Визуализация и анализ географических данных на языке R

Тимофей Самсонов

2017-10-08

Contents

Введение

Добро пожаловать в курс *“Визуализация и анализ географических данных на языке R”*! В данном курсе мы освоим азы программирования на языке **R**, а затем научимся использовать его для решения географических задач. Никаких предварительных знаний и навыков программирования не требуется.

Программное обеспечение

Для успешного прохождения курса на вашем компьютере должно быть установлено следующее программное обеспечение:

- Язык R
- Среда разработки RStudio

Выбирайте инсталлятор, соответствующий вашей операционной системе. Обратите внимание на то, что **RStudio** не будет работать, пока вы не установите базовые библиотеки языка **R**. Поэтому обе вышеуказанные компоненты ПО обязательны для установки.

Установка и подключение пакетов

Существует множество дополнительных пакетов **R** (вы тоже можете написать свой) практически на все случаи жизни. Как и дистрибутив **R**, они доступны через CRAN (Comprehensive R Archive Network). Одним из таких пакетов является, например, пакет `openxlsx`, позволяющий читать и записывать файлы в форматах **Microsoft Excel**.

Существует два способа установки пакетов в **RStudio**.

Во-первых, вы можете сделать это в графическом интерфесе, нажав кнопку *Install* на панели *Packages* (по умолчанию эта панель расположена в нижней правой четверти окна программы). В появившемся окне введите название пакета и нажмите *Install*:

Во-вторых, вы можете вызвать *из консоли* команду `install.packages()`, передав ей в качестве параметра название пакета, заключенное в кавычки:

```
install.packages("openxlsx")
```

> **Никогда не включайте команду `install.packages()` в тело скрипта.** Это приведет к тому, что каждый раз при запуске программы среда **RStudio** будет пытаться заново установить пакет, который уже установлен. Запускайте эту функцию *только из консоли*.
>
> **Если по каким-то причинам вы не можете установить пакет в стандартную системную директорию RStudio** (например, из-за политик безопасности, запрещающих запись в каталог *Program Files* на ОС **Windows**), то необходимо создать директорию вручную в другом месте (куда вы имеете полный доступ) и указать ее адрес в параметре `lib` функции `install.packages()`. Например: `install.packages("xlsx", lib = "C:/Rlib/")`

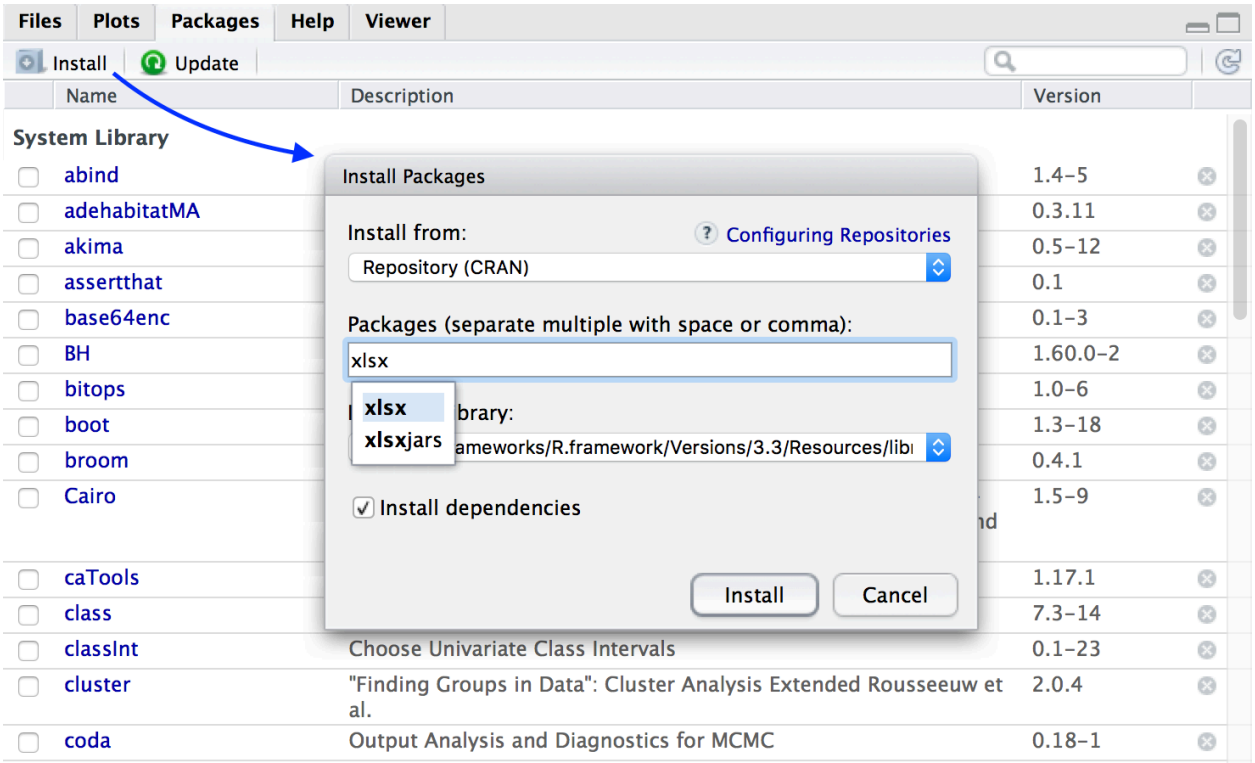


Figure 1: *Установка пакета*

Подключение пакета осуществляется с помощью функции `library()`, при этом название пакета можно в кавычки не заключать:

```
library(openxlsx)
```

> **Если пакет установлен не в стандартный каталог**, а в другое место — например, в каталог `:/Rlib/` (см. выше) — то при вызове функции `library()` необходимо указать местоположение пакета в дополнительном параметре `lib.loc`: `library(xlsx, lib.loc = "C:/Rlib")`

Выполнение программного кода

Существует несколько способов выполнения исходного кода:

- **Выполнить одну строку**: поставить курсор в любую строку и нажать над редактором кода кнопку *Run* или сочетание клавиш `Ctrl+Enter` (`Cmd+Enter` для OS X).
- **Выполнить несколько строк**: выделить необходимые строки и нажать над редактором кода кнопку *Run* или сочетание клавиш `Ctrl+Enter` (`Cmd+Enter` для OS X).
- **Выполнить весь код** можно сразу тремя способами:
  - Выделить весь текст и нажать над редактором кода кнопку *Run* или сочетание клавиш `Ctrl+Enter` (`Cmd+Enter` для OS X)
  - Нажать клавиатурное сочетание `Ctrl+Alt+Enter` (`Cmd+Alt+Enter` для OS X)
  - Нажать в правом верхнем углу редактора кода кнопку *Source*

> **Команды *Source* и `Ctrl+Alt+Enter` могут не сработать**, если у вас не установлена рабочая директория, или если в пути к рабочей директории содержатся кириллические символы (не актуально для Windows 10+ и OS X, которые являются системами, основанными на кодировке Unicode).

Существует также ряд дополнительных опций выполнения кода, которые вы можете найти в меню *Code* > *Run Region*

> **Выполняя код построчно, делайте это последовательно, начиная с первой строки программы.** Одна из самых распространенных ошибок новичков заключается в попытке выполнить некую строку, не выполнив *предыдущий код*. Нет никаких гарантий, что что-то получится, если открыть файл, поставить курсор в произвольную строку посередине программы и попытаться выполнить ее. Возможно, вам и повезет — если эта строка никак не зависит от предыдущего кода. Однако в реальных программах такие строки составляют лишь небольшую долю от общего объема. Как правило, в них происходит инициализация новых переменных стартовыми значениями.

Установка рабочей директории

Вы можете открывать и сохранять любые поддерживаемые файлы в **R**, указывая полный системный путь к файлу. Например, так может выглядеть открытие и сохранение таблицы в формате *CSV* на компьютере *Mac*:

```
d <- read.csv("/Volumes/Data/GitHub/r-geo-course/data/oxr_vod.csv")
write.csv(d, "/Volumes/Data/GitHub/r-geo-course/data/oxr_vod_copy.csv")
```

Однако, если вам требуется открыть или сохранить несколько файлов (и не только данных, но и графиков, карт и т.п.), программа будет выглядеть громоздко. К тому же, прописывать каждый раз полный путь достаточно утомительно и неприятно (даже путем копирования и вставки), а главное — может привести к ошибкам.

Чтобы облегчить работу с файлами, в **R** существует понятие домашней директории. Домашняя директория задается для текущей сессии **R** с помощью функции `setwd()`. После установки домашней директории **R** будет полагать, что все открываемые и сохраняемые файлы должны находиться в ней:

```
setwd("/Volumes/Data/GitHub/r-geo-course/data")

read.csv("oxr_vod.csv")
write.csv(d, "oxr_vod_copy.csv")
```

Как видно, мы добавили дополнительную строчку кода, но сэкономили на длине двух других строк. При увеличении количества обращений к файлам польза домашней директории будет возрастать. При этом вы можете открывать и сохранять файлы в поддиректориях, наддиректориях и соседних директориях, используя синтаксис, стандартный для большинства операционных систем:

```
#                    data
write.csv(d, "data/oxr_vod_copy.csv")

#
write.csv(d, "../oxr_vod_copy.csv")

#                   data,
write.csv(d, "../data/oxr_vod_copy.csv")
```

> **Если вы перенесли код и данные с другого компьютера** (возможно, вы получили их от своего коллеги или скачали с репозитория данного пособия), необходимо заменить путь, указанный в функции `setwd()` на путь к каталогу, в который вы положили данные.
>
> **Рабочая директория и местоположение скрипта могут не совпадать**. Вы можете хранить их в разных местах. Однако рекомендуется держать их вместе, что облегчит передачу вашей программы вместе с данными другим пользователям.

К сожалению, не существует надежного программного способа сказать среде выполнения **R**, что в качестве домашей директории следует использовать директорию в которой лежит сам скрипт (что, вообще говоря, было бы крайне удобно). Возможно, в будущем разработчики языка добавят такую полезную функцию. Однако, если для работы с **R** вы пользуетесь средой **RStudio**, задача может быть решена путем использования проектов. Подробнее читайте здесь.

Диагностические функции

В **R** существует ряд диагностических функций, которые позволяют узнавать информацию об объектах, переменных, а также текущих параметрах среды, оказывающих влияние на результаты выполнения программы. Эти функции полезны, когда необходимо понять, какого типа, размера и содержания данные хранятся в той или иной переменной. Нижеприведенный список функций не являются исчерпывающим, но охватывает наиболее употребительные функции:

Функция	Назначение
`class()`	Класс (тип данных или структура данных) объекта
`str()`	Компактное представление внутренней структуры объекта.
`names()`	Названия элементов объекта
`colnames()`	Названия колонок фрейма данных или матрицы
`rownames()`	Названия строк фрейма данных или матрицы
`mode()`	Режим хранения объекта.
`length()`	Размер (длина) объекта.
`dim()`	Измерение объекта.
`sessionInfo()`	Информация о текущей сессии R и подключенных пакетах.
`options()`	Получение и установка параметров среды.
`getwd()`	Текущая рабочая директория

Получение справки

Любая функция **R** содержит документированное описание ее параметров и правил использования. Справку можно получить несколькими способами:

- Найти интересующую вас функцию вручную на вкладке **Packages**, выбрав нужный пакет
- Воспользоваться строкой поиска на вкладке **Help**
- Ввести знак вопроса и название функции в консоли (будет искать только среди подключенных в настоящий момент пакетов):

```
library(openxlsx)

?read.xlsx #                    help(read.xlsx)
```

- Ввести двойной знак вопроса и название функции в консоли (будет искать по всем установленным пакетам, независимо от того, подключены ли они в настоящий момент):

```
??spsample
```

Во многих пакетах есть также подробная документация с примерами использования функций в виде руководств и так называемых *виньеток* (*vignettes*), которые представляют из себя расширенные руководства (статьи) по использованию пакета. С документацией пакета можно ознакомиться, щелкнув на его названии на вкладке *Packages* и перейдя по ссылке *User guides, package vignettes and other documentation*:

Комментарии

Комментарии — это фрагменты текста программы, начинающиеся с символа **#**. Комментарии не воспринимаются как исполняемый код и служат для документирования программы. При выполнении программы содержимое комментария в зависимости от настроек среды может выводиться или не выводиться в консоль, однако их содержание никак не влияет на результаты выполнения программы.

