МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Дзержинский филиал ННГУ

В. А. Гришин

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ R

Учебно-методическое пособие

Нижний Новгород

2021

УДК - 004.432.42
ББК – 32.973.3
Г-85

Г-85 Гришин В.А.: ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ R: Учебно-методическое пособие — Нижний Новгород: Нижегородский госуниверситет, 2021. – 67 с.

Рецензент: к.техн.н., доцент Васин Д. Ю.

Учебно-методическое пособие предназначено для методической поддержки самостоятельной работы студентов, обучающихся по направлению подготовки 09.03.03 «Прикладная информатика» при изучении дисциплин «Языки программирования для больших данных», «Прикладная статистика», «Математическое и имитационное моделирование» и «Эконометрика». Пособие включает изучение языка программирования R, который может использоваться для автоматизации анализа данных. При освоении R студенты учатся эффективно организовывать данные, что обеспечивает согласованность семантики набора данных со способом их хранения. Знакомятся с функциональным и объектно – ориентированным программированием, математическим моделированием и преобразованием сложных данных в простые форматы. В пособии также приведены примеры программных кодов обработки различных наборов данных на языке R, что значительно облегчит самостоятельную работу студентов.

Методическое пособие является и руководством по использованию R, как мощной программной среды статистических вычислений и предназначено для студентов ННГУ, обучающихся по направлению 09.03.03 «Прикладная информатика», а также для преподавателей, магистрантов, аспирантов и широкого круга специалистов, которые хотели бы применять современные методы автоматизации обработки и анализа своих данных.

Ответственный за выпуск:
председатель Методической комиссии Дзержинского филиала ННГУ
д.э.н., профессор Павленков М. Н.

УДК - 004.432.42
ББК – 32.973.3

Содержание

Введение

Учебно-методическое пособие предназначено для методической поддержки самостоятельной работы студентов, обучающихся по направлению подготовки 09.03.03 «Прикладная информатика» при изучении дисциплин «Языки программирования для больших данных», «Прикладная статистика», «Математическое и имитационное моделирование» и «Эконометрика».

Содержание пособия, как и содержание указанных курсов, разработано в соответствии с требованиями ОС ННГУ по направлению 09.03.03 «Прикладная информатика» в области формирования необходимых компетенций.

В пособии даётся представление об одном из самых мощных, профессиональных и современных языков программирования – языком R. R – это свободно распространяемая программная среда с открытым кодом, развиваемая в рамках проекта GNU. Программная среда R представляет огромный набор инструментов для понимания и автоматизации анализа данных. По своей мощи она сравнима с коммерческими продуктами, а часто и превосходит их в большинстве практических аспектов – разнообразии поддерживаемых операций, программируемости, средствах графического вывода.

R не ограничивается выполнением статистических операций – это язык, который обладает возможностями, присущими объекто – ориентированным и функциональным парадигмам программирования.

Так как R распространяется с открытым исходным кодом, то студенты могут легко получить помощь от сообщества пользователей, что полноценно дополняет самостоятельную работу по дисциплинам «Языки программирования для больших данных», «Прикладная статистика», «Математическое и имитационное моделирование» и «Эконометрика».

В результате чтения этого пособия студенты получат представление о том, как работает R, где можно получить дальнейшую информацию, а также справиться с простыми и достаточно сложными задачами анализа данных.

Предлагаемое пособие поможет студентам сориентироваться при организации текущей аудиторной и внеаудиторной самостоятельной работы, при подготовке к проверочным работам, экзаменам.

1. Основы написания кода в R

1.1. Оператор присваивания

R – объектно-ориентированный язык: переменные, данные, матрицы, функции, результаты, и т.д., хранятся в оперативной памяти компьютера в форме объектов, которые имеют имя. Для того чтобы присвоить значение объекту используется символ "<-". Этот символ пишется вместе со знаком минус так, чтобы они представляли стрелку, которая может быть направлена слева направо, или наоборот, можно также использовать знак равенства:

> m<-33
> 55->n
> k=77

Для отображения значения необходимо напечатать название объекта:

> m
[1] 33
> n
[1] 55
> k
[1] 77

Значение также, может быть результатом арифметического выражения:

> m<-33+22
> m
[1] 55

Можно просто напечатать выражение, не присваивая ему имя, тогда результат будет отображен на экране, но не сохранен в памяти:

> 3*3+3
[1] 12

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.

1.2. Отображение объектов в памяти и удаление

Функция **ls()** отображает список всех объектов текущей среды, находящихся в памяти, возвращая только их имена:

> ls()
[1] "k" "m" "n"

Если объектов в памяти много, может быть целесообразно, перечислить только те объекты, в названии которых содержатся определенные символы. Это может быть сделано с помощью параметра **pattern** (который может быть сокращён: **pat**):

> ls(pat="m")
[1] "m" "name"

Если мы хотим ограничить список объектов, например, названиями которые начинаются с этого символа, то тогда команда отображения запишется так:

> name<-15
> ls(pat="^m")
[1] "m"

Чтобы показать характеристики объектов, можно использовать функцию **ls.str():**

> ls.str()
k : num 77
m : num
n : num 55
name : num 15

Для удаления объекта из памяти, используется функция **rm()**: **rm(x)** удалит объект **x**, **rm(x,y)** - объекты **x** и **y** , **rm(list= ls())** удалит все объекты.

> rm(list=ls())
> ls()
character(0)

1.3. Получение помощи по функциям и средствам

R обладает обширными справочными материалами. Встроенная система помощи содержит подробные разъяснения, ссылки на литературу и примеры для каждой функции из установленных пакетов. Справку можно вызвать при помощи функций, перечисленных в табл. 1.

Таблица 1

Функции вызова справки в R

Функция	Действие
help.start()	Общая справка
help (“*предмет*”) или ? *предмет*	Справка по функции *предмет* (кавычки необязательны)
help.search(“*предмет*”) или ??*предмет*	Поиск в справке записей, содержащих *предмет*
example(“*предмет* ”)	Примеры использования функции *предмет* (кавычки необязательны)
RSiteSearch(“*предмет* ”)	Поиск записей, содержащих *предмет* в онлайн-руководствах и заархивированных рассылках
apropos(“*предмет*”, mode=”function”)	Список всех доступных функций, в названии которых есть *предмет*
library(help="библиотека")	Справка о библиотеке

Функция **help.start()** открывает окно браузера с перечнем доступных руководств разного уровня сложности, часто задаваемых вопросов и ссылок на источники. Функция **RSiteSearch()** осуществляет поиск на заданную тему в онлайн-руководствах и архивах рассылок и представляет результаты в окне браузера.

Другие полезные команды:

- **getwd()** - вывод текущей (рабочей директории);
- ***getwd()***
- ***[1] "C:/Users/Дом/Documents"***
- **setwd**("имя_новой_рабочей_директории") - смена рабочей директории;
- ***setwd("D:/Rdata")***
- **dir()** - выводит список файлов в рабочей директории.

2. Классы объектов, математические и статистические функции преобразования данных в R

2.1. Классы объектов в R

Все данные в R можно поделить на следующие классы **(режимы)**:

1. **numeric** - название класса, а также типа объектов. К нему относятся действительные числа. Объекты данного класса делятся на целочисленные (**integer**) и собственно действительные (**double** или **real**).
2. **complex** - объекты комплексного типа.
3. **logical** - логические объекты, принимают только два значения: **FALSE (F)** и **TRUE (T)**.
4. **character** - символьные объекты (символьные переменные задаются либо в двойных кавычках (“ ”), либо в одинарных (‘’)).

Numeric

Объект класса **numeric** создаётся при помощи команды **numeric(n)**, где **n** количество элементов данного типа. Создаётся нулевой вектор длины **n**.) *(Определение вектора в R дано в п.3.1).*

```
> m <- numeric(6)
> m
[1] 0 0 0 0 0 0
```

В результате создан нулевой вектор типа **numeric** длины 6.

Объекты типов **integer** и **double** создаются, соответственно, при помощи команд **integer(n)** и **double(n)**, а при помощи функций **is.numeric (имя_объекта), is.double (имя_объекта)**, **is.integer (имя_объекта)** можно проверить объект на принадлежность к классу **numeric (double и integer)**. Десятичным разделителем для чисел является точка.

```
> x<-double(2)
> x<-1.2
> is.double(x)
[1] TRUE
> n<-integer(1)
> n<-3
> is.double(n)
[1] TRUE
> is.integer(n)
[1] FALSE
> is.integer(x)
[1] FALSE
```

Переменная **n** была создана целочисленной **(n=integer(1))** и ей присвоили значение **3**. Но после проверки она оказалась не **integer**, а **double**. Почему?

Потому, что по умолчанию, все числа в R являются вещественными. Чтобы сделать их целочисленными, надо воспользоваться командой **as.integer (имя_объекта)**.

Объекты режима **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.

Logical

Перейдём теперь к логическим объектам, т.е. объектам типа **logical**. Объекты этого класса принимают два возможных значения: **TRUE** (истина) и **FALSE** (ложь), и создаются при помощи команды **logical(n)**, где **n** - это длина создаваемого вектора.

```
> x<-logical(4)
> x
[1] FALSE FALSE FALSE FALSE
```

Как видно из примера, при обращении к команде **logical(4)** был создан логический вектор, состоящий только из **FALSE**.

Проверка объекта на принадлежность к логическому типу осуществляется при помощи **is.logical (имя_объекта).**

```
> x<-3;t<-T;
> is.logical(x)
[1] FALSE
> is.logical(T)
[1] TRUE
```

Логические выражения могут составляться с использованием следующих логических операторов:

- "Равно" **= =**
- "Не равно" **!=**
- "Меньше" **<**
- "Больше" **>**
- "Меньше либо равно" **<=**
- "Больше либо равно" **>=**

- "Логическое И" **&**
- "Логическое ИЛИ" **|**
- "Логическое НЕ" **!**

Перевести объект в логический можно при помощи функции **as.logical (имя_объекта)**.

Character

Последний класс объектов, который будет рассмотрен в данном разделе - это объекты типа **character**, т.е символьные объекты. Создаются при помощи команды **character(n)**, результат - пустой символьный вектор размерности **n.**

> ch<-character(1)
> ch
[1] ""

Проверка на принадлежность к данному типу осуществляется при помощи **is.character().**

> is.character(ch)
[1] TRUE

Символьные объекты обязательно задаются в кавычках (одинарных или двойных). Символьным объектом может быть как просто символ, так и строка.

> ax<-'mail'
> ax
[1] "mail"
> xa<-"символьная"
> xa
[1] "символьная"

Объекты любого типа можно перевести в символьные. Для этого нужно воспользоваться командой **as.character (имя_объекта)**.

> t<-F;r<-3.98;
> ct<-as.character(F)
> ct
[1] "FALSE"
> cr<-as.character(r)
> cr
[1] "3.98"

Символьный же объект перевести в иной тип сложнее.

> ch<-'3';n<-as.numeric(ch)
> n
[1] 3
> ch<-"mail";m<-as.numeric(ch);m
Warning message:
NAs introduced by coercion
[1] NA

Символьный объект можно перевести в числовой, если он представляет из себя число, окруженное кавычками. Если же в кавычках стоял

непосредственно символ (или набор символов), то такой перевод приведёт к появлению **NA.**

Символьные переменные **'T'** и **'F'** можно перевести в логические **TRUE** и **FALSE.**

Тип любого объекта можно проверить (и изменить) при помощи функции **mode (имя_объекта)**.

```
> t<-'T'
> ch<-as.logical(t)
> ch
[1] TRUE
> mode(ch)
[1] "logical"
> mode(ch)= "numeric"
> ch
[1] 1
```

Если надо проверить тип переменной в процессе выполнения программы, то применяют функцию **typeof()**:

```
> y <- 21
> typeof(y)
[1] "double"
> x <- "78"
> typeof(x)
[1] "character"
```

2.2. Специальные переменные

В R существует ряд особых объектов:

- **Inf** - бесконечность: положительная **(+∞ Inf)** и отрицательная **(-∞- Inf)**;
- **NA** - отсутствующее значение (Not Available);
- **NaN** - не число (Not a Number);
- **NULL** - ничто.

