col_character(),
##   spelling = col_character(),
```

```
##   count = col_double()
## )
```

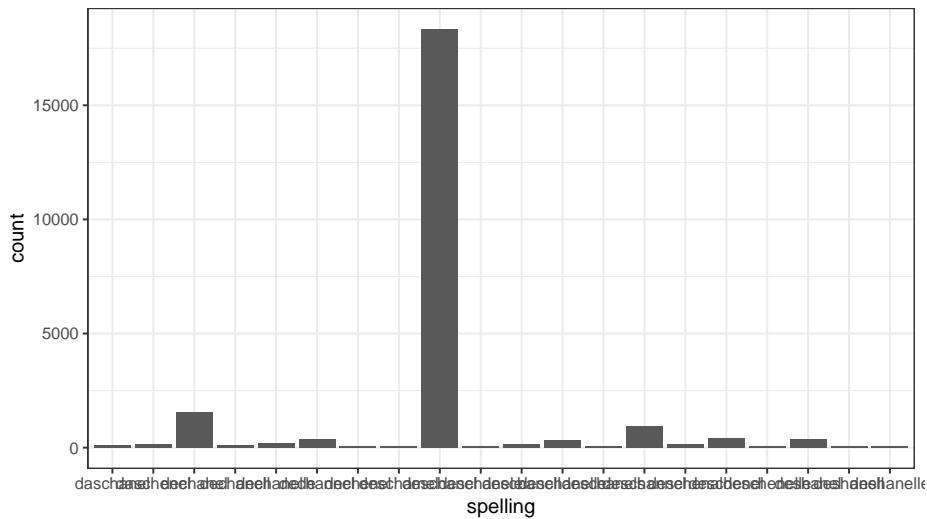
misspelling

```
## # 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
```

- переменные `spelling` **аггрегированы**: для каждого значения представлено значение в столбце `count`, которое обозначает количество каждого из написаний
- переменные `correct` **неаггрегированы**: в этом столбце она повторяется, для того, чтобы сделать вывод, нужно отдельно посчитать количество вариантов

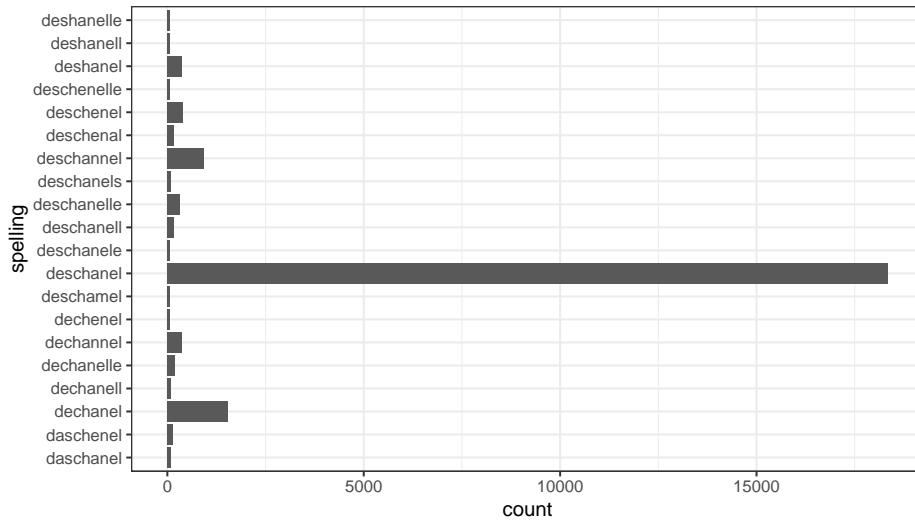
Для агрегированных данных используется `geom_col()`

```
misspelling %>%
  slice(1:20) %>%
  ggplot(aes(spelling, count)) +
  geom_col()
```



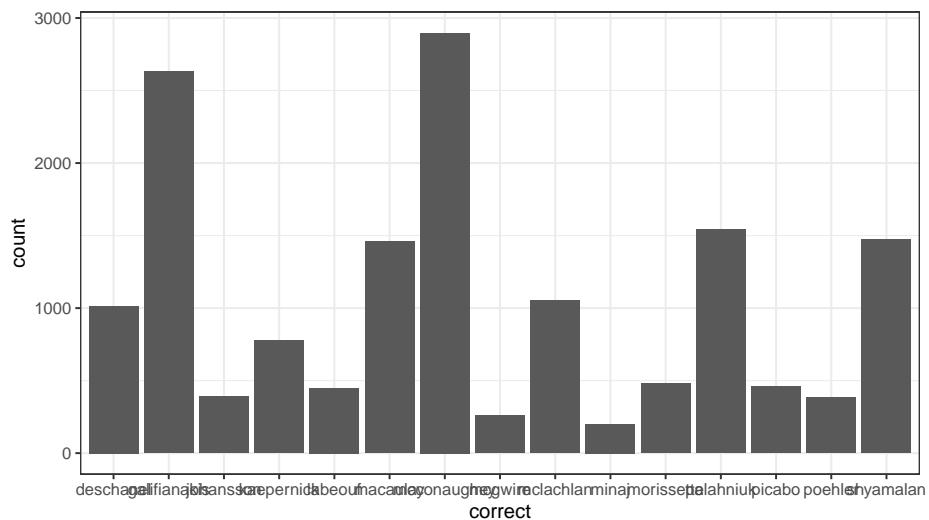
Перевернем оси:

```
misspelling %>%
  slice(1:20) %>%
  ggplot(aes(spelling, count)) +
  geom_col() +
  coord_flip()
```



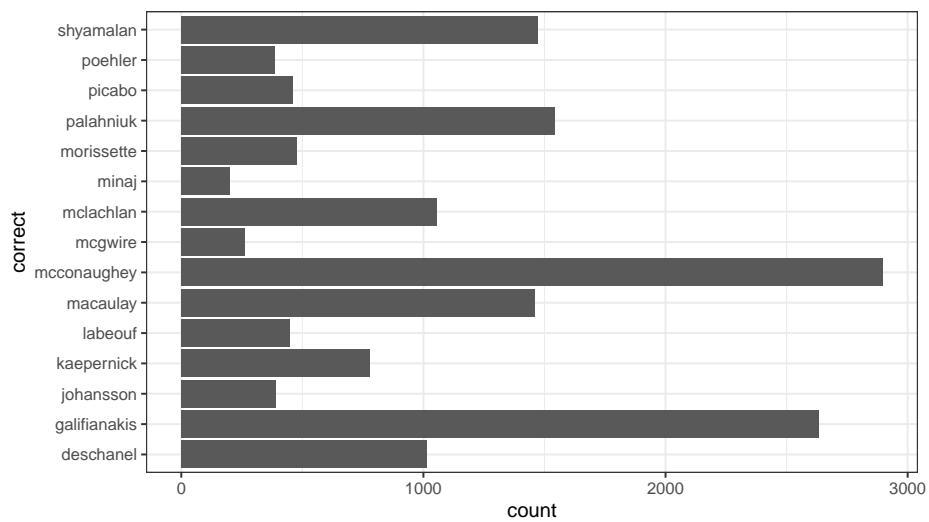
Для неаггрегированных данных используется `geom_bar()`

```
misspelling %>%  
  ggplot(aes(correct)) +  
  geom_bar()
```



Перевернем оси:

```
misspelling %>%  
  ggplot(aes(correct)) +  
  geom_bar() +  
  coord_flip()
```



Неаггрегированный вариант можно перевести в агрегированный:

```
diamonds %>%
  count(cut)
```

```
## # A tibble: 5 x 2
##   cut      n
##   <ord>    <int>
## 1 Fair     1610
## 2 Good     4906
## 3 Very Good 12082
## 4 Premium   13791
## 5 Ideal    21551
```

Агрегированный вариант можно перевести в неаггрегированный:

```
diamonds %>%
  count(cut) %>%
  uncount(n)
```

```
## # A tibble: 53,940 x 1
##   cut
##   <ord>
## 1 Fair
## 2 Fair
## 3 Fair
## 4 Fair
## 5 Fair
## 6 Fair
## 7 Fair
## 8 Fair
## 9 Fair
## 10 Fair
## # ... with 53,930 more rows
```

6.4 Факторы

Как можно заметить по предыдущему разделу, переменные на графике упорядочены по алфавиту. Чтобы это исправить нужно обсудить факторы:

```
my_factor <- factor(misspellings$correct)
head(my_factor)
```

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

```
## 15 Levels: deschanel galifianakis johansson kaepernick labeouf ... shyamalan
```

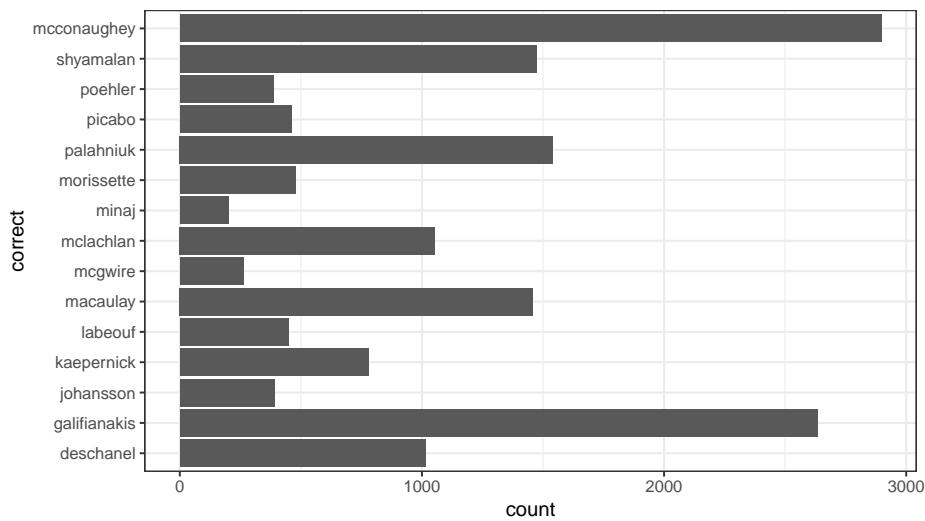
```
levels(my_factor)
```

```
## [1] "deschanel"     "galifianakis"   "johansson"      "kaepernick"    "labeouf"
## [6] "macaulay"      "mcconaughey"    "mcgwire"        "mclachlan"     "minaj"
## [11] "morissette"    "palahniuk"      "picabo"         "poehler"       "shyamalan"
```

```
levels(my_factor) <- rev(levels(my_factor))
head(my_factor)
```

```
## [1] shyamalan shyamalan shyamalan shyamalan shyamalan shyamalan
## 15 Levels: shyamalan poehler picabo palahniuk morissette minaj ... deschanel
```

```
misspelling %>%
  mutate(correct = factor(correct, levels = c("deschanel",
                                                "galifianakis",
                                                "johansson",
                                                "kaepernick",
                                                "labeouf",
                                                "macaulay",
                                                "mcgwire",
                                                "mclachlan",
                                                "minaj",
                                                "morissette",
                                                "palahniuk",
                                                "picabo",
                                                "poehler",
                                                "shyamalan",
                                                "mcconaughey")))) %>%
  ggplot(aes(correct)) +
  geom_bar() +
  coord_flip()
```



Для работы с факторами удобно использовать пакет `forcats` (входит в `tidyverse`, вот ссылка на cheatsheet⁵).

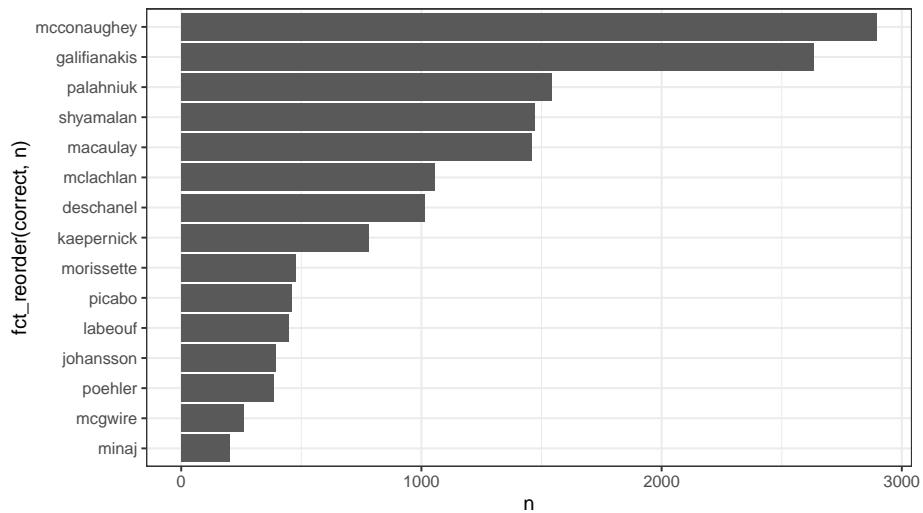
Иногда полезной бывает функция `fct_reorder()`:

```
misspelling %>%
  count(correct)
```

```
## # A tibble: 15 x 2
##   correct      n
##   <chr>     <int>
## 1 deschanel    1015
## 2 galifianakis  2633
## 3 johansson     392
## 4 kaepernick     779
## 5 labeouf       449
## 6 macaulay      1458
## 7 mcconaughey    2897
## 8 mcgwire        262
## 9 mclachlan     1054
## 10 minaj         200
## 11 morissette    478
## 12 palahniuk     1541
## 13 picabo        460
## 14 poehler        386
## 15 shyamalan     1473
```

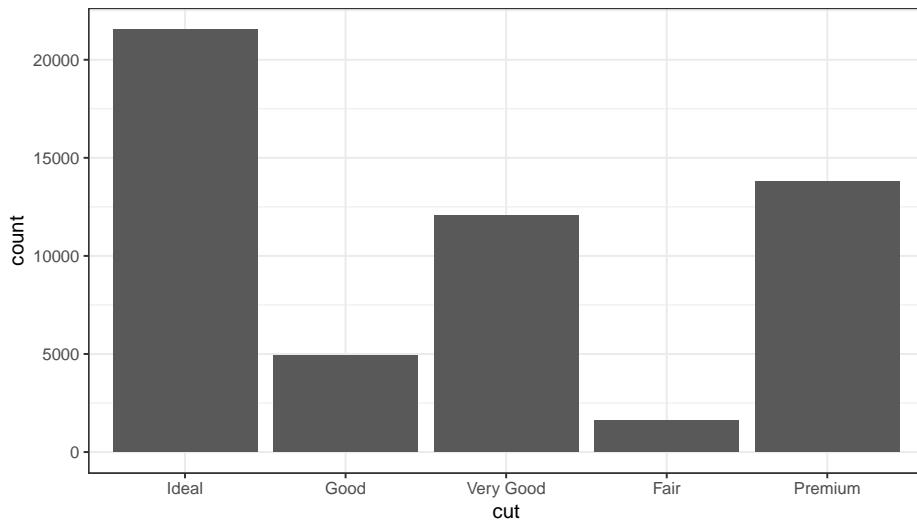
⁵<https://github.com/rstudio/cheatsheets/raw/master/factors.pdf>

```
misspelling %>%
  count(correct) %>%
  ggplot(aes(fct_reorder(correct, n), n)) +
  geom_col() +
  coord_flip()
```



Кроме того, в функцию `fct_reorder()` можно добавить функцию, которая будет считаться на векторе, по которому группируют:

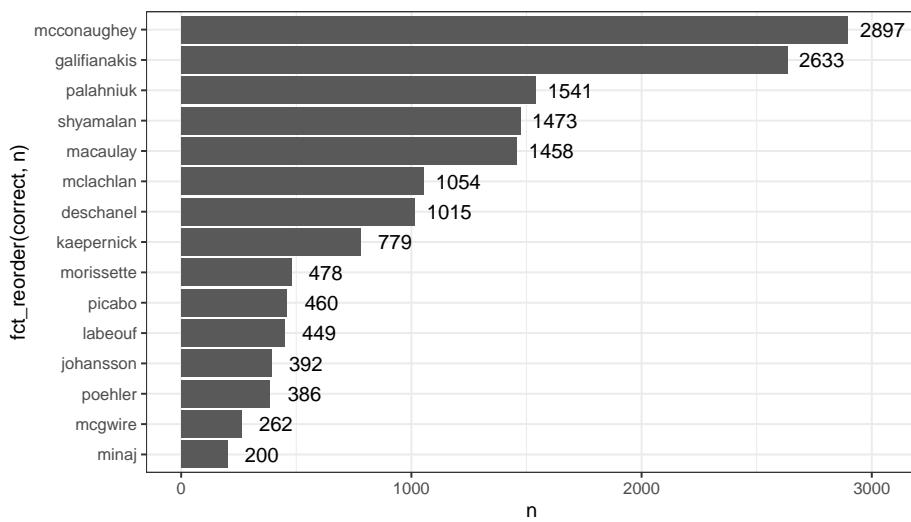
```
diamonds %>%
  mutate(cut = fct_reorder(cut, price, mean)) %>%
  ggplot(aes(cut)) +
  geom_bar()
```



В этом примере переменная `cut` упорядочена по средней `mean` цене `price`. Естественно, вместо `mean` можно использовать другие функции (`median`, `min`, `max` или даже собственные функции).

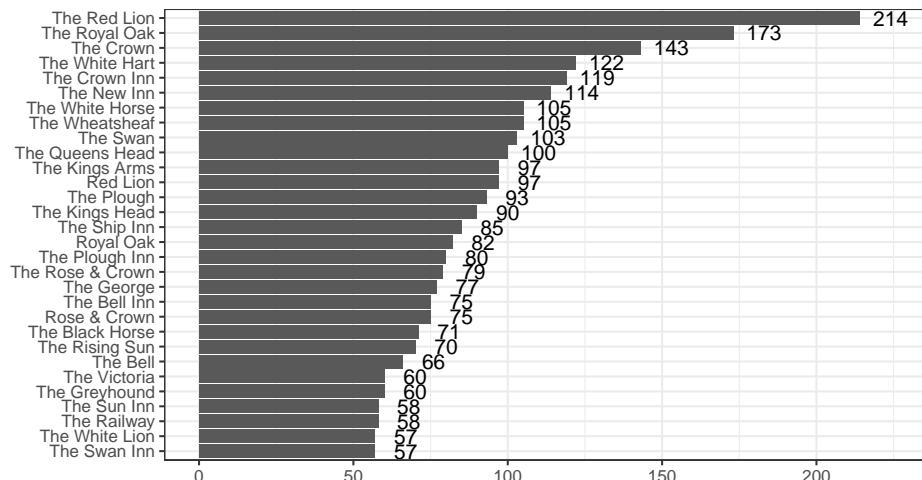
Можно совмещать разные `geom_...`:

```
misspelling %>%
  count(correct) %>%
  ggplot(aes(fct_reorder(correct, n), n, label = n)) +
  geom_col() +
  geom_text(nudge_y = 150) +
  coord_flip()
```





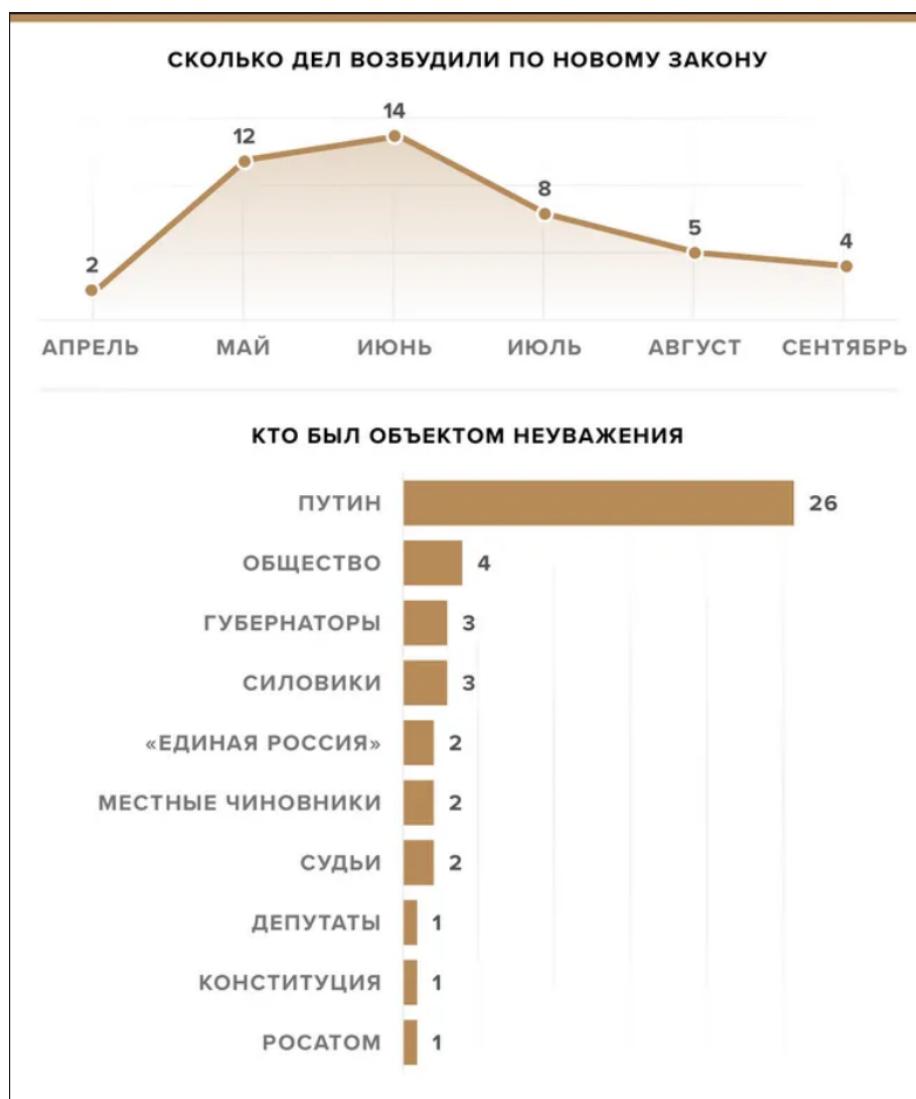
На Pudding вышла статья про английские пабы⁶. Здесь⁷ лежит немного обработанный датасет, которые они использовали. Визуализируйте 30 самых частотных названий пабов в Великобритании.

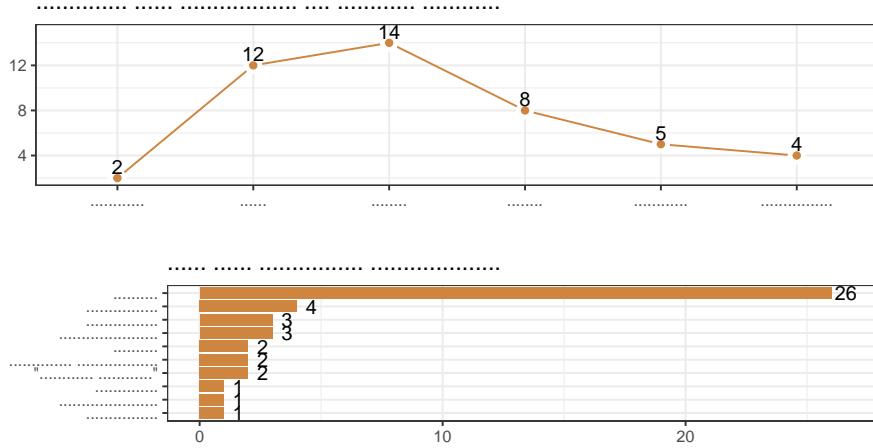


data from <https://pudding.cool/2019/10/pubs/>



На новостном портале meduza.io недавно вышла новость о применения закона “о неуважении к власти в интернете”⁸. Постройте графики из этой новости. При построении графиков я использовал цвет “tan3”.





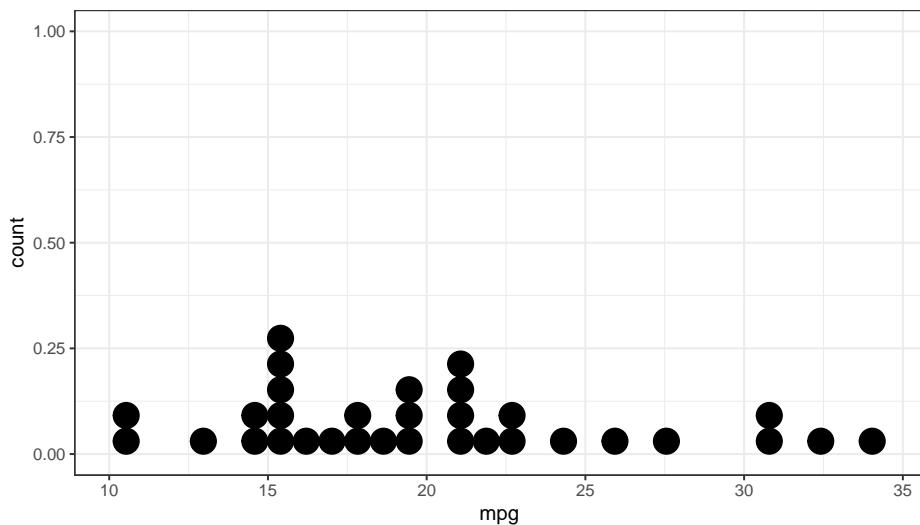
meduza.io

6.5 Дотплот

Иногда для случаев, когда мы исследуем числовую переменную подходит простой график, который отображает распределение наших наблюдений на одной соответствующей числовой шкале.

```
mtcars %>%
  ggplot(aes(mpg)) +
  geom_dotplot(method = "histodot")
```

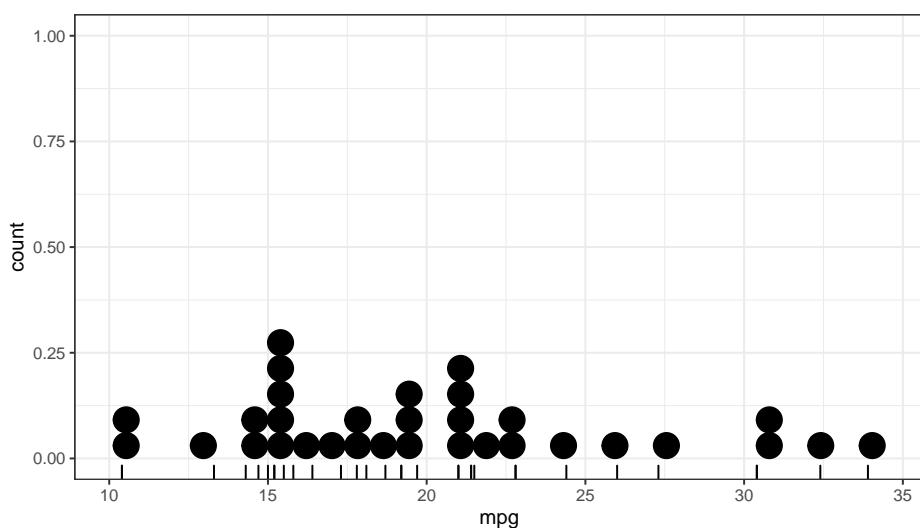
```
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```



По оси х отложена наша переменная, каждая точка – одно наблюдение, а отложенное по оси у стоит игнорировать – оно появляется из-за ограничений пакета `ggplot2`. Возможно чуть понятнее будет, если добавить `geom_rug()`, который непосредственно отображает каждое наблюдение.

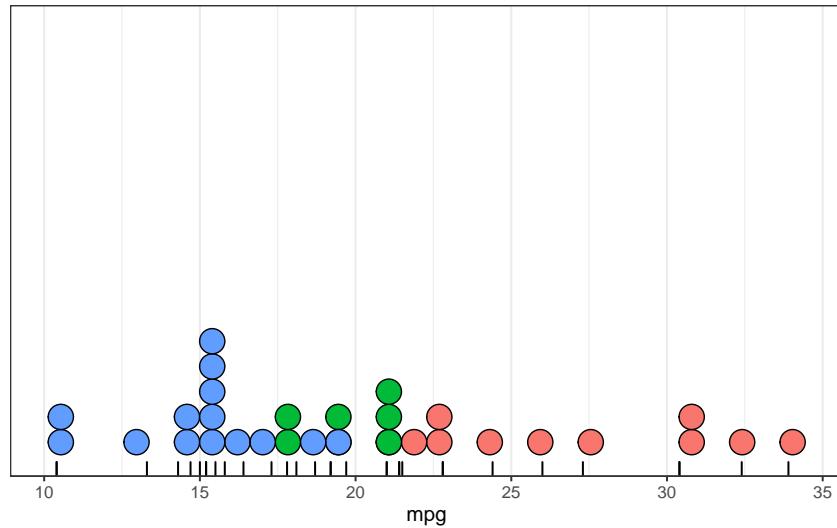
```
mtcars %>%
  ggplot(aes(mpg)) +
  geom_rug() +
  geom_dotplot(method = "histodot")
```

`stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.



Больший смысл имеет раскрашенный вариант:

```
mtcars %>%
  mutate(cyl = factor(cyl)) %>%
  ggplot(aes(mpg, fill = cyl)) +
  geom_rug() +
  geom_dotplot(method = "histodot") +
  scale_y_continuous(NULL, breaks = NULL) # y
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```

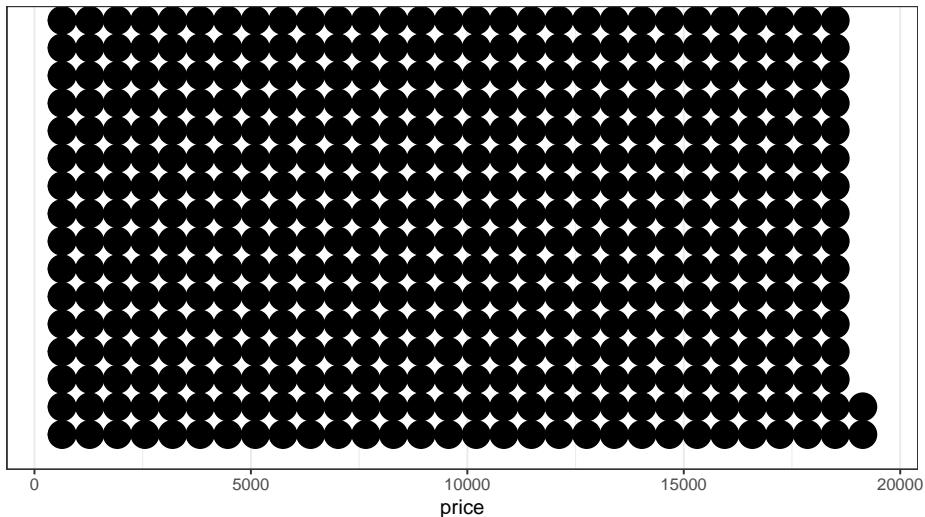


Как видно, на графике, одна синяя точка попала под одну зеленую: значит они имеют общее наблюдение.

6.6 Гистограммы

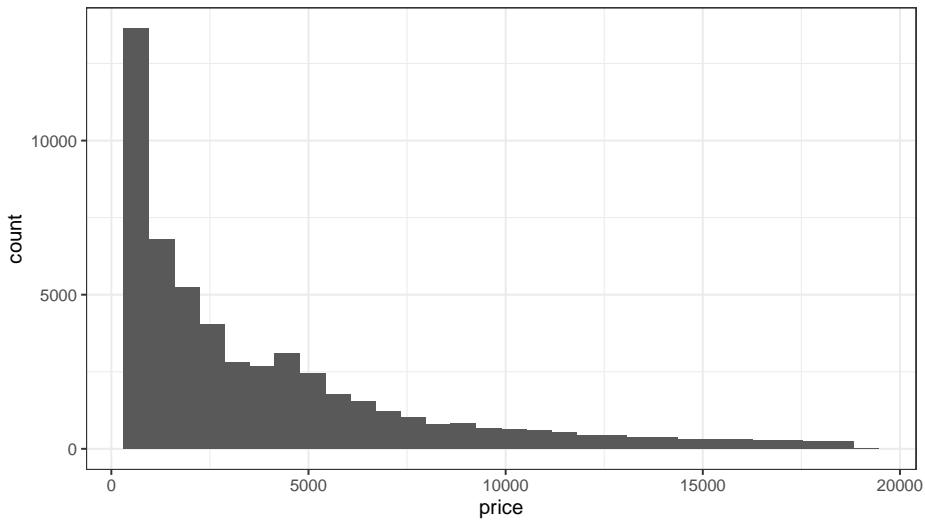
Если наблюдений слишком много, дотплот не имеет много смысла:

```
diamonds %>%
  ggplot(aes(price)) +
  geom_dotplot(method = "histodot") +
  scale_y_continuous(NULL, breaks = NULL) # y
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```



```
diamonds %>%
  ggplot(aes(price)) +
  geom_histogram()
```