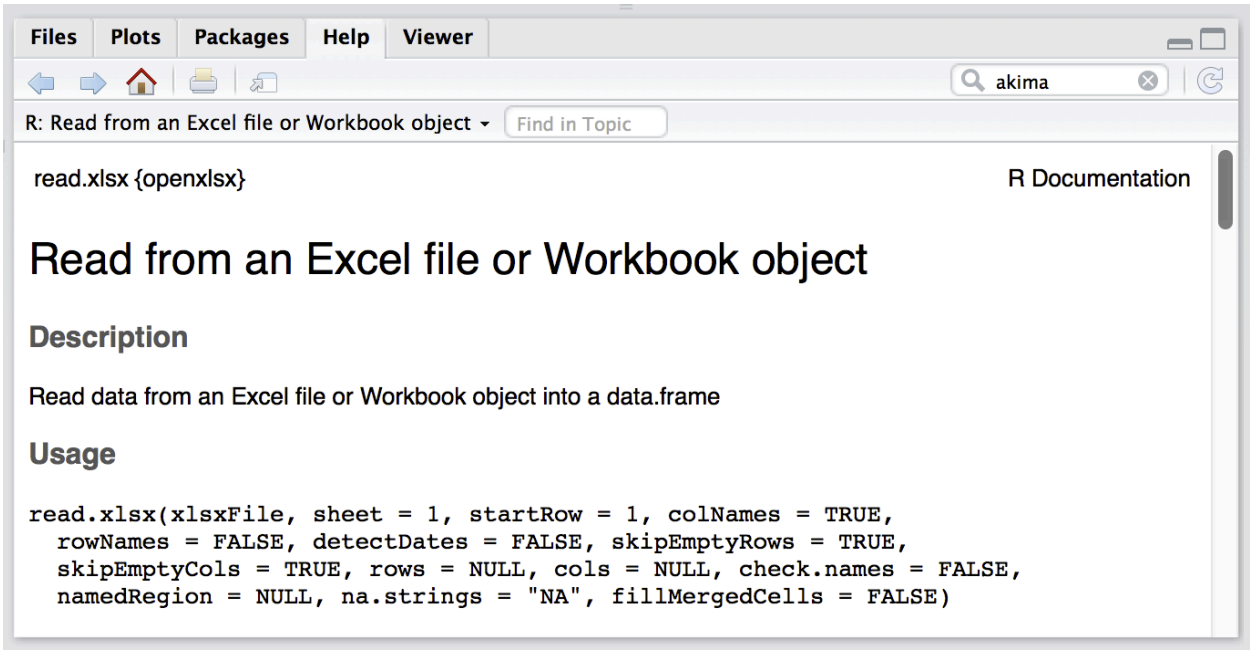


Figure 2: Справка по функции

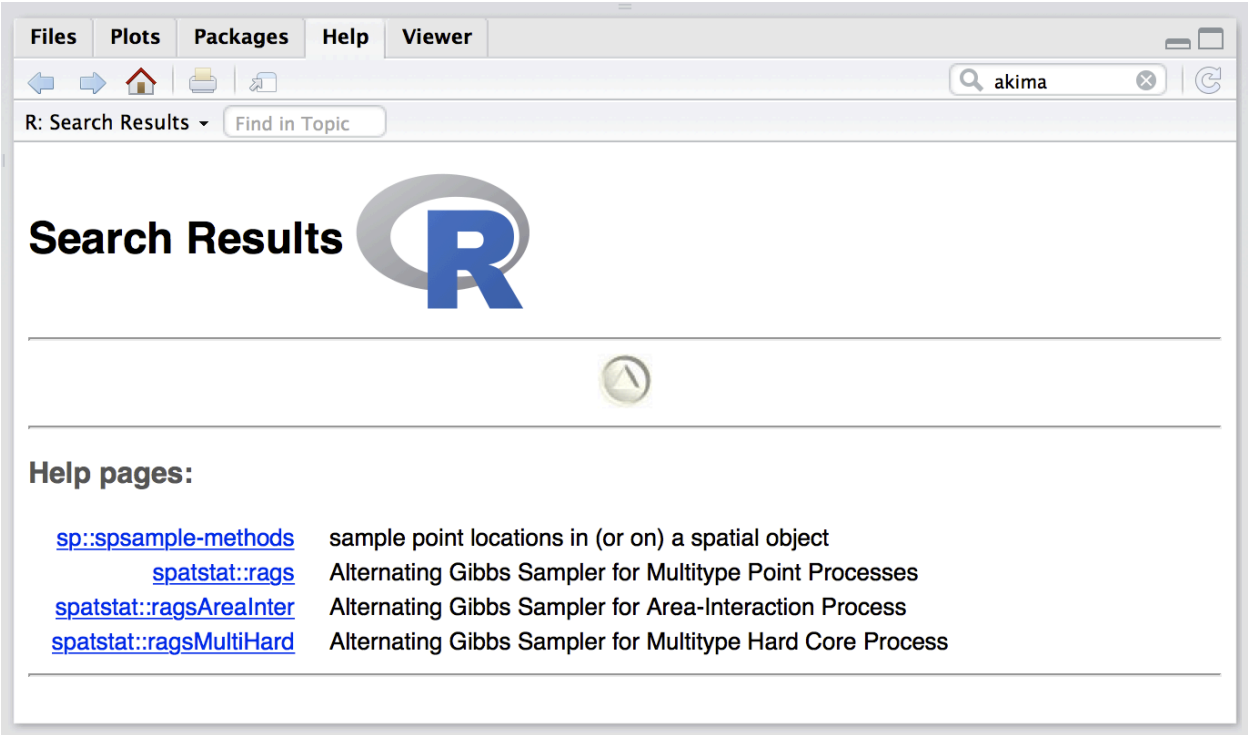


Figure 3: Поиск по функциям

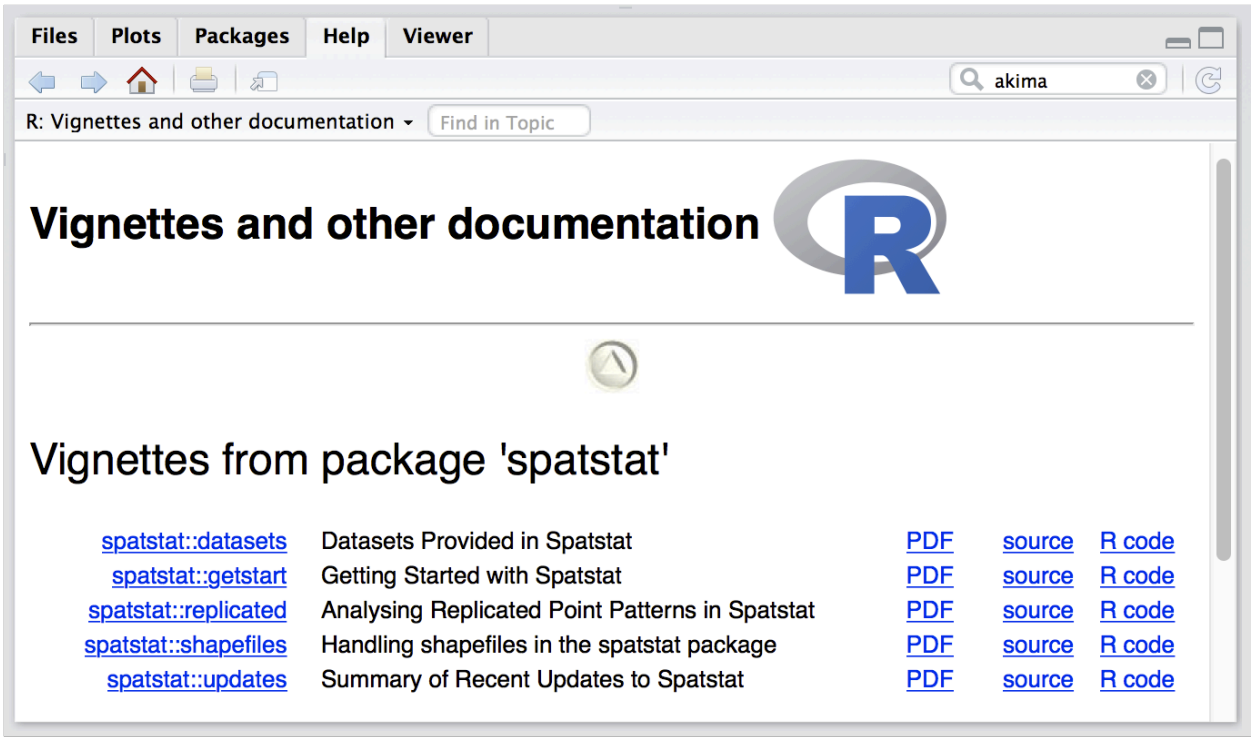


Figure 4: Документация пакета

> **Всегда пишите комментарии**, чтобы по прошествии времени можно было открыть файл и быстро восстановить в памяти логику программы и смысл отдельных операций. Комментарии особенно необходимы, если вашей программой будет пользоваться кто-то другой — без них будет трудно разобраться в программном коде.

Действие комментария продолжается от символа **#** до конца строки. Соответственно, вы можете поставить данный символ в самом начале строки и тогда комментарий будет занимать всю строку. Комментарий также можно расположить справа от исполняемого кода, и тогда он будет занимать только часть строки.

> **Прервать комментарий и написать справа от него исполняемый код нельзя**

Полностроч ные комментарии часто используются для выделения разделов в программе и написания объемных пояснений. Часто в них вводят имитации разделительных линий с помощью символов дефиса (−) или подчеркивания (_), а заголовки набирают прописными буквами. Короткие комментарии справа от фрагментов кода обычно служат пояснением конкретных простых операций. Подобная логика употребления комментариев не является обязательной. Вы можете оформлять их на свое усмотрение. Главное, чтобы они выполняли свою основную функцию — пояснять смысл выполняемых действий. Например:

```
#
# --------------------------
#                                  ,          ,     ,     ,          ,                    :

a <- 3 + 2 #
b <- 4 ^ 8 #
c <- b %% a #

#
d <- c / a
```

```
#
e <- d * b
```

Однако, усердствовать с комментированием каждой мелочи в программе, разумеется, не стоит. Со временем у вас выработается взвешенный подход к документированию программ и понимание того, какие ее фрагменты требуют пояснения, а какие самоочевидны.

> **Для быстрой вставки комментария, обозначающего новый раздел программы**, воспользуйтесь командой меню *Code* > *Insert Section* или клавиатурным сочетанием `Ctrl+Shift+R` (`Cmd+Shift+R` для OS X)

Стандарт оформления кода на R

Очень важно сразу же приучить себя грамотно, структурированно и красиво оформлять код на языке **R**. Это существенно облегчит чтение и понимание ваших программ не только вами, но и другими пользователями и разработчиками. Помимо вышеуказанных рекомендаций по написанию комментариев существует также определенное количество хорошо зарекомендовавших себя и широко используемых практик оформления кода. Эти практики есть в каждом языке программирования и их можно найти в литературе (и в Интернете) в виде негласных сводов правил (*style guides*)

Если вы не хотите быть белой вороной в мире **R**, вам будет полезно внимательно ознакомиться со стандартом оформления кода на R от компании Google, которая широко использует этот язык в своей работе.

> **Стандарт оформления кода иногда также называют *стилем программирования*.** Мы не будем использовать этот термин, поскольку под стилем программирования традиционно понимают фундаментальный подход (*парадигму*) к построению программ: процедурный, функциональный, логический, объектно-ориентированный стиль и некоторые другие.

К числу негласных правил оформления кода на **R** можно отнести следующие:

1. Последовательно используйте знак присвоения `<-` или `=` на протяжении всей программы. Если вы начали использовать `=` – применяйте его на протяжении всей программы, не используя `<-`.

   > **Традиционный подход предполагает использование `<-`**, однако все больше программистов использует знак `=` в своих программах, что делает **R** более похожим на другие языки программирования. Помните, что использование `=` официально не рекомендуется, поскольку существует много старого кода на R, который может ошибочно выполняться в сочетании с кодом, использующим `=`. Но вы, скорее всего, с такими проблемами не столкнетесь. Так что выбор за вами!

2. После запятой всегда ставьте пробел, перед запятой – нет:

```
#      :
a <- c(1, 2, 3, 4)
m <- matrix(a, 2, 2)

#       :
a <- c(1,2,3,4)
a <- c(1 ,2 ,3 ,4)
a <- c(1 , 2 , 3 , 4)
m <- matrix(a,2,2)
m <- matrix(a ,2 ,2)
m <- matrix(a , 2 , 2)
```

3. Отделяйте любые бинарные операторы (такие как `=`, `+`, `-`, `<-`, `*`) пробелами с двух сторон:

```
a <- sin(b + pi * 0.5) #
a<-sin(b+pi*0.5) #
```

4. Между названием функции и открывающей скобкой пробела быть не должно. То же самое касается обращения к элементам вектора, матрицы и т.п.:

```
#      :
sin(b)
a[2]

#       :
sin (b)
a [2]
```

5. В то же время, при вызове команд управления выполнением программы (условные операторы и циклы) перед и после скобок пробел **должен** стоять:

```
#      :
if (a > 0) {
  print(a)
}
i <- 0
while (i < a) {
  print(i)
  i <- i + 1
}

#       :
if(a > 0){
  print(a)
}

i <- 0
while(i < a){
  print(i)
  i <- i + 1
}
```

Зарезервированные слова

В **R** существует небольшое количество зарезервированных слов, которые нельзя использовать в качестве имен переменных, функций и проч. Список этих слов можно получить, набрав в консоли `?Reserved`. К ним относятся:

Слово	Назначение
`if`	Условный оператор ЕСЛИ
`else`	Условный оператор ИНАЧЕ
`repeat`	Цикл без внешнего условия
`while`	Цикл “пока верно условие, повторять”
`function`	Функция
`for`	Цикл “пройти по элементам последовательности”
`in`	Оператор вхождения в множество
`next`	Переход на новую итерацию цикла
`break`	Принудительный выход из цикла или условного оператора
`TRUE`	Логическое значение ИСТИНА
`FALSE`	Логическое значение ЛОЖЬ
`NULL`	Пустое значение

Слово	Назначение
`Inf`	Бесконечность
`NaN`	Нечисловое значние
`NA`	Отсутствующее значение
`NA_integer_`	Отсутствующее целое число
`NA_real_`	Отсутствующее число с плавающей точкой
`NA_complex_`	Отсутствующее комплексное число
`NA_character_`	Отсутствующая строка

Названия переменных

В качестве названий переменных **нельзя использовать заразервированные слова**, а также не рекомендуется использовать названия общеупотребительных (диагностических) функций и констант. Также не следует давать переменным названия, совпадающие с широко распространенными функциями – например, котороткими функциями из пакета `base`, такими как `t()`, `()` и т.д., так как это может привести к путанице в программе и даже ошибкам выполнения кода. Каждый раз, создавая переменную, спрашивайте себя, не совпадает ли ее название с названием одной из используемых вами функций.

Названия специальных символов

В **R**, как и во многих других языках программирования испльзуются различные специальные символы. Их смысл и значение мы узнаем по ходу изучения языка, а пока что выучите их названия, чтобы грамотно употреблять в своей речи

Символ	Название
`$`	доллар
`#`	шарп
`&`	амперсанд (решетка)
`/`	прямой слэш
`\`	обратный слэш
`|`	пайп (вертикальная черта)
`^`	циркумфлекс (крышечка)
`@`	эт (собачка)
`~`	тильда
`''`	одинарные кавычки
`""`	двойные кавычки
`` `` ```	обратные кавычки

Ссылка на пособие

Если этот курс лекций оказался полезным для вас, и вы хотите процитировать его с списке литературы вашей работы, то ссылку можно оформить по следующей форме:

Самсонов Т.Е. **Визуализация и анализ географических данных на языке R.** М.: Географический факультет МГУ, 2017. DOI: 10.5281/zenc

Chapter 1

Типы данных, ввод и вывод

Программный код главы

Тип данных — это класс данных, характеризуемый членами класса и операциями, которые могут быть к ним применены[1]. С помощью типов данных мы можем представлять привычные нам сущности, такие как числа, строки и т.д. В языке R существует 5 базовых типов данных:

Название	Тип данных
`complex`	комплексные числа
`character`	строки
`integer`	целые числа
`logical`	логические (булевы)
`numeric`	числа с плавающей точкой

Помимо этого есть тип `Date`, который позволяет работать с датами. Рассмотрим использование каждого из перечисленных типов.

1.1 Числа

Числа — основной тип данных в R. К ним относятся *числа с плавающей точкой* и *целые числа*. В терминологии R такие данные называются *интервальными*, поскольку к ним применимо понятие интервала на числовой прямой. Целые числа относятся к *дискретным интервальным*, а числа с плавающей точкой — к *непрерывным интервальным*. Числа можно складывать, вычитать и умножать:

```
2 + 3
## [1] 5
2 - 3
## [1] -1
2 * 3
## [1] 6
```

Разделителем целой и дробной части является точка, а не запятая:

```
2.5 + 3.1
## [1] 5.6
```

[1] ISO/IEC/IEEE 24765-2010 Systems and software engineering — Vocabulary

Существует также специальный оператор для возведения в степень. Для этого вы можете использовать или двойной знак умножения (**) или *циркумфлекс* (^), который в обиходе называют просто “крышечкой”:

```
2 ^ 3
## [1] 8
2 ** 3
## [1] 8
```

Результат деления по умолчанию имеет тип с плавающей точкой:

```
5 / 3
## [1] 1.666667
5 / 2.5
## [1] 2
```

Если вы хотите чтобы деление производилось целочисленным образом (без дробной части) необходимо использовать оператор %/%:

```
5 %/% 3
## [1] 1
```

Остаток от деления можно получить с помощью оператора %%:

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

Вышеприведенные арифметические операции являются бинарными, то есть требуют наличия двух чисел. Числа называются “операндами”. Отделять операнды от оператора пробелом или нет — дело вкуса. Я предпочитаю отделять, так как это повышает читаемость кода. Следующие два выражения эквивалентны. Однако сравните простоту их восприятия:

```
5%/%3
## [1] 1
```

```
5 %/% 3
## [1] 1
```

Как правило, в настоящих программах числа в явном виде встречаются лишь иногда. Вместо этого для их обозначения используют переменные. В вышеприведенных выражениях мы неоднократно использовали число 3. Теперь представьте, что вы хотите проверить, каковы будут результаты, если вместо 3 использовать 4. Вам придется заменить все тройки на четверки. Если их много, то это будет утомительная работа, и вы наверняка что-то пропустите. Конечно, можно использовать поиск с автозаменой, но что если тройки надо заменить не везде? Одно и то же число может выполнять разные функции в разных выражениях. Чтобы избежать подобных проблем, в программе вводят переменные и присваивают им значения. Оператор присваивания значения выглядит как <-

```
a <- 5
b <- 3
```

Чтобы вывести значение переменной на экран, достаточно просто ввести его:

```
a
## [1] 5
b
## [1] 3
```

Мы можем выполнить над переменными все те же операции что и над константами:

```
a + b
## [1] 8
a - b
## [1] 2
```

```
a / b
## [1] 1.666667
a %/% b
## [1] 1
a %% b
## [1] 2
```

Легко меняем значение второй переменной с 3 на 4 и выполняем код заново.

```
b <- 4
a + b
## [1] 9
a - b
## [1] 1
a / b
## [1] 1.25
a %/% b
## [1] 1
a %% b
## [1] 1
```

Нам пришлось изменить значение переменной только один раз в момент ее создания, все последующие операции остались неизменны, но их результаты обновились!