Все эти объекты можно использовать в любых выражениях. Рассмотрим их более подробно.

Inf появляется при переполнении и в результате операций вида **a/0**, где **a** действительное число не равное нулю.

```
> x<-5/0
> x
[1] Inf
> y<-log(0)
> y
[1] -Inf
```

Проверить объект на конечность (бесконечность) можно при помощи команд **is.finite() (is.infinite())**:

> is.finite(x)
[1] FALSE
> is.infinite(y)
[1] TRUE

Объект **NaN - не числовой**, появляется при операциях над числами, результат которых не определён (не является числом):

> a<-0/0
> a
[1] NaN
> Inf - Inf
[1] NaN

При помощи **is.nan(имя_объекта)** можно проверить, является ли объект **NaN**.

Отсутствующее значение - **NA** - возникает, если значение некоторого объекта не доступно (не задано). Включает в себя и **NaN**. Проверка, относится ли объект к **NA**, делается при помощи **is.na (имя_объекта)**.

Ничто - **NULL** - нулевой (пустой) объект. Возникает как результат выражений (функций), чьи значения не определены. Обнулить объект можно при помощи команды **as.null (имя_объекта)**, проверить объект на принадлежность к **NULL** можно при помощи функции **is.null (имя_объекта)**.

> x<-99
> a<-as.null(x)
> is.null(x)
[1] FALSE
> is.null(a)
[1] TRUE

2.3. Генерация (создание) данных

Регулярные последовательности

Регулярная последовательность целых (**integer)** чисел, например от 1 до 30, может быть создана следующим оператором:

> x<-1:30
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Результирующий вектор имеет 30 элементов **x**. Оператор ':' имеет приоритет над арифметическими операторами:

> 1:10-1
[1] 0 1 2 3 4 5 6 7 8 9

> 1:(7+1)
[1] 1 2 3 4 5 6 7 8

Функция **seq()** создает последовательности действительных чисел (**double**):

> seq(0,6,0.5)

[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0

– где первый параметр указывает начальное число последовательности, второй конечное число, и третье – приращение. Можно использовать также следующий вариант:

> seq(length=3, from = 2, to =6)

[1] 2 4 6

Также возможно напечатать непосредственно значения, используя функцию **c()**:

> c(0,1,3.1,6,10.9)

[1] 0.0 1.0 3.1 6.0 10.9

Функция **rep()** создает вектор с одинаковыми элементами:

> rep(2,5)

[1] 2 2 2 2 2

Функция **sequence()** создает ряд последовательностей целых чисел каждая, из которых заканчивается числом, которое является параметром функции:

> sequence (2:5)

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

Функция **gl(k,n)** создает правильный ряд факторных переменных, где **k** – количество уровней (или классов) и **n** – количество чисел в каждом уровне.

Дополнительно могут использоваться два параметра: **length** – длина, чтобы определить количество чисел в общей последовательности и **labels**, чтобы определить названия коэффициентов (факторов). Например:

> gl (3,5)

[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3

Levels: 1 2 3

> gl (3,5,30)

[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3

Levels: 1 2 3

> gl (2,5, label=c ("Классификация", "Систем"))

[1] Классификация Классификация Классификация Классификация Классификация Систем

[7] Систем Систем Систем Систем

Levels: Классификация Систем

Случайные последовательности

R позволяет генерировать набор случайных данных для большого количества функций плотности вероятности. Эти функции имеют следующий вид:

> rfunc (n, p [1], p [2]...),

где **func** определяет закон вероятности, **n** - число данных и **p[1], p[2]** значения параметров закона. Следующая ниже таблица отображает значения параметров для каждого закона, и возможные значения по умолчанию (если ни одно значение по умолчанию не обозначено, это означает, что параметр должен быть определен пользователем).

Таблица 2

Генерирование псевдослучайных чисел

Закон	**Параметры**
Гауссовское распределение	*rnorm (n, mean=0, sd=1)*
Экспоненциальное распределение	*rexp (n, rate=1)*
Гамма распределение	*rgamma (n, shape, scale=1)*
Распределение Пуассона	*rpois (n, lambda)*
Распределение Вейбула	*rweibull (n, shape, scale=1)*
Распределение Коши	*rcauchy(n,location=0,scale=1)*
Бета распределение	*rbeta (n, shape1, shape2)*
Распределение Стьюдента(t)	*rt(n, df)*
Распределение Фишера(F)	*rf (n, df1, df2)*
Распределение Пирсона (X)	*rchisq (n, df)*
Биноминальное распределение	*rbinom (n, size, prob)*
Геометрическое распределение	*rgeom (n, prob)*
Гипергеометрическое распределение	*rhyper (nn, m, n, k)*
Логистическое распределение	*rlogis (n, location=0, scale=1)*
Логнормальное распределение	*rlnorm (n, meanlog=0,sdlog=1)*
Отрицательное биноминальное распределение	*rnbinom (n, размер, prob)*
Равномерное распределение	*runif (n, min=0, max=1)*
Распределение Вайлкоксона	*rwilcox (nn, м., n),rsignrank (nn, n)*

Все эти функции можно использовать, заменяя символ **r** на **d**, **p** или **q**, тогда вычисляем плотность вероятности **dfunc(x**), кумулятивная плотность вероятности **pfunc(x),** и значения квантилей **qfunc (p),** (0 <p<1).

Создание выборок из больших наборов данных – обычное дело при поиске структуры в данных или в машинном обучении. Функция **sample()** позволяет создавать случайные выборки (с замещением или без него) заданного объема из анализируемого набора данных.

2.4. Математические и статистические функции

В основном математические функции применяются для преобразования данных. К примеру, данные с положительно асимметричным распределением перед дальнейшей обработкой обычно логарифмируют. Математические функции также используют при составлении формул, создании графиков (например, кривая зависимости **x** от **sin(x)**) и форматировании числовых значений перед выводом на экран.

В табл. 3 приведены примеры применения математических функций к скалярам (отдельным числам). Когда эти функции применяются к числовым векторам, матрицам или таблицам данных, они преобразуют каждое число по отдельности. Например, **sqrt(c(4, 16, 25))** возвращает вектор **c(2, 4, 5)**.

Таблица 3

Математические функции

№	Функция	Описание
1	**abs(x)**	Модуль abs(-4) равно 4
2	**sqrt(x)**	Квадратный корень sqrt(25) равно 5. Это то же, что и 25^(0.5)
3	**ceiling(x)**	Наименьшее целочисленное значение, не меньшее, чем x ceiling(3.457) равно 4
4	**floor(x)**	Наибольшее целочисленное значение, не большее, чем x floor(3.457) равно 3
5	**trunk(x)**	Целое число, полученное при округлении x в сторону нуля trunk(5.99) равно 5
6	**round(x, digits=n)**	Округляет x до заданного числа знаков после запятой round(3.475, digits=2) равно 3.48
7	**signif(x, digits=n)**	Округляет x до заданного числа значащих цифр signif(3.475, digits=2) равно 3.5
8	**cos(x), sin(x), tan(x)**	Косинус, синус и тангенс cos(2) равно –0.416
9	**acos(x), asin(x), atan(x)**	Арккосинус, арксинус и

		арктангенс acos(-0.416) равно 2
10	**log(x, base=n),log(x), log10(x)**	Логарифм x по основанию n Для удобства: log(x) – натуральный логарифм log10(x) – десятичный логарифм log(10) равно 2.3026 log10(10) равно 1
11	**exp(x)**	Экспоненциальная функция exp (2.3026) приблизительно равно 10

Самые распространенные статистические функции перечислены в табл.4. У многих из них есть дополнительные параметры, которые влияют на результат. Например,

y <- mean(x)

позволяет вычислить среднее арифметическое для всех элементов объекта x, а **z <- mean(x, trim = 0.05, na.rm=TRUE)** вычисляет усеченное среднее, исключив 5% наибольших и 5% наименьших значений в выборке, не принимая при этом во внимание пропущенные значения. Используйте команду **help()**, чтобы узнать больше о каждой функции и ее аргументах.

Таблица 4

Статистические функции

№	Функция	Описание
1	**mean(x)**	Среднее арифметическое mean(1:4) равно 2.5
2	**median(x)**	Медиана median(c(1,2,3,4)) равно 2.5
3	**sd(x)**	Стандартное отклонение sd(c(1,2,3,4)) равно 1.291
4	**var(x)**	Дисперсия var(c(1,2,3,4)) равно 1.666667
5	**mad(x)**	Абсолютное отклонение медианы mad(c(1,2,3,4)) равно 1.4826
6	**quantile(x, probs)**	Квантили, где x – числовой вектор, для которого нужно вычислить квантили, а probs –

		числовой вектор с указанием вероятностей в диапазоне [0; 1]. y<-quantile(x,c(0.25,0.75))
7	**range(x)**	Размах значений x <- c(1,2,3,4) равно 1 4.
8	**sum(x)**	Сумма sum(c (1,2,3,4)) равно 10
9	**diff(x, lag=n)**	Разность значений в выборке, взятых с заданным интервалом (lag). По умолчанию интервал равен 1.
10	**min(x)**	Минимум min(c(1,2,3,4)) равно 1
11	**max(x)**	Максимум max(c(1,2,3,4)) равно 4

3. Структуры данных

3.1. 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()**, для создания векторов, содержащих последовательную совокупность чисел, удобна функция **seq()**, векторы, содержащие одинаковые значения, создаются при помощи функции **rep()**.

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:

```
> # создаем текстовый вектор text.vect:
> text.vect <- c("a", "b", "c")
> # объединяем числовой вектор v1 (см. выше)
> # с текстовым вектором text.vect:
> new.vect <- c(v1, text.vect)
> # просмотр содержимого нового вектора new.vect:
> new.vect
[1] "1" "2" "3" "a" "b" "c"
> # все значения нового вектора взяты в кавычки,
> # что указывает на их текстовую природу;
> # для подтверждения этого воспользуемся командой mode():
> 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 обратится к конкретному элементу необходимо указать имя вектора и индекс этого элемента в квадратных скобках:

