Казанский (Приволжский) федеральный университет

СТАТИСТИЧЕСКИЙ ЯЗЫК R

Первоначальные сведения

Григорьева И.С.
Каф. матем. статистики

Оглавление

Введение. Что такое *R* и зачем он нужен?

В начале 90-х годов в *Bell Labs* (там, где в свое время придумали *Unix* и *C*) изобрели язык, специально предназначенный для обработки данных. Этот язык назвали *S*. Затем фирма *MathSoft, Inc*. на основе этого языка разработала программу *S-PLUS*, которая успешно существует и по сей день. В 1997 году группа энтузиастов задумала создать программу, аналогичную *S-PLUS*, но свободно распространяемую. Так появился ***R***.

R – это среда для статистической обработки информации. "Среда" означает не просто набор программ, но и цельный, очень развитый язык программирования высокого уровня, который позволяет быстро и с удобством производить различные вычисления. Например, Вам нужно посчитать среднее для некоторой выборки (назовем ее *x*). ***R*** делает это в одно действие[1] (то, что мы видим на экране, далее будет выделено шрифтом и рамкой):

```
> x       # посмотрим, что это за выборка
> [1] 1 2 3 4 5 6 7 8 9 # вот она
> mean(x)     # посчитаем среднее
> [1] 5      # а вот и ответ
```

В общем-то, среднее для этой выборки можно посчитать и в уме. Но ***R*** позволяет подсчитывать гораздо более сложные вещи, такие как, например, вейвлет-преобразования, главные компоненты, дискриминантные функции и т.п. Это значит, что его можно использовать вместо широко употребляемых программ для статистической обработки данных типа *STATISTICA*. Вот несколько причин:

R, как и его родоначальник, язык *S*, является языком, созданным специально для работы с данными. За годы существования ими накоплен гигантский запас методов и приемов обработки данных практически на все случаи жизни. Если в *STATISTICA* или *SPSS* не реализована какая-то функция, ее почти наверняка можно найти в ***R***.

R поддерживается коллективом специалистов-статистиков, а это значит, что любая новая статистическая методика гораздо быстрее появится в качестве функции для ***R***, нежели для всех остальных программ. Это значит также, что он тщательно оттестирован и практически не содержит столь характерных для многих статистических программ (прежде всего для *STATISTICA*) упрощений и ошибок, а вновь найденные устраняются практически мгновенно.

R относится к свободному программному обеспечению. Любой пользователь может использовать ***R*** бесплатно, скачав версию для своего компьютера непосредственно с *Web*-сайта программы http://www.r-project.org/.

На любой вопрос по ***R***, заданный в одном из листов рассылки, в кратчайшие сроки можно получить квалифицированный ответ одного из ведущих в данной области специалистов.

С другой стороны, хотя ***R*** и превосходно документирован, работа в нем происходит через так называемый "интерфейс командной строки", а привычные большинству пользователей кнопки и выпадающие меню практически отсутствуют.

Но сила ***R*** там же, где его "слабость". Интерфейс командной строки позволяет делать такие вещи, которых рядовой пользователь других статистических программ может достичь только часами ручного труда. Вот, например, простая задача: требуется превратить выборку *x* (см. выше) в матрицу из трех колонок (допустим, это были данные за 3 дня, а

[1] # - это символ комментария, и все, что за ним, **R** не читает. Подразумевается, что в конце каждой строки, начинающейся с >, была нажата клавиша <ENTER>.

каждый день делалось 3 измерения). Чтобы сделать это в *STATISTICA*, требуется: (1) учредить две новые переменные, (2-3) скопировать дважды кусок выборки в буфер, (4-5) скопировать его в одну и другую переменную и (6) уничтожить лишние строки. В ***R*** это делается так:

```
> b <- matrix(x, ncol=3)   # применим команду matrix()
> b          # посмотрим, что вышло
  [,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9
```

R поддерживает широкий спектр статистических и численных методов и обладает хорошей расширяемостью с помощью пакетов. Пакеты представляют собой библиотеки для работы специфических функций или специальных областей применения. В базовую поставку ***R*** включен основной набор пакетов, а всего по состоянию на 2006 год доступно более 800 пакетов.

Еще одной особенностью ***R*** являются графические возможности, заключающиеся в возможности создания качественной графики, которая может включать математические символы.

R и дополнительные пакеты распространяются через *CRAN* (аббревиатура от *Comprehensive* ***R*** Archive *Network*). В настоящее время в мире доступны более 60 зеркал *CRAN*. Головной узел — (http://cran.r-project.org/) расположен в Вене (Австрия).

Два-три раза в год выходит свободно-распространяемый информационный бюллетень. Он содержит информацию по статистической обработке данных и разработке, что может быть интересно как пользователям, так и разработчикам ***R***. Бюллетень выходит с января 2001 года.

Задание 1. Загрузите язык ***R*** с головного узла. На рабочем столе должен появиться ярлык ***R***. Вызовите через него основную консоль. Изучите меню. Закройте консоль (как всякое окно) с сохранением данных. По умолчанию основным каталогом будет назначена папка "Мои документы". Зайдите в нее. Какие файлы в ней появились?

Задание 2. Сменим папку. Создайте свою папку (например, с именем <Группа><Фамилия><***R***>). Запустите ***R*** с рабочего стола. В меню "Файл" выберите пункт "Изменить папку" и выберите из списка свою папку. Закройте окно с сохранением. Проверьте, какие файлы появились в вашей папке? Теперь Вы можете запускать сеанс ***R*** с помощью файла *.RData*. При желании можете создать для этого файла ярлык на рабочем столе (с указанием вашего имени).

Стиль программирования R

R — это объектно-ориентированный язык программирования. Это означает, что теоретически всё что угодно может быть сохранено как объект ***R***. Каждый объект имеет свой класс, описывающий, что содержит этот объект и что могут делать с его содержимым методы языка. Например, *plot*(*x*) выводит один результат, если *x* является регрессией, и другой, если вектором.

Язык ***R*** «общается» с пользователем не с помощью кнопок и меню, а с помощью командной строки. В начале строки может стоять символ > (приглашение системы) или +, если запись команды не закончена и требуется продолжение. Строки команд пишутся красным шрифтом. Результаты работы программы (если они выводятся на экран) записаны синим цветом.

При необходимости можно вызвать на экран одну из предыдущих строк стрелками «вверх» и «вниз». Ее можно редактировать, пока не нажат знак перевода строки. Заметьте, что для движения по строке надо использовать стрелки, а не курсор мышки.

После редактирования *Enter* можно нажимать «в любом месте», т.е. курсор не обязан стоять в конце строки.

Активной является последняя строка, предыдущие можно только просматривать. Можно также их выделять и копировать обычными средствами, в том числе с помощью мыши.

```
> a <- c(1:10,15)
```

Символом присвоения является «<-». Также возможно использовать классический знак «=». Три следующих выражения являются эквивалентными:

```
> a <- 2
> a = 2
> 2 -> a
```

«#» используется для комментариев:

```
# Это комментарий
> 5 + 7 # Это тоже комментарий
```

Команды отделяются точкой с запятой или символом конца строки. Если вы хотите разместить в одной строке более одного выражения, то необходимо использовать разделитель «;».

```
> a <- 1:10; mean(a)
```

И наоборот, вы можете разбить одно выражение на несколько строк кода. Если оператор не закончен, то при нажатии *Enter* в следующей строке появится знак «+» на месте приглашения системы, т.е. знака «>». Он обозначает не сложение, а присоединение строки.

R чувствителен к регистру: «*a*» и «*A*» являются двумя разными объектами.

Традиционно символ подчёркивания «_» не используется в именах. Зачастую лучше использовать символ точки «.». Следует избегать использования символа подчёркивания в качестве первого символа в имени объекта.

Аргументы функций и методов передаются в круглых скобках. Функции можно подставлять одну в другую. Например, вы можете написать:

```
> mean(rnorm(1000)^2)
```

Ясно также, для чего нужны сами круглые скобки. В них, как правило, пишутся *аргументы* команды. Этих аргументов может быть очень много, и один из самых простых способов узнать, какие той или иной команде требуются аргументы, это ввести команду *args*(команда). Например, команда *round*() ("округлить") имеет два аргумента: число, которое нужно округлить, и значение *digits*, сообщающее, до какого знака округлять.

```
> args(round)
function (x, digits = 0)
NULL
```

Заметьте, что в определении функции задано имя одного из аргументов (*digits*), а также его значение по умолчанию, 0.

Система аргументов работает разумно, так что все равно, что написать:

```
> round(1.5, digits=0)
> round(1.5, d=0)
> round(d=0, 1.5)
> round(1.5, 0)
> round(1.5,)
> round(1.5) # здесь учтено значение по умолчанию
```

Если Вы не знаете, как работает та или иная команда, надо вызвать помощь:

```
> help(round)
```

или

```
> ?round
```

Справочные файлы почти всегда содержат примеры, которые можно запустить командой *example*(). Можно также просто скопировать пример из справки и вставить в сеанс работы ***R***.

Аргумент функции *help* рассматривается как строка. В некоторых случаях это надо обозначать кавычками. Например, нельзя записать ?+ или ?<- , но можно ?"+" или ?"<-" (попробуйте!).

Справка *help* выдает информацию только по полному названию объекта. Если же вы не помните его точно, можно использовать оператор *apropos*:

```
> apropos("len")
 [1] ".GlobalEnv" "getSrcFilename" "globalenv" "length"
"length.POSIXlt"
 [6] "length<-" "length<-.factor" "pushBackLength" "seq_len"
"testPlatformEquivalence"
[11] "trySilent" "URLencode"
```

В каждом выведенном названии есть подстрока "*len*". По умолчанию этот оператор не учитывает строчные/прописные буквы. Это можно изменить параметром *ignore.case*.

Другой оператор поиска – *find*.

Очень полезны команды *ls*() и *rm*(), которые нужны для того, чтобы просматривать и удалять существующие в памяти компьютера объекты.

Задание 1. Попробуйте запросить справку по справке, т.е. *help*(*help*)

Задание 2. Запишите (скопируйте) в командной строке следующий текст:

```
a<-c(1:5, seq(2,8, by=2))
a
summary(a)
is.numeric(a)
sort(a)
```

(каждую строку заканчивайте клавишей *Enter*) и посмотрите, что получится. Постарайтесь понять смысл этих операторов.

Язык ***R*** не имел бы, конечно, никакой ценности, если бы команды приходилось каждый раз вводить с клавиатуры. На самом деле вся история команд сохраняется в файле с расширением *.Rhistory* (если, конечно, вы не отказались от этой возможности при выходе из сеанса). Этот файл можно отредактировать обычным текстовым редактором. На основе этой истории можно, убрав из нее все неверные команды, создать скрипт (сценарий). По сути это просто текстовый файл со списком команд (без знаков приглашения системы >). Его можно создать и без помощи ***R***, в любом текстовом редакторе. Для запуска готового скрипта используйте меню "Файл" в окне консоли ***R***, пункт «Загрузить код ***R***» (или пиктограмму "Открыть скрипт"). Там же при необходимости можно поменять папку, в которой сохраняются текущие файлы.

