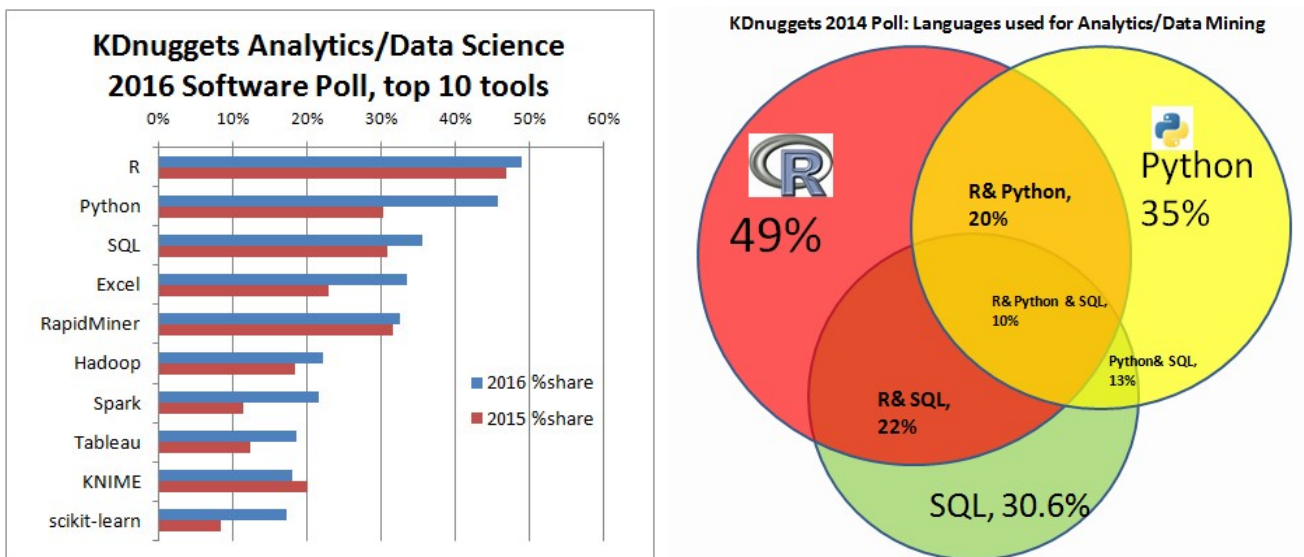


Лекция 2–3 Программная среда анализа данных и язык программирования R

Почему R? Сравнение R и Python. Обзор и история R. Консоль R. Интегрированная среда разработки RStudio. Объекты и атрибуты. Пакеты и библиотеки. Типы данных: векторы, матрицы, факторы, списки, блоки данных (data frames). Чтение и запись данных. Форматы данных. Представление даты и времени; временные ряды. Организация вычислений: функции, ветвления, циклы. Векторизованные вычисления в R

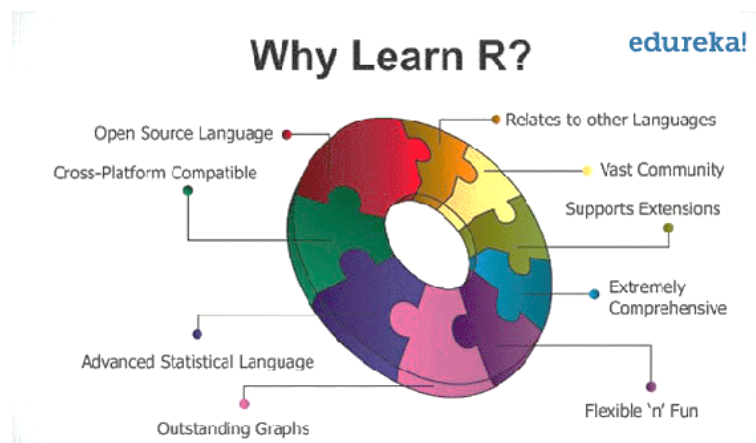
Почему R? Сравнение R и Python



См. также: Data Science Wars: R vs Python

<https://www.datacamp.com/community/tutorials/r-or-python-for-data-analysis#gs.bs1qDmY>

Обзор и история R



R — это среда для статистических расчетов. R задумывался как свободный аналог среды S-Plus, которая, в свою очередь, является коммерческой реализацией языка расчетов S. Язык S — довольно старая разработка. Он возник еще в 1976 году в компании Bell Labs и был назван, естественно, «по мотивам» языка C. Первая реализация S была написана на FORTRAN и работала под управлением операционной системы GCOS. В 1980 г. реализация была переписана под UNIX, и с этого момента S стал распространяться в основном в научной среде.

Начиная с третьей версии (1988 г.), коммерческая реализация S называется S-Plus. Последняя распространялась компанией Insightful, а сейчас распространяется компанией TIBCO Software. Версии S-Plus доступны под Windows и различные версии UNIX — естественно, за плату, причем весьма и весьма немаленькую (версия для UNIX стоит порядка \$6500). Собственно, высокая цена и сдерживала широкое распространение этого во многих отношениях замечательного продукта. Тут-то и начинается история R.



Robert Gentleman и Ross Ihaka

В августе 1993 г. двое молодых новозеландских ученых анонсировали свою новую разработку, которую они называли R (буква «R» была выбрана просто потому, что она стоит перед «S», тут есть аналогия с языком программирования C, которому предшествовал язык B). По замыслу создателей (это были Robert Gentleman и Ross Ihaka), это должна была быть новая реализация языка S,

отличающаяся от S-Plus некоторыми деталями, например обращением с глобальными и локальными переменными, а также работой с памятью. Фактически они создали не аналог S-Plus, а новую «ветку» на «дереве S» (многие вещи, которые отличают R от S-Plus, связаны с влиянием языка Scheme). Проект вначале развивался довольно медленно, но когда в нем появилось достаточно возможностей, в том числе уникальная по легкости система написания дополнений (пакетов), все большее количество людей стало переходить на R с S-Plus. Когда же, наконец, были устранены свойственные первым версиям проблемы работы с памятью, на R стали переходить и «любители» других статистических пакетов (прежде всего тех, которые имеют интерфейс командной строки: SAS, Stata, SYSTAT). Количество книг, написанных про R, за последние годы выросло в несколько раз, а количество пакетов в начале 2017 было более 12 тысяч.

Применение, преимущества и недостатки R. Коротко говоря, R применяется везде, где нужна работа с данными. Это не только статистика в узком смысле слова, но и «первичный» анализ (графики, таблицы сопряженности) и продвинутое математическое моделирование. В принципе, R может использоваться и там, где в настоящее время принято использовать специализированные программы математического анализа, такие как MATLAB или Octave. Но, разумеется, более всего его применяют для статистического анализа — от вычисления средних величин до вейвлет-преобразований и временных рядов. Географически R распространен тоже очень широко. Трудно найти американский или западноевропейский университет, где бы не работали с R. Очень многие серьезные компании (скажем, Boeing) устанавливают R для работы.

У R два главных преимущества: невероятная гибкость и свободный код. Гибкость позволяет создавать приложения (пакеты) практически на любой случай жизни. Нет, кажется, ни одного метода современного статистического анализа, который бы не был сейчас представлен в R. Свободный код — это не просто бесплатность программы (хотя в сравнении с коммерческими пакетами, продающимися за совершенно безумные деньги, это, конечно, преимущество, да еще какое!), но и возможность разобраться, как именно происходит анализ, а если в коде встретилась ошибка — самостоятельно исправить ее и сделать исправление доступным для всех.

У R есть и немало недостатков. Самый главный из них — это трудность обучения программе. Команд много, вводить их надо вручную, запомнить все трудно, а привычной системы меню нет. Поэтому порой очень трудно найти, как именно сделать какой-нибудь анализ. Если функция известна, то узнать, что она делает, очень легко, обычно достаточно набрать команду `help(название функции)`. Увидеть код функции тоже легко, для этого надо просто набрать ее название без скобок или (лучше) ввести команду `getAnywhere(название функции)`.

Не стоит забывать, однако, что сила R — там же, где его слабость. Интерфейс командной строки позволяет делать такие вещи, которых рядовой пользователь других статистических программ может достичь только часами ручного труда.

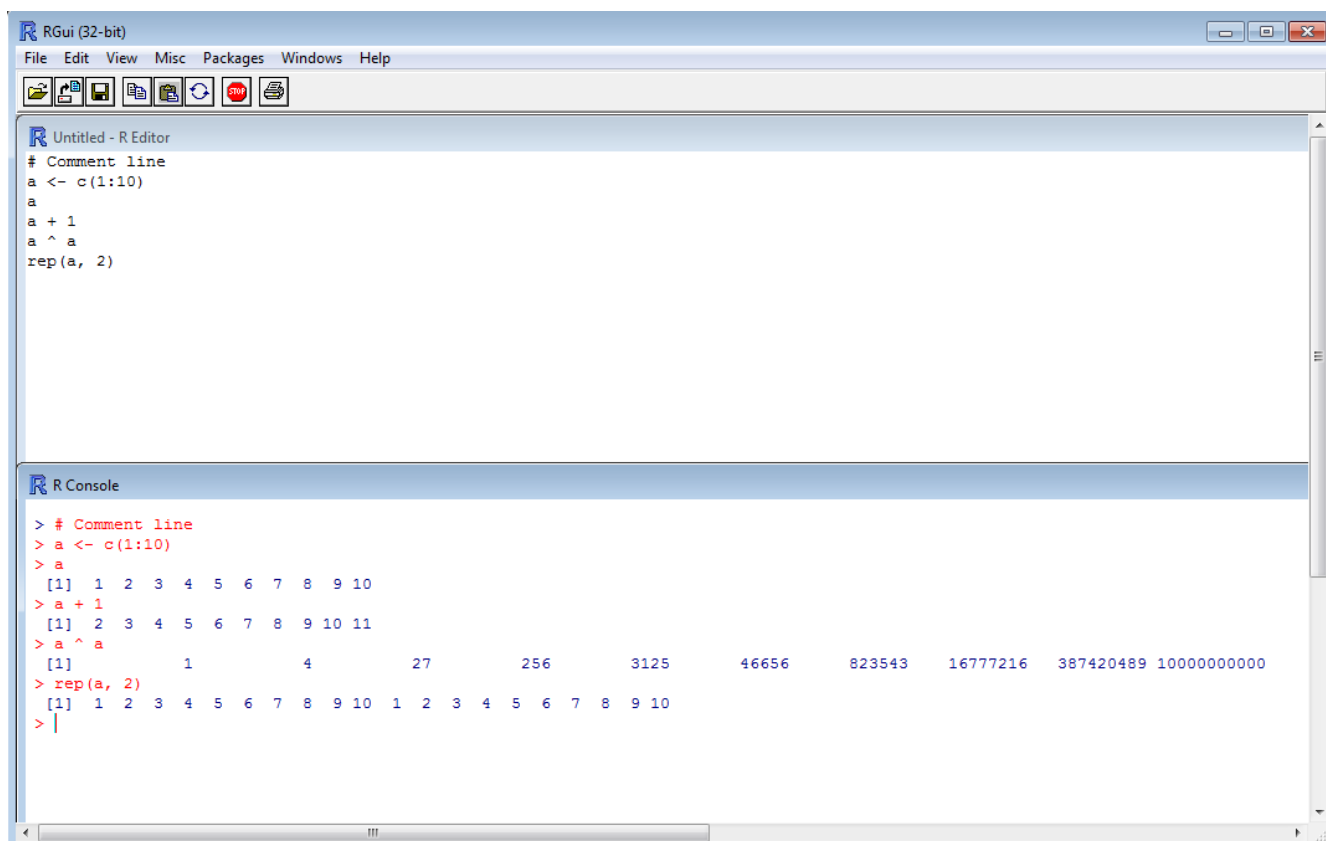
Вот, например, простая задача: требуется превратить выборку, состоящую из цифр от 1 до 9, в таблицу из трех колонок (допустим, это были данные за три дня, и каждый день делалось три измерения). Чтобы сделать это в программе с визуальным интерфейсом, скажем в STATISTICA, требуется: (1) учредить две новые переменные, (2–3) скопировать дважды кусок выборки в буфер, (4–5) скопировать его в одну и другую переменную и (6) уничтожить лишние строки. В R это делается одной командой:

```
> b <- matrix(1:9, ncol=3)
```

Второй недостаток R — относительная медлительность. Некоторые функции, особенно использующие циклы, и виды объектов, особенно списки и таблицы данных, «работают» в десятки раз медленнее, чем их аналоги в коммерческих пакетах. Но этот недостаток преодолевается, хотя и медленно. Новые версии R «умеют» делать параллельные вычисления, создаются оптимизированные варианты подпрограмм, работающие много быстрее, память в R используется все эффективнее, а вместо циклов рекомендуется применять векторизованные вычисления.

Консоль R

Статистическая среда R выполняет любой набор осмысленных инструкций языка R, содержащихся в файле скрипта или представленных последовательностью команд, задаваемых с консоли. Работа с консолью может показаться трудной для современных пользователей, привыкших к кнопочным меню, поскольку надо запоминать синтаксис отдельных команд. Однако, после приобретения некоторых навыков, оказывается, что многие процедуры обработки данных можно выполнять быстрее и с меньшим трудом, чем в том же пакете Statistica.



```
RGui (32-bit)
File Edit View Misc Packages Windows Help

Untitled - R Editor
# Comment line
a <- c(1:10)
a
a + 1
a ^ a
rep(a, 2)

R Console
> # Comment line
> a <- c(1:10)
> a
[1] 1 2 3 4 5 6 7 8 9 10
> a + 1
[1] 2 3 4 5 6 7 8 9 10 11
> a ^ a
[1] 1 4 27 256 3125 46656 823543 16777216 387420489 10000000000
> rep(a, 2)
[1] 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
> |
```

Консоль R представляет собой диалоговое окно, в котором пользователь вводит команды и где видит результаты их выполнения. Это окно возникает сразу при запуске среды (например, после клика мышью на ярлыке R на рабочем столе). Кроме того, стандартный графический пользовательский интерфейс R (RGui) включает окно редактирования скриптов и всплывающие окна с графической информацией (рисунками, диаграммами и проч.) В командном режиме R может работать, например, как обычный калькулятор.

Справа от символа приглашения `>` пользователь может ввести произвольное арифметическое выражение, нажать клавишу Enter и тут же получить результат.

При работе с использованием RGui рекомендуется во всех случаях создавать файл со скриптом (т.е. последовательностью команд языка R, выполняющей определенные действия). Как правило, это обычный текстовый файл с любым именем (но, для определенности, лучше с расширением *.R), который можно создавать и редактировать обычным редактором типа "Блокнот".

Выполнить последовательность команд скрипта можно из пункта меню "Правка > Запустить все". Можно также выделить мышью осмысленный фрагмент из любого места подготовленного скрипта (от имени одной переменной до всего содержимого) и осуществить запуск этого блока на выполнение. Это можно сделать четырьмя возможными способами:

- из основного
- или контекстного меню,
- комбинацией клавиш Ctrl+R
- или кнопкой на панели инструментов.

Таким образом, Редактор R позволяет легко выполнить навигацию по скрипту, редактирование и выполнение любой комбинации команд, поиск и замену определенных частей кода.

R обладает встроенными обширными справочными материалами, которые можно получить непосредственно в RGui. Если подать с консоли команду `help.start()`, то в вашем интернет-браузере откроется страница, открывающая доступ ко всем справочным ресурсам:

- основным руководствам,
- авторским материалам,
- ответам на вероятные вопросы,
- спискам изменений,
- ссылкам на справки по другим объектам R
- и т.д.