```
> # создадим числовой вектор y, содержащий 5 числовых значений:
> y <- c(5, 3, 2, 6, 1)
> # проверим, чему равен третий элемент вектора y:
> y[3]
```

[1] 2

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

> # создадим еще один числовой вектор z, содержащий 3 значения:
> z <- c(0.5, 0.1, 0.6)
> # умножим первый элемент вектора y на третий элемент вектора z
> # (т.е. 5*0.6):
> y[1]*z[3]
[1] 3

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

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

Из этого же вектора можно выбрать, например, только первое и четвертое значения, используя уже известную функцию конкатенации **c()**:

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

Похожим образом можно удалить первое и четвертое значения из вектора **y**, применив знак "минус" перед функцией конкатенации:

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

Для вычисления скалярного произведения двух векторов используется функция **crossprod()**:

> vector1 <- c(2, 4, 6)
> vector2 <- seq(1, 3)
> crossprod(vector1,vector2)
[,1]
[1,] 28

В качестве критерия для выбора значений может служить логическое выражение. Для примера можно выбрать из вектора **y** все элементы , квадраты которых больше 8 , и присвоить этому подвектору имя **z**:

> y<-c(5,2,-3,8)
> z<-y[y*y>8]
> z
[1] 5 -3 8

Фильтрация в R играет настолько важную роль, что стоит изучить подробности того, как R достигает нужной цели. Это выполняется следующим образом:

y<-c(5,2,-3,8)
> y
[1] 5 2 -3 8

> y*y
[1] 25 4 9 64
> y*y>8
[1] TRUE FALSE TRUE TRUE

Выражение **y*y>8** дает вектор логических значений. Необходимо отметить, что в выражении **y*y>8** все компоненты являются векторами или векторными операторами:

- так как **y** является вектором, то и **y*y** тоже будет вектором такой же длины;
- число **8** после переработки превращается в вектор **(8,8,8,8)**;
- операторы **>**, *****, как и **+** являются функциями.

Таким образом, запись

y*y>8 в действительности означает

> ">"(y*y,8)
[1] TRUE FALSE TRUE TRUE

Фильтрация также может выполнятся с использованием функции **subset()**. Применительно к векторам различия между этой функцией и обычной фильтрацией заключается в способе обработки значений **NA**.

> x<-c(6,1:3,NA,15)
> x
[1] 6 1 2 3 NA 15
> x[x>5]
[1] 6 NA 15
> subset(x,x>5)
[1] 6 15

В некоторых случаях достаточно найти позиции **y**, для которых выполняется условие. Это можно сделать при помощи функции выбора **which()**:

> y<-c(5,2,-3,8)
> which(y*y>8)
[1] 1 3 4

Функция **which()** определяет, для каких элементов вектора **y*y** выражение **y*y>8 TRUE**.

Существуют еще две функции, которые могут оказаться полезными при анализе свойств векторов и других совокупностей – **which.min()** и **which.max()**. Как следует из названий, эти функции позволяют выяснить порядковые номера элементов, обладающих минимальным и максимальным значениями соответственно. Если минимальное/максимальное значение принимают несколько элементов в векторе, то будет возвращен порядковый номер первого элемента с этим значением.

В R включена векторизованная функция **ifelse()**, которая вызывается в следующей форме:

ifelse (b,u,v),

где **b** – логический вектор, а **u**, **v** – векторы.

Например:

> x<-1:10

> y<-ifelse(x %% 2 == 0,1,0)

> y

[1] 0 1 0 1 0 1 0 1 0 1

В данном случае создается вектор, который содержит **1** для четных аргументов **x** или **0** для нечетных **x**.

Другой пример:

> y<-c(5,2,-3,8)

> ifelse(y>6,2*y,3*y)

[1] 15 6 -9 16

R возвращает вектор, с элементами **y**, умноженными на **2** или на **3** в зависимости от того, превышает ли элемент **6**.

Проверку равенства векторов выполняют либо с помощью функции **all()**, либо функцией **identical()**:

> y<-c(5,2,-3,8)

> x<-1:4

> x==y

[1] FALSE TRUE FALSE FALSE

> all(x==y)

[1] FALSE

> identical(x,y)

[1] FALSE

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(y)

[1] -3 2 5 8

> sort(y, decreasing = TRUE)

[1] 8 5 2 -3

Элементам вектора можно присвоить имена. Например, допустим у нас есть вектор с численностью населения городов Нижегородской области. Элементам можно присвоить имена, соответствующие названиям малых городов – «Арзамас», «Сергач» и т. д., что позволит выводить названия городов на диаграммах.

Для назначения или чтения имён элементов векторов применяется функция **names()**:

> x<-c(1,2,3)

> names(x)

NULL

> names(x)<-c("a","b","c")

> names(x)

[1] "a" "b" "c"

> x
a b c
1 2 3
Чтобы удалить имена из вектора, присваиваем NULL:
> names(x)<-NULL
> x
[1] 1 2 3
К элементам вектора также можно обращаться по именам:
> x<-c(1,2,3)
> names(x)<-c("a","b","c")
> x['b']
b
2

3.2. Матрицы и массивы

Матрица в R представляет собой вектор, который содержит два дополнительных атрибута: количество строк и количество столбцов. Они являются особым случаем более общей структуры объектов – массивов **(arrays)**. Массивы могут быть многомерными. Например, трёхмерный массив состоит из строк, столбцов и слоёв.

Мощь R в значительной мере происходит от разнообразных операций, которые могут выполнятся с матрицами.

Числовую матрицу можно создать из числового вектора с помощью функции **matrix()**

matrix(x, nrow, ncol, byrow, dimnames)

Для задания матрицы необходим массив данных **x**, нужно указать число строк **nrow = m** и/или число столбцов **ncol=n** (по умолчанию, число строк равняется числу столбцов и равно 1); определить как элементы вектора x заполняют матрицу - по строкам (**byrow=T)** или по столбцам (**byrow=F**) (по умолчанию матрица заполняется по столбцам). В результате элементы из вектора будут записаны в матрицу указанных размеров. Аргумент **dimnames** - список из двух компонент, первая из которых задаёт названия строк, а вторая - названия столбцов (по умолчанию имена строк и столбцов не задаются).

> matrix(1:6, nrow = 2, ncol = 3)
[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
> matrix(1:6, nrow = 2, ncol = 3, byrow=T)
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
> matrix(1:6, nrow = 2, ncol = 3, byrow=T,list(c(1,2),c("A","B","C")))
A B C

1 1 2 3
2 4 5 6

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 **ma** буквами A, B, C и D необходимо выполнить следующее:

```
ma <- matrix(seq(1, 16), nrow = 4, ncol = 4, byrow = TRUE)
rownames(ma) <- c("A", "B", "C", "D")
ma
[,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 **ma** 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:

```
> ma2 <- matrix(seq(1, 12), nrow = 4, ncol = 4, byrow = TRUE)
> ma2
     [,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 видно, для заполнения ячеек последней строки матрицы **ma2** программа снова использовала числа 1, 2, 3, и 4.

Альтернативный способ создания матриц заключается в применении функции **dim()** (от **“dimension”** – размерность). Так, матрицу можно сформировать из одномерного вектора следующим образом:

```
> ma <- 1:16
> dim(ma) <- c(4, 4)
> ma
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

> dim(ma)
[1] 4 4

Обозначать строки, столбцы и элементы матрицы можно при помощи индексов и квадратных скобок. Например, **X[i,]** обозначает **i-ую** строку матрицы **X, X[,j]** – обозначает ее **j-ый** столбец, а **X[i, j**] соответствует элементу этой матрицы, расположенному на пересечении этой строки и этого столбца. В качестве индексов **i** и **j** можно использовать числовые векторы, чтобы обозначить сразу несколько строк или столбцов, как это показано ниже.

> x <- matrix(1:10, nrow=2)
> x
[,1] [,2] [,3] [,4] [,5]
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
> x[2,]
[1] 2 4 6 8 10
> x[,2]
[1] 3 4
> x[1,4]
[1] 7
> x[1, c(4,5)]
[1] 7 9

Сначала создана матрица 2×5, содержащая цифры от 1 до 10. По умолчанию матрица заполнена цифрами по столбцам. Затем выбраны все элементы во второй строке, а далее – все элементы во втором столбце. Потом выбран элемент, который находится в первой строке и в четвертом столбце. Наконец, выбраны элементы первой строки, которые находятся в четвертом и пятом столбцах.

Функции **nrow(), ncol() и dim()** возвращают число строк, число столбцов и размерность матрицы A соответственно.

> A<-matrix(1:12, ncol = 4);A
[,1] [,2] [,3] [,4]
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12
> nrow(A)
[1] 3
> ncol(A)
[1] 4
> dim(A)
[1] 3 4

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

> # Создадим четыре вектора одинаковой длины:

```
> a <- c(1, 2, 3, 4)
> b <- c(5, 6, 7, 8)
> d <- c(9, 10, 11, 12)
> e <- c(13, 14, 15, 16)
> # Объединим этим векторы при помощи функции cbind():
> 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
```

R позволяет сгенерировать единичную матрицу нужного размера при помощи функции **eye()** пакета **matlab**, которой в качестве аргумента передаётся размер матрицы.

Чтобы задать диагональную матрицу достаточно воспользоваться функцией **diag(x,nrow,ncol)**:

```
> diag(1,3,3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Для построения квадратной единичной матрицы нужно задать только число строк **nrow** в матрице (если задать число столбцов **ncol**, то будет выведено сообщение об ошибке).

Если аргумент X функции **diag(X)** есть матрица, то в результате применения функции будет построен вектор из элементов X, расположенных на главной диагонали:

```
> X<-matrix(1:16,nrow=4);X
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> diag(X)
[1]  1  6 11 16
```

Арифметические операции над матрицами осуществляются поэлементно, поэтому, чтобы, к примеру, сложить две матрицы, они должны иметь одинаковые размеры:

```
> A <- matrix(1:9, nrow = 3);A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> B <- matrix(-(1:9), ncol = 3,byrow=T);B
     [,1] [,2] [,3]
[1,]   -1   -2   -3
[2,]   -4   -5   -6
[3,]   -7   -8   -9
> A + B
     [,1] [,2] [,3]
[1,]    0    2    4
[2,]   -2    0    2
[3,]   -4   -2    0
```

Впрочем, можно осуществлять смешанные операции, когда один из операндов - матрица, а другой - вектор. В этом случае матрица рассматривается как вектор, составленный из ее элементов, записанных по столбцам, и действуют те же правила, что и для арифметических операций над векторами:

```
> A+3
     [,1] [,2] [,3]
[1,]    4    7   10
[2,]    5    8   11
[3,]    6    9   12
> B+3
     [,1] [,2] [,3]
[1,]    2    1    0
[2,]   -1   -2   -3
[3,]   -4   -5   -6
> (1:3)*A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    4   10   16
[3,]    9   18   27
> A*(1:3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    4   10   16
[3,]    9   18   27
> (1:9)+A
     [,1] [,2] [,3]
```

```
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
> B^2
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
[3,]   49   64   81
```

Функция **outer(x, y, “операция”)** применяет заданную операцию к каждой паре элементов векторов **x** и **y**. Получим матрицу, составленную из результатов выполнения этой операции. Число строк матрицы - длина вектора **x**, а число столбцов - длина вектора **y**:

```
> x <-1:5; x
[1] 1 2 3 4 5
> y <-2:3;y
[1] 2 3
> outer(x, y, "*")
     [,1] [,2]
[1,]    2    3
[2,]    4    6
[3,]    6    9
[4,]    8   12
[5,]   10   15
```

Транспонирование матрицы осуществляет функция **t(A)**, а матричное произведение - операция **%*%**:

```
> t(A)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> A%*%B
     [,1] [,2] [,3]
[1,]  -66  -78  -90
[2,]  -78  -93 -108
[3,]  -90 -108 -126
```

Для решения системы линейных уравнений **Ax = b** с квадратной невырожденной матрицей **A** есть функция **solve(A, b)**:

```
> A <- matrix(c(3,4,4,4), nrow = 2);A
     [,1] [,2]
[1,]    3    4
[2,]    4    4
> b <- c(1,0)
> solve(A,b)
[1] -1  1
```

Системы линейных уравнений, чьи матрицы коэффициентов имеют верхний треугольный или нижний треугольный вид (т.е, либо все элементы под главной диагональю равны нулю, либо над главной диагональю) можно решать с помощью функций **backsolve (A, b)** и **forwardsolve(B, b)**, где **A** и **B** - верхняя треугольная и нижняя треугольная матрицы, **b** - вектор свободных коэффициентов:

```
> A = matrix(c(3,0,4,4), nrow = 2);A
     [,1] [,2]
[1,]    3    4
[2,]    0    4
> b=c(1,1)
> backsolve(A,b)
[1] 0.00 0.25
> B= matrix(c(3,4,0,4), nrow = 2);B
     [,1] [,2]
[1,]    3    0
[2,]    4    4
> forwardsolve(B,b)
[1]  0.33333333 -0.08333333
```

Функция **det(A)** находит определитель матрицы, а **solve(A)** - обратную матрицу:

```
> det(A)
[1] 12
> solve(A)
          [,1]       [,2]
[1,] 0.3333333 -0.3333333
[2,] 0.0000000  0.2500000
```

Кроме функции **solve()** для нахождения обратной матрицы можно использовать и функцию **ginv()** (но для этого сначала надо подключить пакет MASS):

```
> X=matrix(c(1,2,4,2,1,1,3,1,2),nrow=3);X
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    1    1
[3,]    4    1    2
> solve(X)
     [,1] [,2] [,3]
[1,] -0.2  0.2  0.2
[2,]  0.0  2.0 -1.0
[3,]  0.4 -1.4  0.6
> library(MASS)
> ginv(X)
             [,1] [,2] [,3]
[1,] -2.000000e-01  0.2  0.2
```

[2,] -5.828671e-16 2.0 -1.0
[3,] 4.000000e-01 -1.4 0.6

Рассмотрим ещё ряд функций полезных при работе с матрицами. Это:

- **colSums(X, na.rm)** - сумма элементов по столбцам;
- **rowSums(X, na.rm)** - сумма элементов по строкам;
- **colMeans(X, na.rm)** - средние значения по столбцам;
- **rowMeans(X, na.rm)** - средние значения по строкам.

Аргументы функций: **X** - исходный числовой массив (матрица), **na.rm** - логический аргумент, нужно ли убирать из рассмотрения **NA** (по умолчанию **na.rm=FALSE**):