С самого начала работы следует создавать скрипты, даже для таких задач, которые кажутся пустяковыми – это в будущем значительно сэкономит Ваше бесценное время. Создание скриптов по любому поводу и даже без особого повода – одна из основ культуры работы в ***R***.

Простые примеры

Прежде чем изучать язык ***R*** последовательно, поработаем с простыми скриптами.

Вы можете набрать их с клавиатуры в консоли языка ***R*** или записать в файл, который потом можно вызвать через меню Файл. Если вы пользуетесь электронной версией пособия, просто скопируйте команды из данного файла и вставьте в командную строку. Впрочем, удобнее вводить команды по одной, чтобы разобраться в их действии.

Далее оформление текста соответствует таковому на экране: красным цветом помечены водимые команды, синим – текст, выводимый компьютером.

Пример 1

```
x<- seq(-2,2,by=0.5)
x
x>0
y <- x[x>0]
y
y[x>0] <- 10
y
x^2
sqrt(x)->r
r
is.na(r)
r<1
```

Вот как будет выглядеть на экране результат применения этих команд:

```
> x<- seq(-2,2,by=0.5)
> x
[1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
> x>0
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> y <- x[x>0]
> y
[1] 0.5 1.0 1.5 2.0
> y[x>0] <- 10
> y
[1]  0.5  1.0  1.5  2.0   NA 10.0 10.0 10.0 10.0
> x^2
[1] 4.00 2.25 1.00 0.25 0.00 0.25 1.00 2.25 4.00
> sqrt(x)->r
Предупреждение
In sqrt(x) : созданы NaN
> r
[1]       NaN       NaN       NaN       NaN 0.0000000 0.7071068
1.0000000 1.2247449 1.4142136
> is.na(r)
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
> r<1
[1]    NA    NA    NA    NA  TRUE  TRUE FALSE FALSE FALSE
```

Основным объектом языка *R* является вектор (набор значений). Даже скаляр (переменная, имеющая единственное значение) – это вектор длиной 1. В данном примере *x* создан как числовой вектор (арифметическая прогрессия), результатом сравнения *x* > 0 является вектор такой же длины, состоящий из логических значений.

В квадратных скобках указаны индексы тех значений, которые мы хотим извлечь из вектора. Как мы видим, индексы могут быть логическими. С помощью квадратных скобок можно не только извлекать значения из вектора, но и заменять их и даже создавать но-

вые: заметьте, что при выполнении команды *y*[*x* > 0] <- 10, длина вектора *y* была увеличена в соответствии с длиной вектора *x*.

Вообще, если в выражении два вектора имеют разную длину, то меньший из них повторяется до тех пор, пока их длины не сравняются:

```
> x<-1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x+c(1,0)
 [1]  2  2  4  4  6  6  8  8 10 10
> x+c(1,0,0)
 [1]  2  2  3  5  5  6  8  8  9 11
Предупреждение
In x + c(1, 0, 0) :
  длина большего объекта не является произведением длины меньшего объ-
екта
```

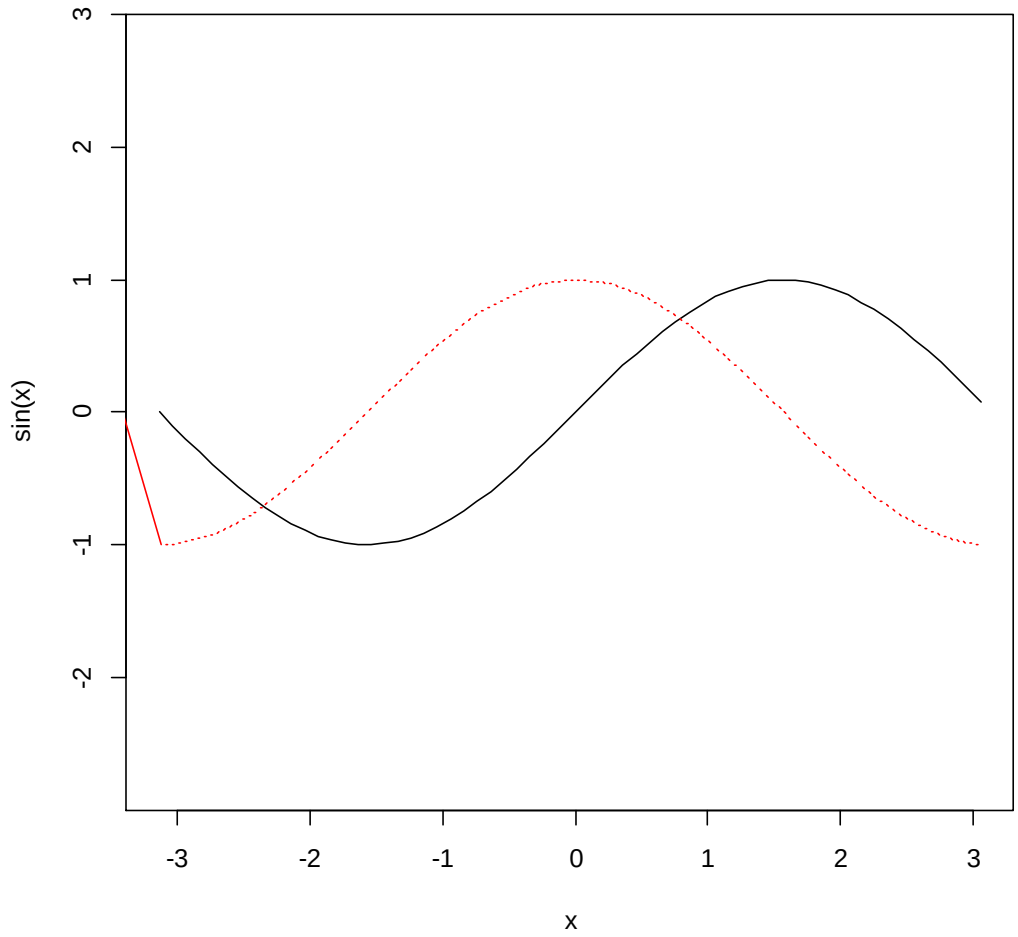
Задание. Попробуйте объяснить смысл выражений *x*[1:2==1], *r*[*is.na*(*r*)]<-0

Пример 2

```
x<-seq(-pi,pi,by=0.1)
plot(x,sin(x),type="l",asp=1)
lines(x,cos(x),lty="dotted",col="red")
```

никогда не присваивайте ничего переменой *pi*, иначе вы потеряете значение π.

Результатом выполнения этой команды будет появление нового (графического) окна, в котором будет такая картинка:

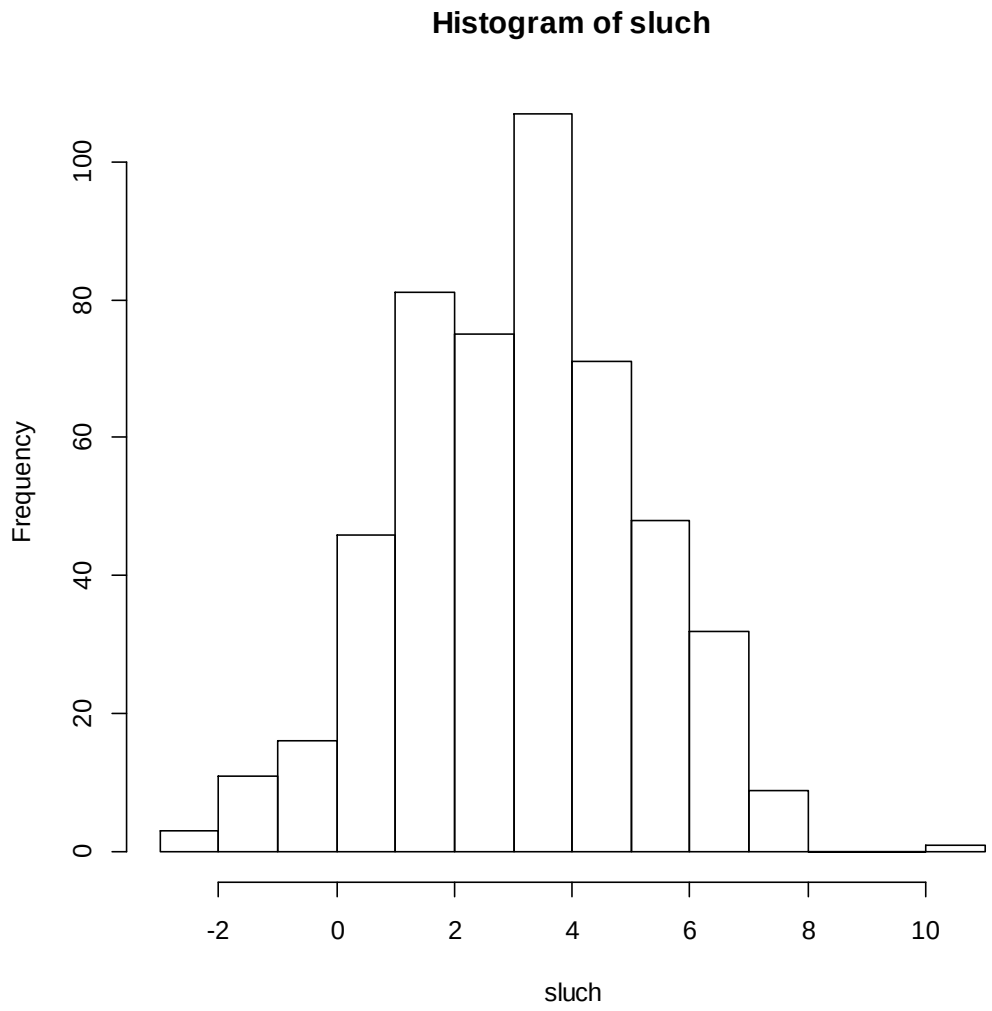


Замечание. Если хотите узнать смысл параметров, наберите ?*plo*t, откроется файл справки по этой функции.

Пример 3

```
sluch <- rnorm(500,3,2)
hist(sluch)->h
```

Первая команда создает выборку размера 500 из нормально распределенной случайной величины. С помощью второй команды мы строим ее гистограмму:



Кроме того, команда *hist*() создает новый объект класса гистограмма:

```
class(h)
[1] "histogram"
```

Его состав можно просмотреть командой *str*() (то есть структура):

```
str(h)
List of 6
 $ breaks  : num [1:15] -3 -2 -1 0 1 2 3 4 5 6 ...
 $ counts  : int [1:14] 3 11 16 46 81 75 107 71 48 32 ...
 $ density : num [1:14] 0.006 0.022 0.032 0.092 0.162 0.15 0.214 0.142
0.096 0.064 ...
 $ mids    : num [1:14] -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5 4.5 5.5 6.5 ...
 $ xname   : chr "sluch"
 $ equidist: logi TRUE
 - attr(*, "class")= chr "histogram"
```

Задание. С помощью команды *ls*() выясните названия всех объектов, которые вы создали к этому времени. С помощью команды *str*() изучите структуру каждого объекта.