Новую переменную можно создать на основе значений существующих переменных:

```
c <- b
d <- a+c
```

Посмотрим, что получилось:

```
c
## [1] 4
d
## [1] 9
```

Вы можете комбинировать переменные и заданные явным образом константы:

```
e <- d + 2.5
e
## [1] 11.5
```

Противоположное по знаку число получается добавлением унарного оператора − перед константой или переменной:

```
f <- -2
f
## [1] -2
f <- -e
f
## [1] -11.5
```

Операция взятия остатка от деления бывает полезной, например, когда мы хотим выяснить, является число четным или нет. Для этого достаточно взять остаток от деления на 2. Если число является четным, остаток будет равен нулю. В данном случае с равно 4, d равно 9:

```
c %% 2
## [1] 0
d %% 2
## [1] 1
```

1.1.1 Числовые функции

Прежде чем мы перейдем к рассмотрению прочих типов данных и структур данных нам необходимо познакомиться с функциями, поскольку они встречаются буквально на каждом шагу. Понятие функции идентично тому, к чему мы привыкли в математике. Например, функция может называться Z, и принимать 2 аргумента: x и y. В этом случае она записывается как Z(x,y). Чтобы получить значение функции, необходимо подставить некоторые значения вместо x и y в скобках. Нас даже может не интересовать, как фактически устроена функция внутри, но важно понимать, что именно она должна вычислять. С созданием функций мы познакомимся позднее.

Важнейшие примеры функций — математические. Это функции взятия корня sqrt(x), модуля abs(x), округления round(x, digits), натурального логарифма abs(x), тригонометрические функции sin(x), cos(x), tan(x), обратные к ним asin(y), acos(y), atan(y) и многие другие. Основные математические функции содержатся в пакете base, который по умолчанию доступен в среде R и не требует подключения.

В качестве аргумента функции можно использовать переменную, константу, а также выражения:

```
sqrt(a)
## [1] 2.236068
sin(a)
## [1] -0.9589243
tan(1.5)
## [1] 14.10142
abs(a + b - 2.5)
## [1] 6.5
```

Вы также можете легко вкладывать функции одна в одну, если результат вычисления одной функции нужно подставить в другую:

```
sin(sqrt(a))
## [1] 0.7867491
sqrt(sin(a) + 2)
## [1] 1.020331
```

Также как и с арифметическими выражениями, результат вычисления функции можно записать в переменную:

```
b <- sin(sqrt(a))
b
## [1] 0.7867491
```

Если переменной b ранее было присвоено другое значение, оно перезапишется. Вы также можете записать в переменную результат операции, выполненной над ней же. Например, если вы не уверены, что a — неотрицательное число, а вам это необходимо в дальнейших расчетах, вы можете применить к нему операцию взятия модуля:

```
b <- sin(a)
b
## [1] -0.9589243
b <- abs(b)
b
## [1] 0.9589243
```

1.2 Строки

Строки — также еще один важнейший тип данных. Строки состоят из символов. Чтобы создать строковую переменную, необходимо заключить текст строки в кавычки:

```
s <- "              ,             ( .     )"
s
## [1] "              ,             ( .     )"
```

Длину строки в символах можно узнать с помощью функции `nchar()`

```
nchar(s)
## [1] 56
```

Строки можно складывать так же как и числа. Эта операция называется *конкатенацией*. В результате конкатенации строки состыковываются друг с другом и получается одна строка. В отличие от чисел, конкатенация производится не оператором `+`, а специальной функцией `paste()`. Состыковываемые строки нужно перечислить через запятую, их число может быть произвольно

```
s1 <- "              ,"
s2 <- "           "
s3 <- "( .     )"
```

Посмотрим содержимое подстрок:

```
s1
## [1] "              ,"
s2
## [1] "           "
s3
## [1] "( .     )"
```

А теперь объединим их в одну:

```
s <- paste(s1, s2)
s
## [1] "              ,           "
s <- paste(s1, s2, s3)
s
## [1] "              ,             ( .     )"
```

Настоящая сила конкатенации проявляется когда вам необходимо объединить в одной строке некоторое текстовое описание (заранее известное) и значения переменных, которые у вас вычисляются в программе (заранее неизвестные). Предположим, вы нашли в программе что максимальная численность населения в Детройте пришлась на 1950 год и составила 1850 тыс. человек. Найденный год записан у вас в переменную `year`, а население в переменную `pop`. Вы их значения пока что не знаете, они вычислены по табличным данным в программе. Как вывести эту информацию на экран “человеческим”” образом? Для этого нужно использовать конкатенацию строк.

Условно запишем значения переменных, как будто мы их знаем

```
year <- 1950
pop <- 1850
```

```
s1 <- "                                  "
s2 <- "         "
s3 <- "  .   "
s <- paste(s1, year, s2, pop, s3)
s
## [1] "                                  1950           1850   .   "
```

Обратите внимание на то что мы конкатенировали строки с числами. Конвертация типов осуществилась автоматически. Помимо этого, функция сама вставила пробелы между строками.

1.3 Даты

Даты являются необходимыми при работе с временными данными. В географическом анализе подобные задачи возникают сплошь и рядом. Точность указания времени может быть самой различной. От года до долей секунды. Чаще всего используются даты, указанные с точностью до дня. Для создания даты используется функция `as.Date()`. В данном случае точка — это лишь часть названия функции, а не какой-то особый оператор. В качестве аргумента функции необходимо задать дату, записанную в виде строки. Запишем дату рождения автора (можете заменить ее на свою):

```
birth <- as.Date('1986/02/18')
## Warning in strptime(xx, f <- "%Y-%m-%d", tz = "GMT"): unknown timezone
## 'default/Europe/Moscow'
birth
## [1] "1986-02-18"
```

Сегодняшнюю дату вы можете узнать с помощью специальной функции `Sys.Date()`:

```
current <- Sys.Date()
current
## [1] "2017-10-08"
```

Даты также можно складывать и вычитать. В зависимости от дискретности данных, вы получите результат в часах, днях, годах и т.д. Например, узнать продолжительность жизни в днях можно так:

```
livedays <- current - birth
livedays
## Time difference of 11555 days
```

Вы также можете прибавить к текущей дате некоторое значение. Например, необходимо узнать, какая дата будет через 40 дней:

```
current + 40
## [1] "2017-11-17"
```

1.4 Логические

Логические переменные возникают там, где нужно проверить условие. Переменная логического типа может принимать значение `TRUE` (истина) или `FALSE` (ложь). Для их обозначения также возможны более компактные константы `T` и `F` соответственно.

Следующие операторы приводят к возникновению логических переменных:

- *РАВНО* (==) — проверка равенства операндов
- *НЕ РАВНО* (!=) — проверка неравенства операндов
- *МЕНЬШЕ* (<) — первый аргумент меньше второго
- *МЕНЬШЕ ИЛИ РАВНО* (<=) — первый аргумент меньше или равен второму
- *БОЛЬШЕ* (>) — первый аргумент больше второго
- *БОЛЬШЕ ИЛИ РАВНО* (>=) — первый аргумент больше или равен второму

Посмотрим, как они работают:

```
a <- 1
b <- 2
a == b
## [1] FALSE
a != b
## [1] TRUE
```

```
a > b
## [1] FALSE
a < b
## [1] TRUE
```

Если необходимо проверить несколько условий одновременно, их можно комбинировать с помощью логических операторов. Наиболее популярные среди них:

- *И* (&&) - проверка истинности обоих условий
- *ИЛИ* (||) - проверка истинности хотя бы одного из условий
- *НЕ* (!) - отрицание операнда (истина меняется на ложь, ложь на истину)

```
c<-3
(b>a) && (c>b)
## [1] TRUE
(a>b) && (c>b)
## [1] FALSE
(a>b) || (c>b)
## [1] TRUE
!(a>b)
## [1] TRUE
```

Более подробно работу с логическими переменными мы разберем далее при знакомстве с условным оператором `if`.

1.5 Определение типа данных

Определение типа данных осуществляется с помощью функции `class()` (см. раздел *Диагностические функции* во Введении)

```
class(1)
## [1] "numeric"
class(0.5)
## [1] "numeric"
class(1 + 2i)
## [1] "complex"
class("sample")
## [1] "character"
class(TRUE)
## [1] "logical"
class(as.Date('1986-02-18'))
## [1] "Date"
```

В вышеприведенном примере видно, что R по умолчанию “повышает” ранг целочисленных данных до более общего типа чисел с плавающей точкой, тем самым закладываясь на возможность точного деления без остатка. Если вы хотите, чтобы данные в явном виде интерпретировались как целочисленные, их нужно принудительно привести к этому типу. Операторы преобразования типов рассмотрены ниже.

1.6 Преобразование типов данных {conversion}

Преобразование типов данных осуществляется с помощью функций семейства `as(d, type)`, где `d` — это входная переменная, а `type` — название типа данных, к которому эти данные надо преобразовать (см. таблицу в начале главы). Несколько примеров:

```
k <- 1
print(k)
## [1] 1
class(k)
## [1] "numeric"

l <- as(k, "integer")
print(l)
## [1] 1
class(l)
## [1] "integer"

m <- as(l, "character")
print(m)
## [1] "1"
class(m)
## [1] "character"

n <- as(m, "numeric")
print(n)
## [1] 1
class(n)
## [1] "numeric"
```

Для функции `as()` существуют обертки (*wrappers*), которые позволяют записывать такие преобразования более компактно и выглядят как `as.<dataype>(d)`, где `datatype` — название типа данных:

```
k <- 1
l <- as.integer(k)
print(l)
## [1] 1
class(l)
## [1] "integer"

m <- as.character(l)
print(m)
## [1] "1"
class(m)
## [1] "character"

n <- as.numeric(m)
print(n)
## [1] 1
class(n)
## [1] "numeric"

d <- as.Date('1986-02-18')
print(d)
## [1] "1986-02-18"
class(d)
## [1] "Date"
```

Если преобразовать число с плавающей точкой до целого, то дробная часть будет отброшена:

```
as.integer(2.7)
## [1] 2
```

После преобразования типа данных, разумеется, к переменной будут применимы только те функции, которые определены для данного типа данных:

```
a <- 2.5
b <- as.character(a)
b + 2
## Error in b + 2:
nchar(b)
## [1] 3
```

1.7 Ввод и вывод данных в консоли

Для ввода данных через консоль можно воспользоваться функцией `readline()`, которая будет ожидать пользовательский ввод и нажатие клавиши Enter, после чего вернет введенные данные в виде строки. Предположим, пользователь вызывает эту функцию и вводит с клавиатуры 1024:

```
a <- readline()
```

Выведем результат на экран:

```
a
## [1] "1024"
```

> Функция `readline()` всегда возвращает строку, поэтому если вы ожидаете ввод числа, полученное значение необходимо явным образом преобразовать к числовому типу.

Для вывода данных в консоль можно воспользоваться тремя способами:

- Просто напечатать название переменной с новой строки (*не работает при запуске программы командой* `Source`)
- Вызвать функцию `print()`
- Вызвать функцию `cat()`

Первый способ мы уже регулярно использовали ранее в настоящей главе. Следует обратить внимание на то, что он хорош для отладки программы, но выглядит некрасиво в рабочих программах, поскольку просто печатая название переменной с новой строки вы как бы явно не говорите о том, что хотите вывести ее значение в консоль, а лишь подразумеваете это. Более того, если скрипт запускается командой `Source`, данный метод вывода перемнной просто не сработает, интерпретатор его проигнорирует.

Поэтому после отладки следует убрать из программы все лишние выводы в консоль, а оставшиеся (действительно нужные) оформить с помощью функций `print()` или `cat()`.

Функция `print()` работает точно так же, как и просто название переменной с новой строки, отличаясь лишь двумя особенностями:

- `print()` явным образом говорит о том, что вы хотите вывести в консоль некую информацию
- `print()` работает при любых методах запуска программы, в том числе методом `Source`.

Например:

```
a <- 1024
a
## [1] 1024
print(a)
## [1] 1024
```

```
b <- "Fourty winks in progress"
b
## [1] "Fourty winks in progress"
print(b)
## [1] "Fourty winks in progress"

print(paste("2       10    ", 2^10))
## [1] "2       10     1024"

print(paste("          - ", Sys.Date()))
## [1] "          -  2017-10-08"
```

Функция `cat()` отличается от `print()` следующими особенностями:

- `cat()` выводит значение переменной, и не печатает ее измерения и внешние атрибуты типа двойных кавычек вокруг строки. Это означает, что `cat()` можно использовать и для записи данных в файл (на практике этим мало кто пользуется, но знать такую возможность надо).
- `cat()` принимает множество аргументов и может осуществлять конкатенацию строк аналогично функции paste()
- `cat()` не возвращает никакого значений, в то время как `print()` возвращает значение, переданное ей в качестве аргумента.
- `cat()` можно использовать только для атомарных типов данных. Для классов (таких как Date) она будет выводит содержимое объекта, которое может не совпадать с тем, что пользователь ожидает вывести

Например:

```
cat(a)
## 1024
cat(b)
## Fourty winks in progress

cat("2       10    ", 2^10)
## 2       10     1024

cat("          -", Sys.Date())
##           - 17447
```

Можно видеть, что в последнем случае `cat()` напечатала отнюдь не дату в ее привычном представлении, а некое число, которое является внутренним представлением даты в типе данных `Date`. Такие типы данных являются классами объектов в R, и у них есть своя функция `print()`, которая и выдает содержимое объекта в виде, который ожидается пользователем. Поэтому пользоваться функцией `cat()` надо с некоторой осторожностью.

1.8 Контрольные вопросы и задачи

1.8.1 Вопросы

1. Какие типы данных поддерживаются в R? Каковы их англоязычные наименования?
2. Что такое переменная?
3. Какой оператор используется для записи значения в переменную?
4. С помощью какой функции можно узнать тип переменной?
5. С помощью какого семейства функций можно преобразовывать типы переменных?
6. Можно ли использовать ранее созданное имя переменной для хранения новых данных другого типа?
7. Можно ли записать в переменную результат выполнения выражения, в котором она сама же и участвует?
8. Какая функция позволяет прочитать пользовательский ввод с клавиатуры в консоли? Какой тип данных будет иметь возвращаемое значение?

9. Какую функцию можно использовать для вывода значения переменной в консоль? Чем отличается использование этой функции от случая, когда вы просто пишете название переменной в строке программы?
10. Какой символ является разделителем целой и дробной части при записи чисел с плавающей точкой?
11. Что такое операторы и операнды? Приведите примеры бинарных и унарных операторов.
12. Какое значение будет имет результат деления на ноль?
13. Какие функции выполняют операторы `%%`, `%/%`, `^`, `**`?
14. Как проверить, является ли число четным?
15. Как определить количество символов в строке?
16. Как называется операция состыковки нескольких строк и с помощью какой функции она выполняется? Как добиться того, чтобы при этом не добавлялись пробелы между строками?
17. С помощью какой функции можно создать дату из строки?
18. Как извлечь из даты год? Месяц? День?
19. Какая функция позволяет получить дату сегодняшнего дня?
20. Можно ли складывать даты и числа? Если да, то в каких единицах измерения будет выражен результат?
21. Какова краткая форма записи логических значений `TRUE` и `FALSE`?
22. Каким числам соответствуют логические значения `TRUE` и `FALSE`?
23. Сколько операндов должно быть верно, чтобы оператор логического И (`&&`) принял значение `TRUE`? Что можно сказать в этом отношении об операторе ИЛИ (`||`)?
24. Можно ли применять арифметические операции к логическим переменным? Что произойдет, если прибавить или вычесть из числа `a` значение `TRUE`? А если заменить `TRUE` на `FALSE`?

1.8.2 Задачи

1. Напишите выражение для вычисления длины отрезка по координатам его вершин, хранящимся в переменных `x1, y1, x2, y2`
2. Не используя оператор `!=` запишите условие неравенства чисел `a` и `b` с помощью других логических операторов.
3. Напишите программу, которая считывает из консоли введенную пользователем строку и выводит в консоль количество символов в этой строке. Вывод оформите следующим образом: `" ... "`, где вместо многоточия стоит вычисленная длина.
4. Напишите программу, которая определяет количество дней в феврале года, который хранится в переменной `y`. Протестируйте программу, меняя значение `y`.