```
> A<-matrix(1:15,nrow=3);A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
> colSums(A)
[1]  6 15 24 33 42
> rowSums(A)
[1] 35 40 45
> colMeans(A)
[1]  2  5  8 11 14
> rowMeans(A)
[1] 7 8 9
```

Одна из самых известных и популярных возможностей R - семейство функций ***apply**.

Команда **apply()** используется тогда, когда нужно применить какую ни будь функцию к строке или столбцу матрицы. Полная форма записи:

apply(X, указатель, функция, ...)

Где:

- **X** - имя матрицы;
- указатель - указывается, к чему применяется функция: **1** - к строкам, **2** - к столбцам, **c(1,2)** - к строкам и столбцам одновременно (если функция применяется ко всем элементам матрицы и результат - матрица, то определяется порядок вывода элементов);
- функция - имя применяемой функции, если нужно применить простые операции вида +, - и т.д., то их необходимо задать в кавычках.

Покажем на примере:

```
> X<-matrix(1:25,nrow=5);X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
```

[5,] 5 10 15 20 25
> #Найдем сумму элементов по строкам
> apply(X,1,sum)
[1] 55 60 65 70 75
> # и по столбцам
> apply(X,2,sum)
[1] 15 40 65 90 115
> #В обоих случаях получили вектора, чьи длины соответствуют числу столбцов и строк соответственно.
> #Найдём корень квадратный по всем элементам матрицы.
> apply(X,1,sqrt)
[,1] [,2] [,3] [,4] [,5]
[1,] 1.000000 1.414214 1.732051 2.000000 2.236068
[2,] 2.449490 2.645751 2.828427 3.000000 3.162278
[3,] 3.316625 3.464102 3.605551 3.741657 3.872983
[4,] 4.000000 4.123106 4.242641 4.358899 4.472136
[5,] 4.582576 4.690416 4.795832 4.898979 5.000000

Собственные вектора и собственные числа матрицы находятся при помощи **eigen(X, symmetric, only.values = FALSE)** где **X** - исходная матрица. **symmetric** - логический аргумент, если его значение есть **TRUE**, то предполагается, что матрица **X** - симметричная (Эрмитова для комплексных чисел), и берутся элементы. лежащие только на главной диагонали и под нею. Если аргумент **symmetric** не задан, то матрица будет проверена на симметричность. Логический аргумент **only.values** определяет, нужно ли выводить только собственные числа или ещё и собственные вектора.

> X
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 2 1 1
[3,] 4 1 2
> eigen(X)
eigen() decomposition
$values
[1] 5.892488 -2.266818 0.374330

$vectors
[,1] [,2] [,3]
[1,] 0.5917695 0.7343437 -0.01899586
[2,] 0.3865001 -0.2573049 -0.83006716
[3,] 0.7074083 -0.6281191 0.55733982

Возвращает вектор собственных значений, расположенных в порядке убывания их модулей (собственные значения могут быть и комплексными) и матрицу, чьи столбцы есть собственные векторы исходной матрицы.

Фильтрация применяется и к матрицам. При этом необходима осторожность, что видно на следующем примере:

```
> X<-matrix(c(1,2,3,2,3,4),nrow =3);X
 [,1] [,2]
[1,]    1    2
[2,]    2    3
[3,]    3    4
> X[X[,2]>=3,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4
```

Анализируем код:

```
> j<-X[,2]>=3;j
[1] FALSE  TRUE  TRUE
```

Здесь рассмотрен вектор **X[,2]** – второй столбец **X** и определено, какой из элементов не меньше **3**. Результат, присваиваемый **j**, представляет собой логический вектор.

Теперь используется **j** в **X**:

```
> X[j,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4
```

Здесь было вычислено **X[j,]**, то есть определены строки **X**, соответствующие истинным элементам **j**.

Другой пример:

```
> A<-matrix(1:6,ncol = 2);A
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> A[A[,1] > 1 & A[,2]>5]
[1] 3 6
```

Здесь используется тот же принцип, но с более сложным набором условий извлечения строк. Сначала выражение **A[,1] > 1** сравнивает каждый элемент первого столбца с 1 возвращает **(FALSE,TRUE, TRUE)**, выражение **A[,2]>5** сравнивает каждый элемент второго столбца A и возвращает **(FALSE,FALSE, TRUE)**. Затем оба результата объединяются логической операцией AND, которая дает результат **(FALSE,FALSE, TRUE)** и в результате получаем третью строку исходной матрицы.

Матрицы - это частный случай многомерных массивов. Матрицы имеют две размерности. В общем случае массивы могут иметь больше размерностей. Работа с многомерными массивами в R во многом аналогична работе с

матрицами. Основной способ их создания - функция **array(X, вектор размерностей)**. Указываются элементы массива и все его размерности.

```
#Создадим массив размерности 3*5*4
> A<-array(1:60, c(3,5,4))
> #При вызове массив выводится послойно.
> A
, , 1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
, , 2
     [,1] [,2] [,3] [,4] [,5]
[1,]   16   19   22   25   28
[2,]   17   20   23   26   29
[3,]   18   21   24   27   30

, , 3
     [,1] [,2] [,3] [,4] [,5]
[1,]   31   34   37   40   43
[2,]   32   35   38   41   44
[3,]   33   36   39   42   45
, , 4
     [,1] [,2] [,3] [,4] [,5]
[1,]   46   49   52   55   58
[2,]   47   50   53   56   59
[3,]   48   51   54   57   60
```

Можно задать имена размерностям:

```
> dim1 =c("A", "B", "C")
> dim2 =c("X", "X2", "X3", "X4", "X5")
> dim3 = c("Зима","Весна","Лето","Осень")
> dimnames(A) = list(dim1, dim2, dim3)
> A
, , Зима
  X X2 X3 X4 X5
A 1  4  7 10 13
B 2  5  8 11 14
C 3  6  9 12 15
, , Весна
   X X2 X3 X4 X5
A 16 19 22 25 28
B 17 20 23 26 29
C 18 21 24 27 30
, , Лето
```

```
   X X2 X3 X4 X5
A 31 34 37 40 43
B 32 35 38 41 44
C 33 36 39 42 45
, , Осень
   X X2 X3 X4 X5
A 46 49 52 55 58
B 47 50 53 56 59
C 48 51 54 57 60
```

и после этого обращаться к элементу по имени его строки, столбца, слоя:

```
> A[,,"Осень"]
   X X2 X3 X4 X5
A 46 49 52 55 58
B 47 50 53 56 59
C 48 51 54 57 60
```

3.3. Списки

В отличие от вектора или матрицы, которые могут содержать данные только одного типа, в список **(list)** 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:
> mylist <- list(Text=vector1, Number=vector2, Logic=vector3)
> # Просмотрим содержимое созданного списка:
> mylist
$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 компонентов **Text**, **Number** и **Logic** из созданного нами списка **mylist** необходимо последовательно ввести следующие команды:

> mylist$Text
[1] "A" "B" "C"
> mylist$Number
[1] 1.0 1.5 2.0 2.5 3.0
> mylist$Logic
[1] FALSE TRUE

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:

> mylist$Text[2]
[1] "B"
>
> mylist$Number[3:5]
[1] 2.0 2.5 3.0
>
> mylist$Logic[1]
[1] FALSE

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

> mylist[[1]]
[1] "A" "B" "C"
>
> mylist[[2]]
[1] 1.0 1.5 2.0 2.5 3.0
>
> mylist[[3]]
[1] FALSE TRUE

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

> mylist[[1]][2]
[1] "B"
> mylist[[2]][3:5]
[1] 2.0 2.5 3.0
> mylist [[3]][1]
[1] FALSE

Созданный список **mylist** содержал всего лишь три небольших вектора, и известно, 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 (*mylist*)
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 **mylist** 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 **(logic)** типам векторов соответственно. Кроме того, команда **str()** выводит на экран первые несколько элементов каждого вектора.

Новые элементы могут добавляться после создания списка:

> x<-list(a="rub",d=1/2)
x
$a
[1] "rub"
$d
[1] 0.5
> x$f<-"prima"#добавляется компонент f
> x
$a
[1] "rub"
$d
[1] 0.5
$f
[1] "prima"

Добавление компонентов также может осуществляться индексированием вектора:

```
> x[[4]]<-'yes'
> x
$a
[1] "rub"
$d
[1] 0.5
$f
[1] "prima"
[[4]]
[1] "yes"
```

Чтобы удалить компонент списка, присвойте ему **NULL**:

```
> x$a<-NULL
> x
$d
[1] 0.5
$f
[1] "prima"
[[3]]
[1] "yes"
```

Учтите, что после удаления **x$a** индексы элементов, следующих за ним, сдвигаются вверх на **1**. Например, бывший элемент **z[[4]]** превращается в **z[[3]]**.

Также возможна конкатенация списков:

```
> y<-c(list("Петя", "студент", "ННГУ"),list("оценка", 5))
> y
[[1]]
[1] "Петя"
[[2]]
[1] "студент"
[[3]]
[1] "ННГУ"
[[4]]
[1] "оценка"
[[5]]
[1] 5
```

Если компонентам списка назначены теги (такие, как **Петя, студент**, **ННГУ**, их можно получить вызовом **names()**:

```
> y<-list(имя="Петя", занятие="студент", вуз="ННГУ")
> y
$имя
[1] "Петя"
```

$занятие
[1] "студент"
$вуз
[1] "ННГУ"
> names(y)
[1] "имя" "занятие" "вуз"

Для получения значений используется функция **unlist()**:

> un<-unlist(y)
> un
имя занятие вуз
"Петя" "студент" "ННГУ"
> class(un)
[1] "character"

Возвращаемое значение **unlist()** представляет собой вектор — в данном случае содержащий символьные строки. При этом имена элементов в векторе берутся из компонентов исходного списка.

Функция **lapply()** (сокращение от «list apply») работает так же, как и матричная функция **apply()**: она вызывает заданную функцию для каждого компонента списка (или вектора, преобразованного к списку) и возвращает другой список. Пример:

> lapply(list(1:3,25:29),median)
[[1]]
[1] 2
[[2]]
[1] 27

R применяет **median()** к **1:3** и **25:29**, возвращая список с элементами **2** и **27**. В некоторых случаях список, возвращаемый **lapply()**, может быть упрощен до вектора или матрицы. Именно это делает функция **sapply()** (от «simplified [l]apply», то есть «упрощенная [l]apply»):

> sapply(list(1:3,25:29),median)
[1] 2 27

Возвращаемое значение **unlist()** представляет собой вектор — в данном случае содержащий символьные строки. При этом имена элементов в векторе берутся из компонентов исходного списка.

Функция **lapply()** (сокращение от «list apply») работает так же, как и матричная функция **apply()**: она вызывает заданную функцию для каждого компонента списка (или вектора, преобразованного к списку) и возвращает другой список. Пример:

> lapply(list(1:3,25:29),median)
[[1]]
[1] 2
[[2]]
[1] 27

R применяет **median()** к **1:3** и **25:29**, возвращая список с элементами **2** и **27**. В некоторых случаях список, возвращаемый **lapply()**, может быть упрощен до вектора или матрицы. Именно это делает функция **sapply()** (от «simplified [l]apply», то есть «упрощенная [l]apply»):

```
> sapply(list(1:3,25:29),median)
[1]  2 27
```

3.4. Кадры данных

На интуитивном уровне **кадр данных** ***(data frame)*** похож на матрицу: он тоже имеет двумерную структуру из строк и столбцов. С другой стороны, кадр данных отличается от матриц тем, что все столбцы могут иметь разные режимы. Скажем, один столбец может содержать числа, а другой — символьные строки. Если списки являются разнородными аналогами векторов в одном измерении, кадры данных являются разнородными аналогами матриц для двумерных данных.

На техническом уровне кадр данных представляет собой список, компоненты которого являются векторами равной длины.

mydata <- data.frame(col1, col2, col3,…),

где – **col1**, **col2**, **col3**,… это векторы любого типа (текстового, числового или логического), которые станут столбцами таблицы. Названия каждого столбца можно прочитать при помощи функции **names()**. Проиллюстрируем сказанное при помощи следующего программного кода:

```
> student<-c(1,2,3,4,5)
> age<-c(18,18,19,20,20)
> kurs<-c(1,1,2,3,3)
> state<-c("Artist","Working","Working","Artist","Artist")
> studentdata<-data.frame(student,kurs,state,age)
> studentdata
  student kurs   state age
1       1    1  Artist  18
2       2    1 Working  18
3       3    2 Working  19
4       4    3  Artist  20
5       5    3  Artist  20
> studentdata[2:3]
  kurs   state
1    1  Artist
2    1 Working
3    2 Working
4    3  Artist
5    3  Artist
```

> studentdata["state"]
state
1 Artist
2 Working
3 Working
4 Artist
5 Artist
> state
[1] "Artist" "Working" "Working" "Artist" "Artist"

Кадр данных создан, теперь можно немного поэкспериментировать. Так как **studentdata** является списком, к нему можно обращаться как по индексам компонентов, так и по именам компонентов:

> studentdata[[1]]
[1] 1 2 3 4 5
> studentdata$student
[1] 1 2 3 4 5

Но с ним также можно работать по тем же правилам, что и с матрицами. Например, можно просмотреть столбец 1:

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

Это сходство с матрицами также проявляется при разборе **studentdata** с использованием **str()**:

> str(studentdata)
'data.frame': 5 obs. of 4 variables:
$ student: num 1 2 3 4 5
$ kurs : num 1 1 2 3 3
$ state : Factor w/ 2 levels "Artist","Working": 1 2 2 1 1
$ age : num 18 18 19 20 20

Рассмотрим три способа обращения к первому столбцу приведенного выше кадра данных: **studentdata [[1]]**, **studentdata [,1]** и **studentdata$kurs**. Третий способ обычно считается более понятным и, что еще важнее, — более безопасным, чем первые два. Он лучше идентифицирует столбец и снижает вероятность случайного обращения к другому столбцу. Но при написании обобщенного кода — допустим, написании пакета R — необходима матричная запись **studentdata [,1]**, что особенно удобно при извлечении подкадров данных.

Рассмотрим таблицу со статистическими данными по продажам подержанных автомобилей.

price	**age**	**color**
10000	3	white
12000	2	red
9300	5	white
15000	1	red
9700	3	white

11000 2 black

Переменные **price** (цена продажи) и **age** (возраст) являются количественными, а переменная **color** является качественной (факторной). Создадим карту данных

```
> cardata <- data.frame(
+   price = c(10000, 12000, 9300, 15000, 9700, 11000),
+   age = c(3, 2, 5, 1, 3, 2),
+   color = factor(c("white", "red", "white",
+                    "red", "white", "black")) )
> cardata
  price age color
1 10000   3 white
2 12000   2   red
3  9300   5 white
4 15000   1   red
5  9700   3 white
6 11000   2 black
```

Как упоминалось ранее, кадр данных может рассматриваться в контексте строк и столбцов. В частности, из него можно извлекать подкадры данных по строкам и столбцам. Пример:

```
> cardata[2:5,]
  price age color
2 12000   2   red
3  9300   5 white
4 15000   1   red
5  9700   3 white
> cardata[2:5,2]
[1] 2 5 1 3
> class(cardata[2:5,2])
[1] "numeric"
> class(cardata[2:5,3])
[1] "factor"
```

Также можно выполнить фильтрацию данных. В следующем примере извлекается подкадр данных всех машин, у которых возраст не больше трех лет:

```
> cardata[cardata$age <=3,]
  price age color
1 10000   3 white
2 12000   2   red
4 15000   1   red
5  9700   3 white
```

6 11000   2 black

В нашем примере вместо команды:

> cardata[cardata$age <=3,]

можно выполнить команду **subset()**, которая игнорирует значения **NA**:

```
> subset(cardata,age<=3)
  price age color
1 10000   3 white
2 12000   2   red
4 15000   1   red
5  9700   3 white
6 11000   2 black
```

Матричные функции **rbind()** и **cbind()** будут работать и с кадрами данных, при условии совместимости размеров. Например, при помощи **cbind()** можно добавить новый столбец с такой же длиной, как у существующих столбцов. При добавлении строки вызовом **rbind()** строка обычно добавляется в форме другого кадра данных или списка.

```
> rbind(studentdata,list(6,3,"Working",21))
  student kurs   state age
1       1    1  Artist  18
2       2    1 Working  18
3       3    2 Working  19
4       4    3  Artist  20
5       5    3  Artist  20
6       6    3 Working  21
```

Также можно создавать новые столбцы на основе уже существующих. Например, можно добавить качественную переменную **gender**, которая характеризует пол студента

```
> studdata<-rbind(studentdata,list(6,3,"Working",21))
> studdata
  student kurs   state age
1       1    1  Artist  18
2       2    1 Working  18
3       3    2 Working  19
4       4    3  Artist  20
5       5    3  Artist  20
6       6    3 Working  21
> gender<-c("M","F","M","F","M","F")
> gender
[1] "M" "F" "M" "F" "M" "F"
> studdata<-cbind(studdata,gender)
> studdata
  student kurs   state age gender
```

```
1      1    1  Artist  18      M
2      2    1 Working  18      F
3      3    2 Working  19      M
4      4    3  Artist  20      F
5      5    3  Artist  20      M
6      6    3 Working  21      F
```

В мире реляционных баз данных одной из важнейших операций является операция соединения (**join**), при которой происходит слияние двух таблиц по значениям общей переменной. В **R** два кадра данных объединяются аналогичным образом при помощи функции **merge()**.

Простейшая форма этой функции выглядит так:

merge(x,y)

Вызов объединяет кадры данных **x** и **y**. Предполагается, что два кадра данных содержат один или несколько столбцов с одинаковыми именами. Пример:

```
> stud<-c("Петр","Семен","Роман","Иван")
> states<-c("Work","Art","Work","Work")
> cd1<-data.frame(stud,states)
> cd1
  stud states
1  Петр   Work
2 Семен    Art
3 Роман   Work
4  Иван   Work

> ages<-c(18,19,19)
> stud<-c("Петр","Сергей","Роман")
> cd2<-data.frame(ages,stud)
> cd2
  ages   stud
1   18   Петр
2   19 Сергей
3   19  Роман

> cd
  stud states ages
1  Петр   Work   18
2 Роман   Work   19
```

Здесь два кадра данных содержат общую переменную **stud**. R находит строки, у которых эта переменная имеет одинаковое значение в двух кадрах

данных **(Петр и Роман)**. Затем создается кадр данных с соответствующими строками и столбцами из обоих кадров данных **(stad, states и ages)**.

Следует помнить, что кадры данных являются особой разновидностью списков; компонентами списков являются столбцы кадра данных. Следовательно, если вызвать **lapply()** для кадра данных с функцией **f()**, функция **f()** будет вызвана для каждого столбца кадра, а возвращаемые значения будут помещены в список.

Например, функция **lapply()** может использоваться следующим образом:

```
> ages<-c(18,19,18)
> stud<-c("Петр","Сергей","Роман")
> cd2<-data.frame(ages,stud)
> cd2
  ages   stud
1   18   Петр
2   19 Сергей
3   18  Роман
> cdm<-lapply(cd2,sort)
> cdm
$ages
[1] 18 18 19
$stud
[1] Петр   Роман  Сергей
Levels: Петр Роман Сергей
```

Итак, **cdm** представляет собой список, состоящий из двух векторов: отсортированных версий **stud** и **ages**.

Следует помнить, что **cdm** — всего лишь список, а не кадр данных. Его можно преобразовать в кадр данных:

```
> as.data.frame(cdm)
  ages   stud
1   18   Петр
2   18  Роман
3   19 Сергей
```

Для сортировки кадров данных в R лучше всего использовать функцию **order()**. По умолчанию данные сортируются в порядке возрастания. Если перед интересующей вас переменной поставить знак минус, значения сортируются в порядке убывания.

3.5. Факторы и таблицы

Факторы лежат в основе многих мощных операций **R**, включая многие операции, выполняемые с табличными данными. Основания для применения факторов исходят от концепции **номинальных** (или **категорийных**) переменных в статистике. Такие значения по своей природе являются

нечисловыми; они соответствуют категориям («мужчина», «женщина», «ребенок» и т. д.), хотя и могут кодироваться в числовом виде.

Фактор **(factor)** R можно рассматривать просто как вектор с дополнительной информацией. Эта дополнительная информация состоит из реестра различных значений в этом векторе, которые называются уровнями **(levels)**. Пример:

```
> x <- c(5,12,13,12)
> xf <- factor(x)
> xf
[1] 5  12 13 12
Levels: 5 12 13
```

Уровнями здесь являются различные значения в **xf** — 5, 12 и 13.

Структура объекта **xf**:

```
> str(xf)
 Factor w/ 3 levels "5","12","13": 1 2 3 2
> unclass(xf)
[1] 1 2 3 2
attr(,"levels")
[1] "5"  "12" "13"
```

В **xf** хранятся не значения (5, 12, 13, 12), а значения (1, 2, 3, 2). Таким образом, данные состоят сначала из значения уровня 1, затем из значений уровней 2 и 3 и, наконец, еще одного значения уровня 2. Таким образом, данные были перекодированы по уровню. Разумеется, сами уровни тоже сохранены, хотя и в виде символов ("5" вместо 5).

Длина фактора определяется в контексте длины данных, а не в контексте количества уровней или еще в каком-нибудь виде:

```
> length(xf)
[1] 4
```

Также можно резервировать будущие новые уровни, как в следующем примере:

```
> x <- c(5,12,13,12)
> xff <- factor(x,levels=c(5,12,13,88))
> xff
[1] 5  12 13 12
Levels: 5 12 13 88
> xff[3] <- 88
> xff
[1] 5  12 88 12
Levels: 5 12 13 88
```

В исходном виде **xff** не содержит значение **88**, но при его определении допускается такая будущая возможность. Позднее это значение действительно было добавлено. Попытка подкинуть «незаконный» уровень по тому же принципу завершается неудачей. Вот как это выглядит:

```
> xff[2] <- 888
```

Warning message:
In `[<-.factor`(`*tmp*`, 2, value = 888) :
invalid factor level, NA generated

С факторами используется еще один представитель семейства функций **apply** — **tapply**. Рассмотрим эту функцию, а также функцию, часто используемую с факторами: **split()**.

Допустим, имеется вектор **x** с возрастами избирателей и фактор **f** с некоторой нечисловой характеристикой избирателей — например, партийная принадлежность (единорос, коммунист, либерал). Требуется найти средний возраст для каждой из партийных групп в **x**.

В типичном варианте использования вызов **tapply(x,f,g)** имеет вектор **x**, фактор или список факторов **f** и функцию **g**. Функцией **g()** в нашем маленьком примере будет встроенная функция R **mean()**. Если надо сгруппировать данные по партийной принадлежности и другому показателю (скажем, полу), фактор **f** должен включать два других фактора (партийная принадлежность и пол).

Каждый фактор в **f** должен иметь такую же длину, как и **x**. В контексте приведенного выше примера с избирателями это логично: количество вариантов партийной принадлежности должно совпадать с количеством возрастов. Если компонент **f** является вектором, он будет преобразован в фактор применением к нему функции **as.factor()**.

Функция **tapply()** (временно) разбивает **x** на группы, соответствующие уровням факторов (или комбинациям уровней в случае нескольких факторов), а затем применяет **g()** к полученным подвекторам **x**. Небольшой пример:

```
> ages <- c(25,26,55,37,21,42)
> affils <- c("ЕР","К","К","ЕР","Л","Л")
> tapply(ages,affils,mean)
  ЕР   К   Л
31.0 40.5 31.5
```

Посмотрим, что здесь произошло. Функция **tapply()** интерпретирует вектор **("ЕР","К","К","ЕР","Л","Л")** как фактор с уровнями **"ЕР"**, **"К"** и **"Л"**. В данных указано, что **"ЕР"** встречается в позициях с индексами **1, 4** ; **"К"** встречается в позициях с индексами **2** и **3**; и **"Л"** встречается в позициях с индексами **5** и **6**. Для удобства будем называть три индексных вектора **(1,4)**, **(2,3)** и **(5,5)** **x**, **y** и **z** соответственно. Затем **tapply()** вычисляет **mean(u[x])**, **mean(u[y])** и **mean(u[z])** и возвращает эти значения в виде вектора из трех элементов. А имена элементов этого вектора **"ЕР"**, **"К"** и **"Л"** отражают уровни фактора, используемые функцией **tapply()**.

А если факторов два и более? Каждый фактор создает набор групп, как в предыдущем примере, и объединяет группы операцией **AND**. Допустим, имеется экономический набор данных с переменными для пола, возраста и дохода. В вызове **tapply(x,f,g)** **x** может быть доходом, а **f** — парой факторов: для пола и для признака возраста (старше/младше 25 лет). Требуется определить средний доход с разбивкой по полу и возрасту. Если задать для **g()** функцию **mean()**, **tapply()** вернет средний доход в каждой из четырех подгрупп:

- мужчины младше 25 лет;
- женщины младше 25 лет;
- мужчины старше 25 лет;
- женщины старше 25 лет.

Небольшой учебный пример для этой конфигурации:

```
> gender <- c("M","M","F","M","F","F")
> age <- c(47,59,21,32,33,24)
> income <-c(55000,88000,32450,76500,123000,45650)
> d <- data.frame(gender,age,income)
> d
  gender age income
1      M  47  55000
2      M  59  88000
3      F  21  32450
4      M  32  76500
5      F  33 123000
6      F  24  45650
> d$over25 <- ifelse(d$age > 25,1,0)
> d
  gender age income over25
1      M  47  55000      1
2      M  59  88000      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0
> tapply(d$income,list(d$gender,d$over25),mean)
     0        1
F 39050 123000.00
M    NA  73166.67
```

В отличие от функции **tapply()**, которая разбивает вектор на группы, а затем применяет заданную функцию к каждой группе, **split()** останавливается на первой стадии и ограничивается формированием групп.

Базовая форма без каких-либо украшений и удобств имеет вид **split(x,f)**. Здесь **x** и **f** играют те же роли, что и при вызове **tapply(x,f,g)**; то есть **x** является вектором или кадром данных, а **f** — фактор или список факторов. Действием является разбиение **x** на группы, которые возвращаются в виде списка. (Заметим, что **x** может быть кадром данных с **split()**, но не с **tapply()**).

```
> d
  gender age income over25
1      M  47  55000      1
2      M  59  88000      1
3      F  21  32450      0
4      M  32  76500      1
```

```
5      F  33 123000      1
6      F  24  45650      0
> split(d$income,list(d$gender,d$over25))
$F.0
[1] 32450 45650

$M.0
numeric(0)

$F.1
[1] 123000

$M.1
[1] 55000 88000 76500
```

Вывод **split()** представляет собой список. Таким образом, последний вектор, например, назывался "M.1"; это имя является результатом объединения "M" из первого фактора и 1 из второго.

Знакомство с таблицами R начнем со следующего примера:

```
> u <- c(22,8,33,6,8,29,-2)
> fl <- list(c(5,12,13,12,13,5,13),c("a","bc","a","a","bc","a","a"))
> tapply(u,fl,length)
   a bc
5  2 NA
12 1  1
13 2  1
```

Здесь **tapply()** снова временно разбивает **u** на подвекторы, как было показано ранее, а затем применяет функцию **length()** к каждому подвектору. (Заметим, что происходящее не зависит от содержимого **u** — нас интересуют исключительно факторы.) Длины подвекторов определяют количества вхождений каждой из 3 × 2 = 6 комбинаций двух факторов. Например, **5** дважды встречается с **"a"** и вовсе не встречается с **"bc"**; отсюда элементы **2** и **NA** в первой строке вывода. В статистике такая структура называется факторной таблицей.

В этом примере есть одна проблема: значение **NA**. В действительности оно должно быть равно 0, это означает, что ни в одном случае первый фактор не имеет уровень **5** при втором факторе с уровнем **"bc"**. Функция **table()** правильно создает факторные таблицы.

```
> table(fl)
    fl.2
fl.1 a bc
  5  2  0
  12 1  1
  13 2  1
```

Первым аргументом вызова **table()** является либо фактор, либо список факторов. В данном случае используются два фактора — **(5,12,13,12,13,5,13)** и **("a","bc","a","a","bc","a","a")**, при этом объект, который может интерпретироваться как фактор, считается таковым.

Обычно в аргументе данных **table()** передается кадр данных. Например, предположим, что у нас есть фрейм, который состоит из данных о воображаемых жуках, состоящий из пяти столбов (признаков): пол жука (POL), цвет жука (CVET), вес жука (VES), рост жука (ROST) и приведенный к росту вес жука (VES.R):

```
> data
   POL CVET   VES  ROST     VES.R
1    1    1 10.68  9.43 1.1325557
2    2    1 10.02 10.66 0.9399625
3    1    2 10.18 10.41 0.9779059
4    2    1  8.01  9.00 0.8900000
5    1    3 10.23  8.98 1.1391982
6    2    3  9.70  9.71 0.9989701
7    2    2  9.73  9.09 1.0704070
8    1    3 11.22  9.23 1.2156013
9    2    1  9.19  8.97 1.0245262
10   2    2 11.45 10.34 1.1073501
```

Для категориальных признаков имеет смысл определить, сколько раз встречается в выборке каждое значение признака:

```
> table(data$POL)
1 2
4 6
```

Определяем, что в выборке содержится 4 самки жуков и 6 самцов.

Можем вычислить характеристики количественных признаков для самцов и самок, например среднее арифметическое:

```
> tapply(data$VES,data$POL,mean)
       1        2
10.577500  9.683333
> tapply(data$ROST,data$POL,mean)
       1        2
9.512500 9.628333
```

Посмотрим, сколько жуков разного цвета среди самцов и самок:

```
> table(data$CVET,data$POL)

  1 2
1 1 3
2 1 2
3 2 1
```

Строки – разные цвета, столбцы – самцы и самки.

В этом примере таблицы, вычислим средние значения веса жуков отдельно для всех комбинаций цвета и пола (для красных самцов, красных самок, зеленых самцов, зеленых самок …):

```
> tapply(data$VES,list(data$POL,data$CVET),mean)
        1     2     3
1 10.680000 10.18 10.725
2  9.073333 10.59  9.700
```

4. Некоторые программные конструкции

В этой главе рассматриваются некоторые основные конструкции R как языка программирования. R позволяет достичь многого, если пользователь в конечном итоге приобретает некоторый уровень программирования, становясь опытным пользователем.

4.1. Вывод сообщения на экран и ввод данных с клавиатуры

В R для вывода сообщения на экран есть две команды: **print() и cat().**

```
> print("There are two commands") # с текстом в кавычках
[1] "There are two commands"
> cat("There are two commands") # с текстом без кавычек
There are two commands
```

При желании, если используем **print()**, кавычки в выдаче можно убрать, скорректировав параметр **quote**:

```
> print("There are two commands", quote=FALSE)
[1] There are two commands
```

Команда **cat()** позволяет не только выводить на экран уже «готовые» элементы, но и склеивать выдачу из нескольких:

```
> w1 <- "there"
> w2 <- "are"
> w3 <- "two"
> w4 <- "commands"
> cat(c(w1, w2, w3, w4), sep=" ") # разделитель - пробел
there are two commands
> cat(c(w1, w2, w3, w4), sep="_") # разделитель - знак подчеркивания
there_are_two_commands
```

Можно выводить каждый элемент с новой строки или с отступом (табуляция):

```
> cat(' 1\n','2\n','3\n') # \n - переход на новую строку
 1
 2
 3
> cat(' 1\t','2\t','3\t') # \t - табуляция
 1       2       3
```

Конкатенация (склевание) строк:

> paste('group', 341) # по умолчанию разделяются пробелом

[1] "group 341"

> paste('01', '10', '2015', sep = '-') # можем задавать разделитель между строками (sep)

[1] "01-10-2015"

Для ввода данных с клавиатуры в R есть функция **readline()**. Для примера попросим пользователя ввести свое имя:

> yname <- readline(prompt = "Введите ваше имя: ")

Введите ваше имя: Пётр

> cat("Привет, ", yname, "!", sep = "")

Привет, Пётр!

>

Функция **cat()** только выводит результат на экран монитора, а функция **print()** ещё и сохраняет его.

Функция **readline()** всегда возвращает текст, то есть объект типа **character**, даже если пользователь ввел число:

> k <- readline(prompt = "Введите число: ")

Введите число: 10

> typeof(k)

[1] "character"

Поэтому необходимо выполнить преобразование символов в числа, например с помощью следующей конструкции:

> k <- as.numeric(readline(prompt = "Введите число: "))

Введите число: 12

> typeof(k)

[1] "double"

> class(k)

[1] "numeric"

Для ввода нескольких чисел необходимо после каждого числа поставить пробел и воспользоваться функцией **strsplit()** и разбить получившуюся строку пробелами. В результате выполнения таких действий получится список **(list)**, который надо преобразовать в вектор:

> vec <- readline(prompt = "Введите числа через пробел: ") # 2 33 45

Введите числа через пробел: 2 33 45

> rez <- strsplit(vec, split = " ")

> rez

[[1]]

[1] "2" "33" "45"

> z <- unlist(rez) # читаем список

> z

[1] "2" "33" "45"

> zv <- as.numeric(z) # список сделаем числовым

> zv
[1] 2 33 45

4.2. Разветвление if - else

Блок **if** может реализовать условную логику, протестировав простое условие:

if (condition) {
 # Выполняется, если условие истинно.
} else {
 # Выполняется, если условие ложно.
}

Конструкция ***if*** позволяет выбирать из двух альтернативных путей, проверяя некое условие, такое как x == 0 или y> 1, и затем следуя по одному или другому пути соответственно. Здесь, например, мы выполняем проверку на предмет наличия отрицательных чисел, перед тем как вычислить квадратный корень:

> x <- -5
> if (x >= 0) {
+ print(sqrt(x)) # Выполняется, если x >= 0.
+ } else {
+ print("negative number") # В противном случае делаем это.
+ }
[1] "negative number"

Можно связать последовательность конструкций **if/else**, чтобы принять ряд решений. Предположим, мы хотим, чтобы значение было ограничено 0 (без отрицательных значений) и ограничено 1. Это можно было бы написать следующим образом:

> x <- -0.3
> if (x < 0) {
+ x <- 0
+ } else if (x > 1) {
+ x <- 1
+ }
> print(x)
[1] 0
>

Когда условие x < 0 неоднозначно, что возможно при анализе содержания векторов, R предоставляет вспомогательные функции **all** и **any**, которые разрешают эту ситуацию. В данном случае обрабатывается вектор логических значений и уменьшают вектора до одного-единственного значения:

> x <- c(-2, -1, 0, 1, 2)
> if (all(x < 0)) {
+ print("all are negative")

```
+ }
> if (any(x < 0)) {
+   print("some are negative")
+ }
[1] "some are negative"
>
```

4.3. Циклы for и while

Для итерации обычно используется конструкция цикла **for**. Если **v** – это вектор или список, цикл **for** выбирает каждый элемент **v** один за другим, присваивает элемент **x** и что-то с ним делает:

```
for (x in v) {
# Что-то делаем с x.
}
```

В R циклы мало распространены, но тем не менее иногда полезны. В качестве иллюстрации этот цикл **for** выводит первые пять целых чисел и их квадраты. Он последовательно устанавливает для **x** значения 1, 2, 3, 4, 5:

```
> for (x in 1:5) {
+   cat(x, x^2, "\n")
+ }
1 1
2 4
3 9
4 16
5 25
```

Пользователь также может перебирать индексы вектора или списка, что полезно для обновления данных на месте. Здесь мы инициализируем **v** с вектором 1:5, затем обновляем его элементы, возводя каждый из них в квадрат:

```
> v <- 1:5
> for (i in v) {
+   v[[i]] <- v[[i]] ^ 2
+ }
> print(v)
[1]  1  4  9 16 25
```

Цикл **while** устроен немного по-другому: действие повторяется до тех пор, пока выполняется некоторое условие.