Список полезных команд

Перечислим здесь некоторые команды и константы, которые упоминались в тексте

Команда	**Смысл**
help() или ?	Справка по команде
apropos()	Поиск названия по его части
ls()	*list*, вывод списка объектов
rm()	*remove*, удаление объектов
args()	Выдает аргументы функции
class()	Выдает класс объекта
str()	Структура объекта
head()	Вывод первых нескольких значений объекта
->, <-	Присвоение значений
[	Извлечение значений из объекта
$	Обращение к элементу списка по имени
c()	Конкатенация, объединение векторов
plot()	Открывает окно графического вывода и выводит на печать
hist()	Строит и выводит на печать гистограмму
lines()	Строит линейные графики
rnorm ()	Создает выборку из нормально распределенной с.в.
runif()	Создает выборку из равномерно распределенной с.в.
mean()	Вычисление среднего
sum()	Вычисление суммы
summary()	Вывод сводки по объекту
table()	Подсчет количества отдельных значений
is.na()	Проверка (является ли *NA*, неопределенным значением)
is.numeric()	Проверка (является ли числовым значением)
as.numeric()	Преобразование в тип «числовой»
vector()	Создание объекта класса «вектор»
as.vector()	Преобразование в объект класса «вектор»
is.vector()	Проверка (является ли объект вектором)
==	Проверка равенства значений
Идентификатор	**Смысл**
TRUE	Истина
FALSE	Ложь
NA	Отсутствующее значение (определенного типа)
NaN	**N***ot* **a N***umber*, нечисловое значение, ошибочный результат
Inf	Бесконечность
pi	Число π. **Не является зарезервированной строкой**!

Некоторые примеры программ (скриптов)

Прежде, чем перейти к более регулярному изложению, приведем примеры программ, включающих в себя основные этапы обработки данных

Ввод данных

Данные записаны в файл *Данные.csv* текущей папки. Они имеют вид

№ пробы	Место	*Cd*	*Pb*	*Co*	*Cu*	*Ni*	*Zn*	*Cr*	*Mn*	*Fe*	Сезон	Категория
7	Апастово	0,32	3,8	0,7	44	1,56	127	0,43	79	240	зимние	загрязн
8	Кунгер	0,15	2,1	0,3	20	1,54	123	2,04	48	190	зимние	фоновая
19	Бимери	0,21	2,7	1,1	16	1,34	100	0,24	29	80	зимние	загрязн
22	Тетеево	0,5	0,8	0,2	15	0,99	120	0,19	77	90	зимние	фоновая
25	Рудник	0,23	3	1,5	34	1,36	110	5,49	47	170	зимние	загрязн
26	Рудник	0,23	2,7	0,5	27	0,6	112	0,02	58	140	осенние	загрязн
27	Рудник	1,14	6,5	0,7	20	0,29	140	0	185	240	летние	загрязн
29	Наб. Челны	0,21	4	1,2	37	1,92	128	2,95	107	180	зимние	загрязн
…	…	…	…	…	…	…	…	…	…	…	…	…

И т.д., всего 53 объекта.

Скрипт записан в файле с расширением *.R*. Вот его содержимое (текст после знака # является комментарием):

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#  Делаем запрос на выбор файла

cat("Какой файл выбрать? По умолчанию - Пример. Расширение csv","\n")

readline()->imja; if (imja=="") {imja <-"Пример"}

putdan<-paste(imja,".csv",sep="") #путь и наименование файла данных

dann<-read.table(putdan,h=T,sep=";",dec=",") # Чтение основных данных

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# выяснение размерности данных: n-число объектов,
# m - число столбцов, имя которых состоит из 2 букв

n<- length(dann[,1]); m<- sum(nchar(attr(dann,"names"))==2)

cat("Обрабатывается файл",putdan,"Число объектов",n,"Число показате-
лей",m,"\n")

dann_num<-dann[3:(2+m)]

dann_kat<-dann[(3+m):(4+m)]
```

После запуска этого сценария на экране появятся следующие записи (вторая строка введена пользователем)

```
Какой файл выбрать? По умолчанию - Пример. Расширение csv
Данные
Обрабатывается файл Данные.csv Число объектов 53 Число показателей 9
```

Задание. Объясните, как действует команда

```
m<-sum(nchar(attr(dann,"names"))==2)
```

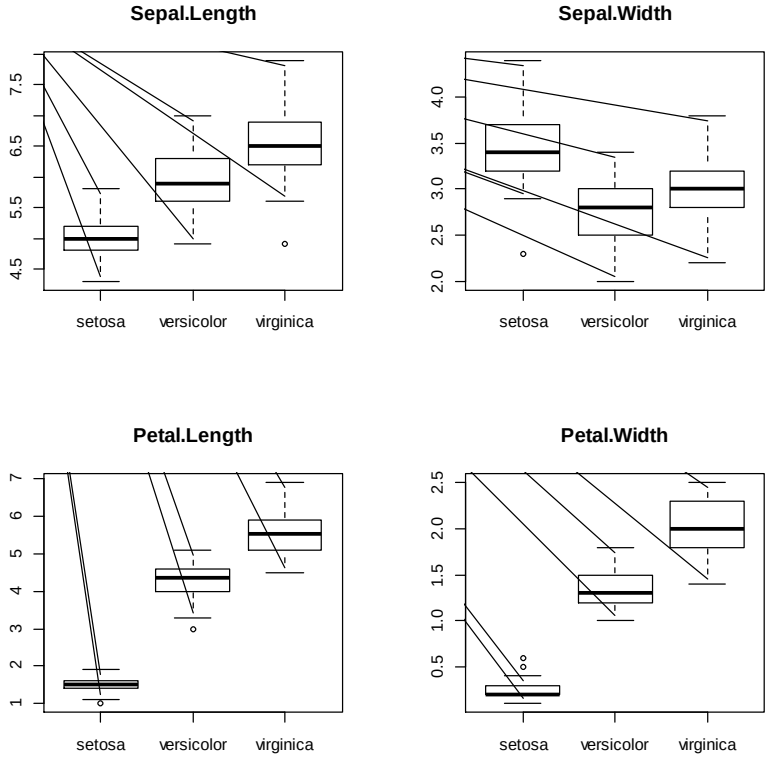
Тест Стьюдента

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#  Исследование встроенного объекта iris
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
cat("Сравниваем средние значения показателей для двух сортов:
setosa,versicolor","\n")
par(mfrow=c(2,2))->o.p # подготовка окна вывода

# Цикл по всем показателям
for (n in 1:4) {
t.test(iris[,n]~iris[,5],subset=iris[,5]==c("setosa","versicolor"))->st

cat("Показатель",names(iris[n]),"p-value =",st$p.value, sep="\t","\t")

if (st$p.value>0.05) cat("Не "); cat("значимо","\n")
boxplot(iris[,n]~iris[,5], main = names(iris[n]))
}
par(o.p) # возвращение в стандартным параметрам вывода
```

На экране появятся следующие результаты:

```
Сравниваем средние значения показателей для двух сортов:
setosa,versicolor
Показатель    Sepal.Length  p-value =    1.124894e-08      значимо
Показатель    Sepal.Width   p-value =    1.504356e-10      значимо
Показатель    Petal.Length  p-value =    3.587603e-24      значимо
Показатель    Petal.Width   p-value =    4.818766e-22      значимо
```

Кроме того, команда *boxplot*() выведет на экран так называемые «ящики с усами», по которым можно получить наглядное представление о значениях показателей:



В этом сценарии показано, как в языке *R* можно организовать цикл, а также проверку условия. Однако без крайней необходимости делать это не рекомендуется: вместо циклов и условных операторов используются векторные вычисления.

Запуск скрипта производится командой *source*() или из меню (Файл –> Загрузить код *R*..).

Типы данных и объекты *R*

Все манипуляции с данными в языке ***R*** основаны на векторах. Вектор представляет собой структуру, содержащую последовательность значений одного типа. Если возникает необходимость использования в программе скаляров, то их можно заменить единичными векторами, т. е. векторами, содержащими только одно значение.

Элементы вектора могут принадлежать к одному из типов данных, определенных в языке ***R***:

numeric – числовой тип, который включает в себя как целые числа, так и дроби;

integer – целочисленный тип;

character – символьный тип данных, каждый элемент в таком векторе является последовательностью из одного или более символов. Следует отметить, что совокупность элементов символьного вектора не является единой строкой;

complex – комплексный тип;

logical – логический тип, принимает значения *TRUE* или *FALSE*.

Кроме конкретных значений, в ***R*** есть специальные константы *NA*, *NaN* и *Inf*. Первое обозначает пропущенное значение (определенного типа!), второе – нечисловое значение (*Not a Number*), третье – бесконечность (положительную).

Numeric

Объекты этого типа могут содержать только числа. Эквивалентные обозначения: *double* или *real*, последнее существует только для сохранения обратной совместимости со старым кодом. Таким образом, в языке ***R*** имеют место три варианта обозначения векторов, содержащих числа с плавающей точкой. Очевидно, что объекты данного типа созданы для выполнения математических операций. Проверка на принадлежность переменной типу *numeric* выполняется функцией *is.numeric*(). Преобразование в числовой тип можно произвести функцией *as.numeric*().

Integer

Этот тип создан для того, чтобы обеспечивать совместимость с кодом, на языках *C* или *Fortran* таким образом, чтобы представить целочисленные данные наиболее компактно. Следует отметить, что в актуальной на сегодня реализации ***R*** используется 32-битный тип *integer*, разброс значений которого ограничен +/- 2*10^9. В свою очередь, объекты типа *double* могут вмещать целые числа из более широкого диапазона. Чтобы убедиться, что некоторый вектор содержит целые числа можно использовать функцию *is.inetger*(). Преобразование в целочисленный тип можно произвести функцией *as.integer*().

Character

Тип данных ch*aracter* создан для выполнения операций с символами. Символом может быть что угодно: буквы алфавита, доступные в той или иной кодировке, цифры, а так же любой другой символ, который пользователь сможет найти на своей клавиатуре. Понятно, что со строковыми векторами невозможно проводить никаких вычислений, даже если в них содержатся цифры. В языке ***R*** есть предопределенные константы '*letters*' и '*LETTERS*' которые содержат соответственно строчные и прописные буквы английского алфавита.

В тех случаях, когда с цифрами, содержащимися в строковых векторах, необходимо провести вычисления их следует преобразовать в числовой тип функциями *as.integer*()

или *as.numeric*(). Соответственно, если мы хотим преобразовать число в строку, то используется функция *as.character*(), а для выяснения типа переменной – *is.character*().