Manuals

[An Introduction to R](#)
[Writing R Extensions](#)
[R Data Import/Export](#)

[The R Language Definition](#)
[R Installation and Administration](#)
[R Internals](#)

Reference

[Packages](#)

[Search Engine & Keywords](#)

Miscellaneous Material

[About R](#)
[License](#)
[NEWS](#)

[Authors](#)
[Frequently Asked Questions](#)
[User Manuals](#)

[Resources](#)
[Thanks](#)
[Technical papers](#)

Material specific to the Windows port

[CHANGES up to R 2.15.0](#)

[Windows FAQ](#)

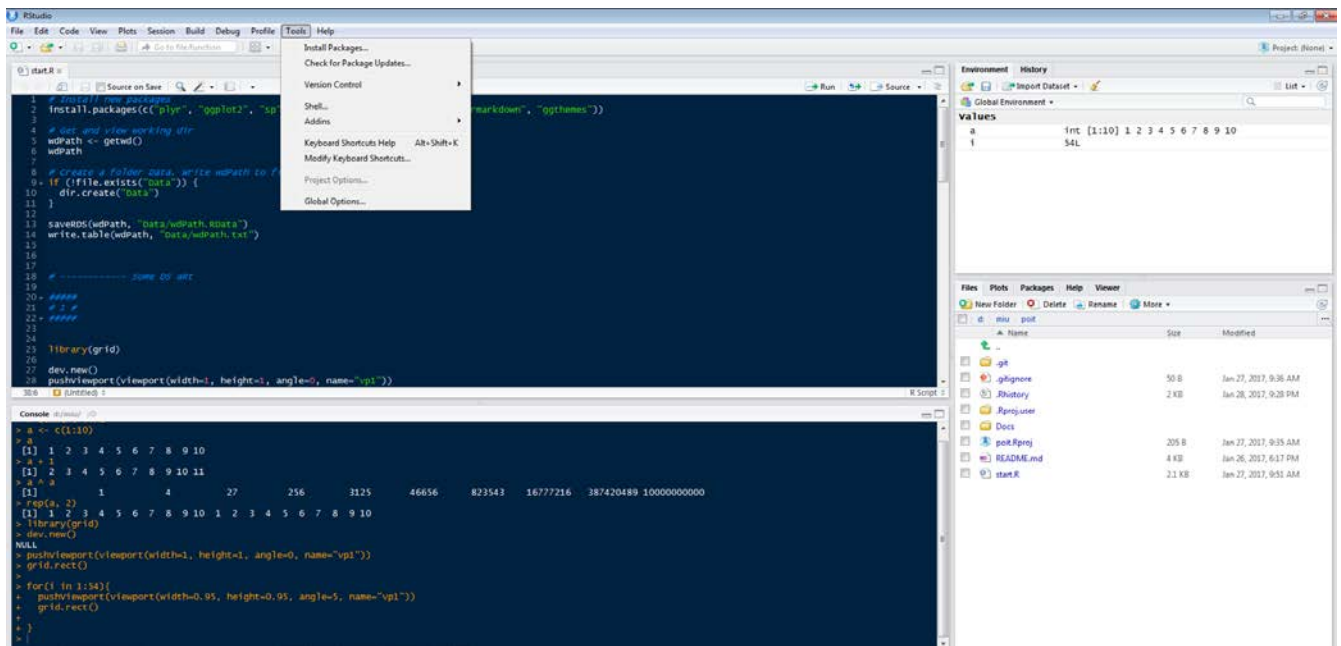
Справку по отдельным функциям можно получить с использованием следующих команд:

- `help("foo")` или `? foo` – справка по функции `foo` (кавычки необязательны);
- `help.search("foo")` или `?? foo` – поиск всех справочных файлов, содержащих `foo`;
- `example("foo")` – примеры использования функции `foo`;
- `RSiteSearch("foo")` – поиск ссылок в онлайн-руководствах и архивах рассылок;
- `apropos("foo", mode="function")` – список всех функций с комбинацией `foo`;
- `vignette("foo")` – список руководств по теме `foo`.

Интегрированная среда разработки RStudio

An integrated development environment (IDE) RStudio позволяет дополнительно выполнять подсветку синтаксиса кода, его автоматическое завершение, "упаковку" последовательности команды в функции для их последующего использования, работу с документами Sweave или TeX и другие операции.

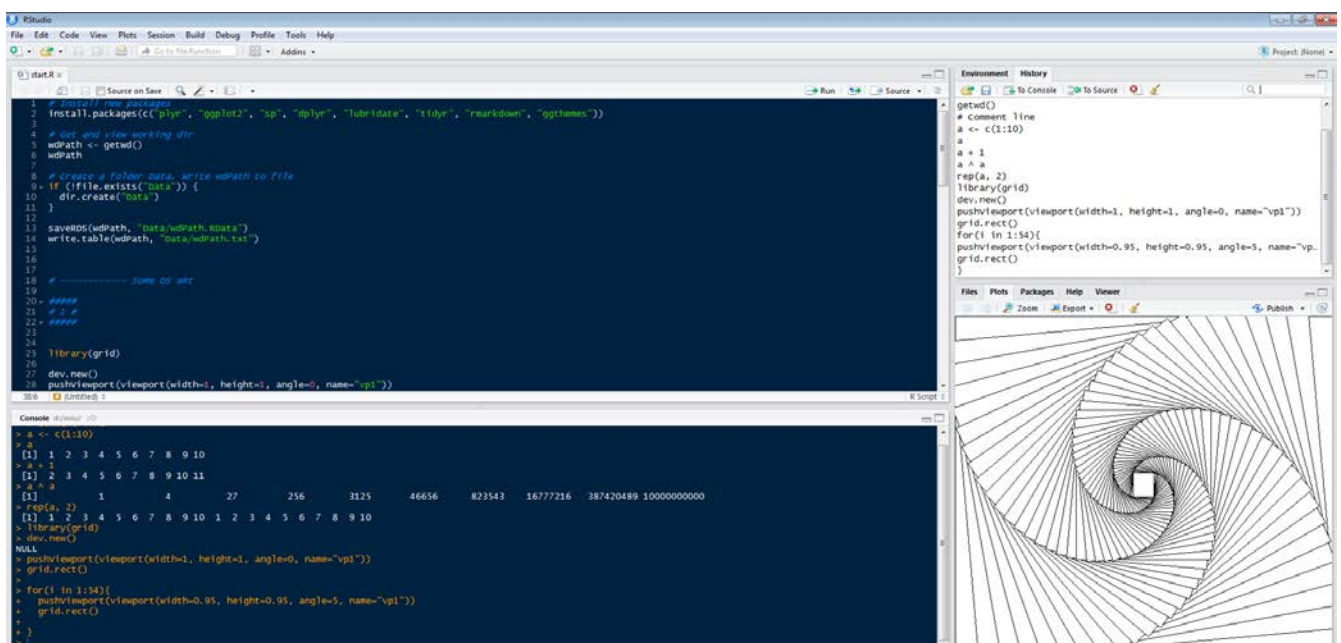
RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux) or in a browser connected to RStudio Server or RStudio Server Pro (Debian/Ubuntu, RedHat/CentOS, and SUSE Linux).



RStudio IDE features:

- runs on most desktops or on a server and accessed over the web;
- integrates the tools you use with R into a single environment;
- includes powerful coding tools designed to enhance your productivity;
- enables rapid navigation to files and functions;
- makes it easy to start new or find existing projects;
- has integrated support for Git and Subversion;
- supports authoring HTML, PDF, Word Documents, and slide shows;
- supports interactive graphics with Shiny and ggvis.

<https://www.rstudio.com/products/rstudio/>



Объекты и атрибуты. Пакеты и библиотеки

Выделяют два основных типа объектов:

1. Объекты, предназначенные для хранения данных ("data objects") – отдельные переменные, векторы, матрицы и массивы, списки, факторы, таблицы данных.
2. Функции ("function objects") – поименованные программы, предназначенные для создания новых объектов или выполнения определенных действий над ними.

Любой объект языка R имеет набор атрибутов (attributes). Этот набор может быть разным для объектов разного вида, но каждый объект обязательно имеет пять встроенных атрибутов:

- длина (length);
- вид (mode);
- класс (class);
- тип (typeof);
- структура (str).

Можно написать функцию, которая будет возвращать значения атрибутов какого-либо объекта:

```
inspect <- function(a) {list(.length = length(a), .mode =  
mode(a), .type = typeof(a), .class = class(a), .str =  
str(a))}
```

Объекты среды R, предназначенные для коллективного и свободного использования, комплектуются в пакеты, объединяемые сходной тематикой или методами обработки данных. Есть некоторое отличие между терминами пакет ("package") и библиотека ("library"). Термин "library" определяет директорию, которая может содержать один или несколько пакетов. Термин "package" обозначает совокупность функций, HTML-страниц руководств и примеров объектов данных, предназначенных для тестирования или обучения.

Пакеты устанавливаются в определенной директории операционной системы или, в неустановленном виде, могут храниться и распространяться в архивных *.zip файлах Windows (версия пакета должна корреспондироваться с конкретной версией вашей R).

Полная информация о пакете (версия, основное тематическое направление, авторы, даты изменений, лицензии, другие функционально связанные пакеты, полный список функций с указанием на их назначение и проч.) может быть получена командой `library(help=<имя_пакета>)`, например:


```
library(help=Matrix)
```

Все пакеты R относятся к одной из трех категорий: базовые ("base"), рекомендуемые ("recommended") и прочие, установленные пользователем. Получить их список на конкретном компьютере можно, подав команду `library()` или:

```
installed.packages(priority = "base")
```

```
installed.packages(priority = "recommended")
```

```
# Получение полного списка пакетов
```

```
packlist <- rownames(installed.packages())
```

```
# Вывод информации в буфер обмена в формате для Excel
```

```
write.table(packlist,"clipboard",sep="\t", col.names=NA)
```

Базовые и рекомендуемые пакеты обычно включаются в инсталляционный файл R. Разумеется, нет необходимости сразу устанавливать "про запас" много разных пакетов. Для установки пакета достаточно в командном окне R Console выбрать пункт меню "Пакеты > Установить пакет(ы)" или ввести, например, команду:

```
install.packages(c("vegan", "xlsReadWrite", "car"))
```

При запуске консоли RGui загружаются только некоторые базовые пакеты. Для инициализации любого другого пакета перед непосредственным использованием его функций нужно ввести команду `library(<имя_пакета>)`.

Установить, какие пакеты загружены в каждый момент проводимой сессии, можно, подав команду:

```
sessionInfo()
```

Получить список аргументов входящих параметров любой функции загруженного пакета можно, подав команду `args()`. Например, при запуске функции получения линейной модели `lm()` задают параметры:

```
> args(lm)
```

```
function (formula, data, subset, weights, na.action,
method = "qr", model = TRUE, x = FALSE, y = FALSE,
qr = TRUE, singular.ok = TRUE, contrasts = NULL,
offset,...)
```

Если ввести команду, состоящую только из аббревиатуры функции (например, вычисляющей межквартильный размах IQR), то можно получить исходный текст функции в кодах языка R:

```
> IQR  
  
function (x, na.rm = FALSE)  
  
diff(quantile(as.numeric(x), c(0.25, 0.75), na.rm = na.rm,  
names = FALSE))
```

Типы данных: векторы, матрицы, факторы, списки, блоки данных (data frames)

Все объекты данных (а, следовательно, и переменные) в R можно разделить на следующие классы (т.е. типы объектов):

- `numeric` – объекты, к которым относятся целочисленные (`integer`) и действительные числа (`double`);
- `complex` – содержат комплексные числа, мнимая часть комплексного числа записывается с символом `i` на конце, например, `c(5.0i, -1.3+8.73i, 2.0)`;
- `logical` – логические объекты, которые принимают только два значения: `FALSE` и `TRUE`;
- `character` – символьные объекты (значения переменных задаются в двойных, либо одинарных кавычках).

В R можно создавать имена для различных объектов (функций или переменных) как на латинице, так и на кириллице (нежелательно), но следует учесть, что `а` (кириллица) и `a` (латиница) – это два разных объекта. Кроме того, среда R чувствительна к регистру, т.е. строчные и заглавные буквы в ней различаются. Имена переменных (идентификаторы) в R должны начинаться с буквы (или точки) и состоять из букв, цифр, знаков точки и подчёркивания.

При помощи команды `exists("<имя>")` можно проверить, существует ли переменная или функция с указанным именем.

Проверка на принадлежность переменной к определенному классу проверяется функциями `is.numeric(<имя_объекта>)`, `is.integer(<имя>)`, `is.logical(<имя>)`, `is.complex(<имя>)`, `is.character(<имя>)` и т.п., а для преобразования объекта в другой тип можно использовать функции `as.numeric(<имя>)`, `as.integer(<имя>)` и т.п., если это возможно.

В R существует ряд специальных объектов:

- Inf – положительная или отрицательная бесконечность (обычно результат деления вещественного числа на 0);
- NA – "отсутствующее значение" (Not Available);
- NaN – "не число" (Not a Number).

Проверить, относится ли переменная к какому-либо из этих специальных типов, можно, соответственно, функциями `is.finite(<имя>)`, `is.na(<имя>)` и `is.nan(<имя>)`.

Выражение (expression) языка R представляет собой сочетание таких элементов, как оператор присваивания, арифметические или логические операторы, имена объектов и имена функций. Результат выполнения выражения, как правило, сразу отображается в командном или графическом окне. Однако при выполнении операции присваивания результат сохраняется в соответствующем объекте и на экран не выводится.

В качестве оператора присваивания в R можно использовать либо символ "=", либо пару символов "<-" (присваивание определенного значения объекту слева) или "->" (присваивание значения объекту справа). Хорошим стилем программирования считается использование "<-".

When the R language (and S before it) was first created, <- was the only choice of assignment operator. This is a hangover from the language APL, where the arrow notation was used to distinguish assignment (assign the value 3 to x) from equality (is x equal to 3?). (On APL keyboards there was an actual key with the arrow symbol on it, so the arrow was a single keystroke back then. The same was true of the AT&T terminals first used for the predecessors of S as described in the Blue Book.) However many modern languages (such as C, for example) use = for assignment, so beginners using R often found the arrow notation cumbersome, and were prone to use = by mistake. But R uses = for yet another purpose: associating function arguments with values.

To make things easier for new users, R added the capability in 2001 to *also* allow = be used as an assignment operator, on the basis that the intent (assignment or association) is usually clear by context.

So, should you use = or <- for assignment? It really boils down to preference. Many people are more used to using = for assignment, and it's one less keystroke if you want to save on typing. On the other hand, many R traditionalists prefer <- for clarity, and if you plan to share or publish your code, other might find code using = for assignment hard to read.

<http://blog.revolutionanalytics.com/2008/12/use-equals-or-arrow-for-assignment.html>

Выражения языка R организуются в скрипте по строкам. В одной строке можно ввести несколько команд, разделяя их символом ";". Одну команду можно также расположить на двух (и более) строках.

Объекты типа numeric могут составлять выражения с использованием традиционных арифметических операций + (сложение), - (вычитание), * (умножение), / (деление), ^ (возведение в степень), %/% (целочисленное деление), %% (остаток от деления). Операции имеют обычный приоритет, т.е. сначала

выполняется возведение в степень, затем умножение или деление, потом уже сложение или вычитание. В выражениях могут использоваться круглые скобки и операции в них имеют наибольший приоритет.