```
> t <- 1 # стартуем со значения 1
>
> while (t < 5){
+   cat(t, "\n") # пока значение меньше 5, выводим его на экран
+   t <- t + 1 # переходим к следующему значению
+ }
1
```

```
2
3
4
```

4.4. Функция swith

При написании программного кода переменная может принимать несколько разных значений и необходимо чтобы программа обрабатывала каждый случай отдельно, в соответствии со значением. В этом случае целесообразно использовать функцию **swith**, которая создает ветку значений, позволяя выбрать способ обработки каждого случая.

Первый аргумент функции – это значение, которое R должен рассмотреть. Остальные аргументы показывают, как обрабатывать каждое возможное значение. Например, в этом вызове функции **switch** мы рассматриваем значение **ho**, а затем возвращаем один из трех возможных результатов:

```
> ho <- 'Moe'
> aire = switch(ho,
+             Moe = "long",
+             Larry = "fuzzy",
+             Curly = "none")
> aire
[1] "long"
> ho <- 'Larry'
> aire = switch(ho,
+           Moe = "long",
+           Larry = "fuzzy",
+           Curly = "none")
> aire
[1] "fuzzy"
```

Очень часто нельзя ожидать, что будут рассмотрены все возможные значения, поэтому функция **switch** позволяет вам определить значение по умолчанию для ситуации, когда ни одна метка не совпадает. Говоря проще, последний по умолчанию без метки. В этом примере мы преобразуем содержимое **s** из **"one"**, **"two"** или **"three"** в соответствующее целое число и возвращаем **NA** для любого другого значения:

```
> s <- 'two'
> num <- switch(s,
+           one = 1,
+           two = 2,
+           three = 3,
+           NA)
> num
```

```
[1] 2
> s <- 'five'
> num <- switch(s,
+             one = 1,
+             two = 2,
+             three = 3,
+             NA)
> num
[1] NA
```

Если метки являются целыми числами, то их необходимо преобразовать в строку символов:

```
> i <- 30
> switch(as.character(i),
+       "10" = "ten",
+       "20" = "twenty",
+       "30" = "thirty",
+       "other")
[1]"thirty"
```

4.5. Конструкция function

Иногда нам нужно, чтобы код решал не одну, а сразу комплекс задач. Эти задачи можно группировать в формате функций. Функции—это очень важные объекты в R. В функцию можно передавать аргументы, а она может возвращать объект. В установленном пакете R есть определенное количество встроенных функций, в том числе: **length(), mean()** и т.д. Каждый раз, когда мы объявляем функцию (или переменную) и вызываем её, она ищется в текущем окружении, а также рекурсивно ищется в родительских окружениях до тех пор, пока значение не будет найдено. У функции есть имя. В теле функции находятся её операторы. Функция может возвращать значение и может принимать ряд аргументов.

Функция создается с использованием ключевого слова **function**, за которым следует список имен параметров, а затем тело функции:

```
name <- function(param1, ..., paramN) {
expr1
.
.
.
exprM
}
```

Вокруг имен параметров ставятся круглые скобки, а фигурные скобки вокруг тела функции, которое представляет собой последовательность из

одного или нескольких выражений. R вычислит каждое выражение по порядку и вернет значение последнего, обозначенное здесь как exprM.

Определения функций – это то, что пользователь устанавливает R: **«Вот как надо выполнить вычисления»**. Например, в R нет встроенной функции для вычисления коэффициента вариации, но можно создать такую функцию, дав ей имя **cvar**:

```
cvar <- function(x) {
  sd(x) / mean(x)
}
```

У этой функции один параметр, **x**, а **sd(x) / mean(x)** – это тело функции.

Когда мы вызываем функцию с аргументом, R устанавливает для параметра **x** это значение, а затем вычисляет тело функции:

```
> cvar(1:10) # Устанавливаем для x значение 1:10 и вычисляем sd(x)/mean(x).
[1]0.5504819
```

Обратите внимание на то, что параметр **x** отличается от любой другой переменной с именем **x**. Например, если в вашей рабочей области есть глобальная переменная **x**, то тот **x** будет отличаться от этого **x** и не будет зависеть от **cvar**. Кроме того, параметр **x** существует только во время выполнения функции **cvar** и после этого исчезает.

У функции может быть более одного аргумента. У следующей функции два аргумента, оба из которых – целые числа. Она реализует алгоритм Евклида для вычисления их наибольшего общего делителя:

```
> gcd <- function(a, b) {
+   if (b == 0) {
+     a # Возвращаем a вызывающей стороне.
+   } else {
+     gcd(b, a %% b) # Рекурсивно вызываем сами себя.
+   }
+ }
> # Каков наибольший одинаковый знаменатель 14 и 21?
> gcd(14, 21)
[1] 7
```

Обычно функция возвращает значение последнего выражения в теле функции. Однако можно вернуть значение раньше, написав **return(expr)**, заставляя функцию остановиться и немедленно вернуть **expr** вызывающей стороне. Рассмотрим следующий пример, кодируя **gcd** несколько иным способом, используя явный возврат:

```
gcd <- function(a, b) {
if (b == 0) {
return(a) # Останавливаемся и возвращаем a.
}
gcd(b, a %% b)
}
```

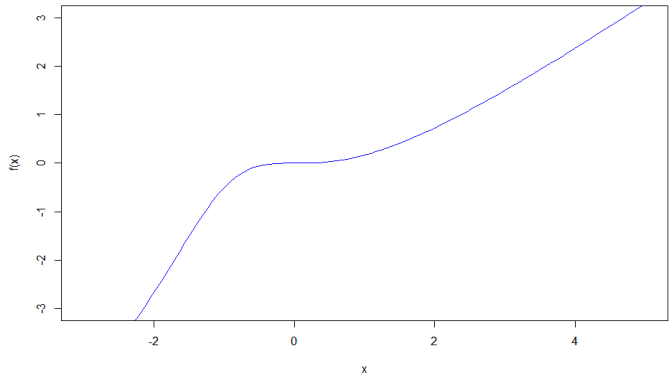
Возвращаемым значением функции может быть любой объект R. Функции в R не умеют возвращать несколько объектов в **return()**. Их необходимо самим объединять либо в вектор, либо в список, либо в какую-то еще структуру:

```
> f <- function(a, b){
+   return(c(a ** b, b * a))
+ }
> f(3,2)
[1] 9 6
```

Возвращаемое значение часто представляет собой список, но им может быть даже другая функция.

При построении графиков сложных аналитических функций с помощью функции **curve()** также возможно использование конструкции **function()**:

```
> f <- function(x){x^3/(x^2+2*x+3)} # задаем функцию
> curve(f, xlim = c(-3, 5), ylim = c(-3, 3), col = "blue") # строим 1-ый график
> y <- function(x){sin(1/x)}
> curve(y, xlim = c(-1, 1), ylim = c(-1, 1),col = "blue",n = 1000) # строим 2-ой график
```



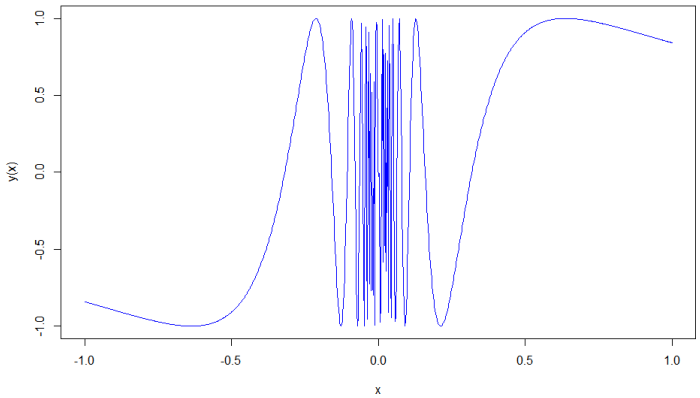


Рис. 1 Графики аналитических функций

Создание локальной переменной для функции выполняется в результате присвоении имени значения. Имя автоматически становится локальной переменной и будет удалено после завершения работы функции.

Приведенная ниже функция отобразит вектор **x** в единичный отрезок. Ей требуются два промежуточных значения, **low** и **high**:

```
> unit <- function(x) {
+ low <- min(x)
+ high <- max(x)
+ (x-low) / (high-low)
+   }
> unit(5:10)
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

> Значения **low** и **high** автоматически создаются операторами присваивания. Поскольку присваивание происходит в теле функции, эти переменные являются локальными для функции. Это дает два важных преимущества.

Во-первых, локальные переменные с именами **low** и **high** отличаются от любых глобальных переменных с именами **low** и **high** в вашем рабочем пространстве. Поскольку они отличаются, здесь нет «столкновения»: изменения в локальных переменных не изменяют глобальные переменные.

Во-вторых, локальные переменные исчезают, когда функция завершает работу. Это предотвращает беспорядок и автоматически освобождает пространство, которое они использовали.

R позволяет устанавливать значения по умолчанию для параметров, включив их в определение **function**:

```
myfun <- function(param = default_value) {
...
}
```

Создадим небольшую функцию, которая приветствует кого-то по имени:

```
> greet <- function(имя) {
+   cat("Привет,", имя, "\n")
+ }
> greet("Петр")
Привет, Петр
```

Если мы вызовем **greet** без аргумента **имя**, то получим вот такую ошибку:

```
> greet()
Ошибка в cat("Привет,", имя, "\n") :
  аргумент "имя" пропущен, умолчаний нет
```

Однако можно изменить определение функции, чтобы определить имя по умолчанию. В этом случае мы по умолчанию используем общее имя ***друг***:

```
> greet <- function(имя = "друг") {
+   cat("Привет,", имя, "\n")
+ }
```

> greet()

Привет, друг

Крайне важно остановить обработку, когда ваш код сталкивается с фатальными ошибками с помощью функции **stop()**.

Проблемы возникают по разным причинам: неверные данные, ошибка пользователя, сбои в сети и недочеты в коде, среди прочего. Этот список бесконечен. Важно, чтобы вы предвидели потенциальные проблемы и писали код соответствующим образом.

Довольно часто вы будете писать функции, которые будут полезны в нескольких сценариях. Например, у вас может быть одна функция, которая загружает, проверяет и очищает ваши данные; теперь вы хотите повторно использовать эту функцию в каждом сценарии, который нуждается в данных.

Большинство новичков просто вырезают и повторно вставляют используемую функцию в каждый сценарий, дублируя код. Это создает серьезную проблему.

Что, если вы обнаружите ошибку в этом дублированном коде? Или что, если вы должны изменить код, чтобы приспособиться к новым обстоятельствам? Вы будете вынуждены отслеживать каждую копию и вносить одинаковые изменения повсюду. Это раздражает, и такой код подвержен ошибкам.

Вместо этого создайте файл, скажем **myLib.R**, и сохраните определение функции там. Содержимое этого файла может выглядеть так:

```
loadData <- function() {
# Здесь идет код для загрузки, проверки и очистки данных.
}
```

Затем, внутри каждого сценария, используйте функцию **source** для чтения кода из файла:

```
source("myLib.R")
```

Когда вы запускаете этот скрипт, функция **source** читает указанный файл, как если бы вы набирали содержимое файла в этом месте в сценарии. Это лучше, чем вырезание и вставка, потому что вы изолировали определение функции в одном известном месте.

5. Задачи

Задача 1. Напишите программу на R определения разности между трехзначным числом и числом, составленным из тех же цифр, но взятых в обратном порядке. Используйте операции численного деления и остатка от деления.

Задача 2. Измените программу так, чтобы для определения цифр трехзначного числа надо было бы делить не только на 10, но и на 100.

Задача 3. Измените программу так, чтобы определялась сумма между четырехзначным числом и числом, составленным из тех же цифр, но взятых в следующем порядке: 1-я цифра становится третьей, а третья -1-ой; 2-я цифра становится четвёртой, четвёртая-2-ой.