Complex

Комплексные векторы содержат комплексные числа. Имя *i* не обозначает мнимую единицу. В этом качестве применяется обозначение 1*i* или *i* с другим коэффициентом.

```
> x = 1+0i,
# тригонометрическая форма
> Phi<-pi/6
> cos(Phi)+sin(Phi)*i
Ошибка: объект 'i' не найден
> cos(Phi)+sin(Phi)*1i
[1] 0.8660254+0.5i # или так:
> complex(1,sqrt(2)/2,sqrt(2)/2)
[1] 0.7071068+0.7071068i
```

Logical

Логические переменные могут принимать только два значения *TRUE* (истина) и *FALSE* (ложь). *TRUE* и *FALSE* – зарезервированные слова языка ***R***, обозначающие логические константы. Эти обозначения в программном коде можно заменять символами '*T*' и '*F* '. Если попытаться конвертировать логический вектор в вектор типа *integer*, то в результате все значения *TRUE* будут заменены на 1, а *FALSE* – на 0.

Логические значения получаются в результате проверок условий. Например,

```
> x
 [1] 4 5 3 2 3 4 1 5 4 4
> x==4 # заметьте, что знак равенства удвоен
 [1] TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE TRUE
> x>=4
 [1] TRUE TRUE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE
> x!=3 # знак ! означает отрицание
 [1] TRUE TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE
```

Логические векторы могут быть использованы в обычных арифметических операциях, и в этом случае они приводятся к числовым векторам, *FALSE* становится 0, а *TRUE* становится 1. Однако есть ситуации, когда логические векторы и их приведения к численному виду отличаются.

```
> c(1,2,NA,NaN)->x
> x==1
[1] TRUE FALSE NA NA
> x==NA
[1] NA NA NA NA
> x==NaN
[1] NA NA NA NA
```

Объекты языка *R*

Вектор является объектом языка ***R*** и, как уже было сказано, может содержать компоненты только одного типа, а так же *NA*-значения, т. е. пропущенные значения (которые также имеют тип). Однако в ***R*** есть и другие объекты: матрицы, списки, таблицы, факторы, методы. Тем не менее, в основе всех этих типов объектов лежит базовый объект – вектор.

Каждый объект имеет ряд атрибутов, характеризующих его свойства. Основные атрибуты это тип объекта и его длина. Под типом объекта понимается тот тип данных, к которому принадлежат его компоненты. Например, вектор, содержащий числовые значения типа *numeric*, является объектом типа *numeric*. В свою очередь длина объекта – это

количество элементов, которые он содержит. Для определения типа и длины объектов используются функции *mode*() и *length*() соответственно. Функция *length*() позволяет еще и задавать размер объекта. Так, строка *length*(*x*) <- 3 задает вектору *x* размер в три единицы. Если ранее в векторе *x* было более трех компонентов, то данное действие удалит все остальные значения, если же длина вектора была менее трех, то функция добавит недостающее количество *NA*-значений. Следует отметить, что даже пустой вектор принадлежит определенному типу, так пустой символьный вектор обозначается *character*(0).

Оба описанных атрибута называют внутренними свойствами объекта. Для того, чтобы получить все остальные, если они есть, необходимо использовать функцию *attributes*(). В том случае, когда мы хотим получить определенные атрибуты объекта, не являющиеся внутренними, используется функция *attr*(). Например, *attr*(*x*,"*name*").

Общее представление о структуре объекта можно получить с помощью команды *str*(), например:

```
> str(tabel)
List of 3
 $ : chr [1:7] "пн" "вт" "ср" "чт" ...
 $ : int [1:7, 1:5] NA NA 1 2 3 4 5 6 7 8 ...
 $ holy: logi [1:31] TRUE TRUE FALSE FALSE FALSE FALSE ...
```

Мы видим, что объект *tabel* есть список (*list*) из трех компонент: символьной, числовой (в виде матрицы 7x5) и логической с именем *holy*. На экран выведены также некоторые значения этих компонент.

Один из атрибутов любого объекта – это класс. Принадлежность к тому или иному классу определяет способ представления данных пользователю, а также их обработку соответствующими методами. Данный атрибут является символьным вектором, содержащим список классов, к которым относится данный объект. Пользователь может изменять значения этого вектора без каких-либо ограничений, при этом проверка на соответствие содержания этого атрибута и фактической принадлежность объекта некоторому классу не производится. Другими словами, можно задать любое, даже ничего не значащее, строковое значение атрибуту *class* через функцию *class*(). Следовательно, пользоваться этой возможностью следует крайне осторожно.

Заметим, что скаляр в ***R*** рассматривается как вектор длины 1. Например

```
> x<-3; x
[1] 3
```

Как мы видим, результат «пронумерован», число 1 в квадратных скобках – это и есть номер компоненты.

```
> x<-1:99; 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] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50
[51] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75
[76] 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97
98 99
```

Здесь вектор уже имеет 99 компонент. В начале строки указывается номер первой из них. Мы можем поменять внешний вид этой нумерации. Например, так:

```
> okon<-c("ый","ой","ий","ый","ый","ой","ой","ой","ый")
> x<-1:9; names(x)<-paste(x,sep="","-",okon)
> x^2
1-ый 2-ой 3-ий 4-ый 5-ый 6-ой 7-ой 8-ой 9-ый
   1    4    9   16   25   36   49   64   81
```

Здесь каждому элементу вектора присвоено имя. На печать выведен вектор x^2. Заметим, что имена указываются не только у первых элементов строки, но и у всех остальных.

Если команда имеет то же название, что и тип объекта, то она создает объект такого типа. Упрощенные варианты таких команд начинаются с символов *as*.

Вот пример из справки:

```
> mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,
+                dimnames = list(c("row1", "row2"),
+                                c("C.1", "C.2", "C.3")))
> mdat
     C.1 C.2 C.3
row1   1   2   3
row2  11  12  13
```

Команда *as.matrix*, практически не содержит аргументов, позволяющих сформировать структуру матрицы. Сравните:

```
1:5
[1] 1 2 3 4 5
> as.matrix(1:5)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
```

Команда *as.matrix*() может применяться к объектам *R*, которые уже имеют структуру. Сравните два объекта: *derevja* (таблица данных или *data.frame*) и *der.m*, тот же объект в виде матрицы.

```
> trees[1:5,]->derevja
> str(derevja)
'data.frame':   5 obs. of  3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7
 $ Height: num  70 65 63 72 81
 $ Volume: num  10.3 10.3 10.2 16.4 18.8
> as.matrix(derevja)->der.m
> str(der.m)
 num [1:5, 1:3] 8.3 8.6 8.8 10.5 10.7 70 65 63 72 81 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:5] "1" "2" "3" "4" ...
  ..$ : chr [1:3] "Girth" "Height" "Volume"
```

При выводе на печать объекты кажутся одинаковыми:

```
> derevja
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
> der.m
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
```

Однако некоторые действия с ним различаются:

```
> derevja$Height
[1] 70 65 63 72 81
> der.m$Height
Ошибка в der.m$Height : $ operator is invalid for atomic vectors
```

Еще:

```
> mean(derevja)
[1] NA
Предупреждение
In mean.default(derevja) :
  аргумент не является числовым или логическим: возвращаю NA
> mean(der.m)
[1] 30.92667
```

Впрочем, некоторые другие команды дают одинаковый результат для объектов того и другого типа:

```
> apply(derevja,2,"mean")
 Girth Height Volume
  9.38  70.20  13.20
> apply(der.m,2,"mean")
 Girth Height Volume
  9.38  70.20  13.20
```

В обоих случаях вычисляется среднее по столбцам.

Векторы

Вектор состоит из любого числа (в том числе 0) компонент одинаковой природы. Тип компоненты считается и типом вектора (а также его классом). Порождается вектор с помощью функции конкатенации *c*, например, `x <- c(1,3,5,7)`. Обращение к отдельной компоненте вектора происходит с помощью квадратных скобок, например, `x[2]` или `x[n*m]`, где числа *n* и *m* должны быть заданы заранее.

Заметим, что индекс вектора также может быть векторным, например,

```
> x<-1:10
> y<-x^2
> n<-c(1,2,4,8)
> y[n]
[1]  1  4 16 64
```

Заметьте, что можно и не водить новый объект *y*, а применять индексирование прямо к алгебраическому выражению:

```
> (x^2)[n]
[1]  1  4 16 64
```

Задание. Для чего здесь поставлены скобки? Что будет, если их опустить?

Если некоторый индекс указан со знаком «–», соответствующий элемент исключается из рассмотрения:

```
> (x<-(1:10)^2)
 [1]  1  4  9 16 25 36 49 64 81 100
> (x[c(-1,-3,-7,-5)]->y)
[1]  4 16 36 64 81 100
# еще один вариант
> b<-1:10; (b<- -b*(b%%2)) # b%%m – остаток от деления b на m
 [1] -1  0 -3  0 -5  0 -7  0 -9  0
# сгенерирован список индексов
> (x[b]->y)
[1]  4 16 36 64 100
# заметьте, что отрицательные индексы не могут смешиваться с положи-
тельными
```

Таким образом, многие операции, которые в других языках требуют использования цикла, в ***R*** выполняются одной командой.

Еще одна возможность для этого – использование логических векторов в качестве индексов. В исходном векторе отбираются только те компоненты, для которых соответствующее значение индекса равно *TRUE*.

```
> x<-1:6
> x[c(F,F,T,T,F,T)]
[1] 3 4 6
```

Логические векторы обычно порождаются при проверке некоторых условий. Пусть, например, *x* задано как вектор из случайных чисел, равномерно распределенных на интервале (0; 1). Мы хотим отобрать из них только те, которые больше 0.75.

```
> x<-runif(20)
> x
 [1] 0.71341024 0.63190557 0.85980973 0.27378609 0.36434490 0.15316421
 [7] 0.16053713 0.92437183 0.43142335 0.73190711 0.28879955 0.24795449
[13] 0.54450789 0.79982371 0.34749438 0.66645002 0.99055378 0.06823902
[19] 0.51944063 0.05652056
> y<-x[x>0.75]
> y
[1] 0.8598097 0.9243718 0.7998237 0.9905538
```

В языке ***R*** есть специфические операторы, применяемые к векторам. Например, функция *diff* вычисляет разность между соседними элементами вектора. Результатом является вектор, у которого компонент на 1 меньше:

```
> x<-c(-1,0,2,4,5,7)
> diff(x)
[1] 1 2 2 1 2
```

С помощью этого оператора можно вычислять разности не только соседних компонент, но и через 1, через 2 … Это указывается в аргументе *lag*, который по умолчанию равен 1. Кроме того, можно вычислять не только первые, но и вторые и т.п. разности: количество указывается в аргументе *differences*:

```
> x<-c(0,1,3,6,10,15,21,28)
> diff(x,lag=2)
[1] 3 5 7 9 11 13
> diff(x,diff=2)
[1] 1 1 1 1 1 1
```

Обратной операцией к *diff* является *diffinv* с теми же аргументами. В качестве исходных данных, кроме разностей, необходимо задать и начальное значение.

```
> x<-c(1,1,1,1,1,1,1,1)
> diffinv(x,lag=2,xi=c(2,1))
 [1] 2 1 3 2 4 3 5 4 6 5
> diffinv(x,diff=2,xi=c(2,1))
 [1] 2 1 1 2 4 7 11 16 22 29
```