Chapter 2

Структуры данных

Программный код главы

Структура данных — это программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных. Структуры данных также являются типами данных, но не простыми, а составными. Поэтому обычно, когда говорят “тип данных”, подразумевают именно простые типы данных, рассмотренные в предыдущей главе. В **R** общеупотребительны следующие структуры данных: векторы, матрицы, массивы, фреймы данных, списки и факторы. Последний тип данных не всегда относят к структурам, хотя это логично, поскольку факторы построены по принципу ассоциативных массивов.

2.1 Векторы

Вектор представляет собой упорядоченную последовательность объектов одного типа. Вектор может состоять *только* из чисел, *только* из строк, *только* из дат или *только* из логических значений и т.д. Числовой вектор легко представить себе в виде набора цифр, выстроенных в ряд и пронумерованных согласно порядку их расстановки.

Вектор является простейшей и одновременно базовой структурой данных в **R**. Понимание принципов работы с векторами необходимо для дальнейшего знакомства с более сложными структурами данных, такими как матрицы, массивы, фреймы данных, списки и факторы.

2.1.1 Создание вектора

Существует множество способов создания векторов. Среди них наиболее употребительны:

1. Явное перечисление элементов
2. Создание пустого вектора (“болванки”), состоящего из заданного числа элементов
3. Генерация последовательности значений

Для создания вектора путем **перечисления** элементов используется функция `c()`:

```
#           -
colors <- c("    ", "    ", "   ", "      ", "     ")
colors
## [1] "    "    "    "    "   "      "     " "     "
```

```
#           -                (                )
lengths <- c(28, 40, 45, 19, 38)
lengths
## [1] 28 40 45 19 38
```

```
#                  -                        (              )
opens <- c(FALSE, TRUE, TRUE, FALSE, FALSE)
opens
## [1] FALSE  TRUE  TRUE FALSE FALSE
```

> Внимание: не используйте латинскую букву ‘c’ в качестве названия переменной! Это приведет к конфликту названия встроенной функции c() и определенной вами переменной

Помимо этого, распространены сценарии, когда вам нужно создать вектор, но заполнять его значениями вы будете по ходу выполнения программы — скажем, при последовательной обработке строк таблицы. В этом случае вам известно только предполагаемое количество элементов вектора и их тип. Здесь лучше всего подойдет **создание пустого вектора**, которое выполняется функцией `vector()`. Функция принимает 2 параметра:

- `mode` отвечает за тип данных и может принимать значения равные `"logical"`, `"integer"`, `"numeric"` (или `"double"`), `"complex"`, `"character"` и `"raw"`
- `length` отвечает за количество элементов

Например:

```
#       5      ,
intvalues <- vector(mode = "integer", length = 5)
intvalues #
## [1] 0 0 0 0 0

#       10      ,                                 (    )
charvalues <- vector("character", 10)
charvalues #
##  [1] "" "" "" "" "" "" "" "" "" ""
```

Обратите внимание на то, что в первом случае подстановка параметров произведена в виде = , а во втором указаны только значения. В данном примере оба способа эквивалентны. Однако первый способ безопаснее и понятнее. Если вы указываете только значения параметров, нужно помнить, что интерпретатор будет подставлять их именно в том порядке, в котором они перечислены в описании функции.

> Описание функции можно посмотреть, набрав ее название в консоли ее название со знаком вопроса в качестве префикса. Например, для вышеуказанной функции надо набрать `?vector`

Наконец, третий распространенный способ создания векторов — это **генерация последовательности**. Чтобы сформировать вектор из натуральных чисел от `M` до `N`, можно воспользоваться специальной конструкцией: `M:N`:

```
index <- 1:5 #         c(1,2,3,4,5)
index
## [1] 1 2 3 4 5
index <- 2:4 #         c(2,3,4)
index
## [1] 2 3 4
```

Существует и более общий способ создания последовательности — функция `seq()`, которая позволяет генерировать вектора значений нужной длины и/или с нужным шагом:

```
seq(from = 1, by = 2, length.out = 10) # 10          ,
##  [1]  1  3  5  7  9 11 13 15 17 19
seq(from = 2, to = 20, by = 3) #   2   20      3 (          )
## [1]  2  5  8 11 14 17 20
seq(length.out = 10, to = 2, by = -2) #                    10                    ,     2
##  [1] 20 18 16 14 12 10  8  6  4  2
```

Как видно, параметры функции `seq()` можно комбинировать различными способами и указывать в произвольном порядке (при условии, что вы используете полную форму (=). Главное, чтобы их совокупность *однозначно*

описывала последовательность. Хотя, скажем, последний пример убывающей последовательности нельзя признать удачным с точки зрения наглядности.

Аналогичным образом можно создавать *последовательности дат*:

```
seq(from = as.Date('2016/09/01'), by = 1, length.out = 7) #                    2016/2017
## Warning in strptime(xx, f <- "%Y-%m-%d", tz = "GMT"): unknown timezone
## 'default/Europe/Moscow'
## [1] "2016-09-01" "2016-09-02" "2016-09-03" "2016-09-04" "2016-09-05"
## [6] "2016-09-06" "2016-09-07"

seq(from = Sys.Date(), by = 7, length.out = 5) #                ,
## [1] "2017-10-08" "2017-10-15" "2017-10-22" "2017-10-29" "2017-11-05"
```

2.1.2 Работа с элементами вектора

К отдельным **элементам вектора** можно обращаться по их индексам:

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

> ВНИМАНИЕ: элементы векторов и других структур данных в языке R индексируются от 1 до N, где N — это длина вектора. Это отличает R от широко распространенных Си-подобных языков программирования (C, C++, C#, Java) и Python, в которых индексы элементов начинаются с 0 и заканчиваются N-1. Например, первый элемент списка (аналог вектора в R) на языке Python извлекался бы как colors[0]. За этим нужно внимательно следить, особенно если вы программируете на нескольких языках.

Количество элементов (длину) вектора можно узнать с помощью функции `length()`:

```
length(colors)
## [1] 5
```

Последний элемент вектора можно извлечь, если мы знаем его длину:

```
n <- length(colors)
colors[n]
## [1] "     "
```

Последовательности удобно использовать для извлечения подвекторов. Предположим, нужно извлечь первые 4 элемента. Для этого запишем:

```
lengths[1:4]
## [1] 28 40 45 19
```

Индексирующий вектор можно создать заранее. Это удобно, если номера могут меняться в программе:

```
m <- 1
n <- 4
index <- m:n
lengths[index]
## [1] 28 40 45 19
```

Обратите внимание на то что по сути один вектор используется для извлечения элементов из другого вектора. Это означает, что мы можем использовать не только простые последовательности натуральных чисел, но и векторы из прозвольных индексов. Например:

```
index <- c(1, 3, 4) #          1, 3  4
lengths[index]
## [1] 28 45 19

index <- c(5, 1, 4, 2) #
lengths[index]
## [1] 38 28 19 40
```

2.1.3 Анализ и преобразования векторов

К числовым векторам можно применять множество функций. Прежде всего, нужно знать функции вычисления базовых параметров статистического ряда — минимум, максимум, среднее, медиана, дисперсия, размах вариации, среднеквадратическое отклонение, сумма:

```
min(lengths) #
## [1] 19
max(lengths) #
## [1] 45
range(lengths) #           =       -
## [1] 19 45
mean(lengths) #
## [1] 34
median(lengths) #
## [1] 38
var(lengths) #       (        -      , variation)
## [1] 108.5
sd(lengths) #                   (standard deviation)
## [1] 10.41633
sum(lengths) #
## [1] 170
```

Одной из мощнейших особенностей R является то что он не проводит различий между числами и векторами чисел. Поскольку R является матричным языком, каждое число представляется как вектор длиной 1 (или матрица 1×1). Это означает, что любая математическая функция, применимая к числу, будет применима и к вектору:

```
lengths * 1000 #
## [1] 28000 40000 45000 19000 38000
sqrt(lengths) #
## [1] 5.291503 6.324555 6.708204 4.358899 6.164414

stations <- c(20, 21, 22, 12, 24) #

dens <- stations / lengths #                     =   -       /
dens
## [1] 0.7142857 0.5250000 0.4888889 0.6315789 0.6315789
```

2.1.4 Поиск и сортировка элементов

К важнейшим преобразованиям векторов относится их **сортировка**:

```
lengths2 <- sort(lengths) #
lengths2 #
## [1] 19 28 38 40 45
```

```
lengths #
## [1] 28 40 45 19 38

lengths2 <- sort(lengths, decreasing = TRUE) #                    .              decreasing
lengths2 #
## [1] 45 40 38 28 19
lengths #
## [1] 28 40 45 19 38
```

Другая распространенная задача — это **поиск индекса** элемента по его значению. Например, вы хотите узнать, какая ветка Московского метро (среди рассматриваемых) является самой длинной. Вы, конечно, легко найдете ее длину с помощью функции `max(lengths)`. Однако это не поможет вам узнать ее название, поскольку оно находится в другом векторе, и его индекс в массиве неизвестен. Поскольку векторы упорядочены одинаково, нам достаточно узнать, под каким индексом в массиве `lengths` располагается максимальный элемент, и затем извлечь цвет линии метро под тем же самым индексом. Дл поиска индекса элемента используется функция `match()`:

```
l <- max(lengths) #
idx <- match(l, lengths) #              ,      l,       lengths
color <- colors[idx] #
color
## [1] "   "
```

Здесь непохо бы лишний раз потренироваться в конкатенации строк, чтобы вывести результат красиво!

```
s <- paste(color, "              -        .                ", l, " ")
s
## [1] "                  -        .                45  "
```

Ну и напоследок пример “матрешки”” из функций — как найти название самой плотной линии одним выражением:

```
colors[match(max(dens),dens)]
## [1] "    "
```

2.2 Матрицы

Матрица — это обобщение понятия вектора на 2 измерения. С точки зрения анализа данных матрицы ближе к реальным данным, посколько каждая матрица по сути представляет собой таблицу со столбцами и строками. Однако матрица, как и вектор, может содержать только элементы одного типа (числовые, строковые, логические и т.д.). Позже мы познакомимся с фреймами данных, которые не обладают подобным ограничением. А пока рассмотрим, как работать с двумерными данными на примере матриц.

Матрица, как правило, создается с помощью функции `matrix`, которая принимает 3 обязательных аргумента: вектор исходных значений, количество строк и количество столбцов:

```
v <- 1:12  #                        1   12
m <- matrix(v, nrow = 3, ncol = 4)
m
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

По умолчанию матрица заполняется данными вектора по столбцам, что можно видеть в выводе программы. Если вы хотите заполнить ее по строкам, необходимо указать параметр `byrow = TRUE`:

```
m <- matrix(v, nrow = 3, ncol = 4, byrow = TRUE)
m
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

Доступ к элементам матрицы осуществляется аналогично вектору, за исключением того что нужно указать положение ячейки в строке и столбце:

```
m[2,4]  # 2    , 4
## [1] 8
m[3,1]  # 3    , 1
## [1] 9
```

Помимо этого, из матрицы можно легко извлечь одну строку или один столбец. Для этого достаточно указать только номер строки или столбца, а номер второго измерения пропустить до или после запятой. Результат является вектором:

```
m[2,]  # 2
## [1] 5 6 7 8
m[,3]  # 3 c
## [1]  3  7 11
```

К матрицам можно применять операции, аналогичные операциям над векторами:

```
log(m)  #
##          [,1]      [,2]     [,3]     [,4]
## [1,] 0.000000 0.6931472 1.098612 1.386294
## [2,] 1.609438 1.7917595 1.945910 2.079442
## [3,] 2.197225 2.3025851 2.397895 2.484907
sum(m)  #
## [1] 78
median(m) #
## [1] 6.5
```

А вот сортировка матрицы приведет к тому что будет возвращен обычный вектор:

```
sort(m)
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

К матрицам также применимы специальные функции, известные из линейной алгебры, такие как транспонирование и вычисление определителя:

```
t(m)  #
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
m2<-matrix(-3:3,nrow = 3, ncol = 3)
## Warning in matrix(-3:3, nrow = 3, ncol = 3):         [7]
##                 [3]
m2
##      [,1] [,2] [,3]
## [1,]   -3    0    3
## [2,]   -2    1   -3
## [3,]   -1    2   -2
det(m2) #
```

```
## [1] -21
det(m)  #    !
## Error in determinant.matrix(x, logarithm = TRUE, ...): 'x'
```

Матрицы также можно перемножать с помощью специального оператора `%*%`. При этом, как мы помним, число столбцов в первой матрице должно равняться числу строк во второй:

```
m2 %*% m
##      [,1] [,2] [,3] [,4]
## [1,]   24   24   24   24
## [2,]  -24  -28  -32  -36
## [3,]   -9  -10  -11  -12
m %*% m2  #    !
## Error in m %*% m2:
```

Функция `match()`, которую мы использовали для поиска элементов в векторе, не работает для матриц. Вместо этого необходимо использовать функцию `which()`. Если мы хотим найти в матрице m позицию числа 8, то вызов функции будет выглядеть так:

```
which(m == 8, arr.ind = TRUE)
##      row col
## [1,]   2   4
```

В данном случае видно, что результат возвращен в виде матрицы 1×2. Обратите внимание на то, что колонки матрицы имеют названия. Попробуем использовать найденные индексы, чтобы извлечь искомый элемент:

```
indexes <- which(m == 8, arr.ind = TRUE)
row <- indexes[1,1]
col <- indexes[1,2]
m[row,col]
## [1] 8
```

Ура! Найденный элемент действительно равен 8.

Еще один полезный способ создания матрицы — это собрать ее из нескольких векторов, объединив их по строкам. Для этого можно использовать функции `cbind()` и `rbind()`. На предыдущем занятии мы создали векторы с длиной и количеством станций на разных ветках метро. Можно объединить их в одну матрицу:

```
lengths <- c(28, 40, 45, 19, 38)
stations <- c(20, 21, 22, 12, 24)
cbind(lengths, stations)  #
##      lengths stations
## [1,]      28       20
## [2,]      40       21
## [3,]      45       22
## [4,]      19       12
## [5,]      38       24
rbind(lengths, stations)  #
##          [,1] [,2] [,3] [,4] [,5]
## lengths    28   40   45   19   38
## stations   20   21   22   12   24
```

Строки и столбцы матрицы можно использовать как векторы при выполнении арифметических операций:

```
mm <- cbind(lengths, stations)
mm[,2]/mm[,1]  #              1
## [1] 0.7142857 0.5250000 0.4888889 0.6315789 0.6315789
```

Результат можно присоединить к уже созданной матрице:

```
dens <- mm[,2]/mm[,1]
mm<-cbind(mm, dens)
mm
##      lengths stations      dens
## [1,]      28       20 0.7142857
## [2,]      40       21 0.5250000
## [3,]      45       22 0.4888889
## [4,]      19       12 0.6315789
## [5,]      38       24 0.6315789
```

Содержимое матрицы можно просмотреть в более привычном табличном виде для этого откройте вкладку *Environment* и щелкните на строку с матрицей в разделе *Data*

Матрицы, однако, не дотягивают по функциональности до представления таблиц, и, в общем-то, не предназначены для объединения разнородных данных в один набор (как мы это сделали). Если вы присоедините к матрице столбец с названиями веток метро, система не выдаст сообщение об ошибке, но преобразует матрицу в текстовую, так как текстовый тип данных способен представить любой другой тип данных:

```
colors <- c("    ", "    ", "   ", "      ", "     ")
mm2<-cbind(mm,colors)
mm2  #               ,
##      lengths stations dens                colors
## [1,] "28"    "20"     "0.714285714285714" "    "
## [2,] "40"    "21"     "0.525"             "    "
## [3,] "45"    "22"     "0.488888888888889" "   "
## [4,] "19"    "12"     "0.631578947368421" "      "
## [5,] "38"    "24"     "0.631578947368421" "     "
```

При попытке выполнить арифметическое выражение над прежде числовыми полями, вы получите сообщение об ошибке:

```
mm2[,2]/mm2[,1]
## Error in mm2[, 2]/mm2[, 1]:
```

2.3 Массивы

Массивы (arrays) — это многомерные структуры данных, с колчеством измерений 3 и более. Трехмерный массив представляет собой куб однородных данных. Массивы возникают тогда, например, когда имеются временные данные, зафиксированные в неких географических локациях. При этом 2 измерения отвечают за местоположение, а третье измерение — за временной срез.

2.4 Фреймы данных

Фреймы данных — это обобщение понятия матрицы на данные смешанных типов. Фреймы данных - наиболее распространенный формат представления табличных данных. Для краткости мы иногда будем называть их просто фреймами.

> Мы специально не используем для перевода слова `data.frame` термин ‘таблица’, поскольку таблица — это достаточно общая категория, которая описывает концептуальный способ упорядочивания данных. В том же языке R для представления таблиц могут быть использованы как минимум две структуры данных: фрейм данных (data.frame) и тиббл (tibble), доступный в соответствующем пакете. Мы не будем

использовать тибблы в настоящем курсе, но после его освоения вы вполне сможете ознакомиться с ними самостоятельною

Для создания фреймов данных используется функция `data.frame()`:

```
t<-data.frame(colors,lengths,stations)
t  #        ,
##       colors lengths stations
## 1              28       20
## 2              40       21
## 3               45       22
## 4             19       12
## 5              38       24
```

К фреймам также можно пристыковывать новые столбцы:

```
t<-cbind(t, dens)
t
##       colors lengths stations      dens
## 1              28       20 0.7142857
## 2              40       21 0.5250000
## 3               45       22 0.4888889
## 4             19       12 0.6315789
## 5              38       24 0.6315789
```

Когда фрейм данных формируется посредством функции `data.frame()` и `cbind()`, названия столбцов берутся из названий векторов. Обратите внимание на то, что листинге выше столбцы имеют заголовки, а строки — номера.

Как и прежде, к столбцам и строкам можно обращаться по индексам:

```
t[2,2]
## [1] 40
t[,3]
## [1] 20 21 22 12 24
t[4,]
##       colors lengths stations      dens
## 4             19       12 0.6315789
```

Вы можете обращаться к отдельным столбцам фрейма данных по их названию, используя оператор $ (доллар):

```
t$lengths
## [1] 28 40 45 19 38
t$stations
## [1] 20 21 22 12 24
```

Так же как и ранее, можно выполнять различные операции над столбцами:

```
max(t$stations)
## [1] 24
t$lengths / t$stations
## [1] 1.400000 1.904762 2.045455 1.583333 1.583333
```

Названия столбцов можно получить с помощью функции `colnames()`

```
colnames(t)
## [1] "colors"   "lengths"  "stations" "dens"
```

Чтобы присоединить строку, сначала можно создать фрейм данных из одной строки:

```
row<-data.frame("      ", 40.5, 22, 22/45)
```

Далее нужно убедиться, что столбцы в этом мини-фрейме называются также как и в той, куда мы хотим присоединить строку. Для этого нужно перезаписать результат, возвращаемый функцией `colnames()`:

```
colnames(row) <- colnames(t)
```

Обратите внимание на синтаксис вышеприведенного выражения. Когда функция возвращает результат, она обнаруживает свойство самого объекта, и мы можем его перезаписать. После того как столбцы приведены в соответствие, можно присоединить новую строку:

```
t<-rbind(t,row)
```

Поскольку названия столбцов хранятся как вектор из строк, мы можем их переделать:

```
colnames(t)<-c("  ","   ","    ","     ")
colnames(t)
## [1] "  "      "   "     "    "   "     "
```

Обратимся по новому названию столбца:

```
t$
## [1] 28.0 40.0 45.0 19.0 38.0 40.5
t
##
## 1          28.0      20 0.7142857
## 2          40.0      21 0.5250000
## 3           45.0      22 0.4888889
## 4         19.0      12 0.6315789
## 5         38.0      24 0.6315789
## 6         40.5      22 0.4888889
```

2.5 Списки

Список — это наиболее общий тип контейнера в R. Список отличается от вектора тем, что он может содержать набор объектов произвольного типа. В качестве элементов списка могут быть числа, строки, вектора, матрицы, фреймы данных — и все это в одном контейнере. Списки используются чтобы комбинировать разрозненную информацию. Результатом выполнения многих функций является список.

Например, можно создать список из текстового описания фрейма данных, самого фрейма данных и обобщающей статистики по нему:

```
d <- "                      6              "
s <- summary(t)  # summary()                    ,
```

Сооружаем список из трех элементов:

```
metrolist <- list(d,t,s)
metrolist
## [[1]]
## [1] "                      6              "
##
## [[2]]
##
## 1          28.0      20 0.7142857
## 2          40.0      21 0.5250000
## 3           45.0      22 0.4888889
```

```
## 4         19.0      12 0.6315789
## 5         38.0      24 0.6315789
## 6         40.5      22 0.4888889
##
## [[3]]
##
##         :1   Min.   :19.00   Min.   :12.00   Min.   :0.4889
##        :1   1st Qu.:30.50   1st Qu.:20.25   1st Qu.:0.4979
##         :1   Median :39.00   Median :21.50   Median :0.5783
##        :1   Mean   :35.08   Mean   :20.17   Mean   :0.5800
##          :1   3rd Qu.:40.38   3rd Qu.:22.00   3rd Qu.:0.6316
##        :1   Max.   :45.00   Max.   :24.00   Max.   :0.7143
```

Можно дать элементам списка осмысленные названия при создании:

```
metrolist <- list(desc = d, table = t, summary = s)
metrolist
## $desc
## [1] "                      6              "
##
## $table
##
## 1          28.0      20 0.7142857
## 2          40.0      21 0.5250000
## 3          45.0      22 0.4888889
## 4         19.0      12 0.6315789
## 5         38.0      24 0.6315789
## 6         40.5      22 0.4888889
##
## $summary
##
##         :1   Min.   :19.00   Min.   :12.00   Min.   :0.4889
##        :1   1st Qu.:30.50   1st Qu.:20.25   1st Qu.:0.4979
##         :1   Median :39.00   Median :21.50   Median :0.5783
##        :1   Mean   :35.08   Mean   :20.17   Mean   :0.5800
##          :1   3rd Qu.:40.38   3rd Qu.:22.00   3rd Qu.:0.6316
##        :1   Max.   :45.00   Max.   :24.00   Max.   :0.7143
```

Теперь можно обратиться к элементу списка по его названию:

```
metrolist$summary
##
##         :1   Min.   :19.00   Min.   :12.00   Min.   :0.4889
##        :1   1st Qu.:30.50   1st Qu.:20.25   1st Qu.:0.4979
##         :1   Median :39.00   Median :21.50   Median :0.5783
##        :1   Mean   :35.08   Mean   :20.17   Mean   :0.5800
##          :1   3rd Qu.:40.38   3rd Qu.:22.00   3rd Qu.:0.6316
##        :1   Max.   :45.00   Max.   :24.00   Max.   :0.7143
```

Поскольку `summary` сама является фреймом данных, из нее можно извлечь столбец:

```
metrolist$summary[,3]
##
## "Min.   :12.00  " "1st Qu.:20.25  " "Median :21.50  " "Mean   :20.17  "
##
## "3rd Qu.:22.00  " "Max.   :24.00  "
```

К элементу списка можно также обратиться по его порядковому номеру или названию, заключив их в *двойные* квадратные скобки:

```
metrolist[[1]]
## [1] "                        6                "
metrolist[["desc"]]
## [1] "                        6                "
```

Использование *двойных скобок* отличает списки от векторов.

2.6 Факторы

2.7 Контрольные вопросы и задачи

2.7.1 Вопросы

1. Что такое вектор в языке R?
2. Какие способы создания векторов существуют?
3. Можно ли хранить в векторе данные разных типов?
4. Как определить длину вектора?
5. Как извлечь из вектора элемент по его индексу?
6. Как извлечь из вектора множество элементов по их индексам?
7. Как извлечь из вектора последний элемент?
8. С помощью какой функции можно сгенерировать последовательность чисел или дат с заданным шагом?
9. Как сгенерировать последовательность целых чисел с шагом 1, не прибегая к функциям?
10. Можно ли применять к векторам арифметические операторы? Что будет результатом их выполнения?
11. С помощью какой функции можно отсортировать вектор? Как изменить порядок сортировки на противоположный?
12. С помощью какой функции можно найти индекс элемента вектора по его значению? Что вернет функция, если этот элемент встречается в векторе несколько раз?
13. Что такое матрица, массив, фрейм данных и список? Чем отличаются эти структуры данных?
14. Какие из рассмотренных в этой главе структур данных могут содержать элементы разного типа, а какие только один?
15. Какая функция позволяет создать матрицу? По строкам или по столбцам заполняется матрица при использовании вектора как источника данных по умолчанию?
16. Как извлечь элемент по его индексам из матрицы, массива, фрейма данных, списка?
17. Как извлечь строку или столбец из матрицы или фрейма данных?
18. С помощью какого специального символа можно обратиться к столбцу фрейма данных по его названию?
19. Как получить или записать названия столбцов фрейма данных?
20. Как получить или записать названия строк фрейма данных?
21. Какая структура данных является результатом сортировки матрицы?
22. Какая функция позволяет осуществить транспонирование матрицы?
23. Какой оператор используется для умножения матриц? Каким критериям должны отвечать перемножаемые матрицы, чтобы эта операция была осуществима?
24. Как добавить новый столбец в фрейм данных? Опишите несколько вариантов.
25. Как добавить новую строку в фрейм данных?
26. Что произойдет, если к целочисленной матрице прибавить столбец, заполненный строками?
27. Какая функция позволяет находить индексы элементов матрицы или фрейма данных по их значениям?

2.7.2 Задачи

Chapter 3

Работа с таблицами

Программный код главы

Необходимые пакеты: `openxlsx, dplyr`

Таблица представляет собой один из способов структурирования данных. Большинство научных данных представляется именно в виде таблиц. В настоящем модуле рассмотрены базовые процедуры обработки таблиц, такие как чтение, сортировка, фильтрация, отбор столбцов (переменных), добавление и вычисление новых строк и столбцов, поиск и исправление ошибок в данных, запись таблиц в файл. Показано, как одни и те же процедуры могут быть реализованы стандартными средствами **R** и с помощью пакета `dplyr`.

Напомним, что для представления таблиц в **R** используются фреймы данных. В рамках данной главы мы будем использовать оба понятия как взаимозаменяемые, предполагая, что речь идет о таблице, представленной в виде фрейма данных.

3.1 Чтение таблиц

Существует множество файловых форматов представления табличных данных. Мы рассмотрим одни из наиболее распространенных — **CSV** и **Microsoft Excel**.

3.1.1 Таблицы CSV

CSV (Comma-separated value) — общее название для формата представления таблиц в виде текстовых файлов, организованных по следующему принципу:

1. Каждая строка в файле соответствует строке в таблице
2. Ячейки отделяются друг от друга символом-разделителем.
3. Если ячейка пустая, то между соседними разделителями не должно быть никаких символов.

Стандартным разделителем ячеек является запятая (`,`), а десятичным разделителем — точка (`.`). Однако это не является строгим правилом.

Например, вот так выглядит таблица в формате CSV, с которой мы дальше будем работать:

```
  ;   ;     ;   ;    ;     ;    ;   ;
1993;27,2;2,5;0,4;4,3;12,1;5,3;1,0;1,6
1994;24,6;2,3;0,4;3,2;11,0;5,0;0,9;1,8
1995;24,5;2,3;0,4;3,5;10,4;5,2;0,9;1,8
1996;22,4;2,2;0,3;3,1;9,8;4,7;0,8;1,5
1997;23,0;2,2;0,3;3,8;9,8;4,4;0,8;1,7
```

Видно, что первая строка в файле занята заголовками столбцов. Формат CSV этот момент не обговаривает, поэтому при чтении таблицы нужно явным образом указывать, что данные начинаются со второй строки, а первую строку следует интерепретировать как заголовочную.

Также следует отметить, что в данном файле в качестве символа-разделителя ячеек используется точка с запятой (`;`), поскольку в русской локали запятая (`,`) зарезервирована под десятичный разделитель. Разделитель ячеек и десятичный разделитель также должны быть указаны при открытии файла, если они не соответствуют стандартным.

Таблицы в формате **CSV** (Comma-Separated Values) можно прочесть с помощью универсальной функции `read.table()`. Ее основные параметры следующие:

- `file` — название файла
- `sep` — разделитель ячеек
- `dec` — десятичный разделитель
- `header` — содержится ли в первой строке заголовок — `encoding` — кодировка символов, в которой сохранен файл (чаще всего `UTF-8` или `CP1251`)

  Стандартной кодировкой для представления текста в UNIX-подобных системах (*Ubuntu*, *macOS* и т.д.) является **UTF-8 (Unicode)**, в русскоязычных версиях *Windows* — **CP1251 (Windows-1251)**. Текстовый файл **CSV**, созданный в разных операционных системах, будет по умолчанию сохраняться в соответствующей кодировке, если вы не указали ее явным образом. Если при загрузке таблицы в **R** вы видите вместо текста нечитаемые символы — *кракозябры* — то, скорее всего, вы читаете файл не в той кодировке, в которой он был сохранен. Попробуйте поменять `UTF-8` на `CP1251` или наоборот. Если вы не знаете, что такое кодировка и Юникод, то вам сюда.

Прочтем таблицу с данными Росстата по объему сброса сточных вод в бассейны некоторых морей России (в млн. м^3):

```
tab <- read.table("oxr_vod.csv",
                  sep = ';',
                  dec = ',',
                  header = TRUE,
                  encoding = 'UTF-8')
str(tab) #
## 'data.frame':    22 obs. of  9 variables:
##  $          : int  1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 ...
##  $         : num  27.2 24.6 24.5 22.4 23 22 20.7 20.3 19.8 19.8 ...
##  $       : num  2.5 2.3 2.3 2.2 2.2 2.2 2.2 2.2 2.1 2 ...
##  $         : num  0.4 0.4 0.4 0.3 0.3 0.3 0.3 0.3 0.3 0.2 ...
##  $        : num  4.3 3.2 3.5 3.1 3.8 3.2 2.5 2 1.9 2 ...
##  $       : num  12.1 11 10.4 9.8 9.8 9.5 9.1 9.2 8.9 9.2 ...
##  $         : num  5.3 5 5.2 4.7 4.4 4.2 4.1 4.2 4.2 4.1 ...
##  $         : num  1 0.9 0.9 0.8 0.8 0.8 0.8 0.9 0.9 0.8 ...
##  $         : num  1.6 1.8 1.8 1.5 1.7 1.8 1.7 1.5 1.5 1.5 ...
```

Существуют также специальные функции для чтения таблиц **CSV**: `read.csv()` и `read.csv2()`. По сути они являются "обертками" (*wrappers*) функции `read.table()` и выполняют ее вызов с автоматической подстановкой параметров `sep`, `dec` и `header`. Обе функции по умолчанию предполагают, что в файле имеется заголовок. `read.csv()` удобна для чтения таблиц с десятичной точкой и запятой-разделителем, а `read.csv2()` — для таблиц с десятичной запятой и точкой-с-запятой в качестве разделителя.

Используем для чтения `read.csv2()`:

```
tab <- read.csv2("oxr_vod.csv", encoding = 'UTF-8')
str(tab) #
## 'data.frame':    22 obs. of  9 variables:
##  $          : int  1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 ...
##  $         : num  27.2 24.6 24.5 22.4 23 22 20.7 20.3 19.8 19.8 ...
##  $       : num  2.5 2.3 2.3 2.2 2.2 2.2 2.2 2.2 2.1 2 ...
```

```
##  $         : num  0.4 0.4 0.4 0.3 0.3 0.3 0.3 0.3 0.3 0.2 ...
##  $        : num  4.3 3.2 3.5 3.1 3.8 3.2 2.5 2 1.9 2 ...
##  $       : num  12.1 11 10.4 9.8 9.8 9.5 9.1 9.2 8.9 9.2 ...
##  $        : num  5.3 5 5.2 4.7 4.4 4.2 4.1 4.2 4.2 4.1 ...
##  $         : num  1 0.9 0.9 0.8 0.8 0.8 0.8 0.9 0.9 0.8 ...
##  $         : num  1.6 1.8 1.8 1.5 1.7 1.8 1.7 1.5 1.5 1.5 ...
```

Данная таблица не отличается от предыдущей, но ее чтение с помощью функции `read.csv2()` более компактно.

3.1.2 Таблицы Microsoft Excel

Чтение таблиц **Microsoft Excel** не входит в возможности стандартной библиотеки R. В то же время, для этих целей существует ряд пакетов, таких как `xlsx`, `readxl` и `openxlsx`. Мы будем пользоваться пакетом `openxlsx`. Подключим его:

```
library(openxlsx)
```

Для чтения таблицы воспользуемся функцией `read.xlsx()` из этого пакета. В качестве обязательных параметров она принимает следующие аргументы:

- `xlsxFile` — название файла
- `sheet` — номер листа

Откроем таблицу с данными Росстата по сбросу загрязненных сточных вод в поверхностные водные объекты (млн м^3).