Логические выражения могут составляться с использованием следующих логических операторов:

- "Равно" ==
- "Не равно" !=
- "Меньше" <
- "Больше" >
- "Меньше либо равно" <=
- "Больше либо равно" >=
- "Логическое И" & (&& – сравнивает только первые элементы)
- "Логическое ИЛИ" |
- "Логическое НЕ" !

```
> fr <- c(TRUE, TRUE, FALSE)
```

```
> sc <- c(TRUE, TRUE, TRUE)
```

```
> th <- c(FALSE, TRUE, TRUE)
```

```
> fr & sc == TRUE
```

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

```
> fr && sc == TRUE
```

```
[1] TRUE
```

```
> fr && th == TRUE
```

```
[1] FALSE
```

Векторы. Вектор представляет собой поименованный одномерный объект, содержащий набор однотипных элементов (числовые, логические, либо текстовые значения – никакие их сочетания не допускаются). Для создания векторов небольшой длины в R используется функция конкатенации `c()` (от "concatenate" – объединять, связывать). В качестве аргументов этой функции через запятую перечисляют объединяемые в вектор значения, например:

```
> my.vector <- c(1, 2, 3, 4, 5)
```

```
> my.vector
```

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

Вектор можно создать также при помощи функции `scan()`, которая "считывает" последовательно вводимые с клавиатуры значения:

```
X <- scan()

1: 2.9 # после каждого нового значения нажать клавишу "Ввод"
2: 3.1
3: 3.4
4: 3.4
5: 3.7
6: 3.7
7: 2.8
8: 2.5

9: # выполнение команды scan завершают введением пустой строки

Read 8 items # программа сообщает о считывании 8 значений

X

[1] 2.9 3.1 3.4 3.4 3.7 3.7 2.8 2.5
```

Один из недостатков создания векторов при помощи функции `scan()` состоит в том, что если при вводе значений с клавиатуры допущена ошибка, то приходится, либо начать ввод заново, либо воспользоваться специальными инструментами корректировки (например, функцией `fix()`).

Для создания векторов, содержащих последовательную совокупность чисел, удобна функция `seq()` (от "sequence" – последовательность). Так, вектор с именем `S`, содержащий совокупность целых чисел от 1 до 7, можно создать следующим образом:

```
S <- seq(1,7)

S

[1] 1 2 3 4 5 6 7
```

Идентичный результат будет получен при помощи команды

```
S <- 1:7
```

В качестве дополнительного аргумента функции `seq()` можно задать шаг приращения чисел:

```
S <- seq(from = 1, to = 5, by = 0.5)
```

```
S
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Векторы, содержащие одинаковые значения, создают при помощи функции `rep()` (от "repeat" – повторять). Например, для формирования текстового вектора `Text`, содержащего пять значений "test", следует выполнить команду

```
Text <- rep("test", 5)
```

```
Text
```

```
[1] "test" "test" "test" "test" "test"
```

Система R способна выполнять самые разнообразные операции над векторами. Так, несколько векторов можно объединить в один, используя уже рассмотренную выше функцию конкатенации:

```
v1 <- c(1, 2, 3)
```

```
v2 <- c(4, 5, 6)
```

```
V <- c(v1, v2)
```

```
V
```

```
[1] 1 2 3 4 5 6
```

Если попытаться объединить, например, текстовый вектор с числовым, сообщение об ошибке не появится – программа просто преобразует все значения в текстовые:

```
text.vect <- c("a", "b", "c")
```

```
new.vect <- c(v1, text.vect)
```

```
new.vect
```

```
[1] "1" "2" "3" "a" "b" "c"
```

```
mode(new.vect)
```

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


Для работы с определенным элементом вектора необходимо иметь способ отличать его от других элементов. Для этого при создании вектора всем его компонентам автоматически присваиваются индексные номера, начиная с 1. Чтобы обратиться к конкретному элементу необходимо указать имя вектора и индекс этого элемента в квадратных скобках:

```
y <- c(5, 3, 2, 6, 1)
y[3]
[1] 2
```

Используя индексные номера, можно выполнять различные операции с избранными элементами разных векторов:

```
z <- c(0.5, 0.1, 0.6)
y[1]*z[3]
[1] 3
```

Индексирование является мощным инструментом, позволяющим создавать совокупности значений в соответствии с определенными критериями. Для вывода на экран 3-го, 4-го и 5-го значений вектора у необходимо выполнить команду

```
y[3:5]
[1] 2 6 1
```

Из этого же вектора мы можем выбрать, например, только первое и четвертое значения, используя уже известную нам функцию конкатенации `c()`:

```
y[c(1, 4)]
[1] 5 6
```

Похожим образом мы можем удалить первое и четвертое значения из вектора у, применив знак "минус" перед функцией конкатенации:

```
y[-c(1, 4)]
[1] 3 2 1
```

В качестве критерия для выбора значений может служить логическое выражение. Для примера выберем из вектора у все значения > 2 :

```
y[y > 2]
[1] 5 3 6
```

Индексирование является также удобным инструментом для внесения исправлений в имеющихся векторах. Например, так можно исправить второе значение созданного нами ранее вектора z с 0.1 на 0.3:

```
z[2] <- 0.3
z
[1] 0.5 0.3 0.6
```

Для упорядочения значений вектора по возрастанию или убыванию используют функцию `sort()` в сочетании с аргументом `decreasing = FALSE` или `decreasing = TRUE` соответственно ("decreasing" значит "убывающий"):

```
sort(z)
# по умолчанию decreasing = FALSE
[1] 0.3 0.5 0.6
sort(z, decreasing = TRUE)
[1] 0.6 0.5 0.3
```

Матрицы. Матрица представляет собой двумерный вектор. В R для создания матриц служит одноименная функция:

```
my.mat <- matrix(seq(1, 16), nrow = 4, ncol = 4)
my.mat
      [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

Обратите внимание на то, что по умолчанию заполнение матрицы происходит по столбцам, т.е. первые четыре значения входят в первый столбец, следующие четыре значения – во второй столбец, и т.д. Такой порядок заполнения можно изменить, придав специальному аргументу `byrow` (от “by row” – по строкам) значение `TRUE`:

```
my.mat <- matrix(seq(1, 16), nrow = 4, ncol = 4, byrow = TRUE)
my.mat
      [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
```

В качестве заголовков строк и столбцов создаваемой матрицы автоматически выводятся соответствующие индексные номера (строки: `[1,]`, `[2,]`, и т.д.; столбцы: `[,1]`, `[,2]`, и т.д.). Для придания пользовательских заголовков строкам и столбцам

матриц используют функции `rownames()` и `colnames()` соответственно. Например, для обозначения строк матрицы `my.mat` буквами A, B, C и D необходимо выполнить следующее:

```
rownames(my.mat) <- c("A", "B", "C", "D")
```

```
my.mat
```

	[,1]	[,2]	[,3]	[,4]
A	1	2	3	4
B	5	6	7	8
C	9	10	11	12
D	13	14	15	16

В матрице `my.mat` имеется 16 значений, которые как раз вмещаются в имеющиеся четыре строки и четыре столбца. Но что произойдет, если, например, попытаться вместить вектор из 12 чисел в матрицу того же размера? В подобных случаях R заполняет недостающие значения за счет "зацикливания" (recycling) короткого вектора. Вот как это выглядит на примере:

```
my.mat2 <- matrix(seq(1, 12), nrow = 4, ncol = 4, byrow = TRUE)
```

```
my.mat2
```

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

Как видим, для заполнения ячеек последней строки матрицы `my.mat2` программа снова использовала числа 1, 2, 3, и 4.

Альтернативный способ создания матриц заключается в применении функции `dim()` (от “dimension” – размерность). Так, матрицу `my.mat` мы могли бы сформировать из одномерного вектора следующим образом:

```
my.mat <- 1:16
```

```
# Задаем размерность 4x4 вектору my.mat:
```

```
dim(my.mat) <- c(4, 4)
```

```
my.mat
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	5	9	13
[2,]	2	6	10	14
[3,]	3	7	11	15
[4,]	4	8	12	16

Функция `dim()` очень полезна. Она позволяет проверить размерность уже имеющейся матрицы (или таблицы данных):

```
dim(my.mat)
[1] 4 4
```

Матрицу можно собрать также из нескольких векторов, используя функции `cbind()` (от `column` и `bind` – столбец и связывать) или `rbind()` (от `row` и `bind` – строка и связывать):

```
a <- c(1, 2, 3, 4)
b <- c(5, 6, 7, 8)
d <- c(9, 10, 11, 12)
e <- c(13, 14, 15, 16)
cbind(a, b, d, e)
      a b  d  e
[1,] 1 5  9 13
[2,] 2 6 10 14
[3,] 3 7 11 15
[4,] 4 8 12 16
```

Объединим те же векторы при помощи функции `rbind()`:

```
rbind(a, b, d, e)
  [,1] [,2] [,3] [,4]
a     1     2     3     4
b     5     6     7     8
d     9    10    11    12
e    13    14    15    16
```

Практически все векторные операции одинаково применимы в отношении матриц и массивов. Так, путем индексирования мы можем извлекать из матриц необходимые элементы и далее подвергать их требуемым преобразованиям. Рассмотрим несколько примеров:

Извлечем элемент матрицы `my.mat`, расположенный на пересечении 2-й строки и 3-го столбца:

```
my.mat[2, 3]
[1] 7
```

Извлечем из матрицы все элементы, находящиеся в 4-м столбце

```
my.mat[, 4]
[1] 4 8 12 16
```

```
# Извлечем из матрицы все элементы, находящиеся в 1-й
# строке
my.mat[1, ]
[1] 1 2 3 4
# Перемножим 1-й и 4-й столбцы матрицы (поэлементно):
my.mat[, 1]*my.mat[, 4]
[1] 4 40 108 208
```

При необходимости матрицу можно транспонировать (т.е. поменять местами строки и столбцы) при помощи функции `t()` (от transpose):

```
t(my.mat)
      A  B  C  D
[1,]  1  2  3  4
[2,]  5  6  7  8
[3,]  9 10 11 12
[4,] 13 14 15 16
```

Факторы. В статистике данные очень часто группируют в соответствии с тем или иным признаком, например, полом, социальным положением, стадией болезни, местом отбора проб и т.п. В R существует специальный класс векторов – факторы (factors), которые являются способом представления номинальных (категориальных) и шкальных типов данных в R. Качественные данные принимают значения в некотором конечном множестве. Поэтому для задания подобных данных достаточно задать это множество (способ кодирования этих данных) — множество градаций, или уровней (levels), фактора.

Часто уровни факторов кодируют в виде чисел. В таких случаях очень важно "проинструктировать" программу так, чтобы она "распознавала" уровни номинальной переменной от чисел как таковых.

Предположим, что в эксперименте по испытанию эффективности нового медицинского препарата было задействовано 10 пациентов-добровольцев, из которых шесть пациентов принимали новый препарат, а четверо остальных – плацебо (например, таблетки активированного угля). Для обозначения членов этих двух групп мы можем использовать коды 1 (препарат) и 0 (плацебо). Соответственно, информацию о всех десяти участниках эксперимента мы могли бы сохранить в виде следующего вектора:

```
treatment <- c(1, 1, 1, 1, 1, 1, 0, 0, 0, 0)
```

При таком подходе, однако, программа будет "рассматривать" вектор `treatment` в качестве числового (проверьте при помощи команды

`class(treatment)`). Это будет ошибкой с нашей стороны, поскольку ноль и единица обозначают лишь два уровня номинальной переменной. С таким же успехом мы могли бы использовать, например, 10 для обозначения контрольной группы пациентов (т.е. пациентов принимавших плацебо) и 110 для обозначения пациентов, принимавших испытываемый препарат. Для преобразования числового (или текстового) вектора в фактор в R существует одноименная функция `factor()`:

```
treatment <- factor(treatment, levels = c(0, 1))
treatment
[1] 1 1 1 1 1 1 0 0 0 0
Levels: 0 1
```

Обратите внимание на то, что теперь при выводе содержимого объекта `treatment` программа подсказывает нам, что этот объект является фактором с двумя уровнями (`Levels: 0 1`). Дополнительно убедиться в этом можно при помощи все той же команды `class(treatment)`:

```
class(treatment)
[1] "factor"
```

Более надежным подходом, позволяющим не запутаться при выполнении анализа, является кодировка уровней факторов при помощи текстовых значений, а не чисел. Например, в нашем примере можно присвоить значение `yes` пациентам, принимавшим препарат, и значение `no` пациентам из контрольной группы. Мы можем перекодировать уровни уже имеющегося фактора `treatment` при помощи функции `levels()`:

```
levels(treatment) <- c("no", "yes")
treatment
[1] yes yes yes yes yes yes no no no no
Levels: no yes
```

Заметьте, что при выводе содержимого вектора `treatment` коды пациентов не заключены в двойные кавычки, как это обычно бывает в случае с текстовыми значениями. Это является одним из внешних признаков того, что мы имеем дело именно с фактором, а не с текстовым вектором, содержащим шесть значений `"yes"` и четыре значения `"no"`.

Те же факторы легко преобразовать обратно в числовой вектор, состоящий из порядковых номеров уровней факторов:

```
as.numeric(treatment)
[1] 2 2 2 2 2 2 1 1 1 1
```


Существует также специальная команда для создания факторов:

```
gl(n, k, length = n*k, labels = 1:n),
```

где n – количество уровней фактора; k – число повторов для каждого уровня; $length$ – размер итогового объекта; $labels$ – необязательный аргумент, который можно использовать для указания названий каждого уровня фактора. Например, выполнение следующей команды приведет к созданию вектора `my.fac`, являющегося фактором с двумя уровнями – `Control` и `Treatment`, причем каждая из меток "Control" и "Treatment" будет повторена по 8 раз:

```
my.fac <- gl(2, 8, labels = c("Control", "Treatment"))
```

```
my.fac
```

```
[1] Control      Control      Control      Control      Control  
[6] Control      Control      Control      Treatment    Treatment  
[11] Treatment    Treatment    Treatment    Treatment    Treatment  
[16] Treatment
```

```
Levels: Control Treatment
```

Списки. В отличие от вектора или матрицы, которые могут содержать данные только одного типа, в список (`list`) или таблицу данных (`data frame`) можно включать сочетания любых типов данных. Это позволяет эффективно, т.е. в одном объекте, хранить разнородную информацию.

Каждый компонент списка может являться переменной, вектором, матрицей, фактором или другим списком. Кроме того, эти элементы могут принадлежать к различным типам: числа, строки символов, булевы переменные. Списки являются наиболее общим средством хранения внутрисистемной информации: в частности, результаты большинства статистических анализов в программе R хранятся в объектах-списках.

Для создания списков в R служит одноименная функция `list()`. Рассмотрим пример. Сначала создадим три разнотипных вектора – с текстовыми, числовыми и логическими значениями:

```
vector1 <- c("A", "B", "C")
```

```
vector2 <- seq(1, 3, 0.5)
```

```
vector3 <- c(FALSE, TRUE)
```

Теперь объединим эти три вектора в один объект-список, компонентам которого присвоим имена `Text`, `Number` и `Logic`:

```
my.list <- list(Text=vector1, Number=vector2,  
Logic=vector3)
```

```
my.list
$Text
[1] "A" "B" "C"
$Number
[1] 1.0 1.5 2.0 2.5 3.0
$Logic
[1] FALSE TRUE
```

К элементам списка можно получить доступ посредством трех различных операций индексации. Для обращения к поименованным компонентам применяют знак \$. Так, для извлечения компонента Text из списка my.list необходимо ввести следующую команду:

```
my.list$Text
[1] "A" "B" "C"
```

Имеется возможность извлекать из списка не только его поименованные компоненты-векторы, но и отдельные элементы, входящие в эти векторы. Для этого необходимо воспользоваться уже рассмотренным ранее способом – индексацией при помощи квадратных скобок. Единственная особенность работы со списками здесь состоит в том, что сначала необходимо указать имя компонента списка, используя знак \$, а уже затем номер(а) отдельных элементов этого компонента:

```
my.list$Number[3:5]
[1] 2.0 2.5 3.0
```

Извлечение компонентов списка можно осуществлять также с использованием двойных квадратных скобок, в которые заключается номер компонента списка:

```
my.list[[1]]
[1] "A" "B" "C"
```

После двойных квадратных скобок с индексным номером компонента списка можно также указать номер(а) отдельных элементов этого компонента:

```
my.list[[3]][1]
[1] FALSE
```

Созданный список my.list содержал всего лишь три небольших вектора, и мы знали, какие это векторы, и на каком месте в списке они стоят. Однако на практике можно столкнуться с гораздо более сложно организованными списками, индексирование которых может быть затруднено из-за отсутствия представлений

об их структуре. Для выяснения структуры объектов в языке R имеется специальная функция `str()` (от structure):

```
str(my.list)
List of 3
 $ Text : chr [1:3] "A" "B" "C"
 $ Number: num [1:5] 1 1.5 2 2.5 3
 $ Logic : logi [1:2] FALSE TRUE
```

Из приведенного примера следует, что список `my.list` включает 3 компонента (List of 3) с именами Text, Number и Logic (перечислены в отдельных строках после знака \$). Эти компоненты относятся к символьному (chr), числовому (num) и логическому (logi) типам векторов соответственно. Кроме того, команда `str()` выводит на экран первые несколько элементов каждого вектора.

Таблица данных (data frame) представляет собой объект R, по структуре напоминающий лист электронной таблицы Microsoft Excel. Каждый столбец таблицы является вектором, содержащим данные определенного типа. При этом действует правило, согласно которому все столбцы должны иметь одинаковую длину (собственно, с "точки зрения" R таблица данных является частным случаем списка, в котором все компоненты-векторы имеют одинаковый размер).

Таблицы данных – это основной класс объектов R, используемых для хранения данных. Обычно такие таблицы подготавливаются при помощи внешних приложений (особенно популярна и удобна программа Microsoft Excel) и затем загружаются в среду R.

Подробнее об импортировании данных в R будет рассказано ниже. Тем не менее, небольшую таблицу можно собрать из нескольких векторов средствами самой системы R. Для этого используют функцию `data.frame()`. Предположим, у нас есть наблюдения по общей численности мужского (Male) и женского (Female) населения в трех городах City1, City2, и City3. Представим эти данные в виде одной таблицы с именем CITY. Для начала создадим текстовые векторы с названиями городов (city) и пола (sex), а также вектор со значениями численности представителей каждого пола (number):

```
city <- c("City1", "City1", "City2", "City2", "City3",
"City3")
sex <- c("Male", "Female", "Male", "Female", "Male",
"Female")
number <- c(12450, 10345, 5670, 5800, 25129, 26000)
```

Теперь объединим эти три вектора в одну таблицу данных и посмотрим, что получилось:

```
CITY <- data.frame(City = city, Sex = sex, Number = number)
CITY
```

	City	Sex	Number
1	City1	Male	12450
2	City1	Female	10345
3	City2	Male	5670
4	City2	Female	5800
5	City3	Male	25129
6	City3	Female	26000

Обратите внимание на синтаксис функции `data.frame()`: ее аргументы перечисляются в формате "заголовок столбца = добавляемый вектор". В качестве заголовков столбцов могут выступать любые пользовательские имена, удовлетворяющие требованиям R.

Извлечь отдельные компоненты таблиц для выполнения необходимых вычислений, как и в примерах со списками, можно с использованием знака `$`, квадратных скобок с указанием двух индексов [`<номер_строки>`, `номер_столбца`], двойных квадратных скобок [`[]`], либо непосредственно по имени столбца:

```
CITY$Sex
[1] Male Female Male Female Male Female
Levels: Female Male
```

Идентичные результаты можно получить при помощи команд:

```
CITY[,2]
CITY[[2]]
```

Тогда как предыдущие команды возвращали список, команда `CITY["Sex"]` вернет таблицу данных:

	Sex
1	Male
2	Female
3	Male
4	Female
5	Male
6	Female

После имени или индексного номера столбца можно указывать индексные номера отдельных ячеек таблицы, что позволяет извлекать содержимое этих ячеек:

```
CITY$Number[4]
[1] 5800
CITY$Number[CITY$Number > 10000]
[1] 12450 10345 25129 26000
CITY$Number[CITY$Sex == "Male"]
[1] 12450 5670 25129
```

Повторяем те же команды, но с использованием []:

```
CITY[4, 3]
[1] 5800
CITY[CITY$Number > 10000, 3]
[1] 12450 10345 25129 26000
CITY[CITY$Sex == "Male", 3]
[1] 12450 5670 25129
```

При работе с большими таблицами данных бывает сложно визуально исследовать всё их содержимое перед началом анализа. Однако визуального просмотра содержимого таблиц и не требуется – полную сводную информацию о них (равно как и о других объектах R) можно легко получить при помощи упомянутой ранее функции `str()`:

```
str(CITY)
'data.frame':
  6 obs. of 3 variables:
 $ City : Factor w/ 3 levels "City1","City2",...: 1 1 2 2 3 3
 $ Sex  : Factor w/ 2 levels "Female","Male": 2 1 2 1 2 1
 $ Number: num 12450 10345 5670 5800 25129...
```

Как следует из представленного отчета, объект `CITY` является таблицей данных, в состав которой входят три переменные с шестью наблюдениями каждая. Две из этих переменных – `City` и `Sex` – программа автоматически распознала как факторы с тремя и двумя уровнями соответственно. Переменная `Number` является количественной. Для удобства выводятся также несколько первых значений каждой переменной.

Часто возникает необходимость выяснить лишь имена переменных, входящих в таблицу данных. Это можно сделать при помощи команды `names()`:

```
names(CITY)
[1] "City" "Sex" "Number"
```

Имеется также возможность быстро просмотреть несколько первых или несколько последних значений каждой переменной, входящей в состав таблицы данных. Для этого используются функции `head()` и `tail()` соответственно:

```
head(CITY, n = 3)
  City      Sex Number
1 City1   Male  12450
2 City1 Female  10345
3 City2   Male   5670
```

При необходимости внесения исправлений в таблицу можно воспользоваться встроенным в R редактором данных. Внешне этот редактор напоминает обычный лист Excel, однако имеет весьма ограниченные функциональные возможности. Все, что он позволяет делать – это добавлять новые или исправлять уже введенные значения переменных, изменять заголовки столбцов, а также добавлять новые строки и столбцы.

Работая в стандартной версии R, редактор данных можно запустить из меню "Файлы > Редактор данных", либо выполнив команду `fix()` (`fix` – исправлять, чинить) из командной строки консоли R (например, `fix(CITY)`). После внесения исправлений редактор просто закрывают – все изменения будут сохранены автоматически.

Заполнение пустых значений. Часто на практике некоторые значения в таблице отсутствуют, что может быть обусловлено множеством причин: на момент измерения прибор вышел из строя, по невнимательности персонала измерение не было занесено в протокол исследования, испытуемый отказался отвечать на определенный вопрос(ы) в анкете, была утеряна проба, и т.п. Ячейки с такими отсутствующими значениями (missing values) в таблицах данных R не могут быть просто пустыми – иначе столбцы таблицы окажутся разной длины. Для обозначения отсутствующих наблюдений в языке R, как указывалось ранее, имеется специальное значение – NA (not available – не доступно). Отметим, что если значение NA имеет смысл нуля (например, экземпляров некоего вида обнаружено не было), то легко произвести эту замену в таблице DF командой

```
DF[is.na(DF)] <- 0
```

Сортировка таблиц. Сортировка строк таблицы по различным ключам не представляет труда. Для этого используется функция `order()`:

```
DF <- data.frame(X1=c(1,15,1,3), X2=c(1,0,7,0),
X3=c(1,0,1,2), X4=c(7,4,41,0), X5=c(1,0,5,3))
row.names(DF) <- c("A", "B", "C", "D")
```

DF1 – таблица, столбцы которой отсортированы по убыванию суммы значений:
`DF1 <- DF[, rev(order(colSums(DF)))]`

DF2 – таблица, строки которой отсортированы в восходящем порядке по 1 столбцу, затем в нисходящем по второму:

```
DF2 <- DF[order(DF$X1, -DF$X2), ]
```

Объединение таблиц. Пусть мы имеем две таблицы:

```
> DF1
  Y  N A B C
1 12 22 0 1 0
2 12 23 1 3 0
3 12 24 0 0 1
```

```
> DF2
  Y  N A B D
1 13 22 0 1 2
2 13 23 0 3 0
3 13 24 1 0 5
```

Объединить их столбцы можно с использованием известной нам функции:

```
cbind(DF1,DF2)
  Y  N A B C  Y  N A B D
1 12 22 0 1 0 13 22 0 1 2
2 12 23 1 3 0 13 23 0 3 0
3 12 24 0 0 1 13 24 1 0 5
```

Для объединения строк мы должны предварительно преобразовать объединяемые таблицы к единому списку столбцов:

```
DF1[,names(DF2)[!(names(DF2) %in% names(DF1))]] <- NA
DF2[,names(DF1)[!(names(DF1) %in% names(DF2))]] <- NA
rbind(DF1,DF2)
  Y  N A B  C  D
1 12 22 0 1  0 NA
2 12 23 1 3  0 NA
3 12 24 0 0  1 NA
4 13 22 0 1 NA  2
5 13 23 0 3 NA  0
6 13 24 1 0 NA  5
```

Аналогичную операцию мы можем выполнить с помощью команды

```
merge(DF1, DF2,all = TRUE)
```

Функция `merge()` позволяет выполнять объединение таблиц всеми распространенными способами join-операций языка SQL.

Tibble. Tibbles are a modern reimaging of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not. The name comes from `dplyr`: originally you created these objects with `tbl_df()`, which was most easily pronounced as “tibble diff”.

Install tibble with: `install.packages("tibble")`

You can create a tibble from an existing object with `as_data_frame()`.

`data_frame()` does much less than `data.frame()`: it never changes the type of the inputs (e.g. it never converts strings to factors!), it never changes the names of variables, and it never creates `row.names()`. You can read more about these features in the vignette, `vignette("tibble")`.

There are two main differences in the usage of a data frame vs a tibble: printing, and subsetting.

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from `str()`:

```
library(nycflights13)
flights
#> Source: local data frame [336,776 x 16]
#>
#>   year month   day dep_time dep_delay arr_time arr_delay carrier tailnum
#>   (int) (int) (int)   (int)     (dbl)   (int)     (dbl)   (chr)   (chr)
#> 1  2013     1     1     517         2     830         11    UA  N14228
#> 2  2013     1     1     533         4     850         20    UA  N24211
#> 3  2013     1     1     542         2     923         33    AA  N619AA
#> 4  2013     1     1     544        -1    1004        -18    B6  N804JB
#> 5  2013     1     1     554        -6     812        -25    DL  N668DN
#> 6  2013     1     1     554        -4     740         12    UA  N39463
#> 7  2013     1     1     555        -5     913         19    B6  N516JB
#> 8  2013     1     1     557        -3     709        -14    EV  N829AS
#> 9  2013     1     1     557        -3     838         -8    B6  N593JB
#> 10 2013     1     1     558        -2     753          8    AA  N3ALAA
#> .. ... ..
#> Variables not shown: flight (int), origin (chr), dest (chr), air_time
#>   (dbl), distance (dbl), hour (dbl), minute (dbl).
```

Tibbles are strict about subsetting. If you try to access a variable that does not exist, you’ll get an error:

```
flights$yea
#> Error: Unknown column 'yea'
```

Tibbles also clearly delineate `[` and `[[`: `[` always returns another tibble, `[[` always returns a vector. No more `drop = FALSE`!

```
class(iris[, 1])
#> [1] "numeric"
class(iris[, 1, drop = FALSE])
#> [1] "data.frame"
class(as_data_frame(iris)[, 1])
#> [1] "tbl_df"      "tbl"          "data.frame"
```

It's possible to change the default printing appearance as follow:

- Change the maximum and the minimum rows to print: `options(tibble.print_max = 20, tibble.print_min = 6)`
- Always show all rows: `options(tibble.print_max = Inf)`
- Always show all columns: `options(tibble.width = Inf)`

`%>%` View

magrittr: Simplifying R code with pipes

R is a functional language, which means that your code often contains a lot of (parentheses). And complex code often means nesting those parentheses together, which make code hard to read and understand. But there's a very handy R package — `magrittr`, by Stefan Milton Bache — which lets you transform nested function calls into a simple pipeline of operations that's easier to write and understand.

Hadley Wickham's `dplyr` package benefits from the `%>%` pipeline operator provided by `magrittr`. Hadley showed at useR! 2014 an example of a data transformation operation using traditional R function calls:

```
hourly_delay <- filter(
  summarise(
    group_by(
      filter(
        flights,
        !is.na(dep_delay)
      ),
      date, hour
    ),
    delay = mean(dep_delay),
    n = n()
  ),
  n > 10
)
```

Here's the same code, but rather than nesting one function call inside the next, data is passed from one function to the next using the %>% operator:

```
hourly_delay <- flights %>%  
  filter(!is.na(dep_delay)) %>%  
  group_by(date, hour) %>%  
  summarise(  
    delay = mean(dep_delay),  
    n = n() ) %>%  
  filter(n > 10)
```

Чтение и запись данных. Форматы данных

Кроме рассмотренных выше команд можно использовать команду `edit()`, которая при вызове не для таблицы запустит редактирование объекта в текстовом редакторе (под Windows это обычно Notepad или Блокнот), и вы сможете отредактировать объект там. Если вы хотите поменять редактор, напишите, например:

```
options(editor="c:\\Program Files\\CrazyPad\\crazypad.exe")
```

Однако лучше всего научиться загружать в R файлы, созданные при помощи других программ, скажем, при помощи Excel.

В целом данные, которые надо обрабатывать, бывают двух типов: текстовые и бинарные. Не вдаваясь в детали, примем, что текстовые – это такие, которые можно прочитать и отредактировать в любом текстовом редакторе (Блокнот, Notepad, TextEdit, Vi). Для того чтобы отредактировать бинарные данные, нужна, как правило, программа, которая эти данные когда-то вывела. Текстовые данные для статистической обработки это обычно текстовые таблицы, в которых каждая строка соответствует строчке таблицы, а колонки определяются при помощи разделителей (обычно пробелов, знаков табуляции, запятых или точек с запятой). Для того чтобы R «усвоил» такие данные, надо, во-первых, убедиться, что текущая папка в R и та папка, откуда будут загружаться ваши данные, – это одно и то же. Для этого в запущенной сессии R надо ввести команду

```
> getwd()  
[1] "d:/programs/R/R-2.14.1"
```

Допустим, что это вовсе не та папка, которая вам нужна. Поменять рабочую папку можно командой:

```
> setwd("e:/wrk/temp")  
> getwd()  
[1] "e:/wrk/temp"
```

Дальше надо проверить, есть ли в нужной поддиректории нужный файл:

```
> dir("data")  
[1] "mydata.txt"
```

Теперь можно загружать данные. (Предполагается, что в текущей директории у вас есть папка data, а в ней файл mydata.txt. Если это не так, нужно поместить туда файл). Данные загружает команда `read.table()`:

```
> read.table("data/mydata.txt", sep=";", head=TRUE)  
  
  a b c  
1 1 2 3  
2 4 5 6  
3 7 8 9
```

Это ровно то, что надо, за тем исключением, что перед нами «рояль в кустах» — авторы заранее знали структуру данных, а именно то, что у столбцов есть имена (`head=TRUE`), а разделителем является точка с запятой (`sep=";"`). Функция `read.table()` очень хороша, но не настолько умна, чтобы определять формат данных «на лету». Поэтому вам придется выяснить нужные сведения заранее, скажем, в том же самом текстовом редакторе. Есть и способ выяснить это через R, для этого используется команда `file.show("data/mydata.txt")`. Она выводит свои данные таким же образом, как и `help()`, — отдельным окном или в отдельном режиме основного окна. Вот что вы должны увидеть:

```
a;b;c  
1;2;3  
4;5;6  
7;8;9
```

Отметим еще несколько важных вещей:

1. Кириллический текст в файлах обычно читается без проблем. Если все же возникли проблемы, то лучше перевести все в кодировку UTF-8 (при помощи любой доступной утилиты перекодирования, скажем `iconv`) и добавить соответствующую опцию:

```
> read.table("data/mydata.txt", sep=";", encoding="UTF-8")
```

2. Иногда нужно, чтобы R прочитал, кроме имен столбцов, еще и имена строк. Для этого используется такой прием:

```
> read.table("data/mydata2.txt", sep=";", head=TRUE)  
  
  a b c  
one 1 2 3  
two  4 5 6  
three 7 8 9
```

В файле `mydata2.txt` в первой строке было три колонки, а в остальных строках — по четыре. Проверьте это при помощи `file.show()`.

Такой способ выглядит сложновато, но позволяет использовать все преимущества программ по набору электронных таблиц и текстовых редакторов.

(подробнее см. файл помощи, доступный по команде `?read.table`).

Аргумент	Назначение
<code>file</code>	Служит для указания пути к импортируемому файлу. Путь приводят либо в абсолютном виде (например, <code>file = "C:/Temp/MyData.dat"</code>), либо указывают только имя импортируемого файла (например, <code>file = "MyData.txt"</code>), но при условии, что последний хранится в рабочей папке программы (см. выше). В качестве имени можно также указывать полную URL-ссылку на файл, который предполагается загрузить из Сети (например: <code>file = "http://somesite.net/YourData.csv"</code>). Начиная с версии R 2.10, появилась возможность импортировать архивированные файлы в zip-формате.
<code>header</code>	Служит для сообщения программе о наличии в загружаемом файле строки с заголовками столбцов. По умолчанию принимает значение <code>FALSE</code> . Если строка с заголовками столбцов имеется, этому аргументу следует присвоить значение <code>TRUE</code> .
<code>row.names</code>	Служит для указания номера столбца, в котором содержатся имена строк (например, в рассмотренном выше примере это был первый столбец, поэтому <code>row.names = 1</code>). Важно помнить, что все имена строк должны быть уникальными, т.е. одинаковые имена для двух или более строк не допускаются.
<code>sep</code>	Служит для указания разделителя значений переменных, используемого в файле (<i>separator</i> – разделитель). По умолчанию предполагается, что значения переменных разделены "пустым пространством", например, в виде пробела или знака табуляции (<code>sep = ""</code>). В файлах формата <code>csv</code> значения переменных разделены запятыми, и поэтому для них <code>sep = ","</code> .
<code>dec</code>	Служит для указания знака, используемого в файле для отделения целой части числа от дроби. По умолчанию <code>dec = "."</code> . Однако во многих странах в качестве десятичного знака применяют запятую, о чем важно вспомнить перед загрузкой файла и, при необходимости, использовать <code>dec = ","</code> . Следите, чтобы <code>dec</code> и <code>sep</code> не были бы одинаковыми.
<code>nrows</code>	Выражается целым числом, указывающим количество строк, которое должно быть считано из загружаемой таблицы. Отрицательные и иные значения игнорируются. Пример: <code>nrows = 100</code> .
<code>skip</code>	Выражается целым числом, указывающим количество строк в файле, которое должно быть пропущено перед началом импортирования. Пример: <code>skip = 5</code>

С электронными таблицами в текстовом формате больших проблем обычно не возникает. Разные экзотические текстовые форматы, как правило, можно преобразовать к «типичным», если не с помощью R, то с помощью каких-нибудь

текстовых утилит (вплоть до «тяжеловесов» типа языка Perl). А вот с «посторонними» бинарными форматами дело гораздо хуже. Здесь возникают прежде всего проблемы, связанные с закрытыми и/или недостаточно документированными форматами, такими, например, как формат программы MS Excel. Надо найти способ, как преобразовать (с минимальными потерями значимой информации, естественно) бинарные данные в текстовые таблицы. Проблем на этом пути возникает обычно не слишком много.

Второй путь — найти способ прочесть данные в R без преобразования. В R есть пакет `foreign`, который может читать бинарные данные, выводимые пакетами MiniTab, S, SAS, SPSS, Stata, Systat, а также формат DBF.

Что касается других распространенных форматов, скажем, форматов MS Excel, здесь дело хуже. Есть не меньше пяти разных способов, как загружать в R эти файлы, но все они имеют ограничения.

Если у вас открыт Excel, то можно скопировать в буфер любое количество ячеек, а потом загрузить их в R командой `read.table("clipboard")`. Это просто и, главное, работает с любой Excel-подобной программой, в том числе с «новым» Excel, Gnumeric и OpenOffice.org / LibreOffice Calc.

Аналогом `read.table()` для считывания csv-файлов является функция `read.csv()`:

```
chem <- read.csv(file = "hydro_chem.csv", header = TRUE)
```

Добавим еще несколько деталей:

1. R может загружать изображения. Для этого есть сразу несколько пакетов, наиболее разработанный из них — `rixmap`. R может также загружать карты в формате ArcInfo и др. (пакеты `maps`, `maptools`) и вообще много чего еще.
2. У R есть собственный бинарный формат (`*.RData`). Он быстро записывается и быстро загружается (команда `ReadRDS()`), но его нельзя использовать с другими программами.
3. Для R написано множество интерфейсов к базам данных, в частности для MySQL, PostgreSQL и sqlite (последний может вызываться прямо из R, см. документацию к пакетам `RSQLite` и `sqldf`).
4. Наконец, R может записывать таблицы и другие результаты обработки данных и, разумеется, графики.

Запись данных во внешние файлы выполняется аналогично.

Представление даты и времени. Временные ряды

Форматы представления даты и времени

Анализ данных, содержащих даты и время, может быть достаточно трудоемким. Причин этому несколько:

- разные годы начинаются в разные дни недели;
- високосные годы имеют дополнительный день в феврале;
- американцы и европейцы по-разному представляют даты (например, 8/9/2011 будет 9-м августа 2011 г. для первых и 8-м сентября 2011 г. для вторых);
- в некоторые годы добавляется так называемая "секунда координации";
- страны различаются по временным поясам и в ряде случаев применяют переход на "зимнее" и "летнее" время.

К счастью, система дат и времени в R такова, что многие из указанных проблем относительно легко преодолеваются. С форматом представления дат и времени в R можно ознакомиться, выполнив команду

```
> Sys.time()  
[1] "2017-01-31 13:47:23 MSK"
```

Как видим, формат строго иерархичен: сначала идет наиболее крупная временная единица – год, потом месяц и день, разделенные дефисом, а затем пробел, час, минуты, секунды и, после еще одного пробела, аббревиатура часового пояса. Отдельные элементы из этого результата можно извлечь при помощи функции `substr()` (от `substring` – часть строки), указав позиции первого и последнего элементов извлекаемой строки:

```
> substr(as.character(Sys.time()), 1, 10)
```

```
[1] "2017-01-31"
```

или

```
> substr(as.character(Sys.time()), 12, 19)
```

```
[1] "13:52:41"
```

Функция `date()` позволяет выяснить текущую дату:

```
[1] "Tue Jan 31 12:53:35 2017"
```


Если выполнить команду

```
> unclass(Sys.time())
```

```
[1] 1485860107
```

то получим время в формате POSIXct, т.е. выраженное в секундах, прошедших с 1 января 1970 г. (его еще трактуют как Unix-время, по названию операционной системы).

The Portable Operating System Interface (POSIX)[1] is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

Такой "машинный" формат удобен для включения в таблицы данных.

Для человека более удобным является представление времени в формате класса POSIXlt. Объекты этого класса представляют собой списки, включающие такие элементы, как секунды, минуты, часы, дни, месяцы, и годы. Например, мы можем конвертировать системное время в объект POSIXlt класса следующим образом:

```
> date <- as.POSIXlt(Sys.time())
```

Из списка date далее легко можно извлечь такие содержащиеся в нем элементы, как sec (секунды), min (минуты), hour (часы), mday (день месяца), mon (месяц), year (год), wday (день недели, начиная с воскресенья = 0), yday (день года, начиная с 1 января = 0), и isdst ("is daylight savings time in operation?" – логическая переменная, обозначающая, используется ли режим перехода на "зимнее" и "летнее" время: 1 если TRUE и 0 если FALSE), например:

```
> date$yday
```

```
[1] 2
```

```
> date$yday
```

```
[1] 30
```

Для просмотра всего содержимого списка date можно использовать функцию unclass() в сочетании с unlist():

```
> unlist(unclass(date))
```

sec	min	hour	mday
"41.5060729980469"	"14"	"14"	"31"
mon	year	wday	yday
"0"	"117"	"2"	"30"
isdst	zone	gmtoff	
"0"	"MSK"	"10800"	

Вычисления с датами и временем

В R можно выполнять следующие типы вычислительных операций с датами и временем:

- число + время;
- время – число;
- время1 – время2;
- время1 "логический оператор" время2 (в качестве логического оператора могут использоваться ==, !=, <=, <, > или >=).

Обратите внимание: в R отсутствует возможность для сложения двух дат.

Например, количество дней между 15 сентября 2011 г. и 15 сентября 2000 года можно найти следующим образом:

```
> t1 <- as.POSIXlt("2011-09-15")
> t2 <- as.POSIXlt("2000-09-15")
> t1 - t2
```

Time difference of 4017 days

Разницу во времени, выраженную в часах, можно рассчитать так:

```
> t3<-as.POSIXlt("2010-09-22 08:30:30")
> t4<-as.POSIXlt("2010-09-22 22:25:30")
> t4-t3
```

Time difference of 13.91667 hours

Еще проще разницу между двумя датами можно найти при помощи готовой функции `difftime()` (от difference – разница, и time – время):

```
> difftime("2011-09-22", "2010-06-22")
```

Time difference of 457 days

Чтобы извлечь непосредственно количество дней из результата выполнения предыдущей команды используйте функцию `as.numeric()`:

```
> as.numeric(difftime("2011-09-22", "2010-06-22"))
[1] 457
```

Измерить продолжительность какого-нибудь вычислительного процесса можно с использованием функции "процессорного времени" `proc.time()`, которая, по существу, работает как секундомер: вы засекаете стартовое время, запускаете процесс и, после его завершения, находите разность времен. Например, чтобы вычислить 10 000 значений арктангенса, потребуется 0.02 сек.:

```
> t1 <- proc.time()

> for (x in 1:10000) y <- atan(x)

> time.result <- proc.time() - t1

> time.result["elapsed"]

elapsed

      0.02
```

Извлечение даты/времени из текстовых переменных

Функция `strptime()` (от strip – раздевать, оголять, и time – время) позволяет извлекать даты и время из различных текстовых выражений. При этом важно верно указать формат (при помощи аргумента `format`), в котором приведены временные величины. Приняты следующие условные обозначения для наиболее часто используемых форматов дат и времени (детали доступны по команде `?strptime`):

`%a` – сокращенное название для недели (англ. яз.)

`%A` – полное название для недели (англ. яз.)

`%b` – сокращенное название месяца (англ. яз.)

`%B` – полное название месяца (англ. яз.)

`%d` – день месяца (01–31)

`%H` – часы от 00 до 23

`%I` – часы от 01 до 12

`%j` – порядковый номер дня года (001–366)

`%m` – порядковый номер месяца (01–12)

`%M` – минуты (00–59)

%S – секунды (00–61, с возможностью добавить "високосную секунду")

%U – неделя года (00–53), первое воскресенье считается первым днем первой недели

%w – порядковый номер дня недели (0–6, воскресенье – 0)

%W – неделя года (00–53), первый понедельник считается первым днем первой недели

%Y – год с указанием века

%y – год без указания века

Рассмотрим пример. Предположим, у нас имеется текстовый вектор, в котором хранятся даты в формате программы Microsoft Excel:

```
> dates.excel <- c("25/02/2008", "24/04/2009",  
"14/06/2009", "25/07/2010", "04/03/2011")
```

Формат имеющихся Excel-дат таков, что сначала идет день месяца, затем порядковый номер самого месяца и, наконец, год с указанием века. Требуется преобразовать эти текстовые выражения в даты формата R. Используя приведенные выше обозначения форматов функции `strptime()`, параметр `format` можно представить в виде `%d/%m/%Y`. Тогда команда для преобразования Excel-дат в R-даты будет выглядеть следующим образом:

```
> strptime(dates.excel, format = "%d/%m/%Y")  
[1] "2008-02-25 EET" "2009-04-24 EEST" "2009-06-14 EEST"  
[4] "2010-07-25 EEST" "2011-03-04 EET"
```

Вот еще один пример, в котором год приведен без указания века, а месяцы приведены в виде их сокращенных названий:

```
> example2 <- c("1jan79", "2jan99", "31jan04", "30aug05")  
  
> strptime(example2, "%d%b%y")  
[1] "1979-01-01 MSK" "1999-01-02 EET" "2004-01-31 EET"  
[4] "2005-08-30 EEST"
```

Часто необходимо обрабатывать календарные даты. По умолчанию `read.table()` считывает все нечисловые даты (например, «12/15/04» или «2004-12-15») как факторы. Поэтому после загрузки таких данных при помощи `read.table()` нужно обязательно применить функцию `as.Date()`.

Она «понимает» описание шаблона даты и преобразует строки символов в тип данных Date. В последних версиях R она работает и с факторами:

```
> dates.df <- data.frame(dates=c("2011-01-01", "2011-01-02", "2011-01-03", "2011-01-04",  
"2011-01-05"))  
  
> str(dates.df$dates)  
  
Factor w/ 5 levels "2011-01-01","2011-01-02",...: 1 2 3 4 5  
  
> dates.1 <- as.Date(dates.df$dates, "%Y-%m-%d")  
  
> str(dates.1)  
  