В простейшем случае *lag* = *differences* = 1 тот же результат можно получить с помощью оператора *cumsum* (кумулятивное суммирование или суммирование с накоплением)

```
> y<-c(1,1,1,1,1,1)
> cumsum(y)->z
> z
[1] 1 2 3 4 5 6
> cumprod(z)
[1] 1 2 6 24 120 720
```

Как мы видим с накоплением можно еще и перемножать элементы вектора. Существуют также операторы *cummax*(*x*) и *cummin*(*x*):

```
> x<-c(5,3,6,4,7,3,1)
> cummin(x)
[1] 5 3 3 3 3 3 1
> cummax(x)
[1] 5 5 6 6 7 7 7
```

Важное действие над векторами – скалярное произведение. Его можно рассматривать как частный случай произведения матриц, *x* %*% *y*. Правда, результатом будет матрица размерности 1 × 1, но ее можно превратить в скаляр (вектор размерности 1) с помощью функции *drop*.

```
> x<-c(1,2,3,4); y<-c(2,3,4,0)
> x%*%y
 [,1]
[1,] 20
> drop(x%*%y)
[1] 20
```

Факторы

Фактор отчасти похож на вектор, но в нем дополнительно выделяются «уровни» (*levels*). Они играют роль «категорий», по которым можно группировать данные. Факторы бывают упорядоченные и нет (это касается уровней). Даже если тип данных фактора указывается как числовой (*numeric*), с ними нельзя производить арифметические операции.

```
> a<-c(1,2,3,4,1,2,3,4)
> (b<-factor(a))
[1] 1 2 3 4 1 2 3 4
Levels: 1 2 3 4
> mode(b)
[1] "numeric"
> b+1
[1] NA NA NA NA NA NA NA NA
       Предупреждение
       In Ops.factor(b, 1) : + не значимо для факторов
> (c<-as.vector(b))
[1] "1" "2" "3" "4" "1" "2" "3" "4"
```

Как мы видим, после обратного преобразования фактора в вектор его компоненты из числовых становятся символьными.

Для обозначения категориальных данных в ***R*** есть несколько способов, разной степени «правильности». Во-первых, можно создать текстовый (*character*) вектор:

```
 > sex <- c( "male", "female", "male", "male", "female", "male",
"male")
 > is.character(sex)
 [1] TRUE
 > is.vector(sex)
 [1] TRUE
 > str(sex)
 chr [1:7] "male" "female" "male" "male" "female" "male" ...
```

Предположим, что *sex* – это описание пола сотрудников небольшой организации. Вот как ***R*** выводит содержимое этого вектора:

```
 > sex
 [1] "male" "female" "male" "male" "female" "male" "male"
```

Можно использовать квадратные скобки для вывода первого элемента:

```
 > sex[1]
 [1] "male"
```

Например, команда *table***()** выводит:

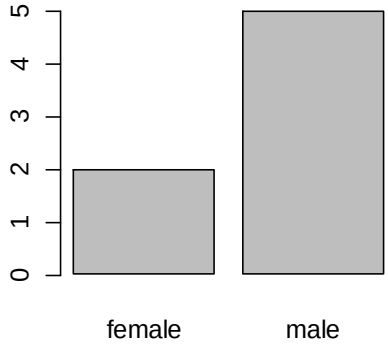
```
> table(sex)
sex
female male
2 5
```

А вот команда *plot***()**, не может сделать с таким вектором ничего вразумительного. В таких случаях пользователь сам должен проинформировать ***R***, что его надо рассматривать как категориальный тип данных. Делается это так:

```
> sex.f <- factor(sex)
> sex.f
[1] male female male male female male male
Levels: female male
```

И теперь команда *plot***()** уже понимает, что ей надо делать:

```
> plot(sex.f)
```



потому что «видит» специальный тип объекта, предназначенный для категориальных данных – фактор с двумя уровнями (*levels*):

```
> is.factor(sex.f)
[1] TRUE
> is.character(sex.f)
[1] FALSE
> str(sex.f)
Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
```

Очень многие функции ***R*** (тот же самый *plot***()**) предпочитают факторы текстовым векторам. При этом некоторые умеют конвертировать текстовые векторы в факторы, а некоторые – нет, поэтому надо быть внимательным.

Есть ещё несколько важных свойств факторов, которые надо знать заранее. Во-первых, подмножество фактора – это фактор с тем же количеством уровней, даже если их в подмножестве не осталось:

```
> sex.f[5:6]
[1] female male
Levels: female male
> sex.f[6:7]
[1] male male
Levels: female male
```

«Избавиться» от лишнего уровня можно, только применив специальный аргумент или выполнив преобразование данных «туда и обратно»:

```
> sex.f[6:7, drop=TRUE]
[1] male male
Levels: male
> factor(as.character(sex.f[6:7]))
[1] male male
Levels: male
```

Во-вторых, факторы в отличие от текстовых векторов можно легко преобразовать в числовые значения:

```
> as.numeric(sex.f)
[1] 2 1 2 2 1 2 2
```

В-третьих, факторы можно упорядочивать, превращая их некое подобие числовых данных. Введём четвёртую переменную: размер маек для тех же самых гипотетических семерых сотрудников:

```
> m <- c( "L", "S", "XL", "XXL", "S", "M", "L")
> m.f <- factor(m)
> m.f
[1] L S XL XXL S M L
Levels: L M S XL XXL
```

Как видно из примера, уровни расположены просто по алфавиту, а нам надо, чтобы *S* (*small*) шёл первым. Кроме того, надо как-то сообщить ***R***, что перед нами не просто категориальные, а упорядочиваемые категориальные данные. Делается это так:

```
> m.o <- ordered(m.f, levels=c( "S", "M", "L", "XL", "XXL"))
> m.o
[1] L S XL XXL S M L
Levels: S < M < L < XL < XXL
```

Теперь ***R*** «знает», какой размер больше.

Функции, создающие и обрабатывающие факторы: *factor*(), *as.factor*(), *gl*,

Матрицы и массивы

Матрицы – чрезвычайно распространённая форма представления данных, организованных в форме таблицы. Про матрицы в ***R***, в общем-то, нужно знать две важные вещи: во-первых, что они бывают разной размерности, и во-вторых, что матриц как таковых в ***R***, по сути, нет.

Начнем с последнего. Матрица в ***R*** – это просто специальный тип вектора, обладающий некоторыми добавочными свойствами («атрибутами»), позволяющими интерпретировать его как совокупность строк и столбцов.

Матрица – это вектор, представленный в виде двумерного массива. Следовательно, как и вектор, матрица может содержать значения только одного типа данных. Для создания матрицы можно использовать функции *matrix*() или *array*(), указав в качестве аргумента исходный вектор, а также размерность массива (для *array*) либо количество столбцов и строк (для *matrix*).

```
> b <-array(a,c(5,5,4))
> b <-matrix(a,nrow=20)
```

Массив, в отличие от вектора, имеет атрибут *dim*, содержащий целочисленные значения, задающие количество элементов для каждого измерения массива. Так, у матрицы 10 × 5 атрибут *dim* задается вектором «10, 5». Отсюда вытекает еще один способ формирования массивов – это задание атрибута *dim* некоторому вектору. Для примера создадим вектор *b* из 100 чисел, `b = 1:100`, а затем превратим его в двумерный массив (матрицу) и в трехмерный массив.

```
> attr(b,"dim")<-c(20,5)
> dim(b)
[1] 20 5
> class(b)
[1] "matrix"
 # другой вариант
> attr(b,"dim")<-c(20,5,1)
> dim(b)
```

```
[1] 20 5 1
> class(b)
[1] "array"
```

Возможна другая запись

```
> dim(b)<- c(20, 5)
```

Следует сказать, что вновь сделать из матрицы вектор можно присвоив атрибуту *dim* значение *NULL*.

```
> dim(b) <- NULL
> b
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
[23] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
[45] 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
[67] 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88
[89] 89 90 91 92 93 94 95 96 97 98 99 100
> is.vector(b)
[1] TRUE
> class(b)
[1] "integer"
```

Если же сделать массив одномерным, то фактически он тоже станет вектором, но формально будет принадлежать классу *array*. Как правило, такие массивы обрабатываются как векторы, но в некоторых случаях могут быть и исключения.

```
> dim(b) <- length(b)
> is.vector(b)
[1] FALSE
> class(b)
[1] "array"
> b
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
[23] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
[45] 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
[67] 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88
[89] 89 90 91 92 93 94 95 96 97 98 99 100
```

Для извлечения из массива определенного элемента используется такой же принцип индексации, как и в векторах, только с учетом размерности массива. Например, чтобы извлечь из матрицы 5 × 10 значение, находящееся в пятом ряду и третьем столбце следует написать *b*[5, 3].

Мы указали лишь два способа создания матриц, а в действительности их гораздо больше. Очень популярно, например, «делать» матрицы из векторов-колонок или строк при помощи команд *cbind***()** или *rbind***()**. Если результат нужно «отразить», используется команда *t***()** (транспонирование).

Иногда матрицу нужно получить в виде «декартова произведения» векторов. Точнее, в виде функции от такого произведения. Это значит, что элемент с номерами *i*, *j* вычисляется как функция $f(x_i,y_j)$. Для такого случая применяется оператор *outer* (т.е. внешнее произведение). Среди аргументов этого оператора есть не только исходные вектора *x*, *y*, но и функция, которая должна к ним применяться. Это может быть стандартная функция (в том числе бинарная операция). Либо ее нужно описать отдельно, заранее.

```
> x<-c(1,2,3,4); y<-c(2,3,4)
> z<-outer(x,y,"+")
> z
 [,1] [,2] [,3]
[1,] 3 4 5
[2,] 4 5 6
```

```
[3,] 5 6 7
[4,] 6 7 8
```

Вариант с функцией:

```
> f<-function(x,y) {x^2+y^2-1}
> z<-outer(x,y,f)
> z
 [,1] [,2] [,3]
[1,] 4 9 16
[2,] 7 12 19
[3,] 12 17 24
[4,] 19 24 31
```

Вместо *outer*(*x*,*y*,"*") можно использовать оператор *x*%*o*%*y*.

```
> x<-1:9; names(x)<-paste("(",x,")",sep="")
# формируем заголовки строк и столбцов
> x%o%x # таблица умножения
  (1) (2) (3) (4) (5) (6) (7) (8) (9)
(1)  1  2  3  4  5  6  7  8  9
(2)  2  4  6  8 10 12 14 16 18
(3)  3  6  9 12 15 18 21 24 27
(4)  4  8 12 16 20 24 28 32 36
(5)  5 10 15 20 25 30 35 40 45
(6)  6 12 18 24 30 36 42 48 54
(7)  7 14 21 28 35 42 49 56 63
(8)  8 16 24 32 40 48 56 64 72
(9)  9 18 27 36 45 54 63 72 81
```

Функция *kronecker* создает такое же произведение, но с другой размерностью. У внешнего произведения количество размерностей суммируется. Например, произведением двух матриц будет четырехмерный массив. Оператор *kronecker* создаст матрицу, но с большего размера. Проверьте это.