```
sewage <- read.xlsx("sewage.xlsx", 1) #
str(sewage) #
## 'data.frame':    97 obs. of  6 variables:
##  $ X1  : chr  "           " "                 " "           " "         " ...
##  $ 2005: num  17727 4341 11 89 155 ...
##  $ 2010: num  16516 3761 77 78 129 ...
##  $ 2011: num  15966 3613 72 75 126 ...
##  $ 2012: num  15678 3651 71 71 124 ...
##  $ 2013: num  15189 3570 71 68 120 ...
```

3.2 Просмотр таблицы

Для просмотра фрейма данных в консоли RStudio вы можете использовать несколько опций. Пусть наш фрейм данных называется `tab`. Тогда:

1. `print(tab)` — выводит фрейм в консоль целиком (можно написать просто `tab` в консоли).
2. `head(tab, n)` — отбирает первые n строк фрейма
3. `tail(tab, n)` — отбирает последние n строк фрейма

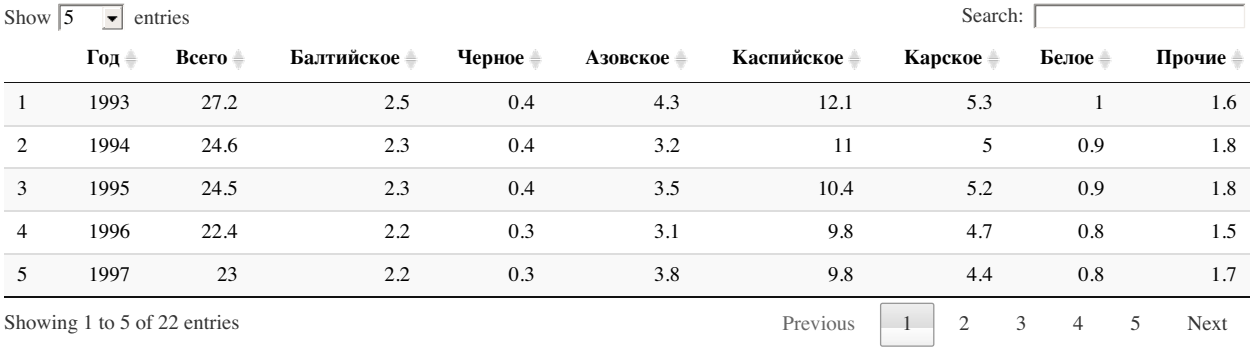
По умолчанию для функций `head()` и `tail()` $n = 6$. Обычно этот параметр опускают, поскольку нужно просмотреть только первые несколько строк и шести вполне достаточно. Если вы напечатаете в консоли `head(tab)` или `tail(tab)`, то для выбранных строк будет вызвана функция `print()`, аналогично выводу всего фрейма:

```
print(tab)
##
## 1  1993  27.2        2.5    0.4      4.3       12.1     5.3   1.0    1.6
## 2  1994  24.6        2.3    0.4      3.2       11.0     5.0   0.9    1.8
## 3  1995  24.5        2.3    0.4      3.5       10.4     5.2   0.9    1.8
## 4  1996  22.4        2.2    0.3      3.1        9.8     4.7   0.8    1.5
## 5  1997  23.0        2.2    0.3      3.8        9.8     4.4   0.8    1.7
```

```
## 6  1998  22.0        2.2    0.3      3.2        9.5     4.2   0.8    1.8
## 7  1999  20.7        2.2    0.3      2.5        9.1     4.1   0.8    1.7
## 8  2000  20.3        2.2    0.3      2.0        9.2     4.2   0.9    1.5
## 9  2001  19.8        2.1    0.3      1.9        8.9     4.2   0.9    1.5
## 10 2002  19.8        2.0    0.2      2.0        9.2     4.1   0.8    1.5
## 11 2003  19.0        2.0    0.2      2.1        8.4     4.0   0.8    1.5
## 12 2004  18.5        2.0    0.2      2.1        8.3     3.8   0.8    1.3
## 13 2005  17.7        2.0    0.2      1.6        8.0     3.8   0.8    1.3
## 14 2006  17.5        1.9    0.2      1.7        7.8     3.8   0.8    1.3
## 15 2007  17.2        1.9    0.2      1.7        7.4     3.8   0.8    1.4
## 16 2008  17.1        1.9    0.2      1.6        7.5     3.9   0.8    1.2
## 17 2009  15.9        1.8    0.2      1.5        6.8     3.5   0.7    1.4
## 18 2010  16.5        2.0    0.2      1.6        7.3     3.3   0.7    1.4
## 19 2011  16.0        1.9    0.2      1.6        7.1     3.2   0.7    1.3
## 20 2012  15.7        1.8    0.2      1.6        7.0     3.0   0.7    1.4
## 21 2013  15.2        1.8    0.2      1.6        6.9     3.0   0.6    1.1
## 22 2014  14.8        1.7    0.2      1.5        6.4     3.2   0.6    1.2
head(tab)
##
## 1 1993  27.2        2.5    0.4      4.3       12.1     5.3   1.0    1.6
## 2 1994  24.6        2.3    0.4      3.2       11.0     5.0   0.9    1.8
## 3 1995  24.5        2.3    0.4      3.5       10.4     5.2   0.9    1.8
## 4 1996  22.4        2.2    0.3      3.1        9.8     4.7   0.8    1.5
## 5 1997  23.0        2.2    0.3      3.8        9.8     4.4   0.8    1.7
## 6 1998  22.0        2.2    0.3      3.2        9.5     4.2   0.8    1.8
tail(tab)
##
## 17 2009  15.9        1.8    0.2      1.5        6.8     3.5   0.7    1.4
## 18 2010  16.5        2.0    0.2      1.6        7.3     3.3   0.7    1.4
## 19 2011  16.0        1.9    0.2      1.6        7.1     3.2   0.7    1.3
## 20 2012  15.7        1.8    0.2      1.6        7.0     3.0   0.7    1.4
## 21 2013  15.2        1.8    0.2      1.6        6.9     3.0   0.6    1.1
## 22 2014  14.8        1.7    0.2      1.5        6.4     3.2   0.6    1.2
```

RStudio, разумеется, имеет "человеческий" интерфейс для просмотра таблиц, в котором таблицу можно сортировать и фильтровать. Чтобы его активировать, надо вызвать функцию `View()`:

```
View(tab)
```


Show 5 entries Search:

	Год	Всего	Балтийское	Черное	Азовское	Каспийское	Карское	Белое	Прочие
1	1993	27.2	2.5	0.4	4.3	12.1	5.3	1	1.6
2	1994	24.6	2.3	0.4	3.2	11	5	0.9	1.8
3	1995	24.5	2.3	0.4	3.5	10.4	5.2	0.9	1.8
4	1996	22.4	2.2	0.3	3.1	9.8	4.7	0.8	1.5
5	1997	23	2.2	0.3	3.8	9.8	4.4	0.8	1.7

Showing 1 to 5 of 22 entries Previous 1 2 3 4 5 Next

Поскольку функции `head()` и `tail()` возвращают строки с хвоста или начала фрейма данных, их можно и подать на вход функции `View()`:

```
View(head(sewage, 3))
```

Show 5 entries Search:

	X1	2005	2010	2011	2012	2013
1	Российская Федерация	17727	16516	15966	15678	15189
2	Центральный федеральный округ	4341	3761	3613	3651	3570
3	Белгородская область	11	77	72	71	71

Showing 1 to 3 of 3 entries Previous 1 Next

Как правило, не следует оставлять вызовы функции `View()` в тексте законченной программы. Это приведет к тому, что при запуске будут открываться новые вкладки с просмотром таблиц, что может раздражать пользователя (в том числе и вас самих). Используйте `View()` для вывода окончательного результата в конце программы или при отладке программы. Все вызовы `View()` в программе можно легко закомментировать или раскомментировать, выполнив поиск с заменой `'View('` на `'# View('` и наоборот.

3.3 Работа со столбцами

3.3.1 Названия столбцов

Для просмотра и изменения названий столбцов фрейма данных следует использовать функцию `colnames()`:

```
#
colnames(sewage)
## [1] "X1"   "2005" "2010" "2011" "2012" "2013"
colnames(tab)
## [1] "  "         "   "       "      " "    "      "     "
## [6] "      " "    "    "    "   "      "    "

#
colnames(sewage) <- c("Region", "Year05", "Year10", "Year11", "Year12", "Year13")
colnames(tab) <- c("Year", "Total", "Baltic", "Black", "Azov", "Caspian", "Kara", "White", "Other")

#
colnames(sewage)
## [1] "Region" "Year05" "Year10" "Year11" "Year12" "Year13"
colnames(tab)
## [1] "Year"    "Total"   "Baltic"  "Black"   "Azov"    "Caspian" "Kara"
## [8] "White"   "Other"
```

3.3.2 Обращение к столбцам

К столбцу можно обращаться по номеру и названию (с помощью оператора $ или в кавычках внутри скобок). Если вы указываете в квадратных скобках номер без запятой, он трактуется именно как номер столбца, а не строки. Тип возвращаемого значения зависит от синтаксиса:

- обращение через $ возвращает вектор;
- обращение в скобках с запятой к одному столбцу возвращает вектор;
- обращение в скобках с запятой к нескольким столбцам возвращает фрейм данных;
- обращение в скобках без запятой возвращает фрейм данных.

Несколько примеров:

```
a <- head(sewage)

#         -
a$Year05       #
## [1] 17727  4341    11    89   155   169
a[, "Year05"] #
## [1] 17727  4341    11    89   155   169
a[, 2]         #
## [1] 17727  4341    11    89   155   169

a["Year05"] #
##   Year05
## 1  17727
## 2   4341
## 3     11
## 4     89
## 5    155
## 6    169
a[2]         #
##   Year05
## 1  17727
## 2   4341
## 3     11
## 4     89
## 5    155
## 6    169

#             -
a[, c(1, 4)]               #
##                          Region Year11
## 1                         15966
## 2                        3613
## 3                             72
## 4                               75
## 5                            126
## 6                             135
a[, c("Region", "Year11")]#
##                          Region Year11
## 1                         15966
## 2                        3613
## 3                             72
## 4                               75
## 5                            126
## 6                             135
a[c("Region", "Year11")]  #
##                          Region Year11
## 1                         15966
## 2                        3613
## 3                             72
## 4                               75
## 5                            126
## 6                             135
a[c(1, 4)]                 #
```

```
##                            Region Year11
## 1                           15966
## 2                          3613
## 3                               72
## 4                                 75
## 5                              126
## 6                               135
```

Использование необходимой формы зависит от контекста и ваших целей.

3.3.3 Выбор и исключение столбцов

Можно создать новую таблицу, выбрав необходимые столбцы, как это показано выше:

```
#                                    :
caspian <- tab[c("Year", "Total", "Caspian")]
str(caspian)
## 'data.frame':    22 obs. of  3 variables:
##  $ Year   : int  1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 ...
##  $ Total  : num  27.2 24.6 24.5 22.4 23 22 20.7 20.3 19.8 19.8 ...
##  $ Caspian: num  12.1 11 10.4 9.8 9.8 9.5 9.1 9.2 8.9 9.2 ...
```

Иногда проще создать новый фрейм данных, исключив из оригинала ненужные столбцы. Исключение делается с помощью знака '-', который ставится перед номером столбца. Например, вот так можно исключить из таблицы `tab` столбцы `Total` (2-й) и `Other` (9-й):

```
cleaned <- tab[c(-2, -9)]
str(cleaned)
## 'data.frame':    22 obs. of  7 variables:
##  $ Year   : int  1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 ...
##  $ Baltic : num  2.5 2.3 2.3 2.2 2.2 2.2 2.2 2.2 2.1 2 ...
##  $ Black  : num  0.4 0.4 0.4 0.3 0.3 0.3 0.3 0.3 0.3 0.2 ...
##  $ Azov   : num  4.3 3.2 3.5 3.1 3.8 3.2 2.5 2 1.9 2 ...
##  $ Caspian: num  12.1 11 10.4 9.8 9.8 9.5 9.1 9.2 8.9 9.2 ...
##  $ Kara   : num  5.3 5 5.2 4.7 4.4 4.2 4.1 4.2 4.2 4.1 ...
##  $ White  : num  1 0.9 0.9 0.8 0.8 0.8 0.8 0.9 0.9 0.8 ...
```

Есть также способ *удалить столбец целиком* (им пользуются довольно редко). Для этого необходимо записать в него значение `NULL`:

```
cleaned$Azov <- NULL
str(cleaned)
## 'data.frame':    22 obs. of  6 variables:
##  $ Year   : int  1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 ...
##  $ Baltic : num  2.5 2.3 2.3 2.2 2.2 2.2 2.2 2.2 2.1 2 ...
##  $ Black  : num  0.4 0.4 0.4 0.3 0.3 0.3 0.3 0.3 0.3 0.2 ...
##  $ Caspian: num  12.1 11 10.4 9.8 9.8 9.5 9.1 9.2 8.9 9.2 ...
##  $ Kara   : num  5.3 5 5.2 4.7 4.4 4.2 4.1 4.2 4.2 4.1 ...
##  $ White  : num  1 0.9 0.9 0.8 0.8 0.8 0.8 0.9 0.9 0.8 ...
```

3.3.4 Добавление и вычисление столбцов {col_add}

Существует простой способ создать новый столбец в таблице — достаточно указать его название после значка $. Если среда R не обнаруживает столбец с таким названием, она его создаст:

```
caspian$CaspianRatio <- round(caspian$Caspian / caspian$Total, 3)
```

```
View(caspian)
```

Show 5 entries Search:

	Year	Total	Caspian	CaspianRatio
1	1993	27.2	12.1	0.445
2	1994	24.6	11	0.447
3	1995	24.5	10.4	0.424
4	1996	22.4	9.8	0.438
5	1997	23	9.8	0.426

Showing 1 to 5 of 22 entries Previous 1 2 3 4 5 Next

Тем не менее, такой метод добавления столбцов нельзя считать правильным. Главный его недостаток — из формы записи неочевидно, что мы добавляем новый столбец, а не перевычисляем уже существующий. Пакет `dplyr`, который мы рассмотрим в конце настоящей главы, решает эту и многие другие стилистические проблемы работы с фреймами данных.

Помимо того что вы можете вычислять столбцы традиционным способом как функцию от других столбцов, есть удобные функции-агрегаторы, позволяющие сделать вычисления по всем столбцам. Это `rowSums()` (сумма всех столбцов в строке) и `rowMeans()` (среднее по всем столбцам в строке). На тот случай, когда в ячейках есть пропущенные значения, в функциях предусмотрен параметр `na.rm = TRUE`.