Задача 4. Вычислить в R определитель по правилу треугольника

$$\begin{vmatrix} 6 & 5 & -2 \\ 9 & -1 & 4 \\ 3 & 4 & 2 \end{vmatrix}$$

Задача 5. Вычислить в R определитель, используя его разложение по первой строке

$$\begin{vmatrix} 2 & -3 & 3 \\ 6 & 9 & -2 \\ 10 & 3 & -3 \end{vmatrix}$$

Задача 6. Создайте в R последовательность целых чисел от 100 до 120.

Задача 7. Создайте в R последовательность произведения 10 целых чисел с числом 2.

Задача 8. Создайте в R последовательность пяти действительных чисел от 4 до 10.

Задача 9. Не применяя функцию конкатенации создайте следующую последовательность целых чисел

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

Задача 10. Создайте правильный, дважды повторяющийся, ряд двух символьных факторных переменных.

Задача 11. Завершите запись нижеприведенных равенств и проверьте их тождество в R

dnorm(0) =…

dnorm(1) =…

Задача 12. Создайте вектор из четырех элементов: 88,10,15,17. Вставьте в него новое число 333 между между третьим и четвертым числом и создайте новый вектор. Примените индексацию элементов вектора.

Задача 13. Из вектора с элементами: 2,4,5,7,9,0 создайте подвектора:

1. X=(2,5)

2. Y=(4,5)
3. Z=(7,9,0)

Задача 14. Дан вектор q, в котором хранятся следующие значения:
1, 0, 2, 3, 6, 8, 12, 15, 0, NA, NA, 9, 4, 16, 2, 0
Выведите на экран:

- первый элемент вектора
- последний элемент вектора
- элементы вектора с третьего по пятый включительно
- элементы вектора, которые равны 2
- элементы вектора, которые больше 4
- элементы вектора, которые кратны 3 (делятся на 3 без остатка)
- элементы вектора, которые больше 4 и кратны 3
- элементы вектора, которые или меньше 1, или больше 5
- индексы элементов, которые равны 0
- индексы элементов, которые не меньше 2 и не больше 8

Задача 15. Известно, что в базе данных хранятся показатели по 3 странам за 5 лет. Таблица выглядит примерно так:

	country	year
1	France	2000
2	France	2001
3	France	2002
4	France	2003
5	France	2004
6	Italy	2000
7	Italy	2001
8	Italy	2002
9	Italy	2003
10	Italy	2004
11	Spain	2000
12	Spain	2001
13	Spain	2002
14	Spain	2003
15	Spain	2004

- Создайте вектор с названиями стран (первый столбец).
- Создайте вектор, который мог бы послужить вторым столбцом в таблице, представленной выше (подумайте, какую длину имеет этот вектор).

Задача 16. Исследователь сохранил доход респондента в переменную dohod:

dohod <- 1573

Исследователь передумал и решил изменить значение этой переменной - сохранить в нее натуральный логарифм дохода.

В результате раздумий он решил создать вектор income, в котором сохранены доходы нескольких респондентов:

income <- c(10000, 32000, 28000, 150000, 65000, 1573)

Исследователю нужно получить вектор income_class, состоящий из 0 и 1: 0 ставится, если доход респондента ниже среднего дохода, а 1 - если больше или равен среднему доходу.

Подсказка: сначала можно посчитать среднее значение по вектору income и сохранить его в какую-нибудь переменную. Пользоваться встроенной функцией mean() нельзя.

Задача 17. Создайте матрицу размерности 3 * 4, состоящую из 3, а затем измените некоторые ее элементы так, чтобы получить следующее:

[3 3 4 3]
[1 3 3 3]
[3 NA 3 1]

Задача 18. Создайте из следующих векторов матрицу, такую, что:

- векторы являются столбцами матрицы
- векторы являются строками матрицы

a <- c(1, 3, 4, 9, NA)
b <- c(5, 6, 7, 0, 2)
c <- c(9, 10, 13, 1, 20)

Дайте (новые) названия строкам и столбцам матрицы.

Задача 19. Может ли матрица состоять из элементов разных типов? Проверьте: составьте матрицу из следующих векторов (по столбцам):

names <- c("Jane", "Michael", "Mary", "George")
ages <- c(8, 6, 28, 45)
gender <- c(0, 1, 0, 1)

Если получилось не то, что хотелось, подумайте, как это можно исправить, не теряя информации, которая сохранена в векторах.

Добавьте в матрицу столбец age_sq – возраст в квадрате.

Задача 20. Выполнить произведение двух матриц C= A*B и для матрицы C найти сумму элементов по всем столбцам и произведение элементов по всем строкам:

	1	2	2	-1		-5	2	-1
	2	3	4	5		-1	7	3
A=	1	3	2	5	B=	-2	4	-3
	3	2	4	-3		1	3	2

Задача 21. Для матрицы A найти обратную и проверить ответ

	4	2	3
A=	1	1	0
	3	2	2

Задача 22. Найти решение системы уравнений и проверить ответ

$$\begin{cases} x + 2y + z = 1, \\ 2x + 3y + 2z = 2, \\ x - y + 3z = 0. \end{cases}$$

Задача 23. Создайте из векторов из задачи 19 список (*list*) и назовите его info.

- Обращаясь к элементам списка, выведите на экран имя Michael.
- Обращаясь к элементам списка, выведите на экран вектор gender.
- Назовите векторы в списке Name, Age, Gender. Выведите на экран элементы вектора Name.
- Добавьте в список вектор drinks, в котором сохранены значения: juice, tea, rum, coffee.
- Добавьте в список данные по еще одному человеку: John, 2 года, мужской пол, любит молоко.

Задача 24. В R есть функция strsplit(), которая позволяет разбивать строки на части по определенным символам.

Пусть у нас есть строка s:

s <- "a,b,c,d"

Мы хотим получить из нее вектор из 6 букв. Применям функцию:

let <- strsplit(s, ",")

Получили почти то, что хотели. Почему почти? Потому что получили не вектор, а список!

class(let)

[1] "list"

Превратим в вектор:

unlist(let)

[1] "a" "b" "c" "d"

Теперь все в порядке, получили вектор из четырех элементов.

Теперь задание. Дана строка index:

index <- "0,72;0,38;0,99;0,81;0,15;0,22;0,16;0,4;0,24"

Получите из этой строки числовой вектор I.

Задача 25.

1. Поставьте библиотеку randomNames.
2. Создайте вектор из 100 испанских имен:

   set.seed(1234) # чтобы у всех получались одинаковые результаты

   names <- randomNames(101, which.names = "first", ethnicity = 4) # чтобы у всех получались одинаковые результаты

   Будем считать, что 101 – количество имен опрошенных респондентов.
3. Создайте случайный вектор функцией *sample()* со значениями возраста респондентов в интервале от 16 до 80.
4. Создайте случайный вектор со значениями политических взглядов респондентов: *right, left, moderate, indifferent*.

5. Создайте из полученных векторов базу данных.

Задача 26.

6. Создайте столбец с порядковыми номерами респондентов.
7. Определите, сколько среди респондентов людей в возрасте от 20 до 35 лет включительно.
8. Определите, какую долю в выборке составляют люди в возрасте от 20 до 35 лет включительно и округлите её до двух знаков после запятой.
9. Выразите эту долю в процентах.

Задача 27.

10. Из имеющейся таблицы, создайте случайную выборку без замещения из 20 строк. С помощью редактора данных удалите данные в трех строках столбца «age».
11. В полученной таблице избавьтесь от наблюдений с неполными данными.
12. В образованной таблице измените имя столбца «polit» на «politic» и замените, имеющееся число в первой строке столбца «age», на число 99, не используя редактор данных.
13. Отсортируйте данные по столбцу «age», по возрастанию.
14. Подсчитайте средний возраст респондентов.

Задача 28. Определить нормальный вес человека и индекс массы его тела по формулам: h*t/240 и m/h^2, где h – рост человека (измеряемый в сантиметрах в первой формуле и в метрах – во второй); t - длина окружности грудной клетки (в см); m – вес (в кг). Индекс массы тела принят Всемирной организацией здравоохранения и не должен превышать 25 пунктов.

Задача 29. Треугольник задан координатами своих вершин. Найти периметр и площадь треугольника.

Задача 30. В каждый подарочный набор входят 1 ручка, 2 линейки, и 4 тетради. Имеется **a** линеек, **b** тетрадей, **c** ручек. Сколько всего получится подарочных наборов?

Задача 31. Дано действительное число x. Вычислить f(x), если

$$f(x)=\begin{cases}0 & \text{при } x\le 0,\\ x^2-x & \text{при } 0<x\le 1,\\ x^2-\sin(\pi x^2) & \text{при } x>1\end{cases}$$

Задача 32. Даны действительные числа x и y. Если оба числа отрицательные, то каждое значение заменить его модулем; если отрицательно только одно из них, то оба значения увеличить на 0.5. В остальных случаях оба числа оставить без изменения.

Задача 33. Дано натуральное число n. Вычислить:

$$\sqrt{2+\sqrt{2+\sqrt{2}+\ldots+\sqrt{2}}} \quad (\textit{всего } n \textit{ корней})$$

Задача 34. Дано действительное число x. Вычислить сумму с помощью цикла for:

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$$

Задача 35. Дано натуральное число n. С помощью цикла while определить: а) сколько цифр в числе n? б) чему равна сумма его цифр? в) найти первую цифру числа n.

Задача 36. Даны 50 целых чисел. Получить сумму тех чисел данной последовательности, которые: а) кратны 5; б) нечетны и отрицательны.

Задача 37. Напишите функцию, которая принимает на вход числовой вектор и возвращает вектор, состоящий из кубов элементов вектора, поданного на вход.

Задача 38. Напишите функцию , которая принимает на вход вектор, и если он числовой, то возвращает вектор из квадратов его элементов, а если нет — возвращает вектор из пропущенных значений и выводит на экран сообщение “Не числовой вектор.”

Задача 39. Создайте функцию для расчета стандартных ошибок.

Задача 40. Напишите функцию для вычисления общих характеристик данных: среднего арифметического значения выборки, стандартного отклонения, медианы и её абсолютное отклонение.

Заключение

В этом учебно – методическом пособии рассмотрен R как язык программирования, а не как инструмент применения конкретных алгоритмов. В пособии представлены основные типы данных и универсальные сематические правила, а также затронуты некоторые темы, которые с успехом могут быть использованы при изучении таких дисциплин, как «Языки программирования для больших данных», «Прикладная статистика», «Математическое и имитационное моделирование» и «Эконометрика». Показано замечательное свойство R – в этой программе всегда есть чему научиться. R – это обширная, мощная и постоянно развивающаяся интерактивная среда и язык программирования, которые востребованы при подготовке бакалавров по направлению 09.03.03 «Прикладная информатика».

В пособии рассмотрены базовые конструкции и основные понятия языка R: переменные, глобальное окружение, управляющие конструкции, векторы. При изучении курса студенты осваивают общие операции над матрицами и списками, кадрами данных, факторами и таблицами. Познают продвинутое программирование, что обеспечивает определенное единообразие работы с данными, т. е одна функция может использоваться для разных типов данных, для которых выбирается подходящий способ обработки.

Как для всех языков функционального программирования, в программировании на R предотвращается явное программирование итераций. За счет векторизации, которая позволяет исключить циклы в программном коде достигается кардинальное ускорение работы программы.

Настоящее учебно – методическое пособие является методической поддержкой указанных выше курсов. Оно будет полезно преподавателям и студентам высших учебных заведений. Правильная организация своей образовательной траектории поможет студентам получить надежную теоретическую и практическую основу для дальнейшего овладения и использования современных методов анализа больших массивов данных.

Список литературы

1. Кабаков Р. И. R в действии. Анализ и визуализация данных в программе R / Кабаков Р. И. – М.: ДМК Пресс, 2014. - 588 с.
2. Мэтлофф Н. Искусство программирования на R. Погружение в большие данные / Мэтлофф Норман – СПб.: Питер, 2019. – 416 с.
3. Гришин В. А., Тихов М. С. Методы обработки данных и моделирование на языке R: Учебно-методическое пособие/ Гришин В. А., Тихов М. С. — Нижний Новгород: Нижегородский госуниверситет, 2019. – 54 с.

Владимир Анатольевич Гришин

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ R

Учебно – методическое пособие

Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского»

603950, Нижний Новгород, пр. Гагарина, 23