Задача. Построить табель-календарь на месяц, в котором 31 день. Первое число попадает на день недели номер *n*.

Р е ш е н и е .

```
> dni<-c("пн","вт","ср","чт","пт","сб","вс")
> den<-1:7; names(den)<-dni
> f<-function(x,y){7*(y-1)+x-n}
> n<-3
> outer(den,1:5,f)->z
> z
 [,1] [,2] [,3] [,4] [,5]
пн -2 5 12 19 26
вт -1 6 13 20 27
ср 0 7 14 21 28
чт 1 8 15 22 29
пт 2 9 16 23 30
сб 3 10 17 24 31
вс 4 11 18 25 32
> z[z>31]<-""; z[z<=0]<-""
> print(z,quote=F)
   [,1] [,2] [,3] [,4] [,5]
пн      5    12   19   26
вт      6    13   20   27
ср      7    14   21   28
чт 1    8    15   22   29
пт 2    9    16   23   30
```

```
сб 3    10   17   24   31
вс 4    11   18   25
```

Конец решения.

Задание. Для чего использован аргумент *quote*?

Умножение матриц обозначается как *x*%*%*y*

```
> y<-c(1,2,0,2,1,0,0,1,2)
> dim(y)<-c(3,3)
> y
 [,1] [,2] [,3]
[1,] 1 2 0
[2,] 2 1 1
[3,] 0 0 2
> z<-c(1,2,3,4,2,3,1,1,0)
> dim(z)<-c(3,3)
> z
 [,1] [,2] [,3]
[1,] 1 4 1
[2,] 2 2 1
[3,] 3 3 0
> y%*%z
 [,1] [,2] [,3]
[1,] 5 8 3
[2,] 7 13 3
[3,] 6 6 0
```

Еще некоторые операторы. Функция *diag* преобразует вектор в диагональную квадратную матрицу.

```
> diag(c(1,2),nrow=4)
   [,1] [,2] [,3] [,4]
[1,]  1  0  0  0
[2,]  0  2  0  0
[3,]  0  0  1  0
[4,]  0  0  0  2
```

Если ее применить к матрице, она создает вектор из элементов, стоящих на главной диагонали.

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

Таблицы данных (структуры)

Все варианты векторов (в том числе матрицы и массивы) имеют общее свойство: все элементы вектора должны принадлежать одному типу данных. Однако часто это неудобно, поэтому в ***R*** есть класс данных *data.frame* (таблица данных). В такой таблице каждый столбец может иметь свой тип. Кроме того, у столбцов могут быть заголовки.

```
> den<-c("пн","вт","ср","чт","пт","сб","вс")
> nach<-c(12,13,12,12,13,14,12)
> (data.frame(den=den,nachalo=nach,konec=nach+2)->raspisan)
 den nachalo konec
1 пн 12 14
2 вт 13 15
```

```
3 ср 12 14
4 чт 12 14
5 пт 13 15
6 сб 14 16
7 вс 12 14
> class(raspisan[,1]) # обращение к столбцу таблицы
[1] "factor"
> class(raspisan$nachalo) # другой способ обращения к столбцу
[1] "numeric"
> str(raspisan)
'data.frame': 7 obs. of 3 variables:
 $ den : Factor w/ 7 levels "вс","вт","пн",..: 3 2 6 7 4 5 1
 $ nachalo: num 12 13 12 12 13 14 12
 $ konec : num 14 15 14 14 15 16 14
```

Заметьте, что первая переменная структуры *raspisan* описана как фактор, хотя исходный объект den фактором не является:

```
> is.factor(den)
[1] FALSE
> is.factor(raspisan$den)
[1] TRUE
```

Сравните также два способа обращения к элементам структуры:

```
> raspisan[1] # аналогично raspisan["den"]
 den
1 пн
2 вт
3 ср
4 чт
5 пт
6 сб
7 вс
> raspisan[,1] # аналогично raspisan[,'den'] или raspisan[['den']]
[1] пн вт ср чт пт сб вс
Levels: вс вт пн пт сб ср чт
```

Можно обратиться и к элементу структуры на пересечении строки и столбца.

```
> raspisan[2,'nachalo']
[1] 13
```

Если надо вывести строку таблицы, ставим после номера запятую, сравните:

```
> raspisan[2]
 nachalo
1 12
2 13
3 12
4 12
5 13
6 14
7 12
> raspisan[2,]
 den nachalo konec
2 вт 13 15
```

Кстати, табель-календарь, полученный в предыдущем разделе, можно также превратить в объект *data.frame*:

```
> data.frame(z)
   X1 X2 X3 X4 X5
пн     5 12 19 26
```

```
вт      6 13 20 27
ср      7 14 21 28
чт  1   8 15 22 29
пт  2   9 16 23 30
сб  3  10 17 24 31
вс  4  11 18 25
```

Задание. Проверьте, к каком типу относятся столбцы этой таблицы.

Списки (перечни)

Списки – ещё один очень важный тип представления данных. Создавать их, особенно на первых порах, скорее всего, не придется, но знать их особенности необходимо. Это нужно, прежде всего, потому, что очень многие функции в ***R*** возвращают именно списки.

Список (*list*) отличается от матрицы тем, что каждая его составляющая может иметь свой тип. Кроме того, отдельные компоненты могут иметь имя. Обращение к отдельной компоненте возможно как по номеру, так и по имени.

```
> den<-c("пн","вт","ср","чт","пт","сб","вс")
> chislo<-c(NA,NA,1:31,NA,NA)
> dim(chislo)<-c(7,5)
> tabel<-list(den,chislo)
> names(tabel)<-c("дни","числа") # присваиваем имена компонентам
```

Посмотрим, какой объект у нас получился:

```
> str(tabel)
List of 2
 $ дни : chr [1:7] "пн" "вт" "ср" "чт" ...
 $ числа: int [1:7, 1:5] NA NA 1 2 3 4 5 6 7 8 ...
```

Можно применить и другой способ:

```
> tabel<-list(den=den,chis=chislo)
> str(tabel)
List of 2
 $ den : chr [1:7] "пн" "вт" "ср" "чт" ...
 $ chis: int [1:7, 1:5] NA NA 1 2 3 4 5 6 7 8 ...
```

Заметьте, что идентификатор *den* используется в команде дважды: *den=den*, причем слева от равенства это имя компоненты, а справа – переменная, созданная ранее.

Как обратиться к отдельному элементу списка или его компоненте? Сравните два способа:

```
> tabel[1]
$den
[1] "пн" "вт" "ср" "чт" "пт" "сб" "вс"

> tabel[[1]]
[1] "пн" "вт" "ср" "чт" "пт" "сб" "вс"
```

При использовании одинарных квадратных скобок результат сам будет списком (из одной компоненты). При двойных скобках результат – объект, который использовался для построения списка.

```
> class(tabel[1])
[1] "list"
> class(tabel[[1]])
[1] "character"
> class(tabel[[2]])
[1] "matrix"
```

Если компонента имеет имя, можно обратиться к ней по имени, заключив его в кавычки:

```
> tabel["den"]
$den
[1] "пн" "вт" "ср" "чт" "пт" "сб" "вс" # это список

> tabel[["den"]]
[1] "пн" "вт" "ср" "чт" "пт" "сб" "вс" # это сама компонента
```

или используя знак доллара $:

```
> tabel$den
[1] "пн" "вт" "ср" "чт" "пт" "сб" "вс" # это сама компонента
```

Кстати, имена можно придавать и элементам вектора или столбцам/строкам матриц. Однако обращаться по именам можно только к элементам списка/структуры.

Способы ввода данных

С клавиатуры

Присваиваем численное или векторное значение переменной. Эта операция является также операцией о б ъ я в л е н и я переменной. Например

```
> a <- 1
> c(1.2, 1.3, 1.7, 4) -> a
```

Вторая строка организует *a* в виде вектора с 4 компонентами. Новое использование той же переменной отменяет как старые значения, так и старый тип данных. Поэтому надо следить за тем, чтобы не потерять информацию при такой отмене. Лучше всего важные переменные обозначать несколькими буквами (а еще лучше - осмысленным словом).

Знак <– по сути является сокращением оператора присваивания:

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

Есть сокращенные способы вводить векторные данные, например, отрезок натурального ряда:

```
> b <- 1:5
```

Или равномерно расположенные числа (последовательность, *sequence*):

```
> seq (0, 1, by=0.1)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq (0, 1, len=11)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Вместо параметра *length* можно использовать *along = a*, тогда длина построенного вектора будет совпадать с длиной вектора *a*.

Задание. Проверьте, как именно называются параметры команды *seq*(): *length* или *len*? Какое написание можно использовать? Можно ли вообще не указывать имена аргументов?

Команда *rep* повторяет набор данных несколько раз. Например,

```
> rep(1:3,5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
> rep(1:3,each=5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
> rep(1:3,1:4)
Ошибка в rep(1:3, 1:4) : неправильный аргумент 'times'
# второй аргумент векторный, его длина должна совпадать с длиной x
> rep(1:3,2:4)
[1] 1 1 2 2 2 3 3 3 3
```

Команда *rev* "обращает" вектор, т.е. переписывает его компоненты от конца к началу.

```
> rev(1:5)
[1] 5 4 3 2 1
```

Чтобы проверить текущее значение объекта, просто напишите его имя в строке и нажмите *Enter*. Того же можно добиться, если оператор, задающий объект, заключить в круглые скобки:

```
> (c(1.2, 1.3, 1.7, 4) -> a)
[1] 1.2 1.3 1.7 4.0
```

Такой оператор производит как присвоение значений, так и вывод их на экран.

Задание. Проверьте, какой получится результат, если применить операцию *c*() к векторам?