`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

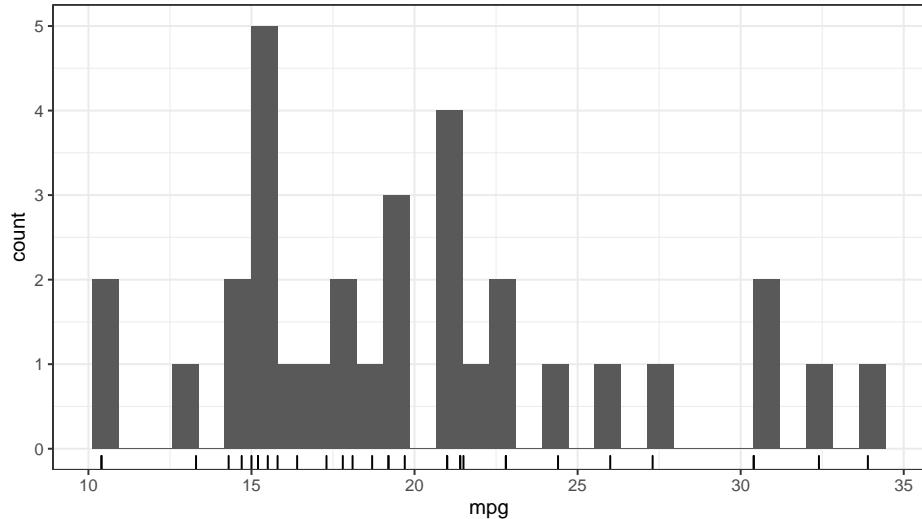


Обсудим на предыдущем примере

```
mtcars %>%
  ggplot(aes(mpg)) +
```

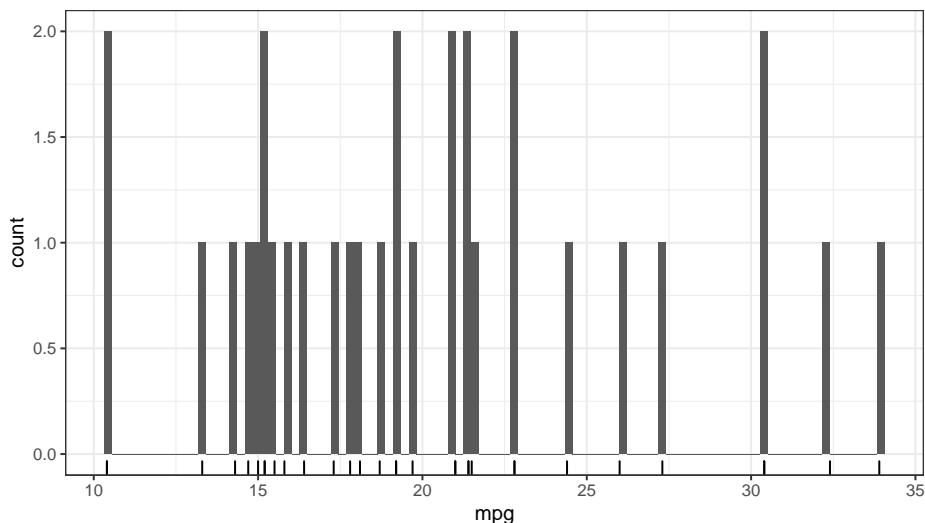
```
geom_rug()+
geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

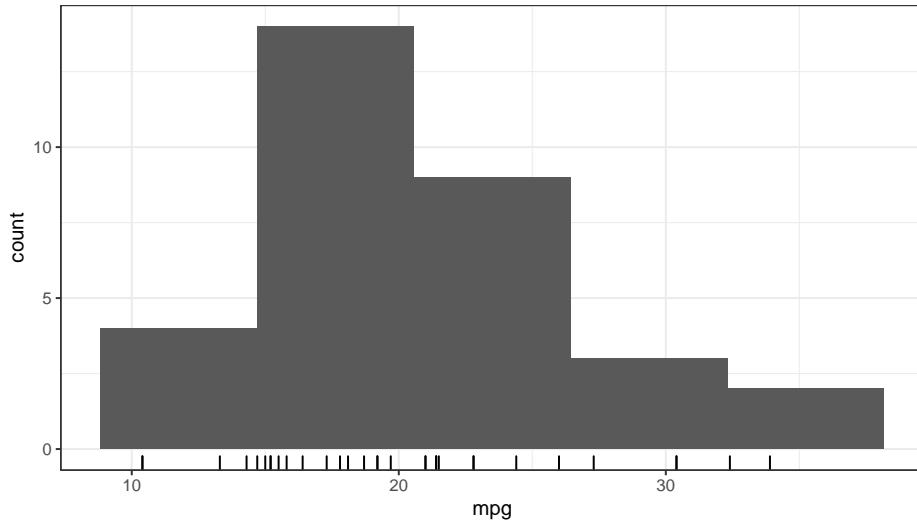


По оси x отложена наша переменная, а высота столбца говорит, сколько наблюдений имеют такое же наблюдение. Однако многое зависит от того, что мы считаем одинаковым значением:

```
mtcars %>%
  ggplot(aes(mpg)) +
  geom_rug()+
  geom_histogram(bins = 100)
```



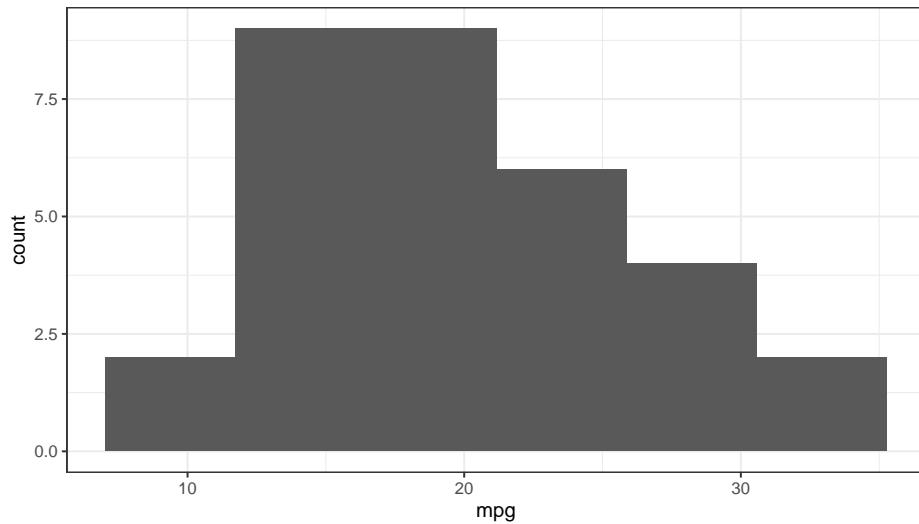
```
mtcars %>%
  ggplot(aes(mpg)) +
  geom_rug() +
  geom_histogram(bins = 5)
```



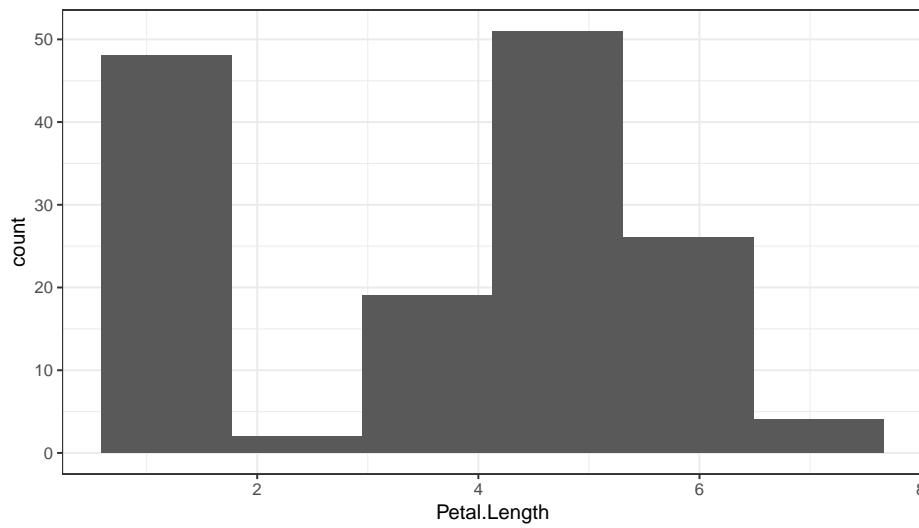
Существует три алгоритма встроенные в R, которые можно использовать и снимать с себя ответственность:

- [Sturges 1926] `nclass.Sturges(mtcars$mpg)`
- [Scott 1979] `nclass.scott(mtcars$mpg)`
- [Freedman, Diaconis 1981] `nclass.FD(mtcars$mpg)`

```
mtcars %>%
  ggplot(aes(mpg)) +
  geom_histogram(bins = nclass.FD(mtcars$mpg))
```



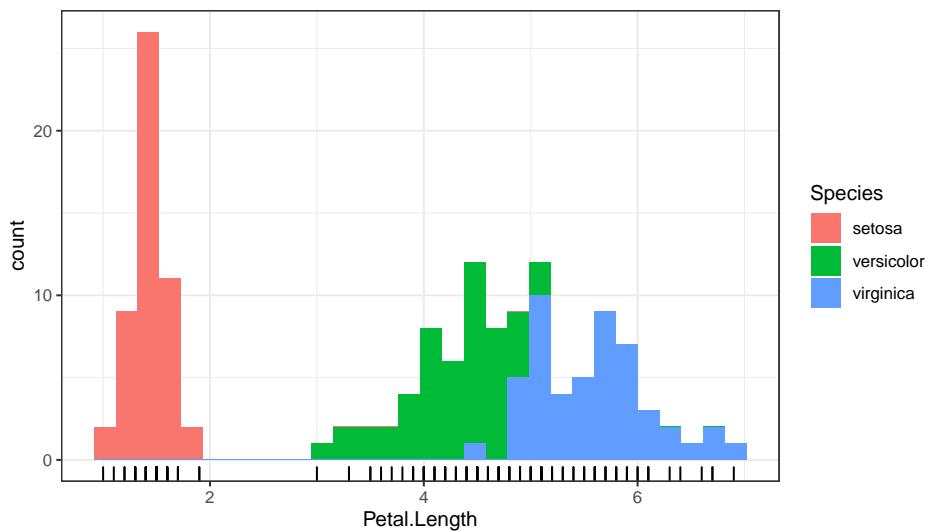
Какой из методов использовался при создании следующего графика на основе встроенного датасета `iris`?



В этом типе графика точно так же можно раскрашивать на основании другой переменной:

```
iris %>%
  ggplot(aes(Petal.Length, fill = Species)) +
  geom_rug() +
  geom_histogram()
```

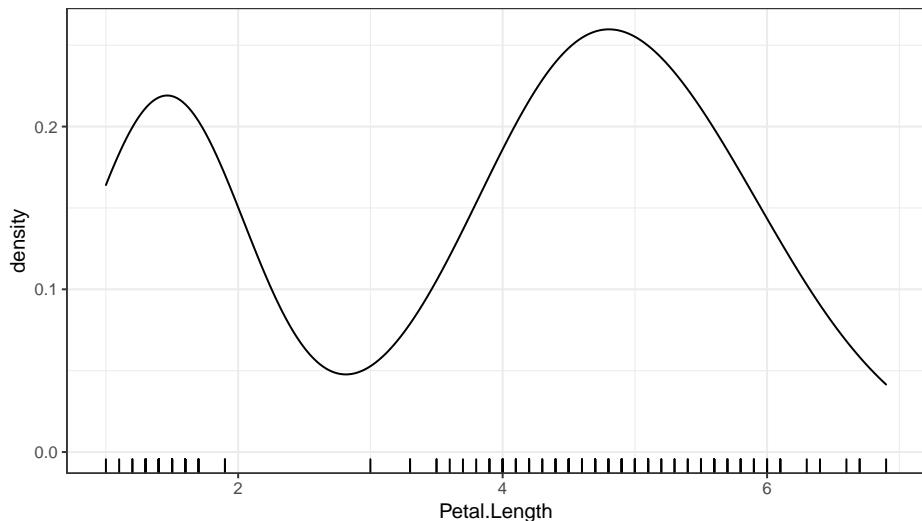
`## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.`



6.7 Функции плотности

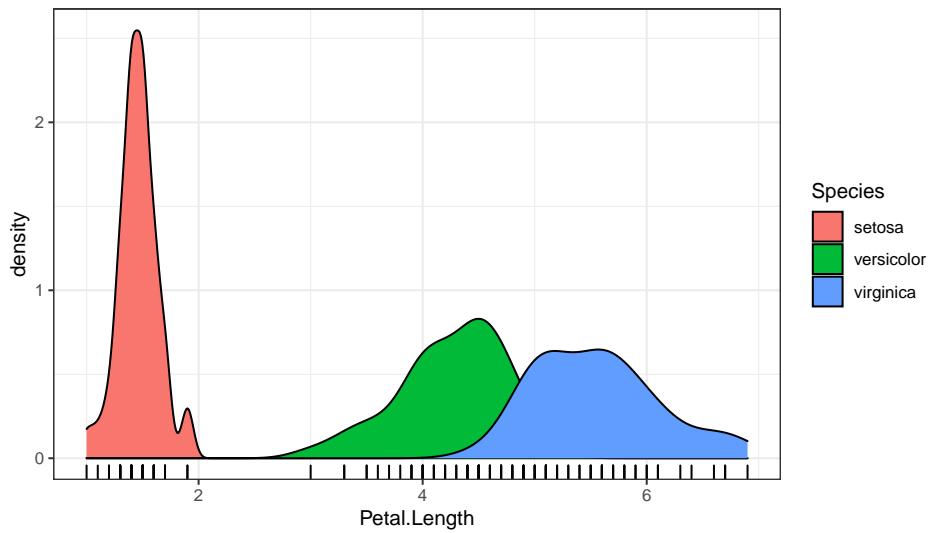
Кроме того, существует способ использовать не такой рубленный график, а его сглаженную вариант, который строиться при помощи функции плотядерной оценки ности. Важное свойство, которое стоит понимать про функцию плотности — что кривая, получаемая ядерной оценкой плотности, не зависит от величины коробки гистделения (хотя есть аргумент, который от `adjust`вечает за степень “близости” функции плотности к гистограмме).

```
iris %>%
  ggplot(aes(Petal.Length)) +
  geom_rug() +
  geom_density()
```



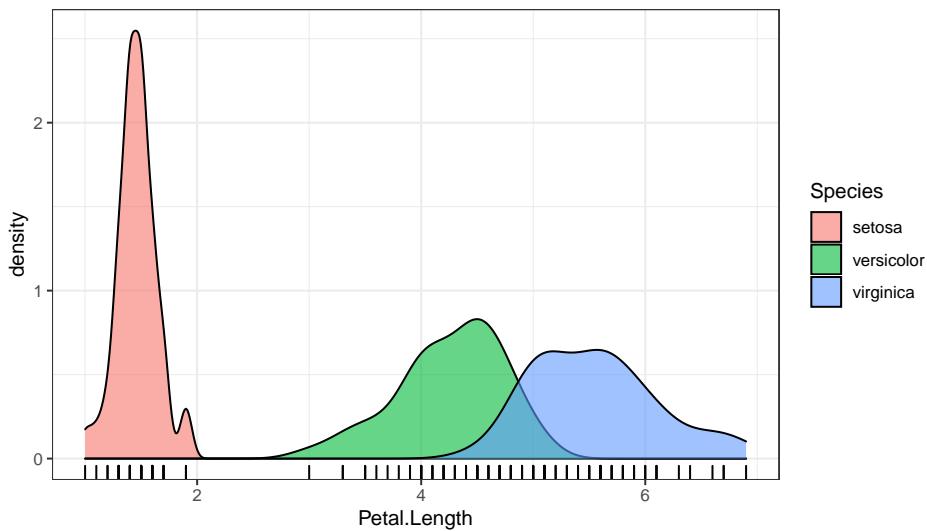
Таким образом мы можем сравнивать распределения:

```
iris %>%
  ggplot(aes(Petal.Length, fill = Species)) +
  geom_rug()+
  geom_density()
```



Часто имеет смысл настроить прозрачность:

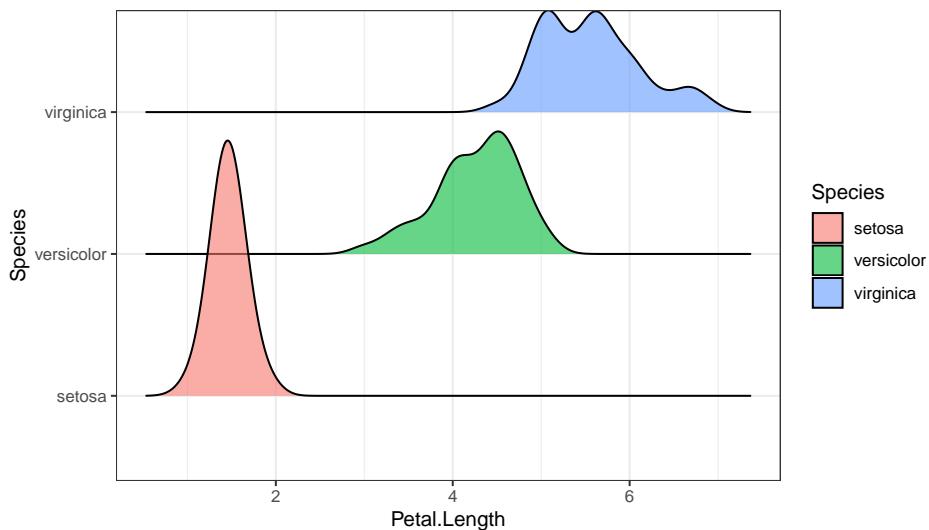
```
iris %>%
  ggplot(aes(Petal.Length, fill = Species)) +
  geom_rug()+
  geom_density(alpha = 0.6) #
```



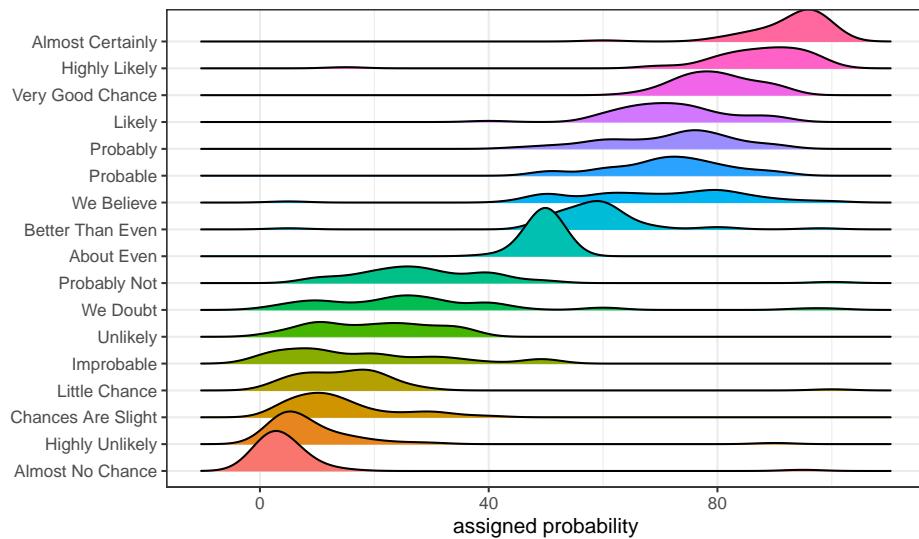
Кроме того, иногда удобно разделять группы на разные уровни:

```
# install.packages(ggridges)
library(ggridges)
iris %>%
  ggplot(aes(Petal.Length, Species, fill = Species)) +
  geom_density_ridges(alpha = 0.6) #
```

```
## Picking joint bandwidth of 0.155
```



В длинный список “2015 Kantar Information is Beautiful Awards” попала визуализация Perceptions of Probability⁹, сделанная пользователем zonination¹⁰ в ggplot2. Попробуйте воспроизвести ее с этими данными¹¹.

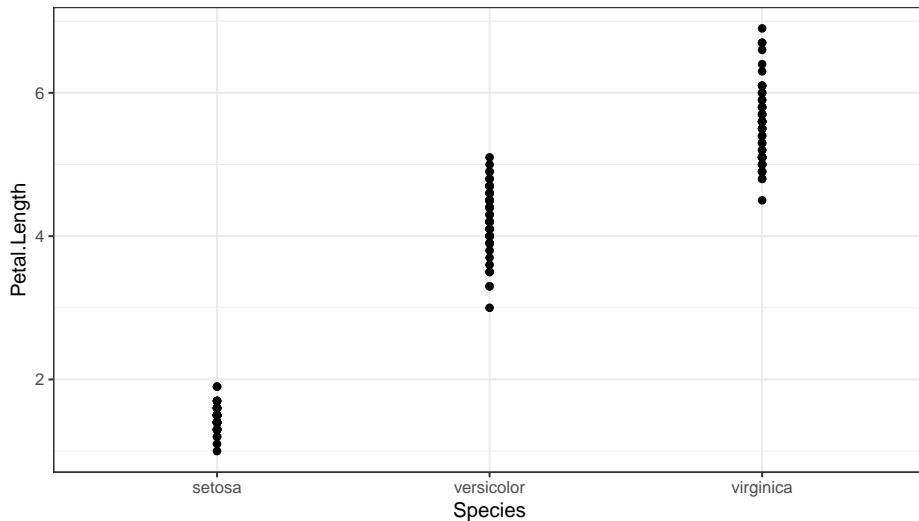


6.8 Точки, джиттер (jitter), вайолинплот (violinplot), ящики с усами (boxplot),

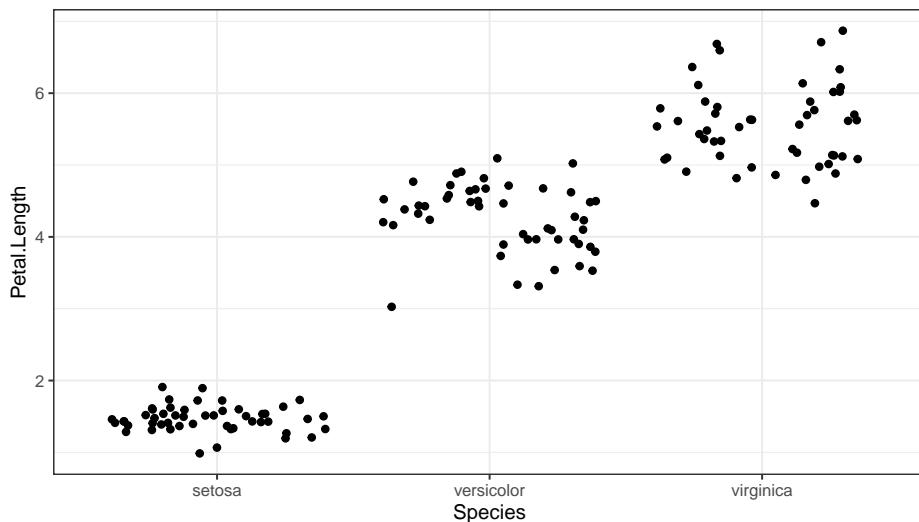
Вот другие способы показать распределение числовой переменной:

6.8. ТОЧКИ, ДЖИТТЕР (JITTER), ВАЙОЛИНПЛОТ (VIOLINPLOT), ЯЩИКИ С УСАМИ (BOXPLOT), 159

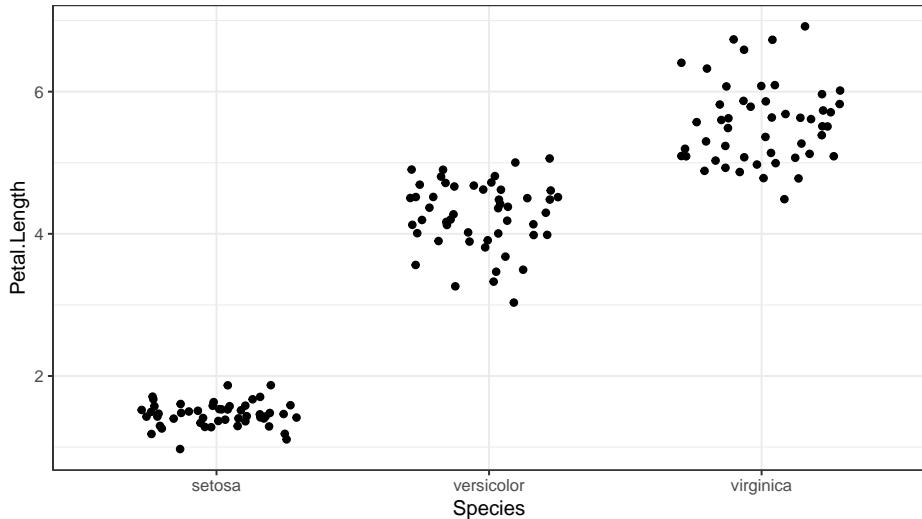
```
iris %>%
  ggplot(aes(Species, Petal.Length)) +
  geom_point()
```



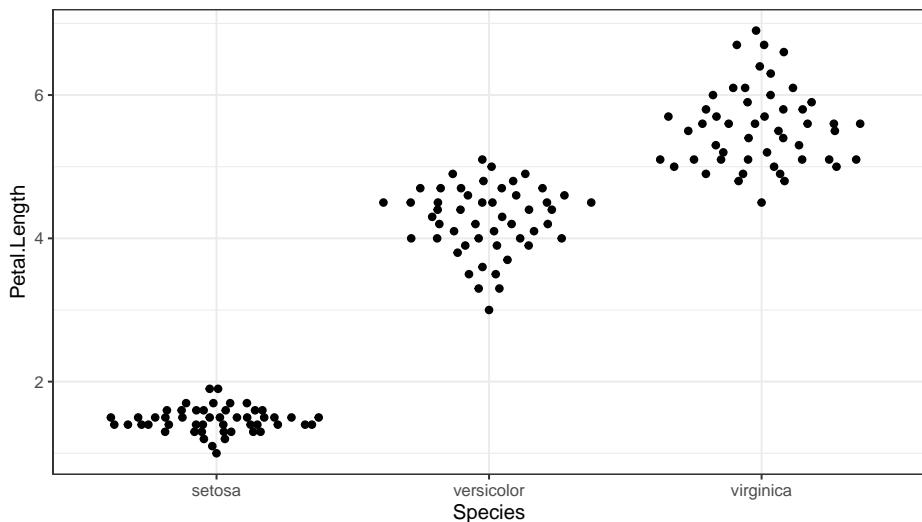
```
iris %>%
  ggplot(aes(Species, Petal.Length)) +
  geom_jitter()
```



```
iris %>%
  ggplot(aes(Species, Petal.Length)) +
  geom_jitter(width = 0.3)
```

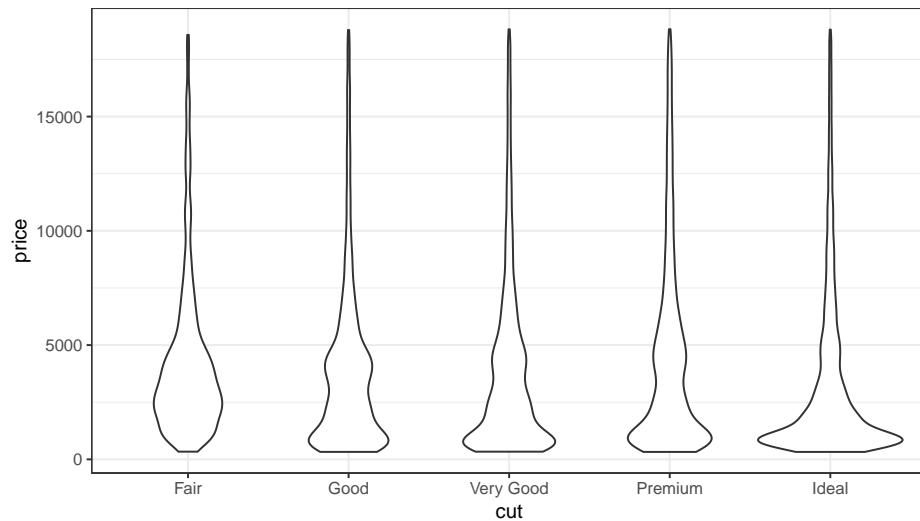


```
library("ggbeeswarm")
iris %>%
  ggplot(aes(Species, Petal.Length)) +
  geom_quasirandom()
```

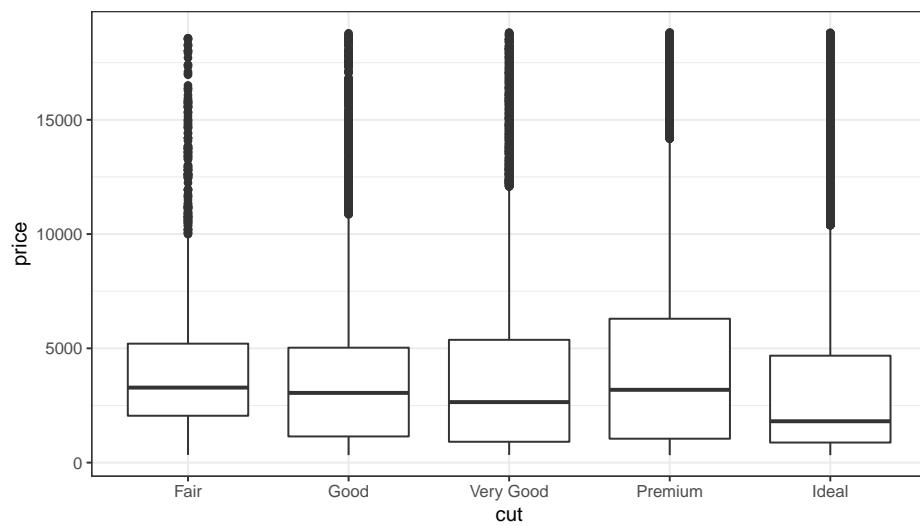


6.8. ТОЧКИ, ДЖИТТЕР (JITTER), ВАЙОЛИНПЛОТ (VIOLINPLOT), ЯЩИКИ С УСАМИ (BOXPLOT), 161

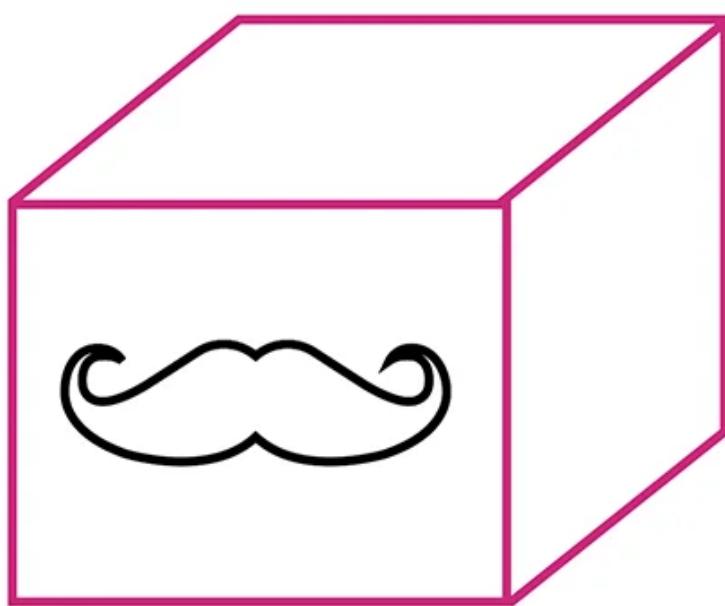
```
diamonds %>%
  ggplot(aes(cut, price)) +
  geom_violin()
```



```
diamonds %>%
  ggplot(aes(cut, price)) +
  geom_boxplot()
```



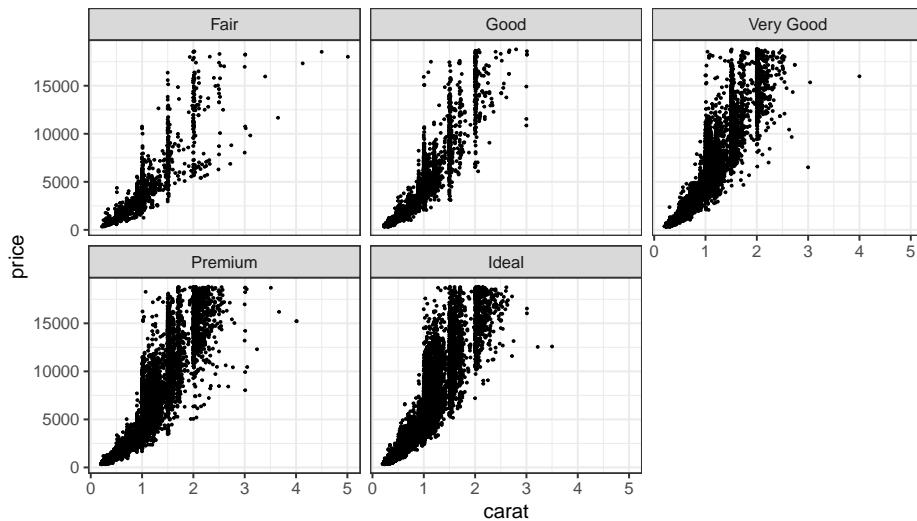
БОКСПЛОТ



6.9 Фасетизация

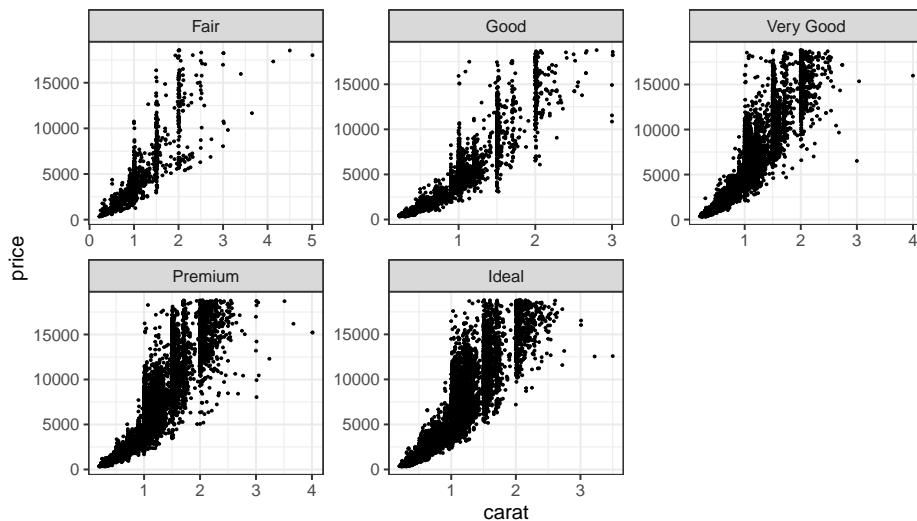
Достаточно мощным инструментом анализа данных является фасетизация, которая позволяет разбивать графики на основе какой-то переменной.