```
years <- sewage[c(-1, -2)] #           2010   2013

colSums(years)  #
## Year10 Year11 Year12 Year13
##     NA     NA     NA     NA
rowMeans(years) #
##         1         2         3         4         5         6         7
## 15837.250  3648.750    72.750    73.000   124.750   132.250    96.500
##         8         9        10        11        12        13        14
##    89.250    45.250    37.000    84.000  1234.500    52.750    86.000
##        15        16        17        18        19        20        21
##    67.500    25.800    93.000   188.750   224.750   922.000  2906.750
##        22        23        24        25        26        27        28
##   190.500   118.000   374.000        NA     0.175   373.750   152.750
##        29        30        31        32        33        34        35
##    96.000   260.000   345.750    87.250    44.000  1239.250  1399.250
##        36        37        38        39        40        41        42
##    27.750    27.250   878.500    62.000   155.250   248.500   389.000
##        43        44        45        46        47        48        49
##    77.750     3.825    31.000    47.000    93.250     0.000  2860.750
##        50        51        52        53        54        55        56
##   319.500    55.750    40.500   483.750   115.250    29.250   381.750
##        57        58        59        60        61        62        63
##   180.250   477.250   125.250   106.000   376.250    59.500   110.250
##        64        65        66        67        68        69        70
##  1745.750    44.750   733.000   184.000        NA    53.250    35.500
##        71        72        73        74        75        76        77
##    95.250   784.500  2094.500     0.275    29.000     8.375    34.500
##        78        79        80        81        82        83        84
```

```
##    10.775    73.500   427.250   582.250   632.750   107.250   166.000
##        85        86        87        88        89        90        91
##    22.750   792.250    82.500    36.250   327.500   183.250    79.000
##        92        93        94        95        96        97
##    21.500    42.250    15.000     4.975        NA        NA
```

Существуют также аналогичные им функции `colSums()` и `colMeans()`, осуществляющие агрегирование данных по столбцам, а не по строкам.

Перечисленные функции являются укороченной версией универсальных функций семейства `apply`, с которыми мы познакомимся далее.

3.4 Сортировка и фильтрация

3.4.1 Сортировка

Распространенные операции с таблицами — это упорядочение по определенному столбцу и фильтрация по значениям. Мы уже знаем что из вектора, матрицы или таблицы можно извлекать элементы: `tab[V, ]`, где `tab` — имя таблицы, `V` — это вектор из номеров элементов. Например, извлечь 5, 2 и 4 строку таблицы можно так:

```
tab[c(5,2,4), ]
##   Year Total Baltic Black Azov Caspian Kara White Other
## 5 1997  23.0    2.2   0.3  3.8     9.8  4.4   0.8   1.7
## 2 1994  24.6    2.3   0.4  3.2    11.0  5.0   0.9   1.8
## 4 1996  22.4    2.2   0.3  3.1     9.8  4.7   0.8   1.5
```

Логично предположить, что таким же образом можно извлечь элементы таблицы в порядке, обеспечивающем возрастание или убывание значений в каком-то столбце. Для этого нужно правильным образом расставить индексы в векторе `c(...)`. Существует специальная функция `order()`, которая позволяет это сделать. Например, отсортируем таблицу по возрастанию сбросов в Каспийское море:

```
indexes<-order(tab$Caspian)
head(tab[indexes, ])
##    Year Total Baltic Black Azov Caspian Kara White Other
## 22 2014  14.8    1.7   0.2  1.5     6.4  3.2   0.6   1.2
## 17 2009  15.9    1.8   0.2  1.5     6.8  3.5   0.7   1.4
## 21 2013  15.2    1.8   0.2  1.6     6.9  3.0   0.6   1.1
## 20 2012  15.7    1.8   0.2  1.6     7.0  3.0   0.7   1.4
## 19 2011  16.0    1.9   0.2  1.6     7.1  3.2   0.7   1.3
## 18 2010  16.5    2.0   0.2  1.6     7.3  3.3   0.7   1.4
```

Если упорядочение несложное, программист его скорее всего вставит непосредственно в инструкцию обращения к таблице:

```
head(tab[order(tab$Caspian), ])
##    Year Total Baltic Black Azov Caspian Kara White Other
## 22 2014  14.8    1.7   0.2  1.5     6.4  3.2   0.6   1.2
## 17 2009  15.9    1.8   0.2  1.5     6.8  3.5   0.7   1.4
## 21 2013  15.2    1.8   0.2  1.6     6.9  3.0   0.6   1.1
## 20 2012  15.7    1.8   0.2  1.6     7.0  3.0   0.7   1.4
## 19 2011  16.0    1.9   0.2  1.6     7.1  3.2   0.7   1.3
## 18 2010  16.5    2.0   0.2  1.6     7.3  3.3   0.7   1.4
```

3.4.2 Фильтрация

Схожим образом реализована *фильтрация данных* по значению. Например, вы хотите извлечь из таблицы только те года, в которых объем сбросов в Каспийское море составил более 10 млн м^3. Здесь используется еще одна возможность извлечения элементов таблицы — с помощью вектора логических значений TRUE/FALSE. Число элементов в этом векторе должно быть равно числу элементов в индексируемом векторе, а значение указывает на то, нужно ли извлекать (TRUE) или нет (FALSE) элемент с текущим индексом. Вектор логических значений получается естественным путем с помощью операции сравнения:

```
condition <- tab$Caspian > 10
condition  #
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
tab[condition, ] #                          :
##   Year Total Baltic Black Azov Caspian Kara White Other
## 1 1993  27.2    2.5   0.4  4.3    12.1  5.3   1.0   1.6
## 2 1994  24.6    2.3   0.4  3.2    11.0  5.0   0.9   1.8
## 3 1995  24.5    2.3   0.4  3.5    10.4  5.2   0.9   1.8
```

Опять же, весьма часто используется запись одним выражением:

```
tab[tab$Caspian > 10, ]
##   Year Total Baltic Black Azov Caspian Kara White Other
## 1 1993  27.2    2.5   0.4  4.3    12.1  5.3   1.0   1.6
## 2 1994  24.6    2.3   0.4  3.2    11.0  5.0   0.9   1.8
## 3 1995  24.5    2.3   0.4  3.5    10.4  5.2   0.9   1.8
```

Часто бывает необходимо отобрать данные из таблицы, содержащей разнородные данные. В частности, в нашей таблице смешаны данные по субъектам и федеральным округам. Предположим, необходимо выгрузить в отдельную таблицу данные по федеральным округам. Для этого нужно найти строки, в которых столбец Region содержит фразу " ". Для поиска по текстовым эталонам импользуется функция grep(), выдающая номера элементов, или ее разновидность grepl(), выдающая список логических констант

```
#          -          ,          -
rows <- grep("          ",sewage$Region)
rows  #     ,              Region
## [1]  2 21 35 42 49 64 73 86
okruga <- sewage[rows,] #
```

```
View(okruga)
```

Show 5 entries Search:

	Region	Year05	Year10	Year11	Year12	Year13
2	Центральный федеральный округ	4341	3761	3613	3651	3570
21	Северо-Западный федеральный округ	3192	3088	2866	2877	2796
35	Южный федеральный округ	1409	1446	1436	1394	1321
42	Северо-Кавказский федеральный округ	496	390	397	395	374
49	Приволжский федеральный округ	3162	2883	2857	2854	2849

Showing 1 to 5 of 8 entries Previous 1 2 Next

Наоборот — для **исключения** найденных объектов удобнее воспользоваться разновидностью grepl(), которая возвращает вектор из логических значений:

```
rows2 <- grepl("          ", sewage$Region)
rows2 #                 grepl
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [67] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

neokruga <- sewage[!rows2, ]
```

```
View(neokruga)
```

Show 5 entries Search:

	Region	Year05	Year10	Year11	Year12	Year13
1	Российская Федерация	17727	16516	15966	15678	15189
3	Белгородская область	11	77	72	71	71
4	Брянская область	89	78	75	71	68
5	Владимирская область	155	129	126	124	120
6	Воронежская область	169	134	135	131	129

Showing 1 to 5 of 89 entries Previous 1 2 3 4 5 … 18 Next

Обратите внимание на восклицательный знак перед `rows2`. Он меняет все значения `TRUE` на `FALSE` и наоборот, что позволяет исключить найденные объекты

В полученной таблице все еще содержится текстовая шелуха типа " ", " ...", а также строка " ". К счастью, функция `grep()` достаточно умна и возволяет искать сразу по нескольким образцам строк. Для этого их нужно разделить вертикальной чертой — *пайпом* (`|`):

```
rows2 <- grepl("  |  |  | | ", sewage$Region)
rows2
##  [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [23] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [67] FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
neokruga <- sewage[!rows2, ] #                                rows2
```

```
View(neokruga)
```

Show 5 entries Search:

	Region	Year05	Year10	Year11	Year12	Year13
3	Белгородская область	11	77	72	71	71
4	Брянская область	89	78	75	71	68
5	Владимирская область	155	129	126	124	120
6	Воронежская область	169	134	135	131	129
7	Ивановская область	144	102	99	97	88

Showing 1 to 5 of 83 entries Previous 1 2 3 4 5 … 17 Next

3.5 Пропущенные значения

Можно ли осуществлять обработку таблицы `sewage`? Попробуем в качестве примера найти минимум сбросов за 2012 год:

```
max(sewage$Year12)
## [1] NA
```

Результат имеет тип `NA`, потому что в данном столбце имеются пропуски. В некоторых статистических задачах это недопустимо. Если вы хотите проигнорировать значения пропусков, следует в вызываемой статистической функции указать дополнительный параметр `na.rm = TRUE`:

```
max(sewage$Year12, na.rm = TRUE)
## [1] 15678
```

Еще один вариант — исключить из таблицы те строки, в которых имеются пропущенные значения (хотя бы одно!). Для этого существует функция `complete.cases()`, возвращающая вектор логических значений:

```
filter<-complete.cases(sewage)
filter  #               .          FALSE -
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [23]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [56]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [67]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [78]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [89]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE

sewage.complete <- sewage[filter, ] #
```

```
View(sewage.complete)
```

Show 5 entries Search:

	Region	Year05	Year10	Year11	Year12	Year13
1	Российская Федерация	17727	16516	15966	15678	15189
2	Центральный федеральный округ	4341	3761	3613	3651	3570
3	Белгородская область	11	77	72	71	71
4	Брянская область	89	78	75	71	68
5	Владимирская область	155	129	126	124	120

Showing 1 to 5 of 93 entries Previous 1 2 3 4 5 … 19 Next

3.6 Преобразование типов и поиск ошибок

Достаточно часто при работе с реальными данными возникает необходимость преобразования их типов. Например, вам необходимо перевести строки в даты, чтобы оперировать ими соответствующим образом. Или принудительным образом указать, что столбец со строками не хранит номинальную переменную (фактор), а его нужно интерпретировать именно как строковый столбец (обычно это полезно, когда столбец содержит какую-то текстовую информацию в виде комментариев по каждому измерению). Наконец, в данных могут быть ошибки, опечатки и так далее, которые могут препятствовать правильному их чтению.

В этом разделе мы рассмотрим, как можно:

1. Найти и исправить множественные варианты одного названия с опечатками
2. Исправить ошибки в числовых данных
3. Преобразовать факторы в строки и наоборот
4. Преобразовать строки в числа и наоборот

Рассмотрим возможные манипуляции с данными на примере таблицы о землепользовании на территории Сатинского учебного полигоны Географического факультета МГУ:

```
tab <- read.csv2("SatinoLanduse.csv", encoding = 'UTF-8')
str(tab) #      ,
## 'data.frame':    160 obs. of  6 variables:
##  $ ID            : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Type          : Factor w/ 12 levels "   ","    ",..: 11 1 1 9 5 1 5 12 9 10 ...
##  $ Administration: Factor w/ 7 levels "","    ",             ",..: 5 5 5 1 1 5 1 5 5 6 ...
##  $ Comment       : Factor w/ 35 levels "","  \"   \"",..: 30 1 1 1 1 1 1 3 2 1 ...
##  $ Perimeter     : Factor w/ 160 levels "1014.155593894044800",..: 67 155 51 104 78 153 17 19 108 57
##  $ Area          : Factor w/ 160 levels "0.238070145845919",..: 73 49 121 100 63 72 88 128 99 24 ...
```

```
View(tab)
```

Show 5 entries Search:

	ID	Type	Administration	Comment	Perimeter	Area
1	1	Территории населенных пунктов	Совьяковская сельская администрация	Село Бениды	2395.725463843722500	286159.158855028570000
2	2	Выгоны	Совьяковская сельская администрация		922.314163446381600	21651.963989321295000
3	3	Выгоны	Совьяковская сельская администрация		2180.787622893539300	56826.463220403260000
4	4	Пашни			3947.940266502537700	450293.758854912190000
5	5	Леса			278.569432215831970	2612.615315620203500

Showing 1 to 5 of 160 entries Previous 1 2 3 4 5 … 32 Next

Видно, что все столбцы, кроме двух, хранящих идентификаторы, были прочитаны как строки и преобразованы в факторы (номинальные переменные). Это означает, что мы не сможем работать привычным образом со столбцами периметра и площади, а столбец комментариев теперь также является номинальной переменной, что противоречит здравому смыслу (он вообще переменной не является).

Когда вы отображаете таблицу в консоли или графическом интерфейсе, факторы выглядят и ведут себя как обычные строки. Подвох заключается в том, что хранятся они в виде пар “ключ — значение” (об этом мы говорили выше) и все операции преобразования осуществляются **над ключами**, а не значениями. Рассмотрим, как следует правильно преобразовывать номинальные переменные в **R**.

Чтобы привести столбцы к нужному типу, необходимо использовать преобразования типов. Для этого можно воспользоваться функциями семейства `as`, описанными в Главе 1.

Для начала преобразуем столбец `Comment` к обычному символьному представлению:

```
tab$Comment <- as.character(tab$Comment)
str(tab)
## 'data.frame':    160 obs. of  6 variables:
##  $ ID            : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Type          : Factor w/ 12 levels "   ","    ",..: 11 1 1 9 5 1 5 12 9 10 ...
##  $ Administration: Factor w/ 7 levels "","  ","              ",..: 5 5 5 1 1 5 1 5 5 6 ...
##  $ Comment       : chr  "       " "" "" "" ...
##  $ Perimeter     : Factor w/ 160 levels "1014.155593894044800",..: 67 155 51 104 78 153 17 19 108 57
##  $ Area          : Factor w/ 160 levels "0.238070145845919",..: 73 49 121 100 63 72 88 128 99 24 ...
```

Посмотрим теперь, что произойдет, если мы попытаемся преобразовать столбец `Perimeter` к числовому виду:

```
as.numeric(tab$Perimeter)
##   [1]  67 155  51 104  78 153  17  19 108  57   7   3 158 159 156  50  91
##  [18] 143   6  58   4   5 131 148 113 128 147 114   9  18 118 132  84 134
##  [35]  81  40 130  98  83 157  42  95  71 141   8 100  34   1  87  77 160
##  [52]  93 119  90  74  35 125 150 101 136  31 109 110 103  75  14  32  63
##  [69] 145  56 102  25  65  88  72  53  92  30 117  73  43  54 121  44  52
##  [86]  27 115 149 120  45  26  41   2  60  36 123  29 151 144 106 127  12
## [103] 116  94  82 146 142  69  21  48 139 105 154 124  47  61  33  80  97
## [120]  64  10  76 111  11 112  89  28 129  68  39  49  86  96  59  24 137
## [137]  46 152  55  15  99  85  22 126  16 122  79  66 133  23 107  38 138
## [154]  13 135  37 140  70  20  62
```

Вместо значений перметра мы получили загадочные числа, которых в таблице нет. Это и есть ключи факторов. Чтобы получить их значения, необходимо использовать функцию `levels()` (для краткости выведем первые 10 значений):

```
levels(tab$Perimeter)[1:10]
##  [1] "1014.155593894044800" "1019.457949256323400" "1020.278536197552200"
```

```
##  [4] "1021.109926202218700" "1041.122684298658400" "1060.678503301135200"
##  [7] "1081.964408568060900" "1094.945610298295600" "114.701418496307100"
## [10] "1155.916232728818800"
```

Обратите внимание на то, что значения фактора отсортированы в алфавитном порядке, без учете порядка их встречаемости в исходной таблице. Для корректного преобразования факторов в числа необходимо сначала привести их к обычному строковому виду:

```
tab$Perimeter <- as.numeric(as.character(tab$Perimeter))
str(tab)
## 'data.frame':    160 obs. of  6 variables:
##  $ ID            : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Type          : Factor w/ 12 levels "   ","    ",..: 11 1 1 9 5 1 5 12 9 10 ...
##  $ Administration: Factor w/ 7 levels "","  ","             ",..: 5 5 5 1 1 5 1 5 5 6 ...
##  $ Comment       : chr  "       " "" "" "" ...
##  $ Perimeter     : num  2396 922 2181 3948 279 ...
##  $ Area          : Factor w/ 160 levels "0.238070145845919",..: 73 49 121 100 63 72 88 128 99 24 ...