```
> c(1:7, seq(1,5,2)) -> a
```

Операция *c* называется «конкатенация», однако она не совпадает с конкатенацией строк.

Если команда содержит вычисление, но не содержит присвоения, то результат просто выводится на экран, а потом теряется.

Задание 2. Как Вы думаете, совпадают ли выражения 1 : *n* – 1 и 1 : (*n* – 1). Присвойте *n* какое-нибудь значение и сравните их.

Сравните также выражения 1 : *n*^2, 1 : (*n*^2) и (1 : *n*)^2.

Из файла

Пусть данные записаны в текстовом файле, в виде таблицы. Между элементами одной строки задан разделитель (по умолчанию – пробел). В конце строки должен быть знак конца строки, другой разделитель ставить не нужно.

Вместо текстового можно использовать файл с расширением *.csv*, получаемый из *Excel*-таблицы. В этом файле в качестве разделителя предполагаются точки с запятой (даже если они не видны на экране).

Данные из такого файла можно прочитать в таблицу с помощью функции *read.table* :

```
> myfile <- read.table(file="c:/myfile.csv", header=TRUE,
sep=";",dec=",")
```

Здесь параметр *sep* показывает тип разделителя, параметр *header* – признак того, нужно ли считать первую строку заголовками. Эту запись можно сократить до *h* = *T*. Параметр *dec* указывает знак для десятичной точки. Язык ***R*** использует для этих целей точку, в то время как внешние файлы могут использовать запятую.

При указании пути *backslash* заменяются на прямые *slash* или на двойные *backslash*. Название файла записывается как строка (т.е. в кавычках).

Можно не указывать полный путь для файла. Поиск файла будет производиться в текущем каталоге. По умолчанию текущим каталогом является «Мои документы».

Если вы не знаете, какой каталог текущий, можно узнать это командой

```
> getwd()
```

Изменить текущий каталог можно с помощью меню Файл в основном меню консоли. При желании можно сделать это и с помощью командной строки:

```
> setwd("e:\\temp")
```

Можно использовать также файлы с расширением *.r*, создав их изнутри программы.

Суммируя, можно сказать, что один из возможных способов работы с данными в ***R*** такой:

- данные набирают в какой-нибудь "внешней" программе,
- записывают их в текстовый или табличный файл с разделителями (см. выше),
- загружают как объект в ***R*** и работают с ним, возможно, внося изменения,
- если были изменения, командой *write.table*() записывают обратно в файл,
- импортируют как текстовый файл в исходную программу и работают дальше, и т.д.

Такой способ несколько сложен, но позволяет использовать все преимущества программ по набору электронных таблиц и текстовых редакторов. Можно, конечно, поступить проще и вообще не использовать ничего, кроме ***R***. Тогда есть два пути сохранения объекта между сессиями: либо (1) записывать его в виде *текстового* файла (см. выше), либо (2) сохранять как *бинарный* файл ***R***. Второе требует использования либо команды *save*(), либо команды *save.image*(), причем последняя сохранит *все* объекты, накопленные Вами к текущему моменту. Учтите, что бинарный файл проще сохранить и быстрее загрузить (командой *load*()), но его нельзя прочитать из других программ, а размер его значительно больше, чем у текстового файла.

Удобно принять соглашение, по которому все текстовые файлы данных, годные для загрузки в ***R***, должны оканчиваться на **.dat*, а бинарные файлы ***R*** должны оканчиваться на **.rd*. Сразу скажем, что программы на языке ***R*** записываются в текстовые файлы, оканчивающиеся просто на **.r* (а загружаются они командой *source*()).

Порождение с помощью датчика случайных значений.

Основной объект статистики – выборка (*sample*). В *R* предоставлена возможность создавать выборки из самых разных генеральных совокупностей и случайных величин.

Команда создания выборки из произвольного набора данных так и называется *sample*(). Например,

```
> sample(1:10)
[1] 2 1 3 6 8 7 10 5 4 9
```

задает выборку из массива 1:10 с равновероятными исходами, без возвращения. В результате получим перестановку чисел 1, 2, …, 10. Вместо 1:10 можно подставить любой другой вектор, содержащий значения искомой величины.

Кроме того, можно указать *size* (размер выборки), признак возвращений *replace* (*FALSE* означает выборку без возвращения, значение по умолчанию). Можно также установить весовые коэффициенты, т.е. смоделировать произвольное дискретное распределение. Для этого есть параметр *prob*, по умолчанию равен *NULL*.

```
> week
[1] "пн" "вт" "ср" "чт" "пт" "сб" "вс"
> sample(week)
[1] "ср" "вс" "пт" "сб" "пн" "чт" "вт"
> sample(week,10)
Ошибка в sample(week, 10) :
 не могу сделать выборку бОльшую чем популяция если 'replace = FALSE'
> sample(week,10,replace=T)
 [1] "чт" "пт" "ср" "пт" "ср" "чт" "вс" "пт" "вт" "вт"
```

Для получения целочисленного вектора (т.е. вектора, значения которого будут натуральными числами) можно использовать функцию *sample.int*(*n*, *size* = *n*, *replace* = *FALSE*, *prob* = *NULL*). В описании функции указаны значения параметров по умолчанию.

```
> sample.int(n, replace = TRUE)
```

Кроме того, существуют датчики случайных чисел с разными распределениями. Некоторые из них приведены в таблице:

Название функции	*rbinom*	*runif*	*rnorm*	*rchisq*	*rf*	*rt*
Тип распределения	Биномиальное	Равномерное	Нормальные	Хи-квадрат	*F* - распределение	Распределение Стьюдента

Пример:

```
> rnorm(n) # равносильно, rnorm(n, mean = 0, sd = 1)
```

Задание. Просмотрите справку по одной из указанных функций. Заметьте, что каждая из них принадлежит «кусту» функций с аналогичными названиями, например,

dnorm(*x*, *mean* = 0, *sd* = 1, *log* = *FALSE*)

pnorm(*q*, *mean* = 0, *sd* = 1, *lower.tail* = *TRUE*, *log.p* = *FALSE*)

qnorm(*p*, *mean* = 0, *sd* = 1, *lower.tail* = *TRUE*, *log.p* = *FALSE*)

rnorm(*n*, *mean* = 0, *sd* = 1)

Для чего предназначена каждая из команд?

Вычисления в R

В языке ***R*** существуют операторы для арифметических и логических вычислений, а также действий со строками. Их можно применять не только к скалярам (которых в чистом виде в ***R*** и не существует), но и к векторам. При этом унарные операции (функции) применяются к каждой компоненте. Бинарные операции применяются к соответственным компонентам двух векторов. Если их длины не совпадают, то меньший вектор повторяется циклически.

Правила порождения атрибутов результата довольно сложны. Большинство из них берутся из более длинного аргумента, а если длины равны – из первого. Имена берутся из того аргумента, длина которого совпадает с длиной результата (предпочтительно - первого). Для массивов (и операций, результаты которых являются массивами) размерности и имена строк/столбцов берутся из первого аргумента, если это массив, в противном случае – из второго.

Арифметические операции

Эти операции применяются к числовым (*numeric*) или комплексным векторам, или объектам, которые могут быть преобразованы (? *coerced*) в них. Обычно результат является вектором той же длины. Если длины векторов-аргументов не совпадают, более короткий из них повторяется циклически.

```
> x<-1:10; y<-(1:3)^2
> 10*x+y
 [1] 11 24 39 41 54 69 71 84 99 101
Предупреждение
In x + y :
 длина большего объекта не является произведением длины меньшего объ-
екта
```

Как мы видим, операция была совершена, хотя длина вектора *x* не равна и даже не кратна длине *y*.

В качестве арифметических операций используются следующие: *x* + *y*, *x* – *y*, *x* * *y*, *x* / *y*, *x* ^ *y*, *x* %% *y*, *x* %/% *y*. Последние два оператора обозначают остаток от деления *x* на *y* и деление нацело. Они могут применяться и к нецелым числам, но результат может быть неожиданным. Например, 1 %/% 0.2 может оказаться равным 4.

```
> x<-10^10; x^2%%3
[1] 1
Предупреждение
возможна полная потеря точности остатка от деления
```

Правила возведения в степень не совсем совпадают с принятыми в математике. Например, 1^0 = 1, 0^0 = 1, 1^*Inf* = 1, 2^*Inf* = *Inf*, 2^+*Inf* = *Inf*, 2^-*Inf* = 0. Сложности могут возникнуть при возведении в степень отрицательных чисел. Например, (–8)^1 = –8, (–8)^2 = 64, но (–8)^(1/3) = *NaN*.

Кроме бинарных арифметических операций существуют и стандартные функции, объединенные в группы

Math

"*abs*", "*sign*", "*sqrt*",

"*ceiling*", "*floor*", "*trunc*" (это различные способы округления),

"*cummax*", "*cummin*", "*cumprod*", "*cumsum*" (действия «с накоплением»),

"*exp*", "*expm*1", "*log*", "*log*10", "*log*2", "*log*1*p*", "*cos*", "*cosh*", "*sin*", "*sinh*", "*tan*", "*tanh*", "*acos*", "*acosh*", "*asin*", "*asinh*", "*atan*", "*atanh*", …

*Math*2
"*round*", "*signif*" (это также способы округления),
Summary
"*max*", "*min*", "*range*", "*prod*", "*sum*",
"*any*", "*all*" (применяются к логическим векторам)
Complex
"*Arg*", "*Mod*", "*Im*", "*Re*", "*Conj*" (сопряженное)
И другие, которые можно просмотреть в группе *Special*.

Операции со строками

Для работы со строками используются команды *paste*, *substr*, *nchar, grep, sub* и другие. Для объединения двух строк в одну (конкатенации) используется команда *paste*().

```
> n<-5
> paste("В строке",n,"знаков")->a
> a
[1] "В строке 5 знаков"
```

Команда *substr* извлекает часть строки, заданную ее местоположением:

```
> substr(a,2,5)
[1] " стр"
> substr(a,3,7)<-"столбец"
> a
[1] "В столбе 5 знаков"
```

Как вы видите, подстроку можно не только «извлекать» из строки, но и заменять ее новыми значениями.

При использовании команды *paste*() между объединяемыми строками вставляется пробел (по умолчанию). Его можно заменить и другим знаком с помощью параметра *sep*.

Если аргумент является вектором, то команда применяется к каждой его компоненте:

```
> paste("a",c(1,2),sep="")
[1] "a1" "a2"
> paste("a",1,2)
[1] "a 1 2"
> paste(c(1,2))
[1] "1" "2"
```

Заметьте, что *paste*() с одним аргументом равносилен оператору *as.character*().

Если же надо объединить компоненты вектора в одну строку, нужно использовать параметр *collapse*. Он указывает также, какой символ ставится между компонентами:

```
a<-1:8
> a
[1] 1 2 3 4 5 6 7 8
> paste(a)
[1] "1" "2" "3" "4" "5" "6" "7" "8"
> paste(a,collapse="")
[1] "12345678"
> paste(a,collapse=",")
[1] "1,2,3,4,5,6,7,8"
```

Если *collapse=NULL*, то элементы вектора не объединяются в одну строку:

```
> paste(a,collapse=NULL)
[1] "1" "2" "3" "4" "5" "6" "7" "8"
#результат такой же, как просто paste
```

Оператор *substr* может как выделить подстроку, так и заменить ее чем-нибудь. Аргументы – строка, *start*, *stop*. Последние два указывают на первый и последний номер элементов подстроки. Эти аргументы могут быть векторными.

```
> dni<-c("понедельник","вторник","среда","четверг","пятница",
+"суббота","воскресенье")
> substr(dni,1,c(3,2,2,3,3,3,3) )
[1] "пон" "вт" "ср" "чет" "пят" "суб" "вос"
```

Для замены подстроки используется оператор присвоения. При этом длина строки символов не меняется: если на место подстроки длиной 5 вы присвоите строку длиной 1, то все 4 остальных символа останутся незамененными.

```
> sun<-"воскресенье"
> substr(sun,2,nchar(sun)-1)<-"-"
> sun
[1] "в-скресенье"
```

Здесь использован оператор *nchar*(), который задает число символов в строке.

Функция *strsplit*() разбивает строку на подстроки. Простейший случай – на отдельные буквы.