```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point(size = 0.3) +
  facet_wrap(~cut)
```



При этом иногда так бывает, что наличие какой-то одного значение в одном из фасетов, заставляет иметь одну и ту же шкалу для всех остальных. Это можно изменить при помощи аргумента `scales`:

```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point(size = 0.3) +
  facet_wrap(~cut, scales = "free")
```

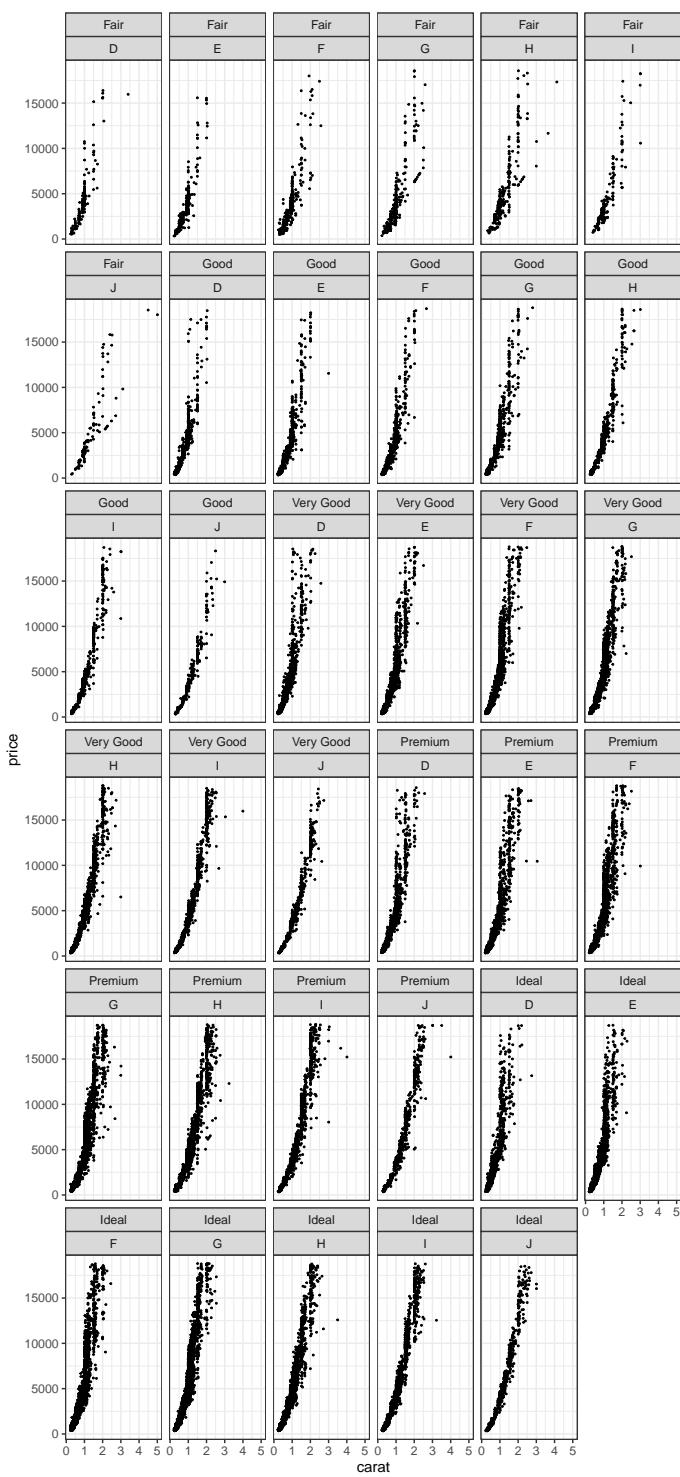


Кроме того, можно добавлять дополнительные аргументы:

```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point(size = 0.3) +
  facet_wrap(~cut + color)
```

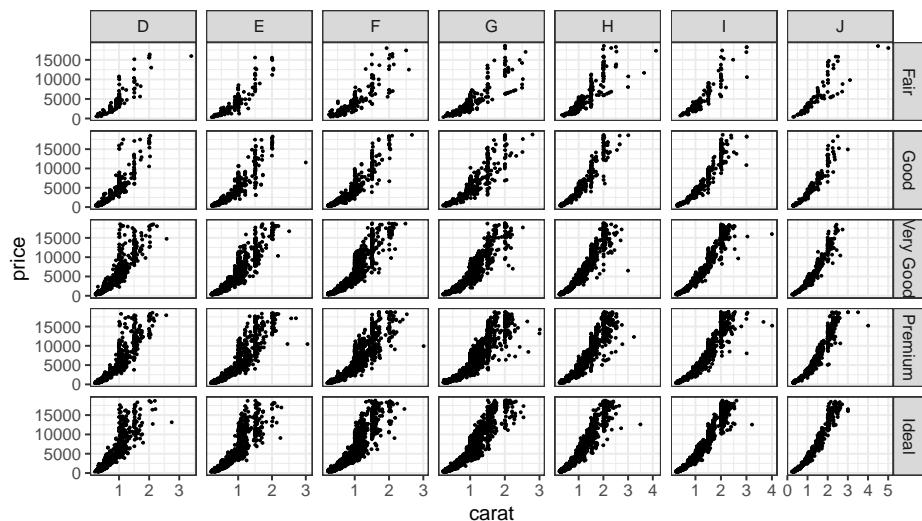
6.9. ФАСЕТИЗАЦИЯ

165



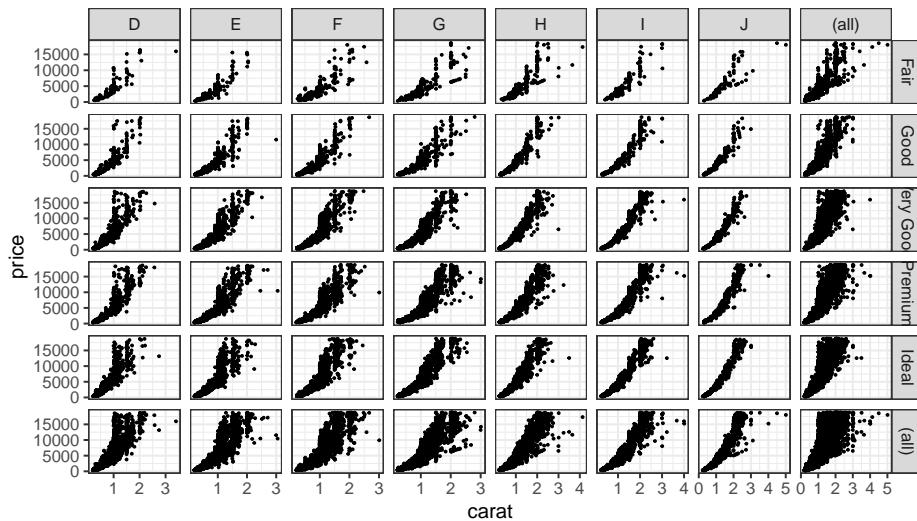
Кроме того, можно создавать сетки переменных используя `geom_grid()`, они `facet_grid()` выше места, чем `facet_wrap()`:

```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point(size = 0.3) +
  facet_grid(cut~color, scales = "free")
```



Кроме того `facet_grid()` позволяет делать обобщающие поля, где представлены все данные по какой-то строчке или столбцу:

```
diamonds %>%
  ggplot(aes(carat, price)) +
  geom_point(size = 0.3) +
  facet_grid(cut~color, scales = "free", margins = TRUE)
```



6.10 Визуализация комбинаций признаков

6.10.1 Потоковая Диаграмма (Sankey diagram)

Один из способов визуализации отношений между признаками называется потоковая диаграмма¹².

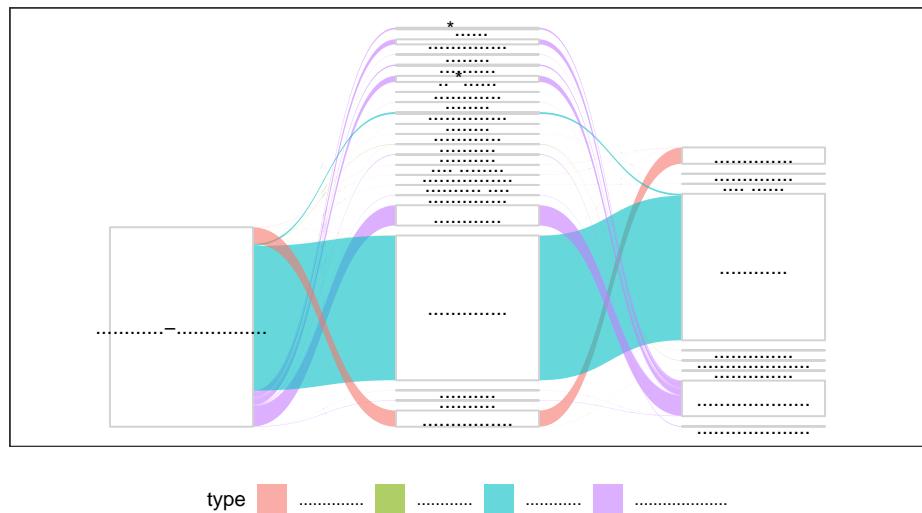
```
library("ggforce")
zhadina <- read_csv("https://raw.githubusercontent.com/agricolamz/2020-2021-ds4dh/master/data/zhadina.csv")

## 
## -- Column specification -----
## cols(
##   word_1 = col_character(),
##   word_2 = col_character(),
##   word_3 = col_character(),
##   type = col_character(),
##   n = col_double()
## )

zhadina %>%
  gather_set_data(1:3) %>%
  ggplot(aes(x, id = id, split = y, value = n)) +
  geom_parallel_sets(aes(fill = type), alpha = 0.6, axis.width = 0.5) +
  geom_parallel_sets_axes(axis.width = 0.5, color = "lightgrey", fill = "white") +
```

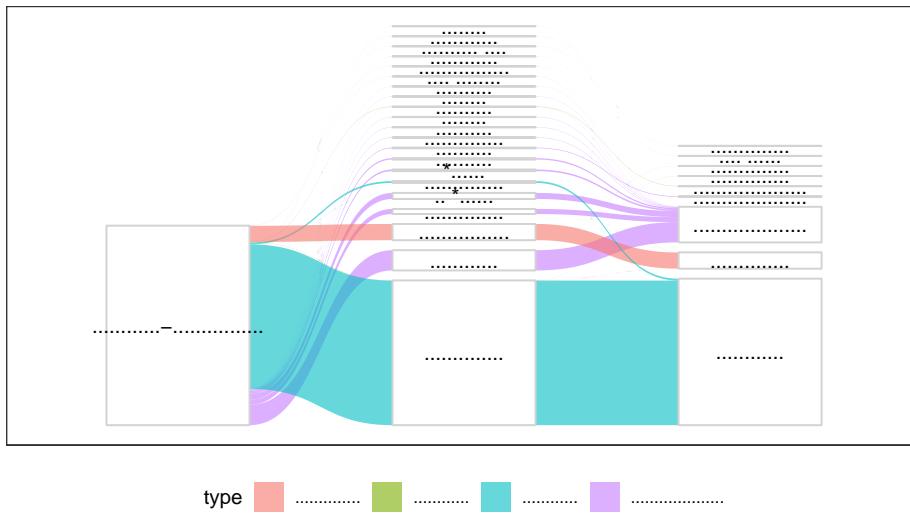
¹²https://en.wikipedia.org/wiki/Sankey_diagram

```
geom_parallel_sets_labels(angle = 0) +
theme_no_axes() +
theme(legend.position = "bottom")
```



А как поменять порядок? Снова факторы.

```
zhadina %>%
gather_set_data(1:3) %>%
mutate(y = fct_reorder(y, n, mean)) %>%
ggplot(aes(x, id = id, split = y, value = n)) +
geom_parallel_sets(aes(fill = type), alpha = 0.6, axis.width = 0.5) +
geom_parallel_sets_axes(axis.width = 0.5, color = "lightgrey", fill = "white") +
geom_parallel_sets_labels(angle = 0) +
theme_no_axes() +
theme(legend.position = "bottom")
```



Можно донастроить, задав собственный порядок в аргументе `levels` функции `factor()`.

6.10.2 UpSet Plot

Если диаграмма Sankey визуализирует попарные отношения между переменными, то график UpSet потенциально может визуализировать все возможные комбинации и является хорошей альтернативой диаграмме Вена, с большим количеством переменных (см. эту статью Лауры Эллис¹³).

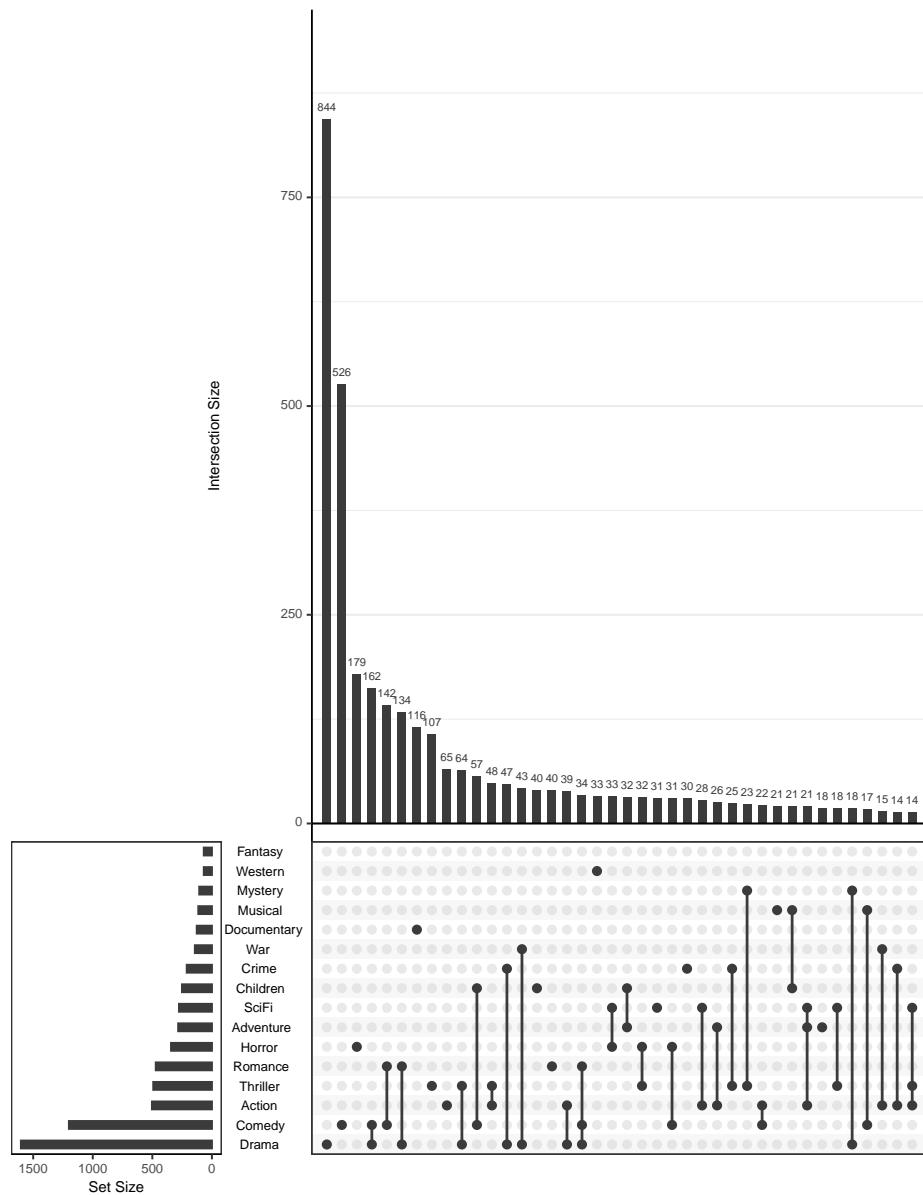
```
library(UpSetR)
movies <- read.csv( system.file("extdata", "movies.csv", package = "UpSetR"), header=TRUE, sep=";")
str(movies)

## 'data.frame': 3883 obs. of  21 variables:
## $ Name      : chr  "Toy Story (1995)" "Jumanji (1995)" "Grumpier Old Men (1995)" "Waiting to ...
## $ ReleaseDate: int  1995 1995 1995 1995 1995 1995 1995 1995 1995 ...
## $ Action     : int  0 0 0 0 1 0 0 1 1 ...
## $ Adventure  : int  0 1 0 0 0 0 1 0 1 ...
## $ Children   : int  1 1 0 0 0 0 0 1 0 0 ...
## $ Comedy     : int  1 0 1 1 1 0 1 0 0 0 ...
## $ Crime      : int  0 0 0 0 1 0 0 0 0 ...
## $ Documentary: int  0 0 0 0 0 0 0 0 0 ...
## $ Drama      : int  0 0 0 1 0 0 0 0 0 ...
## $ Fantasy    : int  0 1 0 0 0 0 0 0 0 ...
## $ Noir       : int  0 0 0 0 0 0 0 0 0 ...
## $ Horror     : int  0 0 0 0 0 0 0 0 0 ...
```

¹³<https://www.littlemissdata.com/blog/set-analysis>

```
## $ Musical      : int  0 0 0 0 0 0 0 0 0 0 ...
## $ Mystery      : int  0 0 0 0 0 0 0 0 0 0 ...
## $ Romance      : int  0 0 1 0 0 0 1 0 0 0 ...
## $ SciFi        : int  0 0 0 0 0 0 0 0 0 0 ...
## $ Thriller     : int  0 0 0 0 0 1 0 0 0 1 ...
## $ War           : int  0 0 0 0 0 0 0 0 0 0 ...
## $ Western       : int  0 0 0 0 0 0 0 0 0 0 ...
## $ AvgRating    : num  4.15 3.2 3.02 2.73 3.01 3.88 3.41 3.01 2.66 3.54 ...
## $ Watches       : int  2077 701 478 170 296 940 458 68 102 888 ...
```

```
upset(movies[,3:19], nsets = 16, order.by = "freq")
```



Глава 7

Работа со строками

7.1 Работа со строками в R

Для работы со строками можно использовать:

- базовый R
- пакет `stringr` (часть `tidyverse`)
- пакет `stringi` – отдельный пакет, так что не забудьте его установить:

```
install.packages("stringi")
library(tidyverse)
library(stringi)
```

Мы будем пользоваться в основном пакетами `stringr` и `stringi`, так как они в большинстве случаев удобнее. К счастью функции этих пакетов легко отличить от остальных: функции пакета `stringr` всегда начинаются с `str_`, а функции пакета `stringi` — с `stri_`.

Существует cheat sheet по `stringr`¹.

7.2 Как получить строку?

- следите за кавычками

```
"the quick brown fox jumps over the lazy dog"
```

```
## [1] "the quick brown fox jumps over the lazy dog"
```

¹<https://github.com/rstudio/cheatsheets/raw/master/strings.pdf>