#                       Area
temp <- as.numeric(as.character(tab$Area))
## Warning:                       NA
temp[1:10]
##  [1]  286159.159   21651.964   56826.463  450293.759    2612.615
##  [6]   28608.401 3469445.793   62299.631  450291.261  147943.134
```

Все прошло вроде бы успешно, но с предупреждением, что некоторые значения были преобразованы в `NA` (*Not Available*) — отсутствующие значения. По всей видимости, данные в соответствущих ячейках не соответствуют представлениям **R** о том, как должно выглядеть число: ячейка или пустая, или число набрано с ошибкой/опечаткой.

Чтобы найти и исправить все неверно заданные данные, необходимо выполнить следующие действия:

1. Получить индексы всех элементов, имеющих значение `NA`.
2. Просмотреть, какие значения были в исходных данных под этими индексами
3. Исправить ошибки в этих значениях, если это поддается автоматизации
4. Повторить конвертацию в числовой тип данных

Проверку на отсутствующие данные осуществляют с помощью функции `is.na()`. Передав ей в качестве аргумента вектор значений, вы получите вектор булевых значений, в котором `TRUE` будет стоять для пустых элементов. Проверим с помощью него, какие элементы столюца `Area` привели к ошибкам конвертации данных:

```
tab[is.na(temp), "Area"]
## [1] 89499,573298880117000 11922,638460079328000 5153,570673500797100
## 160 Levels: 0.238070145845919 ... 9865.323033935605100
```

Видно, что **R** не справился с преобразованием типов там, где содержится опечатка в десятичном разделителе — вместо точки указана запятая.

Для исправления этой ошибки мы можем воспользоваться стандартной функцией замены символа `gsub(pattern, replacement, x)`. Ее стандартные параметры означают соответственно: что искать, на что заменять, где искать:

```
tab$Area <- gsub(',', '.', tab$Area) #
tab$Area <- as.numeric(as.character(tab$Area)) #
str(tab)
## 'data.frame':    160 obs. of  6 variables:
##  $ ID            : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Type          : Factor w/ 12 levels "   ","    ",..: 11 1 1 9 5 1 5 12 9 10 ...
##  $ Administration: Factor w/ 7 levels "","  ","             ",..: 5 5 5 1 1 5 1 5 5 6 ...
##  $ Comment       : chr  "       " "" "" "" ...
```

```
##  $ Perimeter     : num  2396 922 2181 3948 279 ...
##  $ Area          : num  286159 21652 56826 450294 2613 ...
```

Теперь необходимо навести порядок в значениях факторов, убедившись, что и там нет опечаток. Выведем все уникальные значения с помощью функции `levels()`:

```
levels(tab$Type)
##  [1] "    "                          "     "
##  [3] "              "        "            "
##  [5] "   "                             "         "
##  [7] "   "                             "       "
##  [9] "    "                           "   "
## [11] "                  " "             "
levels(tab$Administration)
## [1] ""
## [2] " "
## [3] "                "
## [4] "                      "
## [5] "                     "
## [6] "                     "
## [7] "                    "
```

Видно, что если с типами все в порядке, то в данных об административном подчинении содержится 5 вариантов названия одной и той же Совьяковской сельской администрации. Помимо этого, пустые ячейки хорошо бы заменить на значение " ".

Чтобы найти все строчки, относящиеся к одному и тому же объекту, можно воспользоваться уже знакомой нам функцией `grep()`, передав ей подстроку, которая является для них общей. Например, " " (хотя в данном случае было бы вообще достаточно одной буквы " ").

```
filter <- grep("      ", tab$Administration) #
tab[filter, "Administration"] <- "                    " #
tab$Administration <- droplevels(tab$Administration) #
levels(tab$Administration)
## [1] ""
## [2] " "
## [3] "                    "
```

Пустые строки можно также найти с помощью `grep()`, но мы этого делать не будем, так как это требует дополнительных знаний о регулярных выражениях. Вместо этого воспользуемся тем, что пустые строки имеют длину 0. Обратите внимание ниже, что преобразование в вектор столбца `Administration` необходимо, т.к. `nchar()` не понимает объекты типа `data.frame`, которыми являются не только таблицы, но и их столбцы:

```
filter <- nchar(as.vector(tab$Administration)) == 0 # TRUE           0
#          :
tab[filter, "Administration"] <- "    "
## Warning in `[<-.factor`(`*tmp*`, iseq, value = c("    ", "    ", :
##                  ,        NA
```

Ошибка выше связана с тем, что **R** строго следит за неизменностью набора значений фактора для того чтобы избежать всевозможных ошибок при работе с данными (опечаток и т.д.). Предыдущий раз мы заменили все значаниея одним из существующих. В данном случае необходимо ввести новое значение фактора. Чтобы это сделать, придется преобразовать данные в символьные, произвести замену срок и после этого снова конвертировать столбец в фактор:

```
tab$Administration <- as.character(tab$Administration)
tab[filter, "Administration"] <- "    "
tab$Administration <- as.factor(tab$Administration)
```

```
levels(tab$Administration)
## [1] "    "
## [2] " "
## [3] "                    "
```

Теперь таблица готова к работе. Можно, например, подсчитать по ней сводную статистику:

```
summary(tab)
##        ID                                  Type
##  Min.   :  1.00                            :52
##  1st Qu.: 40.75                           :27
##  Median : 80.50                        :22
##  Mean   : 80.50                            :15
##  3rd Qu.:120.25                            :11
##  Max.   :160.00                    : 8
##                   (Other)                      :25
##                               Administration   Comment
##                                   :76     Length:160
##                                    : 3     Class :character
##                      :81     Mode  :character
##
##
##
##
##    Perimeter              Area
##  Min.   :    3.087   Min.   :      0
##  1st Qu.:  421.431   1st Qu.:   5087
##  Median :  939.369   Median :  21260
##  Mean   : 1761.654   Mean   : 125002
##  3rd Qu.: 2135.987   3rd Qu.:  83019
##  Max.   :23920.945   Max.   :3469446
##
```

Обратите внимание, что строки, интервальные и номинальные (факторы) переменные обрабатываются функцией `summary()` по-разному.

3.7 Сохранение таблиц

Одной из завершающих стадий анализа данных, помимо графиков и отчетов, часто являются новые табличные представления, которые было бы неплохо сохранить в виде файлов. К счастью, сохранение таблиц в **R** столь же просто, как и чтение. Для текстовых файлов в формате **CSV** можно использовать функции `write.table()`, `write.csv()` и `write.csv2()`. Для файлов **Microsoft Excel** используйте функцию `write.xlsx()` из пакета `openxlsx` соответственно.

> **По умолчанию функции `write.table()`, `write.csv()` и `write.csv2()` записывают в таблицы в качестве первого столбца названия (номера) строк таблиц.** Если вы не хотите, чтобы это происходило, укажите дополнительный параметр `row.names=FALSE`.

Сохраним таблицы `okruga` и `neokruga`, раздельно хранящие статистику по объему сброса сточных в поверхностные водные объекты по федеральным округам и субъектам соответственно:

```
write.csv2(okruga, "okruga.csv", fileEncoding = 'UTF-8') #                CSV        Unicode
write.xlsx(neokruga, "neokruga.xlsx") #                XLSX

#     ,                          :
```

```
okruga.saved <- read.csv2("okruga.csv", encoding = 'UTF-8')
head(okruga.saved)
##    X                              Region Year05 Year10 Year11 Year12
## 1  2                              4341   3761   3613   3651
## 2 21      -                      3192   3088   2866   2877
## 3 35                                 1409   1446   1436   1394
## 4 42    -                        496    390    397    395
## 5 49                               3162   2883   2857   2854
## 6 64                                 1681   1860   1834   1665
##   Year13
## 1   3570
## 2   2796
## 3   1321
## 4    374
## 5   2849
## 6   1624

neokruga.saved <- read.xlsx("neokruga.xlsx",1)
head(neokruga.saved)
##                  Region Year05 Year10 Year11 Year12 Year13
## 1                  11     77     72     71     71
## 2                    89     78     75     71     68
## 3                 155    129    126    124    120
## 4                 169    134    135    131    129
## 5                 144    102     99     97     88
## 6                   99     92     88     84     93
```

Видно, что в файле **CSV** присутствует также дополнительный столбец с названиями строк, а в файле **XLSX** его нет. Если вы не задавали названия строк явным образом и они не несут какого-то смысла, всегда указывайте параметр `row.names=FALSE`

> **Вы можете дать строкам таблицы названия и извлечь их, используя функцию `row.names()`** аналогично функции `colnames()` для столбцов.

3.8 Обработка таблиц в dplyr

3.9 Правила подготовки таблиц для чтения в R

С таблицами, которые мы использовали в настоящем модуле, все прошло гладко, поскольку они были подготовлены специальным образом. Несмотря на то, что каких-то четких правил подготовки таблиц для программной обработки не существует, можно дать несколько полезных рекомендаций по данному поводу:

1. В первой строке таблицы должны располагаться названия столбцов.
2. Во второй строке таблицы должны начинаться данные. Не допускайте многострочных заголовков.
3. В названиях столбцов недопустимы объединенные ячейки, покрывающие несколько столбцов. Это может привести к неверному подсчету количества столбцов и, как следствие, некорректному чтению таблицы в целом.
4. Названия столбцов должны состоять из латинских букв и цифр, начинаться с буквы и не содержать пробелов. Сложносочиненные названия выделяйте прописными буквами. Плохое название столбца: 2015 . Хорошее название столбца: GDP2015.
5. Некоторые ошибки данных в таблицах (такие как неверные десятичные разделители), проще найти и исправить в табличном/текстовом редакторе, нежели после загрузки в **R**.

Следование этим правилам значительно облегчит работу с табличными данными.

3.10 Контрольные вопросы и задачи

3.10.1 Вопросы

3.10.2 Задачи

Chapter 4

Основы графики в R

Программный код главы

Данный модуль посвящен введению в работу с графическим представлением информации в R. Построение графиков на языке R сродни работе с конструктором: вы собираете изображение по кирпичикам из множества настроек и компонент. Поняв основные принципы базовой графической подсистемы **R** из пакета **graphics**, вы сможете освоить дополнительные библиотеки **lattice**, **ggplot2** и **plotly**, предоставляющие еще более интересные возможности с точки зрения функциональности и дизайна.

Прежде чем мы приступим к построению графиков, необходимо подготовить исходные данные. Мы будем работать с теми же таблицами, что и в предыдцщей лекции: экспорт/импорт продукции по регионам России (млн долл. США) и объем сброса сточных вод по морям России (млрд м^3). На этот раз мы воспользуемся пакетом readxl, который возволяет задать типы столбцов при чтении:

```
library(readxl)

#                /
types <- c("text", rep("numeric", 12))
tab <- as.data.frame(read_excel("ExpImp.xlsx", 1, col_types = types))
## Warning in strptime(x, format, tz = tz): unknown timezone 'default/Europe/
## Moscow'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in E10 / R10C5: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D11 / R11C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D14 / R14C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D17 / R17C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in B27 / R27C2: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in C27 / R27C3: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D27 / R27C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in E27 / R27C5: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in F27 / R27C6: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
```

```
## shim, : Expecting numeric in H27 / R27C8: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in J27 / R27C10: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D37 / R37C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D38 / R38C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in E38 / R38C5: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in F38 / R38C6: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in H38 / R38C8: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in J38 / R38C10: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in F45 / R45C6: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in H45 / R45C8: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in J45 / R45C10: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in L45 / R45C12: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D46 / R46C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D47 / R47C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D49 / R49C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in E49 / R49C5: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D54 / R54C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D59 / R59C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in E72 / R72C5: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in H72 / R72C8: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in E76 / R76C5: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in B78 / R78C2: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in C78 / R78C3: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D78 / R78C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in E78 / R78C5: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in F78 / R78C6: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in H78 / R78C8: got '-'
```

```
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in J78 / R78C10: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in L78 / R78C12: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in H90 / R90C8: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D94 / R94C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in H94 / R94C8: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D96 / R96C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in F96 / R96C6: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in J96 / R96C10: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in D97 / R97C4: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in F97 / R97C6: got '-'
## Warning in read_fun(path = path, sheet = sheet, limits = limits, shim =
## shim, : Expecting numeric in H97 / R97C8: got '-'
str(tab)
## 'data.frame':    96 obs. of  13 variables:
##  $         : chr  "           " "                " "           " "         " ...
##  $       : num  16196.2 4552.9 221.9 28.5 177.9 ...
##  $       : num  43076 22954 614 650 454 ...
##  $       : num  371791.8 204331.7 64.7 5 0.9 ...
##  $       : num  3613.6 1660.3 24.1 20.5 16.7 ...
##  $       : num  30739.2 8442.7 33.3 24.5 87.7 ...
##  $       : num  50129.5 34870.4 242.8 71.7 419 ...
##  $       : num  10965.8 1101.6 6.2 23.4 57.1 ...
##  $       : num  6641.5 3942.6 43.3 44.9 9 ...
##  $       : num  40859.3 9877 2014.1 50.5 29 ...
##  $       : num  22017.4 11763.6 1207.3 68.3 56.6 ...
##  $       : num  28338.5 12845.9 84.1 143.2 286.1 ...
##  $       : num  154371 96196 1710 823 469 ...
```

```
#
filter <- grep("         ", tab$   )
okr <- tab[filter, ]

#                                    :
okr <- okr[order(okr$   ), ]
```

```
View(okr)
```

Show 5 entries Search:

	Регион	ПродЭкспорт	ПродИмпорт	ТЭКЭкспорт	ТЭКИмпорт	ХимЭкспорт	ХимИмпорт
87	Дальневосточный федеральный округ	2440.3	1138.9	18872.1	158	49.2	985.4
50	Приволжский федеральный округ	836.8	978	45600.8	475.1	11688.4	3321.7
21	Северо-Западный федеральный округ	2485.2	12527	29969.4	332.7	4104.8	6002.8
42	Северо-Кавказский федеральный округ	283	584.1	22.9	12	759.5	141.8
74	Сибирский федеральный округ	593.2	722.5	14473	505.6	2462.8	2534

Showing 1 to 5 of 8 entries Previous 1 2 Next

```
#
filter <- grepl("          |     |   ",tab$    )
sub <- tab[!filter, ]
```

```
View(sub)
```

Show 5 entries Search:

	Регион	ПродЭкспорт	ПродИмпорт	ТЭКЭкспорт	ТЭКИмпорт	ХимЭкспорт	ХимИмпорт	Д
3	Белгородская область	221.9	613.8	64.7	24.1	33.3	242.8	
4	Брянская область	28.5	650.3	5	20.5	24.5	71.7	
5	Владимирская область	177.9	454.3	0.9	16.7	87.7	419	
6	Воронежская область	373.7	181.9	38.9	38	976.9	84.9	
7	Ивановская область	3.1	105.5	0.1	0.1	40.8	57.4	

Showing 1 to 5 of 85 entries Previous 1 2 3 4 5 … 17 Next

4.1 Столбчатые графики

Столбчатые графики — **barplot** — отображают вектор числовых данных в виде столбиков. Это простейший вид графика (наряду с *dotchart*), который используется для сравнения асболютных величин. Для построения достаточно вызвать функцию `barplot()` и передать ей столбец таблицы:

```
barplot(okr$      )
```