Date[1:5], format: "2011-01-01" "2011-01-02" "2011-01-03" "2011-01-04" "2011-01-05"
```

Временные ряды

Во многих областях деятельности людей замеры показателей проводятся не один раз, а повторяются через некоторые интервалы времени. Иногда этот интервал равен многим годам, как при переписи населения страны, иногда — дням, часам, минутам и даже секундам, но интервал между измерениями во временном ряду есть всегда. Его называют *интервалом выборки* (sampling interval). А образующийся в результате выборки ряд данных называют *временным рядом* (time series).

В любом временном ряду можно выделить две компоненты:

- 1) неслучайную (детерминированную) компоненту;
- 2) случайную компоненту.

Неслучайная компонента обычно наиболее интересна, так как она дает возможность проверить гипотезы о производящем временной ряд явлении. Математическая модель неслучайной компоненты может быть использована для прогноза поведения временного ряда в будущем. Если явление, результатом которого является изучаемый ряд, зависит от времени года (или времени суток, или дня недели, или иного фиксированного периода календаря), то из неслучайной компоненты может быть выделена еще одна компонента — сезонные колебания явления. Ее следует отличать от циклической компоненты, не привязанной к какому-либо естественному календарному циклу.

Под *трендом* (*тенденцией*) понимают неслучайную и непериодическую компоненту ряда. Первый вопрос, с которым сталкивается исследователь, анализирующий временной ряд, — существует ли в нем тренд? При наличии во временном ряде тренда и периодической компоненты значения любого последующего значения ряда зависят от предыдущих. Силу и знак этой связи можно измерить *коэффициентом корреляции*. Корреляционная зависимость между последовательными значениями временного ряда называется *автокорреляцией*.

Коэффициент автокорреляции первого порядка определяет зависимость между соседними значениями ряда t_n и t_{n-1} , больших порядков — между более отдаленными значениями. *Лаг (сдвиг) автокорреляции* — это количество периодов временного ряда, между которыми определяется коэффициент автокорреляции. Последовательность коэффициентов автокорреляции первого, второго и других порядков называется автокорреляционной функцией временного ряда. Анализ автокорреляционной функции позволяет найти лаг, при котором автокорреляция наиболее высокая, а следовательно, связь между текущим и предыдущими уровнями временного ряда наиболее тесная.

Если значимым оказался только первый коэффициент автокорреляции (коэффициент автокорреляции первого порядка), временной ряд, скорее всего, содержит только тенденцию (тренд). Если значимым оказался коэффициент автокорреляции, соответствующий лагу n , то ряд содержит циклические колебания с периодичностью в n моментов времени. Если ни один из коэффициентов автокорреляции не является значимым, то можно сказать, что либо ряд не содержит тенденции (тренда) и циклических колебаний, либо ряд содержит нелинейную тенденцию, которую линейный коэффициент корреляции выявить не способен.

Взаимная корреляция (кросс-корреляция) отражает, есть ли связь между рядами. В этом случае расчет происходит так же, как и для автокорреляции, только коэффициент корреляции рассчитывается между основным рядом и рядом, связь с которым основного ряда нужно определить. Лаг (сдвиг) при этом может быть и отрицательной величиной, поскольку цель расчета взаимной корреляции — выяснение того, какой из двух рядов «ведущий».

Анализ временного ряда часто строится вокруг объяснения выявленного тренда и циклических колебаний значений ряда в рамках некоторой статистической модели.

Найденная модель позволяет: прогнозировать будущие значения ряда (*forecasting*), генерировать искусственный временной ряд, все статистические характеристики которого эквивалентны исходному (*simulation*), и заполнять пробелы в исходном временном ряду наиболее вероятными значениями.

Нужно отличать *экстраполяцию* временного ряда (прогноз будущих значений ряда) от *интерполяции* (заполнение пробелов между имеющимися данными ряда). Не всегда модели ряда, пригодные для интерполяции, можно использовать для прогноза. Например, полином (степенное уравнение с коэффициентами) очень хорошо сглаживает исходный ряд значений и позволяет получить оценку показателя, который описывает данный ряд в промежутках между значениями ряда. Но если мы попытаемся продлить полином за стартовое значение ряда, то получим совершенно случайный результат. Вместе с тем обычный линейный тренд, хотя и не столь изошренно следует изгибам внутри ряда, дает устойчивый прогноз развития ряда в будущем.

Разные участки ряда могут описывать различные статистические модели, в этом случае говорят, что ряд *нестационарный*. Нестационарный временной ряд во многих случаях удастся превратить в стационарный путем преобразования данных.

В R существует специальный класс объектов для работы с данными, представляющими собой временные ряды – `ts` (от *time series* – временной ряд), который представляет собой временной ряд, состоящий из значений, разделенных одинаковыми интервалами времени. Для создания объектов этого класса служит одноименная функция `ts()`.

В качестве примера рассмотрим ежемесячные данные по рождаемости в г. Нью-Йорк, собранные в период с января 1946 г. по декабрь 1959 г. Пример заимствован из электронной книги *A Little Book of R for Time Series* и исходные данные можно загрузить с сайта проф. Роба Хиндмана (Rob J. Hyndman) следующим образом:

```
> birth <- scan("http://robjhyndman.com/tsdldata/data/nybirths.dat")
Read 168 items
```

Объект `birth` представляет собой вектор со всеми 168 ежемесячными значениями рождаемости (в тыс. человек). Функция `head()` позволяет просмотреть первые несколько значений вектора `birth` (по умолчанию первые 6 значений):

```
> head(birth)
[1] 26.663 23.598 26.931 24.740 25.806 24.364
```

Преобразовать объект `birth` во временной ряд очень просто:

```
> birth.ts <- ts(birth, start = c(1946, 1), frequency = 12)
```

В приведенной команде аргумент `start` был использован для того, чтобы указать дату, с которой начинается временной ряд `birth.ts` (1946 год, 1-й месяц). Дополнительный аргумент `frequency` (частота) позволяет задать шаг приращения последующих дат – в рассматриваемом примере год разбивается на 12 промежутков, так что шаг приращения составляет 1 месяц. Созданный таким образом объект `birth.ts` при просмотре внешне напоминает матрицу. При этом строкам и столбцам этой матрицы были автоматически, исходя из значений аргументов `start` и `frequency`, присвоены соответствующие имена:

```
> birth.ts
      Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
1946 26.663 23.598 26.931 24.740 25.806 24.364 24.477 23.901 23.175 23.227 21.672 21.870
1947 21.439 21.089 23.709 21.669 21.752 20.761 23.479 23.824 23.105 23.110 21.759 22.073
1948 21.937 20.035 23.590 21.672 22.222 22.123 23.950 23.504 22.238 23.142 21.059 21.573
1949 21.548 20.000 22.424 20.615 21.761 22.874 24.104 23.748 23.262 22.907 21.519 22.025
1950 22.604 20.894 24.677 23.673 25.320 23.583 24.671 24.454 24.122 24.252 22.084 22.991
```

```

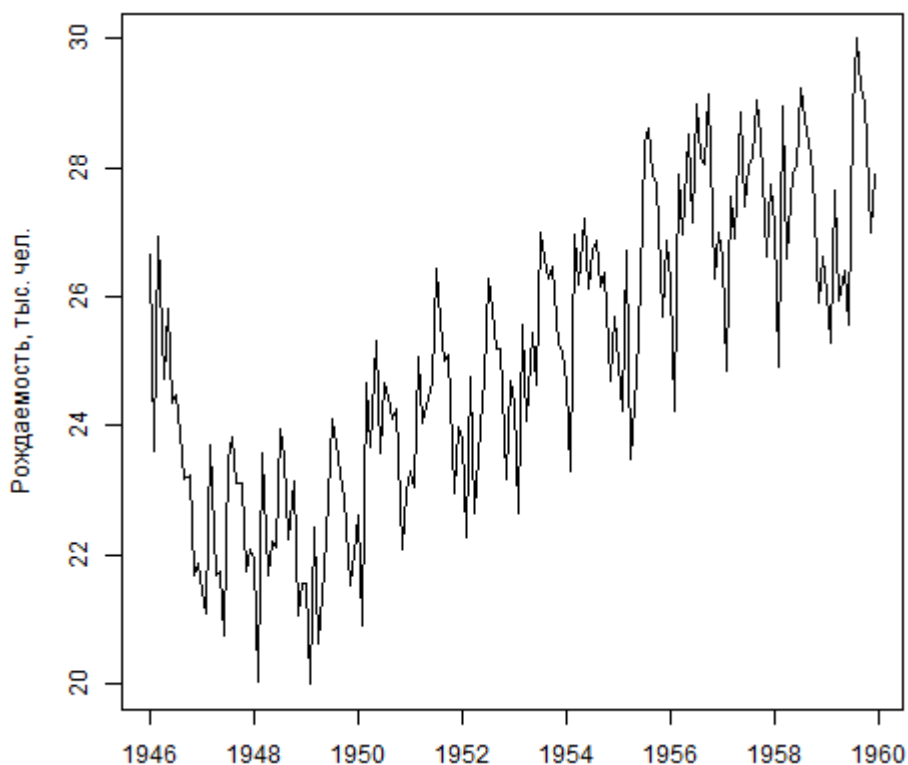
1951 23.287 23.049 25.076 24.037 24.430 24.667 26.451 25.618 25.014 25.110 22.964 23.981
1952 23.798 22.270 24.775 22.646 23.988 24.737 26.276 25.816 25.210 25.199 23.162 24.707
1953 24.364 22.644 25.565 24.062 25.431 24.635 27.009 26.606 26.268 26.462 25.246 25.180
1954 24.657 23.304 26.982 26.199 27.210 26.122 26.706 26.878 26.152 26.379 24.712 25.688
1955 24.990 24.239 26.721 23.475 24.767 26.219 28.361 28.599 27.914 27.784 25.693 26.881
1956 26.217 24.218 27.914 26.975 28.527 27.139 28.982 28.169 28.056 29.136 26.291 26.987
1957 26.589 24.848 27.543 26.896 28.878 27.390 28.065 28.141 29.048 28.484 26.634 27.735
1958 27.132 24.924 28.963 26.589 27.931 28.009 29.229 28.759 28.405 27.945 25.912 26.619
1959 26.076 25.286 27.660 25.951 26.398 25.565 28.865 30.000 29.261 29.012 26.992 27.897

```

Функция `is.ts()` позволяет проверить, действительно ли созданный нами объект `birth.ts` является временным рядом.

В R имеется достаточно большой набор методов для работы с объектами класса `ts`. В частности, при помощи функции `plot()` можно быстро изобразить временной ряд графически:

```
> plot(birth.ts, xlab = "", ylab = "Рождаемость, тыс. чел.")
```



Временные ряды могут быть образованы и неравномерно отстоящими друг от друга значениями. В этом случае следует воспользоваться специальными типами данных — `zoo` и `its`, которые становятся доступными после загрузки пакетов с теми же именами.

А вот как создаются временные ряды типа `ts`:

```
> ts(1:10, frequency = 4, start = c(1959, 2))
#поквартально, начинаем во втором квартале 1959 года
```


	Qtr1	Qtr2	Qtr3	Qtr4
1959		1	2	3
1960	4	5	6	7
1961	8	9	10	

Можно конвертировать сразу матрицу, тогда каждая колонка матрицы станет отдельным временным рядом:

матрица данных из трех столбцов

```
> z <- ts(matrix(rnorm(300), 100, 3), start=c(1961, 1), frequency=12)
> z
```

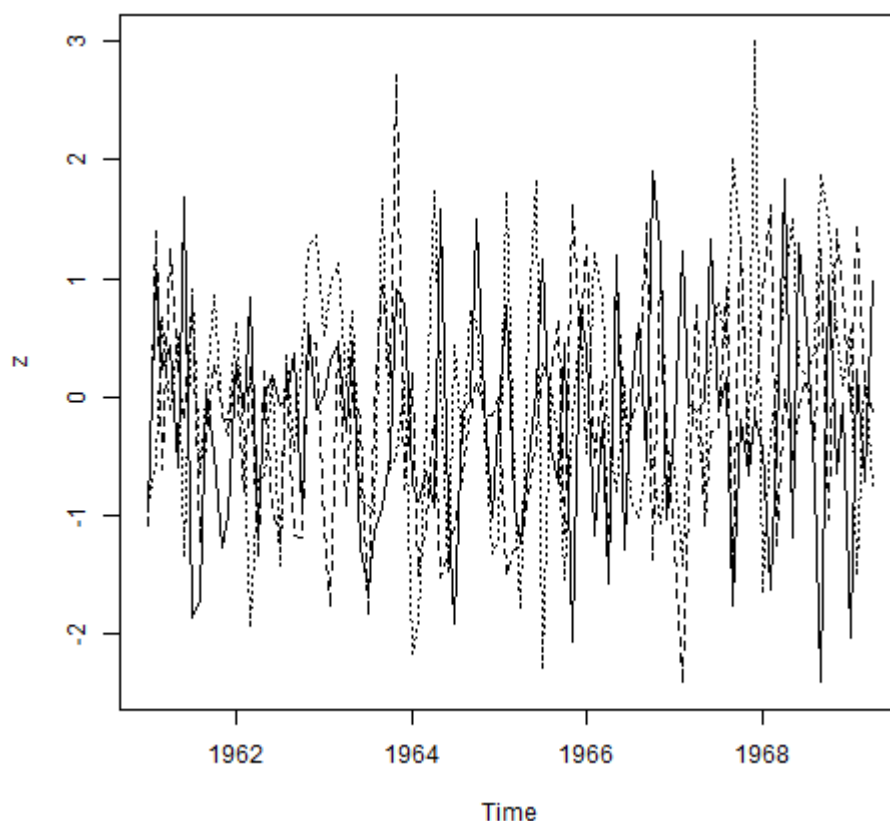
	Series 1	Series 2	Series 3
Jan 1961	-0.977653264	-1.090035629	-0.922699535
Feb 1961	1.079056556	1.403451197	-0.545109138
Mar 1961	0.222396367	-0.617287491	0.680068624
Apr 1961	0.441329301	1.246996869	-0.131372101
...			
Mar 1969	-0.712414781	0.099381786	0.215910024
Apr 1969	0.977615684	-0.146646371	-0.786762814

```
> class(z)
[1] "mts"      "ts"       "matrix"
```

Ряды отображаются графически с помощью стандартной функции `plot()`:

```
> plot(z, plot.type="single", lty=1:3)
```

Методы для анализа временных рядов и их моделирования включают ARIMA-модели, реализованные в функциях `arima()`, `AR()` и `VAR()`, структурные модели в `StructTS()`, функции автокорреляции и частной автокорреляции в `acf()` и `pacf()`, классическую декомпозицию временного ряда в `decompose()`, STL-декомпозицию в `stl()`, скользящее среднее и авторегрессивный фильтр в `filter()`.



Организация вычислений: функции, ветвления, циклы

Абсолютное большинство процедур обработки данных в R реализуется с помощью функций. Функции представляют собой поименованный программный код, состоящий из некоторого набора переменных, констант, операторов и других функций, и предназначенный для выполнения конкретных операций и задач. Как правило (но не всегда), функции возвращают результат своего выполнения в виде объекта языка R – переменной определенного класса: вектора, списка, таблицы и т.д.

По своему назначению функции можно разделить на характерные группы: арифметические, символьные, статистические и прочие. Функции могут быть встроенными (т.е. представленными в базовых или подгружаемых пакетах) и собственными (т.е. написанными непосредственно самими пользователями). Некоторые наиболее употребимые встроенные функции представлены ниже.

Три характерными чертами языка R как языка высокого уровня являются модульность построения, ориентация на объекты, и векторизация вычислений. Под модульностью понимается широкое использование групп выражений и функций.

Выражения `expr`, состоящие из объектов данных, вызовов функций и других операторов языка могут группироваться в фигурных скобках: `{expr_1; ...; expr_m}`, и значение, которое возвращает эта группа, представляет собой результат выполнения последнего выражения. Поскольку такая группа является также выражением, то она может быть, например, включена в круглые скобки и использоваться как часть еще более общего выражения.

Вызов функции и описание	Пример и результат
Арифметические функции	
abs (x) – модуль величины x	abs(-1) ⇒ 1
ceiling (x) – округление до целого в большую сторону	ceiling(9.435) ⇒ 10
floor (x) – округление до целого в меньшую сторону	floor(2.975) ⇒ 2
round (x, digits=n) – округление до указанного числа digits знаков после десятичной точки	round(5.475, 2) ⇒ 5.48
signif (x, digits=n) # округление до указанного числа digits значащих цифр	signif(3.475, 2) ⇒ 3.5
trunc (x) – округление до целого числа	trunc(4.99) ⇒ 4
exp (x) – e^x	exp(2.87) ⇒ 17.637
log (x) – логарифм натуральный x	log(3.12) ⇒ 1.137
log10 (x) – логарифм десятичный x	log(3.12) ⇒ 0.494
sqrt (x) # корень квадратный x	sqrt(2.12) ⇒ 1.456
cos (x) sin (x) tan (x) acos (x) cosh (x) acosh (x) – тригонометрические функции от x	cos(1.27*pi) ⇒ -0.661
Функции для работы с символьными типами данных	
grep (pattern, x, ignore.case=FALSE, fixed=FALSE) – возврат индекса первого найденного элемента pattern в x	grep("A", c("x", "y", "A", "z"), fixed=TRUE) ⇒ 3
substr (x, start=n1, stop=n2) – выбор или замена символов в строках символьного вектора x	substr("язык R", 2, 4) ⇒ "зык"
paste (..., sep="") – объединение символов или строк через значение разделителя sep	paste("x", 1:3, sep="") ⇒ "x1" "x2" "x3"
strsplit (x, split) – разделяет элементы вектора по разделителям split	strsplit("абв", "") ⇒ "a" "б" "в"
toupper (x) и tolower (x) – преобразуют буквы текстового вектора x в прописные и обратно	toupper("Мал") ⇒ "МАЛ" toupper("БАЛ") ⇒ "бал"

Например, группа команд ниже выполняет расчет среднего и стандартного отклонения натурального ряда чисел от 1 до 10 и возвращает вектор из этих значений:

```
> {aver <- mean(1:10); stdev <- sd(1:10); c(MEAN=aver, SD=stdev)}
```

```
MEAN      SD
```

```
5.50000 3.02765
```

Однако если этот расчет необходимо выполнить неоднократно для различных наборов исходных данных, то его стоит оформить в виде функции. Общий синтаксис оформления собственной функции пользователя таков:

```
имя_функции <- function(arg1, arg2,...) {  
  
  группа_выражений  
  
  return(object) }
```

где `имя_функции` – имя создаваемой функции, `arg1, arg2, ...` – формальные аргументы функции. Оператор `return()` нужен в случаях, когда группа выражений не возвращает целевого результата.

Перед своим первым выполнением функция должна быть определена в текущем скрипте, либо загружена с помощью команды `source()` из скриптового файла, где она была предварительно подготовлена. Тогда вызов функции может быть осуществлен как

```
имя_функции (arg1, arg2, ...)
```

где `arg1, arg2, ...` – фактические аргументы, связанные с формальными параметрами функции по порядку их следования, либо по наименованиям.

Для представленного выше примера можно оформить функцию:

```
stat_param <- function(x){  
  aver <- mean(x); stdev <- sd(x)  
  c(MEAN=aver, SD=stdev)}
```

и включить ее в коллекцию собственных функций, расположенных в файле `my_func.R`.

Тогда необходимый нам результат, приведенный выше, можно получить, выполнив

```
source("my_func.R")  
  
stat_param (1:10)
```

Компоненты списка аргументов в заголовке функций могут быть обязательными или принимать опциональные значения. Например, следующая функция возводит числовой объект `x` в степень `n`, но если степень не указана, то автоматически происходит возведение в куб:

```
power <- function(x, n = 3){ x^n }
```

Аргументами функций могут быть объекты самого разного типа, например, названия других функций. Так, следующая функция выполняет произвольные преобразования случайных равномерно распределенных величин:

```
my_examp1 <- function(n, func_trans)  
{ x <- 1:n ; abs(func_trans(x)) }
```

Тогда сгенерировать 5 прологарифмированных значений можно, если записать:

```
my_examp1(5, log)
```

```
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

Условия и циклы

Как и в любом алгоритмическом языке, в R широко используются ветвления и циклы вычислительного процесса. Условный оператор имеет следующую структуру:

```
if( логическое_выражение )  
  
{ группа_выражений_1 если логическое_выражение равно TRUE }  
  
else { группа_выражений_2 в противном случае }
```

Например, следующая функция сравнивает размеры двух векторов:

```
> compare <- function(x, y){  
    n1 <- length(x)  
    n2 <- length(y)  
    if(n1 != n2){  
        if(n1 > n2) {  
            z <- (n1 - n2)  
            cat("Первый вектор имеет на ", z, " элементов больше \n")  
        } else {  
            z <- (n2 - n1)  
            cat("Второй вектор имеет на ", z, " элементов больше \n")  
        }  
    } else {  
        cat("Количество элементов одинаково ", n1, "\n")  
    }  
}  
  
> x <- c(1:4)  
  
> y <- c(1:9)
```

```
> compare(x, y)
```

Первый вектор имеет на 5 элементов больше

Имеется также сокращенная форма реализации ветвлений:

```
ifelse(логическое_выражение, группа_выражений_1, группа_выражений_2)
```

Повторение в **цикле** одних и тех же вычислительных операций осуществляется с использованием конструкций `for()`, `while()` или `repeat()`, которые имеют следующий синтаксис:

```
for (index in for_object) { группа_выражений }
```

```
while(логическое_выражение) { группа_выражений }
```

```
repeat { группа_выражений ; break }
```

Здесь объект `for_object` может быть вектором, массивом, таблицей, или списком, а группа_выражений выполняется каждый раз для каждого элемента `index` этого объекта.

Рассмотрим в качестве примера функцию оценки доверительного интервала среднего значения для выборки размером `n` с использованием непараметрического бутстрепа. Необходимо отметить, что устоявшегося перевода термина "bootstrap" с английского языка на русский не существует. Используются разные варианты: "бутстреп", "бутстрэп", "бутстрап", "размножение выборок", "метод псевдовыборок" и даже "ресамплинг" (от англ. "resampling"). Несмотря на сложности с русскоязычным названием, суть метода, тем менее, весьма проста и подробно изложена в оригинальных работах Б. Эфрона (1979-1988). Бутстреп-метод с использованием R для решения различных задач представлен в ряде монографий (Chernick, LaBudde, 2011; Zieffler et al. 2011; Fox, Weisberg, 2012; Шитиков, Розенберг, 2014).

Предположим, что у нас есть выборка некоторого ограниченного объема, и мы имеем основания полагать, что эта выборка является репрезентативной (т.е. хорошо отражает свойства генеральной совокупности, из которой она была взята). Идея бутстреп-метода заключается в том, что мы можем рассматривать саму эту выборку в качестве "генеральной совокупности" и, соответственно, можем извлечь большое число случайных выборок из этой исходной совокупности для расчета интересующего нас параметра (или параметров). Очевидно, что благодаря случайному процессу формирования этих новых выборок, будет наблюдаться определенная вариация значений оцениваемого параметра. Другими словами, мы получим некоторое распределение значений этого параметра.

Рассчитав стандартное отклонение этого распределения, мы получим оценку стандартной ошибки параметра, которая при большом числе наблюдений будет

асимптотически приближаться к истинной стандартной ошибке. Аналогично можно получить оценки границ доверительного интервала.

Итак, будем генерировать из исходной выборки множество псевдовыборок того же размера, состоящих из случайных комбинаций исходного набора элементов. При этом используем алгоритм "случайного выбора с возвратом" (random sampling with replacement), т.е. извлеченный элемент возвращается в исходную совокупность и имеет шанс быть выбранным снова. В результате некоторые члены в каждой отдельной псевдовыборке могут повторяться два или более раз, тогда как другие – отсутствовать.

Этот алгоритм в R реализован в функции `sample(data, replace = T)`. Для каждой псевдовыборки мы рассчитаем значение среднего, а в качестве границ 95%-ного доверительного интервала (bootstrap percentile interval) примем 2.5% и 97.5% квантили бутстреп-распределения:

```
> boot_np <- function(data, Nboot = 5000) {  
  
  boots <- numeric(Nboot) # Пустой вектор для хранения результатов  
  
  for (i in 1:Nboot) { boots[i] <- mean(sample(data, replace = T)) }  
  
  CI <- quantile(boots, prob = c(0.025, 0.975))  
  
  return (c(m = mean(data), CI))  
  
}  
  
> x <- c(5, 5, 8, 10, 10, 10, 19, 20, 20, 20, 30, 40, 42, 50, 50)  
  
> boot_np(x)  
  
      m      2.5%      97.5%  
22.60000 15.20000 31.06667
```

Здесь конструкция `for()` осуществляет формирование `Nboot = 5000` значений средних для генерируемых псевдовыборок.

Векторизованные вычисления в R

Встроенные операции языка R векторизованы, то есть выполняются покомпонентно. В таком случае достаточно быстро встает вопрос, каким образом осуществляются операции в случае, если операнды имеют разную длину (например, при сложении вектора длины 2 и длины 4). За это отвечают так называемые правила переписывания (recycling rules):

1. Длина результата совпадает с длиной операнда наибольшей длины.

2. Если длина операнда меньшей длины делит длину второго операнда, то такой операнд повторяется (переписывается) столько раз, сколько нужно до достижения длины второго операнда. После этого операция производится покомпонентно над операндами одинаковой длины.
3. Если длина операнда меньшей длины не является делителем длины второго операнда (то есть она не укладывается целое число раз в длину большего операнда), то такой операнд повторяется столько раз, сколько нужно для перекрытия длины второго операнда. Лишние элементы отбрасываются, производится операция и выводится предупреждение.

Как следствие этих правил, операции типа сложения числа (то есть вектора единичной длины) с вектором выполняются естественным образом:

```
> 2 + c(3, 5, 7, 11)
```

```
[1] 5 7 9 13
```

```
> c(1, 2) + c(3, 5, 7, 11)
```

```
[1] 4 7 8 13
```

```
> c(1, 2, 3) + c(3, 5, 7, 11)
```

```
[1] 4 7 10 12
```

Warning message:

```
In c(1, 2, 3) + c(3, 5, 7, 11) :
```

```
longer object length is not a multiple of shorter object length
```

Большинство встроенных функций языка R так или иначе векторизованы, то есть выдают «естественный» результат при передаче в качестве аргумента вектора. К этому необходимо стремиться при написании собственных функций, так как это, как правило, является ключевым фактором, влияющим на скорость выполнения программы. Разберем простой пример (написанный, очевидно, человеком, хорошо знакомым с языком типа C, но малознакомым с R):

```
> p <- 1:20
```

```
> lik <- 0
```

```
> for (i in 1:length(p)) {
```

```
+ lik <- lik + log(p[i])
```

```
+ }
```


Это же самое действие можно реализовать существенно проще и короче (кроме того, обеспечив корректную работу в случае, если вектор `p` имел бы нулевую длину):

```
> lik <- sum(log(p))
```

Отметим, что «проще» имеется в виду не только с точки зрения количества строк кода, но и вычислительной сложности: первый образец кода выполняется полностью на интерпретаторе, второй же использует эффективные и быстрые встроенные функции.

Второй образец кода работает потому, что функции `log()` и `sum()` векторизованы. Функция `log()` векторизована в обычном смысле: скалярная функция применяется поочередно к каждому элементу вектора, таким образом результат `log(c(1, 2))` идентичен результату `c(log(1), log(2))`.

Функция `sum()` векторизована в несколько ином смысле: она берет на вход вектор и считает что-то, зависящее от вектора целиком. В данном случае вызов `sum(x)` полностью эквивалентен выражению `x[1] + x[2] + ... + x[length(x)]`.

Очень часто векторизация кода появляется сама собой за счет наличия встроенных функций, правил переписывания для арифметических операций и т. п. Однако (особенно часто это происходит при переписывании кода с других языков программирования) код следует изменить для того, чтобы он стал векторизованным. Например, код

```
> v <- NULL

> v2 <- 1:10

> for (i in 1:length(v2)) {
+   if (v2[i] %% 2 == 0) {
+     v <- c(v, v2[i]) # 7 строка
+   }
+ }
```

плох сразу по двум причинам: он содержит цикл там, где его можно избежать, и, что совсем плохо, он содержит вектор, растущий внутри цикла. Среда R действительно прячет детали выделения и освобождения памяти от пользователя, но это вовсе не значит, что о них не надо знать и их не надо учитывать. В данном случае в седьмой строке происходят выделение памяти под новый вектор `v` и копирование в этот новый вектор элементов из старого. Таким образом,

вычислительные затраты этого цикла (в терминах числа копирования элементов) пропорциональны квадрату длины вектора `v2`!

Оптимальное же решение в данном случае является очень простым:

```
> v <- v2[v2 %% 2 == 0]
```

Векторизацию отдельной функции можно произвести при помощи функции `Vectorize()`, однако не стоит думать, что это решение всех проблем — это исключительно изменение внешнего интерфейса функции, внутри она по-прежнему будет вызываться для каждого элемента по отдельности (хотя в отдельных случаях такого решения оказывается достаточно).

Стандартной проблемой при векторизации является оператор `if`. Один из вариантов замены его векторизации был рассмотрен выше, но очень часто подобного рода преобразования невозможны. Например, рассмотрим код вида

```
if (x > 1) y <- -1 else y <- 1
```

В случае когда `x` имеет длину 1, получаемый результат вполне соответствует ожиданиям. Однако как только длина `x` становится больше 1, все перестает работать. В данном случае код можно векторизовать при помощи функции `ifelse()`:

```
ifelse(x > 1, -1, 1)
```

```
> x <- c(3, 5, 1, 6, 0)
```

```
> ifelse(x > 1, -1, 1)
```

```
[1] -1 -1  1 -1  1
```

В общем случае синтаксис функции следующий:

```
ifelse(cond, vtrue, vfalse)
```

Результатом функции `ifelse()` является вектор той же длины, что и вектор-условие `cond`. Векторы `vtrue` и `vfalse` переписываются до этой длины. В вектор результата записывается элемент вектора `vtrue` в случае, если соответствующий элемент вектора `cond` равен `TRUE`, либо элемент вектора `vfalse`, если элемент `cond` равен `FALSE`. В противном случае в результат записывается `NA`.

Стандартным желанием при выполнении векторизации является использование функций из `apply()`-семейства: `lapply()`, `sapply()` и т. п. (будут рассмотрены ниже).

В большинстве случаев результат получится гораздо хуже, чем если бы векторизации не было вообще (в этом примере мы предварительно явно выделили память под вектор `v`, чтобы исключить влияние менеджера памяти на результат):

```
> d <- 1:1000000

> v <- numeric(length(d))

> system.time(for (i in 1:length(d)) v[i] <- pi*d[i]^2/4)

  user  system elapsed 
 1.86    0.00    1.86 

> system.time(v <- sapply(d, function(d) pi*d^2/4))

  user  system elapsed 
 2.08    0.00    2.08 

> system.time(v <- pi*d^2/4)

  user  system elapsed 
 0.02    0.00    0.01
```

Такие результаты для времени выполнения вполне объяснимы. Первый фрагмент кода выполняется целиком на интерпретаторе. Последний за счет использования векторизованных операций проводит большую часть времени в ядре среды R (то есть в неинтерпретируемом коде). Второй же фрагмент совмещает худшие стороны первого и третьего фрагментов:

- Функция `sapply()` выполняется в неинтерпретируемом коде.
- Второй аргумент функции `sapply()` является интерпретируемой функцией.

Таким образом, для вычисления одного элемента вектора `d` необходимо запустить интерпретатор для тривиальной функции, выполнить эту функцию и получить результат. Как следствие накладные расходы, необходимые на запуск интерпретатора, доминируют во времени исполнения кода.

Излишняя векторизация может приводить не только к увеличению времени выполнения, но и к существенному расходу памяти. Предположим, что встала задача заменить все отрицательные элементы таблицы данных `df` на 0. Естественно, это можно сделать в полностью векторизованном виде:

```
df[df < 0] <- 0
```

Однако, как только таблица данных `df` станет очень большой, эта конструкция будет требовать памяти в два раза больше, нежели размер таблицы данных, тем самым потенциально приводя к нехватке свободной памяти для среды.

Альтернативой может быть заполнение по строкам:

```
for (i in nrow(df)) df[i, df[i, ] < 0] <- 0
```

или же по столбцам:

```
for (i in ncol(df)) df[df[, i] < 0, i] <- 0
```

Выбор того или иного варианта зависит от соотношения числа строк и столбцов в таблице данных и от того, что является более важным — скорость исполнения кода или потребление памяти (хотя как только заканчивается доступная физическая память и начинается использование файла подкачки, ни о какой производительности не может быть и речи).