```
'the quick brown fox jumps over the lazy dog'
```

```
## [1] "the quick brown fox jumps over the lazy dog"
```

```
"the quick 'brown' fox jumps over the lazy dog"
```

```
## [1] "the quick 'brown' fox jumps over the lazy dog"
```

```
'the quick "brown" fox jumps over the lazy dog'
```

```
## [1] "the quick \"brown\" fox jumps over the lazy dog"
```

- пустая строка

```
""
```

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

```
''
```

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

```
character(3)
```

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

- преобразование

```
typeof(4:7)
```

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

```
as.character(4:7)
```

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

- встроенные векторы

```
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"
```

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"
```

month.name

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

- Создание рандомных строк

```
set.seed(42)
stri_rand_strings(n = 10, length = 5:14)
```

```
## [1] "uwHpd"          "Wj8ehS"         "ivFSwy7"        "TYu8zw5V"
## [5] "OuRpjo0g0"       "p0CubNR2yQ"      "xtdyckLOm2k"    "fAGVfylZqBGp"
## [9] "gE28DTCiONV0a"    "9MemYE55If0Cvv"
```

- Перемешивает символы внутри строки

```
stri_rand_shuffle(" , - , - , - ")
```

```
## [1] " , - , - , - "
```

```
stri_rand_shuffle(month.name)
```

```
## [1] "aJayunr"      "eyrabraFu"    "achMr"      "Aplri"      "ayM"        "Jnue"
## [7] "uJly"         "usuAgt"      "tpebermSe"  "t0ecrbo"    "oeNembvr"   "Dmceerbe"
```

- Генерирует псевдорандомный текст²

```
stri_rand_lipsum(nparagraphs = 2)
```

```
## [1] "Lorem ipsum dolor sit amet, donec sit nunc urna sed ultricies ac pharetra orci luctus iad
## [2] "Risus eleifend magnis neque diam, suspendisse ullamcorper nulla adipiscing malesuada mass
```

7.3 Соединение и разделение строк

Соединить строки можно используя функцию `str_c()`, в которую, как и в функции `()`, можно перечислять элементы через запятую:

²Лорем ipsum — классический текст-заполнитель на основе трактата Марка Туллия Цицерона “О прелестах добра и зла”. Его используют, чтобы посмотреть, как страница смотрится, когда заполнена текстом

```
tibble(upper = rev(LETTERS), smaller = letters) %>%
  mutate(merge = str_c(upper, smaller))

## # A tibble: 26 x 3
##       upper smaller   merge
##       <chr>  <chr>    <chr>
## 1 Z      a        Za
## 2 Y      b        Yb
## 3 X      c        Xc
## 4 W      d        Wd
## 5 V      e        Ve
## 6 U      f        Uf
## 7 T      g        Tg
## 8 S      h        Sh
## 9 R      i        Ri
## 10 Q     j        Qj
## # ... with 16 more rows
```

Кроме того, если хочется, можно использовать особенный разделитель, указав его в аргументе `sep`:

```
tibble(upper = rev(LETTERS), smaller = letters) %>%
  mutate(merge = str_c(upper, smaller, sep = "_"))

## # A tibble: 26 x 3
##       upper smaller   merge
##       <chr>  <chr>    <chr>
## 1 Z      a        Z_a
## 2 Y      b        Y_b
## 3 X      c        X_c
## 4 W      d        W_d
## 5 V      e        V_e
## 6 U      f        U_f
## 7 T      g        T_g
## 8 S      h        S_h
## 9 R      i        R_i
## 10 Q     j        Q_j
## # ... with 16 more rows
```

Аналогичным образом, для разделяние строки на подстроки можно использовать функцию `separate()`. Это функция разносит разделенные элементы строки в соответствующие столбцы. У функции три обязательных аргумента: `col` — колонка, которую следует разделить, `into` — вектор названий новых столбец, `sep` — разделитель.

```
tibble(upper = rev(LETTERS), smaller = letters) %>%
  mutate(merge = str_c(upper, smaller, sep = "_")) %>%
  separate(col = merge, into = c("column_1", "column_2"), sep = "_")
```

```
## # A tibble: 26 x 4
##   upper smaller column_1 column_2
##   <chr>  <chr>    <chr>    <chr>
## 1 Z      a        Z        a
## 2 Y      b        Y        b
## 3 X      c        X        c
## 4 W      d        W        d
## 5 V      e        V        e
## 6 U      f        U        f
## 7 T      g        T        g
## 8 S      h        S        h
## 9 R      i        R        i
## 10 Q     j        Q        j
## # ... with 16 more rows
```

Кроме того, есть инструмент `str_split()`, которая позволяет разбивать строки на подстроки, но возвращает список.

```
str_split(month.name, "r")
```

```
## [[1]]
## [1] "Janua" "y"
##
## [[2]]
## [1] "Feb"   "ua"   "y"
##
## [[3]]
## [1] "Ma"    "ch"
##
## [[4]]
## [1] "Ap"    "il"
##
## [[5]]
## [1] "May"
##
## [[6]]
## [1] "June"
##
## [[7]]
## [1] "July"
```

```

## 
## [[8]]
## [1] "August"
## 
## [[9]]
## [1] "Septembe" ""
## 
## [[10]]
## [1] "Octobe" ""
## 
## [[11]]
## [1] "Novembe" ""
## 
## [[12]]
## [1] "Decembe" ""

```

7.4 Количество символов

7.4.1 Подсчет количества символов



```
tibble(mn = month.name) %>%
  mutate(n_charactars = str_count(mn))
```

```

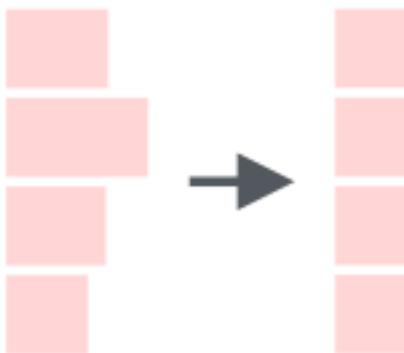
## # A tibble: 12 x 2
##   mn      n_charactars
##   <chr>        <int>
## 1 January       7
## 2 February      8
## 3 March         5
## 4 April         5
## 5 May          3

```

```
##  6 June      4
##  7 July      4
##  8 August     6
##  9 September   9
## 10 October    7
## 11 November   8
## 12 December   8
```

7.4.2 Подгонка количества символов

Можно обрезать строки, используя функцию `str_trunc()`:



```
tibble(mn = month.name) %>%
  mutate(mn_new = str_trunc(mn, 6))
```

```
## # A tibble: 12 x 2
##       mn      mn_new
##   <chr>    <chr>
## 1 January  Jan...
## 2 February Feb...
## 3 March   March
## 4 April   April
## 5 May     May
## 6 June   June
## 7 July   July
## 8 August August
## 9 September Sep...
## 10 October Oct...
## 11 November Nov...
```

```
## 12 December Dec...
```

Можно решить с какой стороны обрезать, используя аргумент `side`:

```
tibble(mn = month.name) %>%
  mutate(mn_new = str_trunc(mn, 6, side = "left"))
```

```
## # A tibble: 12 x 2
##   mn      mn_new
##   <chr>   <chr>
## 1 January ...ary
## 2 February ...ary
## 3 March   March
## 4 April   April
## 5 May     May
## 6 June   June
## 7 July   July
## 8 August  August
## 9 September ...ber
## 10 October ...ber
## 11 November ...ber
## 12 December ...ber
```

```
tibble(mn = month.name) %>%
  mutate(mn_new = str_trunc(mn, 6, side = "center"))
```

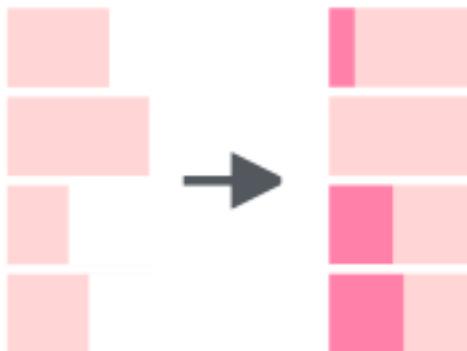
```
## # A tibble: 12 x 2
##   mn      mn_new
##   <chr>   <chr>
## 1 January Ja...y
## 2 February Fe...y
## 3 March   March
## 4 April   April
## 5 May     May
## 6 June   June
## 7 July   July
## 8 August  August
## 9 September Se...r
## 10 October Oc...r
## 11 November No...r
## 12 December De...r
```

Можно заменить многоточие, используя аргумент `ellipsis`:

```
tibble(mn = month.name) %>%
  mutate(mn_new = str_trunc(mn, 3, ellipsis = ""))
```

```
## # A tibble: 12 x 2
##   mn      mn_new
##   <chr>    <chr>
## 1 January Jan
## 2 February Feb
## 3 March Mar
## 4 April Apr
## 5 May May
## 6 June Jun
## 7 July Jul
## 8 August Aug
## 9 September Sep
## 10 October Oct
## 11 November Nov
## 12 December Dec
```

Можно наоборот “раздуть” строку:



```
tibble(mn = month.name) %>%
  mutate(mn_new = str_pad(mn, 10))
```

```
## # A tibble: 12 x 2
##   mn      mn_new
##   <chr>    <chr>
## 1 January " January"
## 2 February " February"
```

```
##  3 March      "      March"
##  4 April      "      April"
##  5 May       "      May"
##  6 June      "      June"
##  7 July      "      July"
##  8 August     "      August"
##  9 September  "      September"
## 10 October    "      October"
## 11 November   "      November"
## 12 December   "      December"
```

Опять же есть аргумент `side`:

```
tibble(mn = month.name) %>%
  mutate(mn_new = str_pad(mn, 10, side = "right"))
```

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

Также можно выбрать, чем “раздувать строку”:

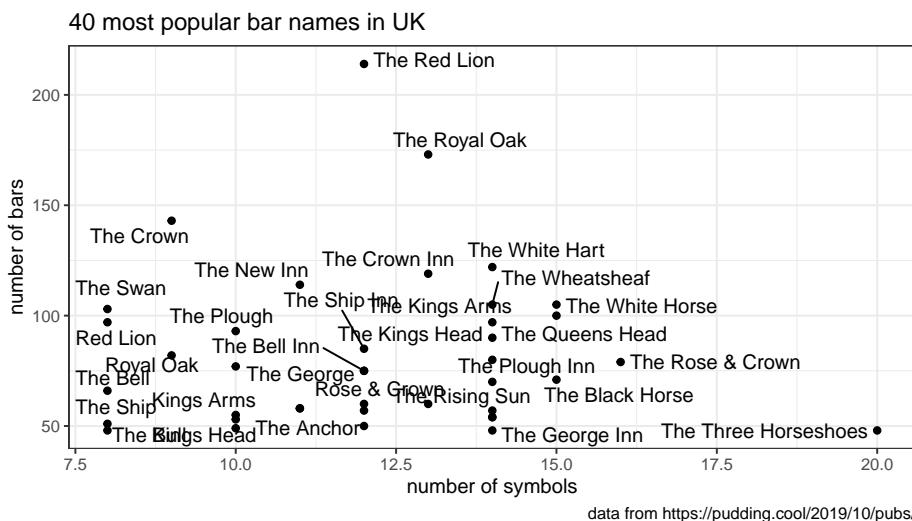
```
tibble(mn = month.name) %>%
  mutate(mn_new = str_pad(mn, 10, pad = "."))
```

```
## # A tibble: 12 x 2
##   mn      mn_new
##   <chr>   <chr>
## 1 January ...January
## 2 February ..February
## 3 March   .....March
## 4 April   .....April
## 5 May    .....May
## 6 June   .....June
```

```
##  7 July      .....July
##  8 August     ....August
##  9 September .September
## 10 October    ...October
## 11 November   ..November
## 12 December   ..December
```



На Pudding вышла статья про английские пабы³. Здесь⁴ лежит немного обработанный датасет, которые они использовали. Визуализируйте 40 самых частотных названий пабов в Великобритании, отложив по оси x количество символов, а по оси y – количество баров с таким названием.



□ список подсказок □

□ Датасет скачался, что дальше? □ Перво-наперво следует создать переменную, в которой бы хранилось количество каждого из баров.

□ А как посчитать количество баров? □ Это можно сделать при помощи функции `count()`.

□ Бары пересчитали, что дальше? □ Теперь нужно создать новую переменную, где бы хранилась информация о количестве символов.

□ Все переменные есть, теперь рисуем? □ Не совсем. Перед тем как рисовать нужно отфильтровать 50 самых популярных.

□ Так, все готово, а какие `geom_()`? □ На графике `geom_point()` и `geom_text_repel()` из пакета `ggrepel`.

□ А-а-а! could not find function "geom_text_repel" □ А вы включили библиотеку `ggrepel`? Если не включили, то функция, естественно будет недоступна.

¶ A-a-a-a! `geom_text_repel` requires the following missing aesthetics:
`label`" ¶ Все, как написала программа: чтобы писать какой-то текст в функции `aes()` нужно добавить аргумент `label = pub_name`. Иначе откуда он узнает, что ему писать?

¶ Фуф! Все готово! ¶ А оси подписаны? А заголовок? А подпись про источник данных?

7.5 Сортировка

Для сортировки существует базовая функция `sort()` и функция из `stringr` `str_sort()`:

```
unsorted_latin <- c("I", " ", "N", "Y")
sort(unsorted_latin)
```

```
## [1] " " "I" "N" "Y"
```

```
str_sort(unsorted_latin)
```

```
## [1] " " "I" "N" "Y"
```

```
str_sort(unsorted_latin, locale = "lt")
```

```
## [1] " " "I" "Y" "N"
```

```
unsorted_cyrillic <- c(" ", "i", " ")
str_sort(unsorted_cyrillic)
```

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

```
str_sort(unsorted_cyrillic, locale = "ru_UA")
```

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

Список локалей на компьютере можно посмотреть командой `stringi::stri_locale_list()`. Список всех локалей вообще приведен на этой странице⁵. Еще полезные команды: `stringi::stri_locale_info` и `stringi::stri_locale_set`.

Не углубляясь в разнообразие алгоритмов сортировки⁶, отмечу, что алгоритм по-умолчанию хуже работает с большими данными:

⁵https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes

⁶<https://www.youtube.com/watch?v=BeoCbJPuvSE>

```
set.seed(42)
huge <- sample(letters, 1e7, replace = TRUE)
head(huge)

## [1] "q" "e" "a" "y" "j" "d"

system.time(
  sort(huge)
)

##      user  system elapsed
##     6.785   0.020   6.806

system.time(
  sort(huge, method = "radix")
)

##      user  system elapsed
##     0.277   0.028   0.305

system.time(
  str_sort(huge)
)

##      user  system elapsed
##     5.949   0.050   5.999

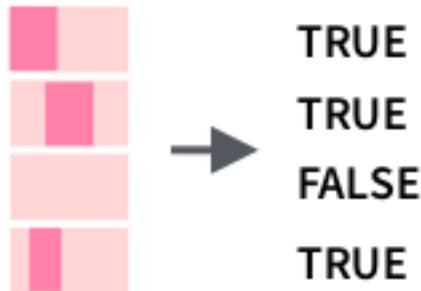
huge_tbl <- tibble(huge)
system.time(
  huge_tbl %>%
    arrange(huge)
)

##      user  system elapsed
##    34.201   0.043  34.246
```

Предварительный вывод: для больших данных – `sort(..., method = "radix")`.

7.6 Поиск подстроки

Можно использовать функцию `str_detect()`:



```
tibble(mn = month.name) %>%
  mutate(has_r = str_detect(mn, "r"))
```

```
## # A tibble: 12 x 2
##   mn      has_r
##   <chr>    <lgl>
## 1 January  TRUE
## 2 February TRUE
## 3 March   TRUE
## 4 April   TRUE
## 5 May     FALSE
## 6 June    FALSE
## 7 July    FALSE
## 8 August  FALSE
## 9 September TRUE
## 10 October TRUE
## 11 November TRUE
## 12 December TRUE
```

Кроме того, существует функция, которая возвращает индексы, а не значения



TRUE/FALSE:

```
tibble(mn = month.name) %>%
  slice(str_which(mn, "r"))
```

```
## # A tibble: 8 x 1
##   mn
##   <chr>
## 1 January
## 2 February
## 3 March
## 4 April
## 5 September
## 6 October
## 7 November
## 8 December
```

Также можно посчитать количество вхождений какой-то подстроки:



```
tibble(mn = month.name) %>%
  mutate(has_r = str_count(mn, "r"))
```

```
## # A tibble: 12 x 2
##   mn      has_r
##   <chr>    <int>
## 1 January     1
## 2 February    2
## 3 March       1
## 4 April       1
## 5 May         0
## 6 June        0
## 7 July         0
## 8 August      0
## 9 September   1
```

```
## 10 October      1
## 11 November     1
## 12 December     1
```

7.7 Изменение строк

7.7.1 Изменение регистра

```
latin <- "tHe QuIcK BrOwN f0x JuMpS OvEr ThE lAzY d0g"
cyrillic <- " , , , - , - "
str_to_upper(latin)

## [1] "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"

str_to_lower(cyrillic)

## [1] " , , , - , - "

str_to_title(latin)

## [1] "The Quick Brown Fox Jumps Over The Lazy Dog"
```

7.7.2 Выделение подстроки

Подстроку в строке можно выделить двумя способами: по индексам функцией `str_sub()`, и по подстроке функцией `str_png()`.

```
extract(images/5.07_str_sub.png)
```

```
tibble(mn = month.name) %>%
  mutate(mutate = str_sub(mn, start = 1, end = 2))

## # A tibble: 12 x 2
##   mn      mutate
##   <chr>    <chr>
## 1 January  Ja
## 2 February Fe
## 3 March   Ma
## 4 April   Ap
## 5 May     Ma
## 6 June    Ju
## 7 July    Ju
## 8 August  Au
```

```
##  9 September Se
## 10 October  Oc
## 11 November No
## 12 December De
```



```
tibble(mn = month.name) %>%
  mutate(mutate = str_extract(mn, "r"))
```

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

По умолчанию функция `str_extract()` возвращает первое вхождение подстроки, соответствующей шаблону. Также существует функция `str_extract_all()`, которая возвращает все вхождения подстрок, соответствующих шаблону, однако возвращает объект типа список.

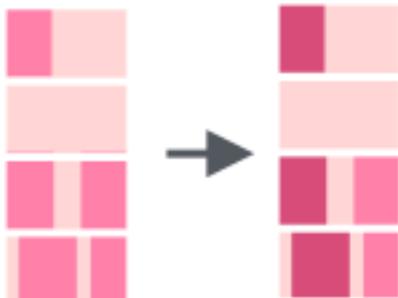
```
str_extract_all(month.name, "r")
```

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

```
## [[2]]
## [1] "r" "r"
##
## [[3]]
## [1] "r"
##
## [[4]]
## [1] "r"
##
## [[5]]
## character(0)
##
## [[6]]
## character(0)
##
## [[7]]
## character(0)
##
## [[8]]
## character(0)
##
## [[9]]
## [1] "r"
##
## [[10]]
## [1] "r"
##
## [[11]]
## [1] "r"
##
## [[12]]
## [1] "r"
```

7.7.3 Замена подстроки

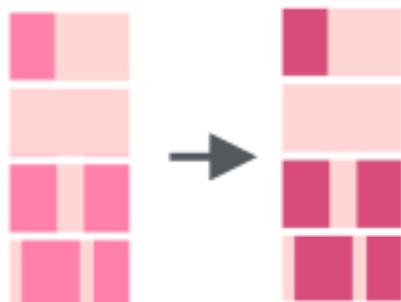
Существует функция `str_replace()`, которая позволяет заменить одну подстроку в строке на другую:



```
tibble(mn = month.name) %>%
  mutate(mutate = str_replace(mn, "r", "R"))
```

```
## # A tibble: 12 x 2
##   mn      mutate
##   <chr>    <chr>
## 1 January  JanuaRy
## 2 February FebRuary
## 3 March   MaRch
## 4 April   ApRil
## 5 May     May
## 6 June    June
## 7 July    July
## 8 August  August
## 9 September SeptembeR
## 10 October  OctobeR
## 11 November NovembeR
## 12 December DecembeR
```

Как и другие функции `str_replace()` делает лишь одну замену, чтобы заменить все вхождения подстроки следует использовать функцию `str_replace_all()`:



```
tibble(mn = month.name) %>%
  mutate(mutate = str_replace_all(mn, "r", "R"))

## # A tibble: 12 x 2
##   mn      mutate
##   <chr>    <chr>
## 1 January JanuaRy
## 2 February FebRuaRy
## 3 March   MaRch
## 4 April   ApRil
## 5 May     May
## 6 June   June
## 7 July   July
## 8 August August
## 9 September SeptembeR
## 10 October OctobeR
## 11 November NovembeR
## 12 December DecembeR
```

7.7.4 Удаление подстроки

Для удаления подстроки на основе шаблона, используется функция `str_remove()` и `str_remove_all()`

```
tibble(month.name) %>%
  mutate(mutate = str_remove(month.name, "r"))

## # A tibble: 12 x 2
##   month.name mutate
##   <chr>      <chr>
## 1 January    Januay
## 2 February   Febuary
## 3 March     Mach
## 4 April     Apil
## 5 May       May
## 6 June     June
## 7 July     July
## 8 August   August
## 9 September Septembe
## 10 October  Octobe
## 11 November Novembe
## 12 December Decembe
```

```
tibble(month.name) %>%
  mutate(mutate = str_remove_all(month.name, "r"))

## # A tibble: 12 x 2
##   month.name     mutate
##   <chr>        <chr>
## 1 January      Januay
## 2 February    Febuay
## 3 March       Mach
## 4 April       Apil
## 5 May         May
## 6 June        June
## 7 July         July
## 8 August      August
## 9 September  Septembe
## 10 October    Octobe
## 11 November   Novembe
## 12 December   Decembe
```

7.7.5 Транслитерация строк

В пакете `stringi` существует достаточно много методов транслитераций строк, которые можно вызвать командой `stri_trans_list()`. Вот пример использования некоторых из них:

```
stri_trans_general("stringi", "latin-cyrillic")
```

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

```
stri_trans_general(" ", "cyrillic-latin")
```

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

```
stri_trans_general("stringi", "latin-greek")
```

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

```
stri_trans_general("stringi", "latin-armenian")
```

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



Вот два датасета:

- список городов России⁷
- частотный словарь русского языка [Шаров, Ляшевская 2011]⁸

Определите сколько городов называется обычным словом русского языка (например, город Орёл)? Не забудьте поменять ё на е.

список подсказок

Датасеты скачались, что дальше? Надо их преобразовать к нужному виду и объединить.

А как их соединить? Что у них общего?

В одном датасете есть переменная `city`, в другом – переменная `lemma`. Все города начинаются с большой буквы, все леммы с маленькой буквы. Я бы уменьшил букву в датасете с городами, сделал бы новый столбец в датасете с городами (например, `town`), соединил бы датасеты и посчитал бы сколько в результирующем датасете значений `town`.

А как соединить? Я бы использовал `dict %>% ... %>% inner_join(cities)`. Если в датасетах разные названия столбцов, то следует указывать какие столбцы, каким соответствуют:`dict %>% ... %>% inner_join(cities, by = c("lemma" = "city"))`

Соединилось вроде... А как посчитать? Я бы, как обычно, использовал функцию `count()`.

7.8 Регулярные выражения

Большинство функций из раздела об операциях над векторами (`str_detect()`, `str_extract()`, `str_remove()` и т. п.) имеют следующую структуру:

- строка, с которой работает функция
- образец (pattern)

Дальше мы будем использовать функцию `str_view_all()`, которая позволяет показывать выделенное образцом в исходной строке.

```
str_view_all("c", "c") #
```

Я всегда путаю с и

7.8.1 Экранирование метасимволов

```
a <- "      ,   4$\\"2 + 3$ * 5 = 17$?  ?      (      ) . [|}^{|}"  
str_view_all(a, "$")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\$")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\.")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\*")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\+")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\?")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\(")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\)")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\|")
```

Всем известно, что $4\$\\2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}^{|}]

```
str_view_all(a, "\\"^")
```

Всем известно, что $4\$ \cdot 2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|]{|}

```
str_view_all(a, "\\[")
```

Всем известно, что $4\$ \cdot 2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}{|}

```
str_view_all(a, "\\]"")
```

Всем известно, что $4\$ \cdot 2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}{|]

```
str_view_all(a, "\\{")
```

Всем известно, что $4\$ \cdot 2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}{|]

```
str_view_all(a, "\\}")")
```

Всем известно, что $4\$ \cdot 2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|]{|}

```
str_view_all(a, "\\\\")")
```

Всем известно, что $4\$ \cdot 2 + 3\$ * 5 = 17\$$? Да? Ну хорошо (а то я не был уверен). [|}{|}

7.8.2 Классы знаков

- \\d – цифры. \\D – не цифры.

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "\\d")
```

два . . . !

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "\\D")
```

. . . !

- \\s – пробелы. \\S – не пробелы.

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "\\s")
```

два 15 42. 42 15. 37 08 5. 20 20 20!

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "\s")
```

два 15 42. 42 15. 37 08 5. 20 20 20!

- `\w` – не пробелы и не знаки препинания. `\W` – пробелы и знаки препинания.

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "\w")
```

два 15 42. 42 15. 37 08 5. 20 20 20!

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "\W")
```

два 15 42. 42 15. 37 08 5. 20 20 20!

- произвольная группа символов и обратная к ней

```
str_view_all("      ,           ", "[ ]")
```

Умей мечтать, не став рабом мечтанья

```
str_view_all("      ,           ", "[^ ]")
```

И мыслить, мысли не обожествив

- встроенные группы символов

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "[0-9]")
```

два 15 42. 42 15. 37 08 5. 20 20 20!

```
str_view_all("      ,           ", "[ - ]")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", "[ - ])")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", "[ - ])")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

```
str_view_all("The quick brown Fox jumps over the lazy Dog", "[a-z])")
```

The quick brown Fox jumps over the lazy Dog

```
str_view_all(" 15 42. 42 15. 37 08 5. 20 20 20!", "[^0-9])")
```

два 15 42. 42 15. 37 08 5. 20 20 20!

regexp	matches
ab d	or
[abe]	one of
[^abe]	anything but
[a-c]	range

- выбор из нескольких групп

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", "[ | ])")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

- произвольный символ

```
str_view_all("Везет Сеняка Саньку с Сонькой на санках. Санки скок, Сеняку с ног, Соньку в лоб, все - в сугроб", "[ . ])")
```

Везет Сеняка Саньку с Сонькой на санках. Санки скок, Сеняку с ног, Соньку в лоб, все – в сугроб

- знак начала и конца строки

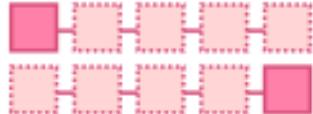
```
str_view_all("          .", ".")
```

от топота копыт пыль по полю летит.

```
str_view_all(" - , - ", "- $")
```

у ежа – ежата, у ужа – ужата

- есть еще другие группы и другие обозначения уже приведенных групп, см. ?regex



regexp

^
 a\$

matches

start of string
end of string

7.8.3 Квантификация

- ? – ноль или один раз

```
str_view_all("          ", "?")
```

хорошее длинношеее животное

- * – ноль и более раз

```
str_view_all("          ", "*")
```

хорошее длинношеее животное

- + – один и более раз

```
str_view_all("          ", "+")
```

хорошее длинношеее животное

- {n} – n раз

```
str_view_all("           ", " {2}")
```

хорошее длинношеее животное

- $\{n, \}$ – n раз и более

```
str_view_all("           ", " {1,}")
```

хорошее длинношеее животное

- $\{n, m\}$ – от n до m . Отсутствие пробела важно: $\{1, 2\}$ – правильно, $\{1, \sqcup 2\}$ – неправильно.

```
str_view_all("           ", " {2,3}")
```

хорошее длинношеее животное

- группировка символов

```
str_view_all(" , , , ", "( )+")
```

Пушкиновед, Лермонтовед, Лермонтолововед

```
str_view_all(" , , , ", "( )+")
```

беловатый, розоватый, розововатый

- жадный vs. нежадный алгоритмы

```
str_view_all(" , , , ", ". * ")
```

Пушкиновед, Лермонтовед, Лермонтолововед

```
str_view_all(" , , , ", ". *? ")
```

Пушкиновед, Лермонтовед, Лермонтолововед

regexp	matches
	zero or one
	zero or more
	one or more
	exactly m
	m or more
	between n and m

7.8.4 Позиционная проверка (look arounds)

Позиционная проверка – выглядит достаточно непоследовательно даже в свете остальных регулярных выражений.

Давайте найдем все *a* перед *p*:

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", " ", "(?= )")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

А теперь все *a* перед *p* или *l*:

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", " ", "(?=[ ])")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

Давайте найдем все *a* после *p*

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", " ", "(?=< ) ")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

А теперь все *a* после *p* или *l*:

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", " ", "(?=<[ ]) ")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

Также у этих выражений есть формы с отрицанием. Давайте найдем все *p* не перед *a*:

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", " (?!)")
```

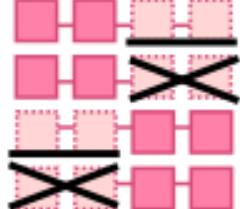
Карл у Клары украл кораллы, а Клара у Карла украла кларнет

А теперь все *p* не после *a*:

```
str_view_all("Карл у Клары украл кораллы, а Клара у Карла украла кларнет", "(?! ) ")
```

Карл у Клары украл кораллы, а Клара у Карла украла кларнет

Запомнить с ходу это достаточно сложно, так что подсматривайте сюда:

	regexp	matches
	<code>a(?=c)</code>	followed by
	<code>a(?:!c)</code>	not followed by
	<code>(?<=b)a</code>	preceded by
	<code>(?<!b)a</code>	not preceded by



Вот отсюда⁹ можно скачать файл с текстом стихотворения Н. Заболоцкого “Меркнут знаки здиака”. Посчитайте долю женских (ударение падает на предпоследний слог рифмующихся слов) и мужских (ударение падает на последний слог рифмующихся слов) рифм в стихотворении.

🔗 список подсказок ↳

🔗 Датасеты скачиваются с ошибкой, почему? ↳ Дело в том, что исходный файл в формате `.txt`, а не `.csv`. Его нужно скачивать, например, командой `read_lines()`

🔗 Ошибка: `... applied to an object of class "character"` ↳

Скачав файл Вы получили вектор со строками, где каждая элемент вектора – строка стихотворения. Создайте `tibble()`, тогда можно будет применять стандартные инструменты `tidyverse`.

🔗 Хорошо, `tibble()` создан, что дальше? ↳ Дальше нужно создать переменную, из которой будет понятно, мужская в каждой строке рифма, или женская.

🔗 А как определить, какая рифма? Нужно с словарем сравнивать? ↳ Формально говоря, определять рифму можно по косвенным признакам. Все стихотворение написано четырехстопным хореем, значит в нем либо 7, либо 8 слов. Значит, посчитав количество слов, мы поймем, какая перед нами рифма.

🔗 А как посчитать гласные? ↳ Нужно написать регулярное выражение... вроде бы это тема нашего занятия...

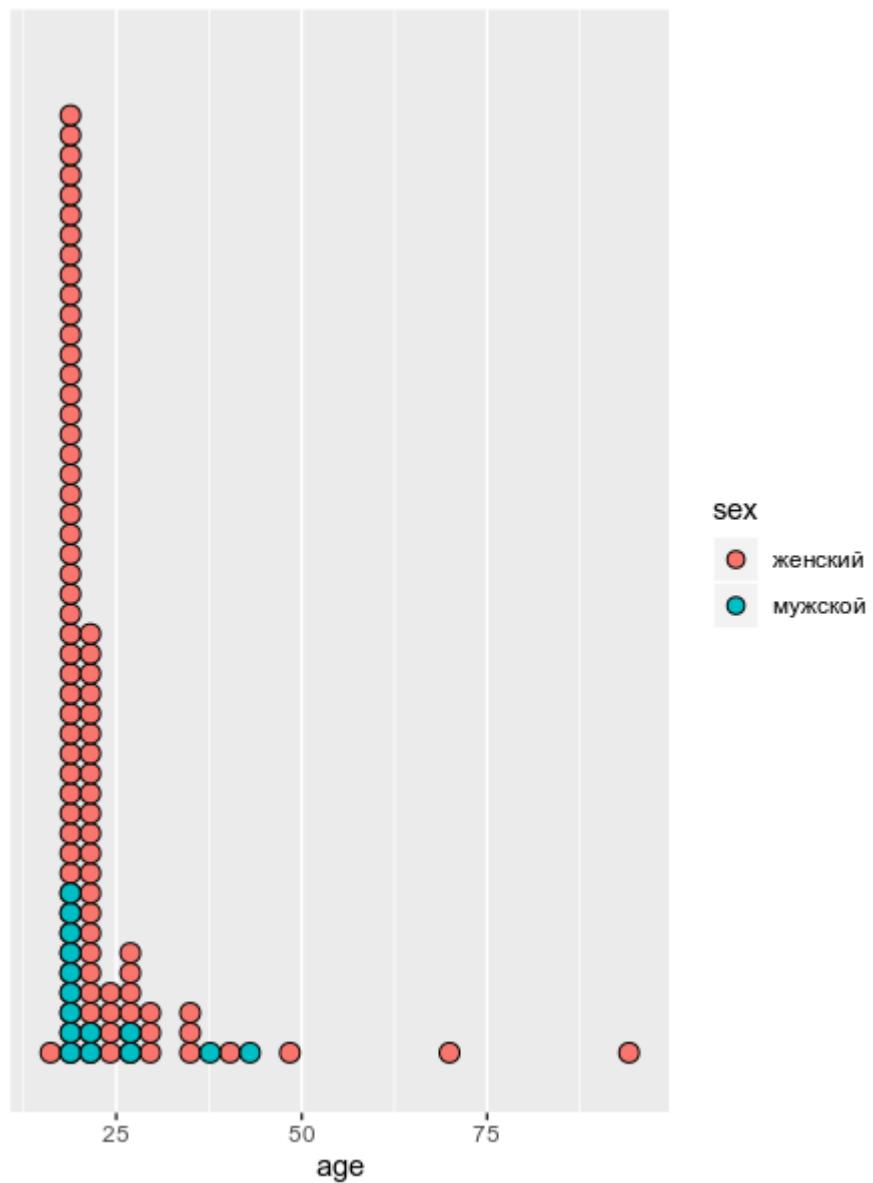
Гласные посчитаны. А что дальше? Ну теперь нужно посчитать, сколько каких длин (в количестве слогов) бывает в стихотворении. Это можно сделать при помощи функции `count()`.

А почему у меня есть строки длины 0 слогов? Ну, видимо, в стихотворении были пустые строки. Они использовались для разделения строф.

А почему у меня есть строки длины 6 слогов? Ну, видимо, Вы написали регулярное выражение, которое не учитывает, что гласные буквы могут быть еще и в начале строки, а значит написаны с большой буквы.



В ходе анализа данных чаще всего бороться со строками и регулярными выражениями приходится в процессе обработки неаккуратнособранных анкет. Предлагаю обработать переменные `sex` и `age` такой вот неудачно собранной анкеты¹⁰ и построить следующий график:



☞ список подсказок ☞

☞ А что это за `geom_...()`? ☞ Это `geom_dotplot()`¹¹ с аргументом `method = "histodot"` и с удаленной осью у при помощи команды `scale_y_continuous(NULL, breaks = NULL)`

¹¹[#D0%B4%D0%BE%D1%82%D0%BF%D0%BB%D0%BE%D1%82](https://agricolamz.github.io/DS_for_DH/viz-1.html)

❑ Почему на графике рисуется каждое значение возраста? ❑ Если Вы все правильно преобразовали, должно помочь преобразование строковой переменной age в числовую при помощи функции as.integer().

7.9 Определение языка

Для определения языка существует два пакета cld2 (вероятностный) и cld3 (нейросеть).