Принцип векторизованных вычислений применим не только к векторам как таковым, но и к более сложным объектам R — матрицам, спискам и таблицам данных (для R разницы между последними двумя типами объектов не существует: фактически таблица данных является списком из нескольких компонентов — векторов одинакового размера).

В базовой комплектации R имеется целое семейство функций, предназначенных для организации векторизованных вычислений над такими объектами. В названии всех этих функций имеется слово `apply` (англ. применить), которому предшествует буква, указывающая на принцип работы той или иной функции (см. подробнее в справочном файле `?apply`). При этом:

- `apply()` в отличие от `for()` можно легко распараллелить (просто переименовав функцию в ее параллельную версию из пакета `snow`);
- алгоритмы, записанные без циклов, легче модифицируются, содержат меньше ошибок и легче набираются в командной строке;
- результат работы `apply()` может быть аргументом функции, не говоря уже о том что любая функция может быть вставлена в качестве аргумента в `apply()`.

В ответе на один из вопросов, опубликованных на сайте stackoverflow.com, который мы постоянно используем для поиска информации по R, был дан замечательный обзор `apply`-функций с примерами их использования. Ниже приводится перевод этого сообщения (с некоторыми изменениями и дополнениями).

Функция **apply()** – используется в случаях, когда необходимо применить какую-либо функцию ко всем строкам или столбцам матрицы (или массивам большей размерности):

```
apply(x, MARGIN, FUN, ...)
```

где *x* – это преобразуемый объект, *MARGIN* – индекс, обозначающий направление процесса вычислений (по столбцам или строкам), *FUN* – применяемая для вычислений функция, а *...* – это любые другие параметры применяемой функции. Для матрицы или таблицы данных *MARGIN* = 1 обозначает строки, а *MARGIN* = 2 – столбцы. Поскольку *FUN* означает любую функцию R, в том числе и ту, которую вы сами написали, то функция *apply()* – это мощное средство модульной обработки данных.

```
# Создадим обычную двумерную матрицу:
```

```
M <- matrix(seq(1,16), 4, 4)
```

```
# Найдем минимальные значения в каждой строке матрицы
```

```
apply(M, 1, min)
```

```
[1] 1 2 3 4
```

```
# Найдем минимальные значения в каждом столбце матрицы
```

```
apply(M, 2, max)
```

```
[1] 4 8 12 16
```

```
# Пример с трехмерным массивом:
```

```
M <- array(seq(32), dim = c(4,4,2))
```

```
# Применим функцию sum() к каждому элементу M[, , ], т.е. выполним суммирование по измерениям 2 и 3:
```

```
apply(M, 1, sum)
```

```
# Результат – одномерный вектор:
```

```
[1] 120 128 136 144
```

```
# Применим функцию sum() к каждому элементу M[, *, ], т.е. выполним суммирование по третьему измерению:
```

```
apply(M, c(1, 2), sum)
```

```
# Результат - матрица:
```

```
      [,1] [,2] [,3] [,4]  
[1,]   18   26   34   42  
[2,]   20   28   36   44  
[3,]   22   30   38   46  
[4,]   24   32   40   48
```

При необходимости вычисления сумм и средних значений по строкам или столбцам матриц рекомендуется также использовать очень быстрые и специально оптимизированные для этого функции `colSums()`, `rowSums()`, `colMeans` и `rowMeans()`.

Функция **`lapply()`** – используется в случаях, когда необходимо применить какую-либо функцию к каждому компоненту списка и получить результат также в виде списка (буква "l" в названии `lapply()` означает list – "список").

```
# Создадим список с тремя компонентами-векторами:
```

```
x <- list(a = 1, b = 1:3, c = 10:100)
```

```
# Выясним размер каждого компонента списка x (функция  
length()):
```

```
lapply(x, FUN = length)
```

```
$a
```

```
[1] 1
```

```
$b
```

```
[1] 3
```

```
$c
```

```
[1] 91
```

```
# Выполним суммирование элементов в каждом компоненте  
списка x:
```

```
lapply(x, FUN = sum)
```

```
$a
```

```
[1] 1
```

```
$b
```

```
[1] 6
```

```
$c
```

```
[1] 5005
```

Функция **sapply()** – используется в случаях, когда необходимо применить какую-либо функцию к каждому компоненту списка, но результат вывести в виде вектора (буква "s" в названии sapply() означает simplify – "упростить").

```
# Список из трех компонентов:
x <- list(a = 1, b = 1:3, c = 10:100)
# Выясним размер каждого компонента списка x:
sapply(x, FUN = length)
  a  b  c
  1  3 91
# Суммирование всех элементов в каждом компоненте списка x:
sapply(x, FUN = sum)
  a    b    c
  1    6 5005
```

В некоторых более "продвинутых" случаях sapply() может выдать результат в виде многомерного массива. Например, если применяемая нами функция возвращает векторы одинаковой длины, sapply() объединит эти векторы в матрицу (по столбцам):

```
sapply(1:5, function(x) rnorm(3, x))
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.4522206 4.3936827 2.9647418 3.113249 4.736281
[2,] 0.5352991 1.6610181 3.0150867 3.014616 2.488926
[3,] 0.4529069 0.8300361 0.8131918 3.782212 6.019982
```

Если применяемая функция возвращает матрицу, то sapply() преобразует каждую матрицу в вектор и объединит такие векторы в одну большую матрицу (звучит не очень понятно, но пример хорошо поясняет эту идею):

```
sapply(1:5, function(x) matrix(x, 2, 2))
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     2     3     4     5
[2,]     1     2     3     4     5
[3,]     1     2     3     4     5
[4,]     1     2     3     4     5
```

Поведение sapply(), продемонстрированное в последнем примере, можно отменить при помощи аргумента simplify = "array" – в этом случае матрицы будут объединены в один многомерный массив:

```
sapply(1:5, function(x) matrix(x, 2, 2), simplify =
"array")
```

```

, , 1
      [,1] [,2]
[1,]    1    1
[2,]    1    1
, , 2
      [,1] [,2]
[1,]    2    2
[2,]    2    2
, , 3
      [,1] [,2]
[1,]    3    3
[2,]    3    3
, , 4
      [,1] [,2]
[1,]    4    4
[2,]    4    4
, , 5
      [,1] [,2]
[1,]    5    5
[2,]    5    5

```

Функция **replicate()** является своего рода "оберткой" для функции `sapply()` и позволяет провести серию вычислений с целью генерации набора чисел по заданному алгоритму. Синтаксис функции имеет вид:

```
replicate(n, expr, simplify=TRUE)
```

где `n` – число повторов, `expr` – функция или группа выражений, которые надо повторить `n` раз, `simplify = TRUE` – необязательный параметр, который пробует упростить результат и представить его в виде вектора или матрицы значений.

Рассмотрим пример использования функции `replicate()` для проверки статистической гипотезы о равенстве медиан двух выборок бутстреп-методом. Созданная нами функция `boot_med()` из каждой исходной выборки `x` и `y` извлекает по `N` псевдовыборок, используя алгоритм "случайного выбора с возвратом", и находит разность их медиан. Здесь функция `sample.int()` формирует целочисленные наборы случайных индексов `indx` и `indy` для каждого из сравниваемых векторов:


```
boot_med <- function(x, y, N = 100) {
  replicate(N, {
    indx <- sample.int(length(x), length(x), replace = T)
    indy <- sample.int(length(y), length(y), replace = T)
    median(x[indx]) - median(y[indy])
  })
}
```

Для тестирования функции проверим однородность медиан для двух случайных выборок из нормального распределения со средним `mean = 10` и стандартным отклонением `sd = 4`:

```
Y <- rnorm(100, sd=4, mean=10)
X <- rnorm(100, sd=4, mean=10)
quantile(boot_med(Y,X,10000), probs = c(0.025, 0.975))
      2.5%      97.5%
-1.7906891  0.9941717
```

Поскольку доверительный интервал разности медиан включает 0, то нулевую гипотезу отклонять не следует.

Функция **vapply()** – схожа с `sapply()`, но работает несколько быстрее за счет того, что пользователь однозначно указывает тип возвращаемых значений (буква "v" в названии `vapply()` означает velocity – "скорость"; пример сравнения `sapply()` и `vapply()` можно найти на stackoverflow.com). Такой подход позволяет также избегать сообщений об ошибках (и прерывания вычислений), возникающих при работе с `sapply()` в некоторых ситуациях. При вызове `vapply()` пользователь должен привести пример ожидаемого типа возвращаемых значений. Для этого служит аргумент `FUN.VALUE`:

Аргументу `FUN.VALUE` присвоено логическое значение `FALSE`. Этим задается тип возвращаемых функцией значений, который ожидает пользователь

```
a <- vapply(NULL, is.factor, FUN.VALUE = FALSE)
# Функция sapply() применена к тому же NULL-объекту:
b <- sapply(NULL, is.factor)
# Проверка типа переменных:
is.logical(a)
[1] TRUE
is.logical(b)
[1] FALSE
```

Функция **mapply()** – используется в случаях, когда необходимо поэлементно применить какую-либо функцию одновременно к нескольким объектам (например, получить сумму первых элементов векторов, затем сумму вторых элементов векторов, и т.д.). Результат возвращается в виде вектора или массива другой размерности (см. примеры для `sapply()` выше). Буква "m" в названии означает *multivariate* – "многомерный" (имеется в виду одновременное выполнение вычислений над элементами нескольких объектов).

```
mapply(sum, 1:5, 1:5, 1:5)
```

```
[1] 3 6 9 12 15
```

Функция **rapply()** – используется в случаях, когда необходимо применить какую-либо функцию к компонентам вложенного списка (буква "r" в названии означает *recursively* – "рекурсивно").

```
# Пользовательская функция, которая добавляет ! к элементу
# объекта x если этот элемент является текстовым выражением
# или добавляет 1 если этот элемент является числом:
```

```
myFun <- function(x){
  if (is.character(x)){
    return(paste(x, "!", sep = ""))
  }
  else{
    return(x + 1)
  }
}
```

```
# Пример вложенного списка:
```

```
l <- list(a = list(a1 = "Boo", b1 = 2, c1 = "Eeek"), b = 3,
c = "Yikes", d = list(a2 = 1, b2 = list(a3 = "Hey", b3 = 5)))
```

```
# Рекурсивное применение функции myFun к списку l:
```

```
rapply(l, myFun)
      a.a1      a.b1      a.c1      b      c      d.a2  d.b2.a3  d.b2.b3
"Boo!"      "3"    "Eeek!"    "4"  "Yikes!"    "2"    "Hey!"    "6"
```

Если необходимо вернуть результат в виде вложенного списка, можно воспользоваться аргументом `how = "replace"`. В этом случае исходные значения в списке `l` будут заменены на новые:

```

rapply(1, myFun, how = "replace")
$a
$a$a1
[1] "Boo!"
$a$b1
[1] 3
$a$c1
[1] "Eeek!"
$b
[1] 4
$c
[1] "Yikes!"
$d
$d$a2
[1] 2
$d$b2
$d$b2$a3
[1] "Hey!"
$d$b2$b3
[1] 6

```

Функция **tapply()** – используется в случаях, когда необходимо применить какую-либо функцию `fun` к отдельным группам элементов вектора `x`, заданным в соответствии с уровнями какого-либо фактора `group`:

```
tapply(x, group, fun, ...)
```

Например, в следующем фрагменте кода функция `sample()` используется для создания двух случайных выборок: из 50 значений целых чисел от 1 до 4 и связанных с ними меток четырех групп A-D. Функция `tapply()` подсчитывает суммы `x` для каждого из значений фактора:

```

> x <- sample(1:4, size = 50, replace = T)

> gr <- as.factor(sample(c("A","B","C","D"), size = 50, replace = T))

> x
[1] 3 1 2 4 1 2 4 2 4 3 2 4 2 2 3 2 2 4 1 1 1 3 4 4 1 2 2 4 2 1 3 1 3 2 3
[36] 4 1 4 3 2 1 4 4 1 2 4 2 2 1 4

> gr
[1] D C C D A C C C A C D B C C D A B D C B D A C A C B A B B C C B D C B
[36] C A C C B D B D D D D B B A D

```

```
Levels: A B C D
```

```
> tapply(x, gr, sum)
```

```
A B C D
```

```
18 29 41 36
```

Функция **outer()** позволяет выполнить комбинаторную операцию `fun` над элементами двух массивов или векторов `x` и `y`, не прибегая к явному использованию "двойного" цикла. По умолчанию осуществляется операция попарного перемножения.

```
outer(x, y, fun="*", ...)
```

```
x <- 1:5
```

```
y <- 1:5
```

```
outer(x, y)
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     2     3     4     5
[2,]     2     4     6     8    10
[3,]     3     6     9    12    15
[4,]     4     8    12    16    20
[5,]     5    10    15    20    25
```

Используя, например, функцию `outer()` вместе с функцией `paste()`, можно сгенерировать все возможные попарные комбинации "связок" элементов символьного и целочисленного векторов:

```
> x <- c("A", "B", "C", "D")
```

```
> y <- 1:10
```

```
> outer(x, y, paste, sep = "")
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] "A1" "A2" "A3" "A4" "A5" "A6" "A7" "A8" "A9" "A10"
[2,] "B1" "B2" "B3" "B4" "B5" "B6" "B7" "B8" "B9" "B10"
[3,] "C1" "C2" "C3" "C4" "C5" "C6" "C7" "C8" "C9" "C10"
[4,] "D1" "D2" "D3" "D4" "D5" "D6" "D7" "D8" "D9" "D10"
```

Наряду с функциями `apply()`-семейства, существует еще несколько полезных векторизованных функций. Самая интересная из них, наверное, функция **do.call()**, которая вызывает выражение из своего первого аргумента для каждого элемента своего второго аргумента-списка. Это очень удобно, когда нужно преобразовать список в матрицу или таблицу данных, поскольку ни `cbind()`, ни `rbind()` не могут обрабатывать списки. Вот как это делается:

```

> (spisok <- strsplit(c("Вот как это делается", "Другое немного похожее предложение"), " "))

[[1]]

[1] "Вот" "как" "это" "делается"

[[2]]

[1] "Другое" "немного" "похожее" "предложение"

> do.call("cbind", spisok)
      [,1]      [,2]
[1,] "Вот"      "Другое"
[2,] "как"      "немного"
[3,] "это"      "похожее"
[4,] "делается" "предложение"

> do.call("rbind", spisok)
      [,1]      [,2]      [,3]      [,4]
[1,] "Вот"      "как"      "это"      "делается"
[2,] "Другое"   "немного" "похожее" "предложение"

```

Мы сделали здесь полезную вещь — разобрали две строки текста на слова и разместили их по ячейкам. Это часто требуется при конвертации данных. Обратите внимание на первую команду — она выводит свой результат одновременно и на экран, и в объект `spisok`! Таков эффект наружных круглых скобок.

Литература:

Маслицкий, С. Э. Статистический анализ и визуализация данных с помощью R [Электронный ресурс] / С. Э. Маслицкий, В. К. Шитиков. – 2014. – Режим доступа: <http://www.ievbras.ru/ecostat/Kiril/R/Mastitsky%20-and%20Shitikov%202014.pdf>. – Дата доступа: 01.09.2016.

Шипунов, А. Б. Наглядная статистика. Используем R! [Электронный ресурс] / А. Б. Шипунов, Е. М. Балдин, П. А. Волкова, А. И. Коробейников, С. А. Назарова, С. В. Петров, В. Г. Суфиянов. – 2014. – Режим доступа: <https://cran.r-project.org/doc/contrib/Shipunov-rbook.pdf>. – Дата доступа: 01.09.2016.