```
udhr_24 <- read_csv("https://raw.githubusercontent.com/agricolamz/DS_for_DH/master/data/article_24.csv")  
  
## # A tibble: 6 x 1  
##   article_text  
##   <chr>  
## 1 , ~  
## 2 Everyone has the right to rest and leisure, including reasonable limitation o~  
## 3 Toute personne a droit au repos et aux loisirs et notamment à une limitation ~  
## 4 Toda persona tiene derecho al descanso, al disfrute del tiempo libre, a una l~  
## 5 ~  
## 6  
  
cld2::detect_language(udhr_24$article_text)  
  
## [1] "ru" "en" "fr" "es" "ar" "zh"  
  
cld2::detect_language(udhr_24$article_text, lang_code = FALSE)  
  
## [1] "RUSSIAN" "ENGLISH" "FRENCH" "SPANISH" "ARABIC" "CHINESE"  
  
cld3::detect_language(udhr_24$article_text)  
  
## [1] "ru" "en" "fr" "es" "ar" "zh"
```

```

cld2::detect_language("    ?      -      ?")
## [1] "bg"

cld3::detect_language("    ?      -      ?")
## [1] NA

cld2::detect_language("  .          ,          ,          .")
## [1] "ru"

cld3::detect_language("  .          ,          ,          .")
## [1] "ru"

cld2::detect_language("  ...  '          ,          ,          .")
## [1] "uk"

cld3::detect_language("  ...  '          ,          ,          .")
## [1] "uk"

cld2::detect_language_mixed("OpenDataScience state-of-the-art"
## $classification
##   language code latin proportion
## 1 RUSSIAN   ru FALSE     0.87
## 2 ENGLISH   en  TRUE     0.11
## 3 UNKNOWN   un  TRUE     0.00
##
## $bytes
## [1] 353
##
## $reliabale
## [1] TRUE

cld3::detect_language_mixed("OpenDataScience state-of-the-art"

```

```
##   language probability reliable proportion
## 1      ru    0.9983915    TRUE 0.88951844
## 2      en    0.9992564    TRUE 0.05099150
## 3      sr    0.4266235   FALSE 0.04815864
```

7.10 Расстояния между строками

Существует много разных метрик для измерения расстояния между строками (см. ?'stringdist-metrics'), в примерах используется расстояние Дамерау — Левенштейна. Данное расстояние получается при подсчете количества операций, которые нужно сделать, чтобы перевести одну строку в другую.

- вставка ab → aNb
- удаление aOb → ab
- замена символа aOb → aNb
- перестановка символов ab → ba

```
library(stringdist)

## 
## Attaching package: 'stringdist'

## The following object is masked from 'package:tidyverse':
## 
##     extract

stringdist(" ", " ")

## [1] 0

stringdist(" ", c(" ", " ", " ", " ", " ", " ", " "))

## [1] 4 6 6 1 5

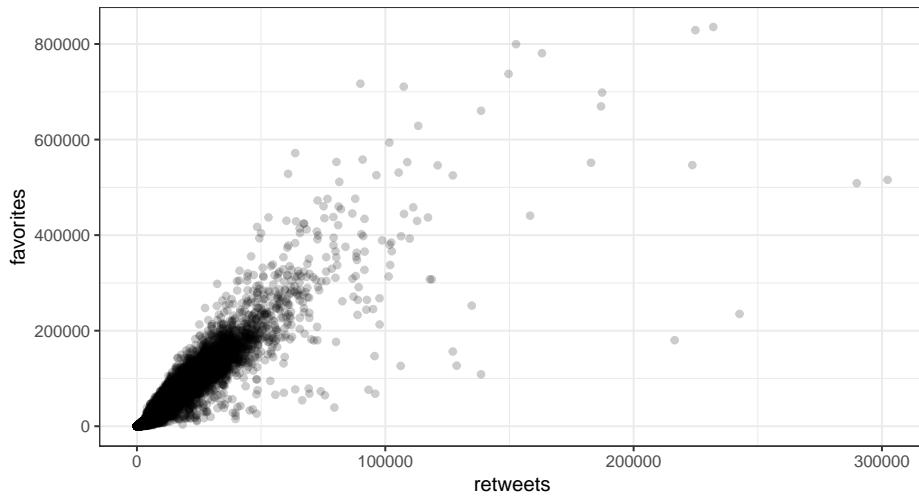
amatch(c(" ", " "), c(" ", " ", " ", " ", " ", " ", " "), maxDist = 2)

## [1] 2 4
```

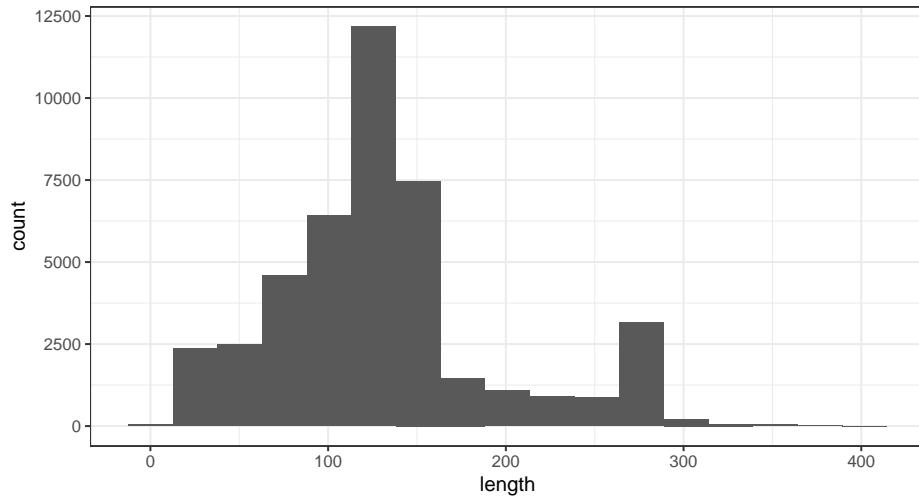
7.11 Дополнительные задания:



В датасет¹² записаны твиты Дональда Трампа взятые с kaggle¹³. Постройте график рассеяния, которые показывает связь количества ретвитов и лайков. Чтобы убрать научную запись больших чисел используйте команду `options(scipen = 999)`.

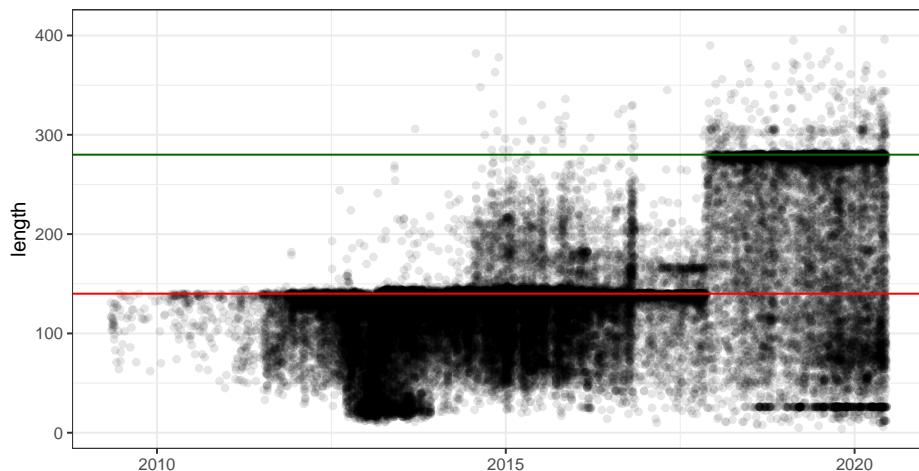


Постройте гистограмму, которая показывает распределения длины твитов в символах. Какой метод определения размера ячейки использован на приведенном графике? [Sturgers 1926], [Scott 1979] или [Freedman, Diaconis 1981]?





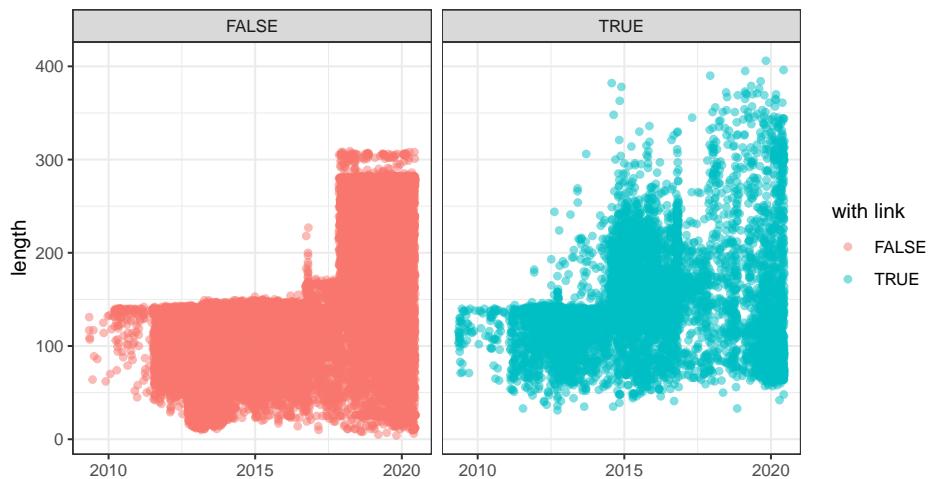
Постройте график рассеивания, который бы показывал связь с длиной твита во времени. Используя `geom_hline()`, наложите две линии: 140 символов и 280. Сделайте прозрачность 0.1.



data from www.kaggle.com/austinreese/trump-tweets



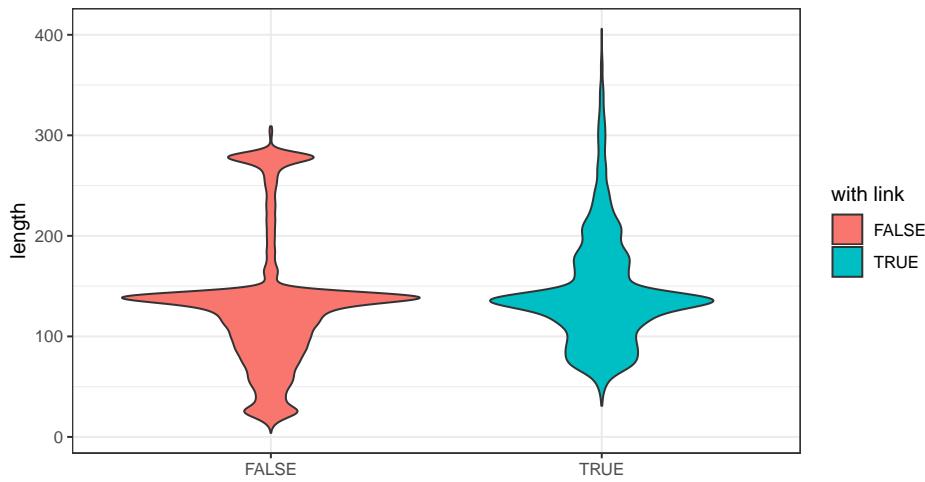
Постройте график рассеивания, который бы показывал связь с длиной твита во времени. Разбейте и раскрасьте твиты на основании наличия в них интернет ссылок.



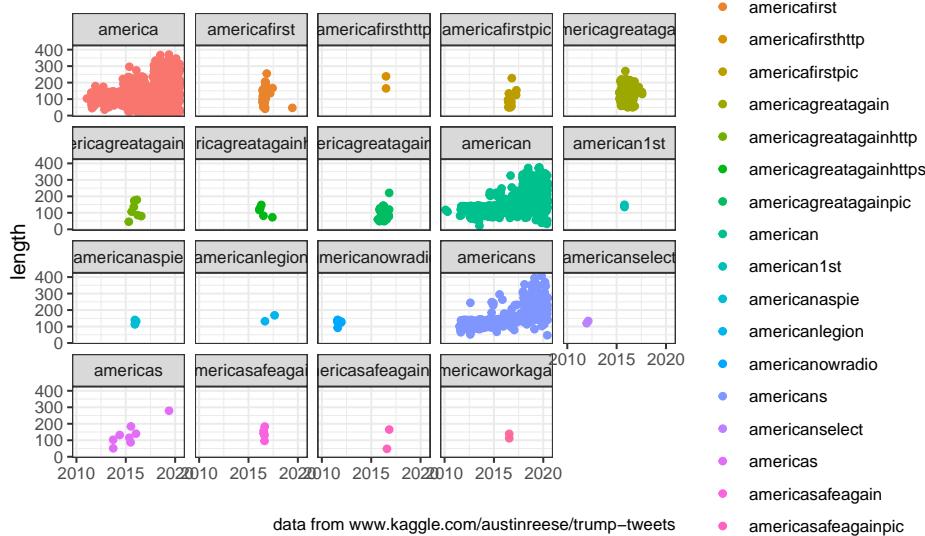
data from www.kaggle.com/austinreese/trump-tweets



Можно ли утверждать, что твиты со ссылками длиннее? Постройте вайолин-плот, которые показывает распределение значений длины твитов в зависимости от наличия в них интернет ссылок.



Найдите твиты которые содержат корень america, которые встречаются больше одного раза, и фасетизируйте по таким словам.



Глава 8

Представление данных: rmarkdown

Достаточно важной частью работы с данными является их представление. Мы рассмотрим наиболее распространенный варианты: `rmarkdown`, `flexdashboard` и `shiny`. Смотрите книжку (?)(<https://bookdown.org/yihui/rmarkdown/>) или cheatsheet¹.

8.1 rmarkdown

`rmarkdown` – это пакет, который позволяет соединять R команды и их исполнения в один документ. В результате можно комбинировать текст и исполняемый код, что в свою очередь позволяет делать: * документы в формате `.html`, `.pdf` (используя \LaTeX , мы почти не будем это обсуждать), `.docx` * презентации в формате `.html`, `.pdf` (используя \LaTeX пакет `beamer`) `.pptx`-презентации * набор связанных `.html` документов (полнцененный сайт или книга)

8.1.1 Установка

Как и все пакеты `rmarkdown` можно установить из CRAN

```
install.packages("rmarkdown")
```

8.1.2 Составляющие rmarkdown-документа

- `yaml`² шапка (факультативна)

¹<https://rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf>

²<https://en.wikipedia.org/wiki/YAML>

- обычный текст с markdown³ форматированием (расширенный при помощи Pandoc⁴)
- блоки кода (не обязательно на языке R), оформленные с двух сторон тройным бэктиком “ (у меня на клавиатуре этот знак на букве ё).

8.1.3 Пример rmarkdown-документа

Создайте файл .Rmd в какой-нибудь папке (в RStudio, это можно сделать File > New file > R Markdown). Скомпилировать файл можно командой:

```
rmarkdown::render(" _ .Rmd")
```

или кнопкой  . Вот пример кода:

```
---
output: html_document
---

## R

```{r}
summary(iris)
```

## R

```{r}
library(tidyverse)
iris %>%
 ggplot(aes(Sepal.Length, Sepal.Width)) +
 geom_point()
```

```

Результат⁵.



Создайте и скомпилируйте свой rmarkdown-документ с заголовком, текстом и кодом.

³<https://en.wikipedia.org/wiki/Markdown>

⁴<https://en.wikipedia.org/wiki/Pandoc>

⁵https://raw.githubusercontent.com/agricolamz/DS_for_DH/master/rmd_examples/ex_01.html

8.1.4 Markdown

Универсальный язык разметки, работает во многих современных он-лайн системах создания текста.

8.1.4.1 Заголовки

```
##          2
###         4
```

8.1.4.2 Форматирование

```
-      -      *      *
--     --     **     **
~~     ~~
```

italic или *другой italic*

жирный или **другой жирный**

зачеркивание

8.1.4.3 Списки

```
*
```

```
*
```

```
*
```

```
1.
  1.
2.
```

```
+
-
```

```

  · кролик
  · заяц
    – заяц серый
```

```

1. машины
  1. автобус
2. самолеты
```

```

  · можно еще ставить плюс
  · и минус
```

8.1.4.4 Ссылки и картинки

[1] (https://agricolamz.github.io/2018_ANDAN_course_winter/2_ex.html)