```
> strsplit(sun,NULL)->s
> s
[[1]]
 [1] "в" "о" "с" "к" "р" "е" "с" "е" "н" "ь" "е"
```

Результатом является, как вы видите, список. Операция имеет много параметров. Например, параметр *split* (который в предыдущем примере приравнен *NULL*) указывает подстроку, которая удаляется из строки и на месте которой происходит разбиение:

```
> strsplit(sun,"е")->s
> s
[[1]]
[1] "воскр" "с" "нь"
```

Вообще говоря, аргумент *split* является маской, задаваемой как регулярной выражение (*regular expression*). Сравните два результата:

```
> x<-"1.1"
> strsplit(x,".")
[[1]]
[1] "" "" ""
> strsplit(x,".",fixed=TRUE)
[[1]]
[1] "1" "1"
```

Пример применения различных функций:

```
> strReverse <- function(x)
+ sapply(lapply(strsplit(x, NULL), rev), paste, collapse="")
> strReverse(c("abc", "Statistics"))
[1] "cba" "scitsitatS"
```

Как мы видим, функция сначала разбивает строку на отдельные символы, полученный вектор переставляет в обратном порядке (команда *rev*()) и снова собирает в строку.

Логические операции

Логические величины получаются как результат логических операций <, <=, >, >=, == для точного равенства и != для не-равенства. Кроме того, если *c*1 и *c*2 являются логическими выражениями, тогда *c*1 & *c*2 является их умножением ("и"), *c*1 | *c*2 их сложением ("или") и !*c*1 является отрицанием *c*1.

Можно также применять кванторы всеобщности и существования, *all*() и *any*(). Они задают логическое умножение/сложение всех элементов вектора.

```
> x<-runif(10,-1,1)
> x
 [1] -0.2156970 -0.3799545 -0.5039355 -0.5556247 0.3971427 0.6049843
```

```
 [7] -0.7325494 -0.1530170 0.3604098 0.7171559
> x<0
 [1] TRUE TRUE TRUE TRUE FALSE FALSE TRUE TRUE FALSE FALSE
> any(x<0)
[1] TRUE
> all(x<0)
[1] FALSE
```

Вместо слов "*TRUE*" и "*FALSE*" можно получить номера тех компонент вектора, для которых выполняется условие. Это делается с помощью команды *which*().

```
> x<-rnorm(10,1)
> x
 [1] 2.0404450 0.4099973 1.5408356 0.6081870 1.8492207 1.0676334
 [7] 1.1600879 1.1812413 1.1590843 2.1752097
> which(x<1)
[1] 2 4
> x[which(x<1)]
[1] 0.4099973 0.6081870
> x[x<1]
[1] 0.4099973 0.6081870
```

Как мы видим, здесь результаты команд *x*[*which*(*x*<1)] и *x*[*x*<1] совпадают.

```
> x<-c(1,2,NA,4,5)
> x[x<4]
[1] 1 2 NA
> x[which(x<4)]
[1] 1 2
```

Разница заметна, если вектор содержит пропущенные значения.

Если нужно подсчитать число значений, удовлетворяющих заданному условию, можно использовать функцию *sum*. Она просуммирует логический вектор, при этом *TRUE* заменится на 1, а *FALSE* – на 0, так что сумма как раз будет равна числу значений *TRUE*.

```
> (x<-sample(1:20,10))
[1] 5 6 13 7 17 19 12 15 20 14
> sum(x%%2 == 1) # ищем число нечетных значений
[1] 6
```

Операции с множествами

В качестве множества можно рассматривать набор компонент вектора. Для множеств *x*, *y* определены операции объединения *union*(*x*, *y*), пересечения *intersect*(*x*, *y*), разности *setdiff*(*x*, *y*) и проверка равенства *setequal*(*x*, *y*). Кроме того, можно проверить принадлежность элемента множеству операцией *%in%*. Вместо этого оператора можно использовать также *is.element*(*el*, *set*). Функция *match* не только проверяет вхождение элемента в множество, но и указывает (первую) позицию такого вхождения.

Заметьте, что перед проверкой все значения приводятся к символьному виду:

```
x<-1:5
> c(1,6) %in% x
[1] TRUE FALSE
> c("1","6") %in% x
[1] TRUE FALSE
```

Некоторые более сложные объекты (например, таблицы) не рассматриваются как множества своих значений.

Задание. Найдите и изучите способы обработки дат в ***R***.

Для вычисления основных статистических параметров существуют функции *mean*() (среднее), *sd*() (стандартное отклонение), *var*() (дисперсия), *median*() (медиана), *quantile*() (квантиль), *cor*() (корреляция) и т.п. Можно объединить основные характеристики вектора командой *summary*().

```
> summary(rasp[[2]])
 Min. 1st Qu. Median Mean 3rd Qu. Max.
 12.00 12.00 12.00 12.57 13.00 14.00
```

Результатом применения этого оператора является таблица (*table*) с 6 именованными компонентами. Если присвоить значение *summary* переменной *x*, то к отдельным ее компонентам можно обратиться так: *x*["*Min*."], тот же результат даст обращение *x*[1].

Команда *summary* имеет много модификаций (методов) в зависимости от класса объекта, к которому его применяют. Для перечисления этих методов надо сделать соответствующий запрос системе, *methods*(*summary*)

Основные характеристики можно применять как к векторам, так и к матрицам (получаем единственное значение!). Чтобы усреднить или просуммировать матрицу по столбцам (строкам), нужно применять другие операторы: *colSums*(), *rowSums*(), *colMeans*(), *rowMeans*() (не забудьте прописные буквы).

Другой результат получается, если применить *mean*() к таблице данных. Применим его к уже известной нам таблице данных *rasp*:

```
> mean(rasp)
[1] NA 12.57143 14.57143
Предупреждение
In mean.default(X[[1L]], ...) :
 аргумент не является числовым или логическим: возвращаю NA
```

Кстати, функция *sum*() и в этом случае даст единственное значение (сумму данных во всей таблице). Она применима только к чисто числовым таблицам данных.

Задание 1. Как воздействуют на таблицу с данными смешанного типа функции *mean*() и *median*()?

Задание 2. Изучите по справочной системе функцию *rowsum*() (суммирование по группам).

Заметим, что для такой характеристики, как мода, в *R* нет специальной команды: функции *mod*() и *mode*() вычисляют нечто другое (а что именно?). Но такую функцию можно построить самостоятельно. Например, так: `x[which(table(x)==max(table(x)))]`.

Покажем, как она работает. Пусть *x* – некоторый вектор.

```
> x
 [1] 3 3 6 4 5 3 1 3 1 3 5 7
> table(x)
x
1 3 4 5 6 7
2 5 1 2 1 1
> which(table(x)== max(table(x)))
3
2
> x[which(table(x)== max(table(x)))]
[1] 3
```

Функция *table*() подсчитывает число вхождений каждого значения. Функция *which*() находит порядковый номер нужного значения (в данном случае – максимума), после чего мы можем вывести на печать именно это значение.

Задание. Что является результатом (вычисляемым объектом) команды *which*()?

Более широкие возможности дают методы группы *apply*. В качестве аргументов этой команды выбираются обрабатываемое измерение (например, 1 – строки, 2 – столбцы) и тип применяемой функции, "*sum*", "*mean*" и т.п.

```
> (aa<-rasp[-1]) # убираем первый (символьный) столбец
 NA NA
1 12 14
2 13 15
3 12 14
4 12 14
5 13 15
6 14 16
7 12 14
> mean(aa)
[1] 12.57143 14.57143
> apply(aa,2,"mean")
[1] 12.57143 14.57143 # то же, что и mean
> apply(aa,1,"mean")
[1] 13 14 13 13 14 15 13
```

У этого метода есть много модификаций, например, *lapply*() (результат – *list*, т.е. список). Метод *sapply*() дает результат любого типа. Аналогичный ему *vapply*() дает результат определенного типа, соответствующего самим исходным данным. Более сложным является метод *tapply*(), который позволяет перед вычислением произвольно группировать данные. Порядок группировки задается вторым параметром *INDEX*.

```
> grup<-rep(c("a","b","c"),5)
> grup
 [1] "a" "b" "c" "a" "b" "c" "a" "b" "c" "a" "b" "c" "a" "b" "c"
> tapply(10:24,grup,sum)
 a b c
80 85 90
```

В данном случае функция *sum* применяется к элементам, помеченным одной буквой. То есть элементы суммируются через два на третий: 10 + 13 + 16 + 19 + 22 = 80 и т.д. Соответственно, получаем три значения, которые помечены теми же «именами» *a*, *b*, *c*.

Можно применить произвольную функцию к отдельным строкам таблицы:

```
> (data.frame(x=1:6,y=11:16)->xy)
  x  y
1 1 11
2 2 12
3 3 13
4 4 14
5 5 15
6 6 16
> aggregate(xy,by=list(rep(1:2,3)),FUN=mean)
  Group.1 x  y
1       1 3 13
2       2 4 14
```

В данном случае усреднение происходит отдельно для 1, 3 и 5-ой строк, и отдельно – для 2, 4 и 6-ой. Потому что в качестве шаблона используется вектор

```
> rep(1:2,3)
```

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

Обработка пропущенных данных

В реальных данных часто бывают пропущенные значения. Они обозначаются символом *NA*. При обработке таких данных результат также будет *NA*. Кроме того, неверные данные *NaN* могут возникнуть как результат применения некоторых операций. Поэтому при обработке их надо пропустить или чем-нибудь заменить. Самый простой способ – оставить только существующие данные. Для этого можно применить проверку *is.na*(). Например, так:

```
> k<-sqrt(-2:4)
Предупреждение
In sqrt(-2:4) : созданы NaN
> k
[1]       NaN       NaN 0.000000 1.000000 1.414214 1.732051 2.000000
> sum(k)
[1] NaN
> sum(k[!is.na(k)])
[1] 6.146264
```

В последней строчке команда *sum*() применяется только к «правильным» значеиям вектора *k*. Так же можно исключить строки, содержащие пропущенные или неверные значения в таблице:

```
> cbind(-2:4,k)->nk
> nk
                 k
[1,] -2       NaN
[2,] -1       NaN
[3,]  0 0.000000
[4,]  1 1.000000
[5,]  2 1.414214
[6,]  3 1.732051
[7,]  4 2.000000
> nk[!is.na(k),]
                k
[1,] 0 0.000000
[2,] 1 1.000000
[3,] 2 1.414214
[4,] 3 1.732051
[5,] 4 2.000000
```

Однако такой способ неудобен, если неверные данные есть в разных столбцах. Поэтому созданы другие команды и аргументы команд. Например, можно указать параметр *na.rm* = *TRUE* (*remove NA*).

```
> apply(nk,2,mean)
      k
  1 NaN
> apply(nk,2,mean,na.rm=TRUE)
                  k
1.000000 1.229253
```

Заметьте, что число 1 получено усреднением по всем строкам, а 1,229253 – только по пяти последним. Это не всегда удобно. В некоторых задачах нужно применять операции к одним и тем же строкам. Можно исключить все строки, в которых есть хотя бы одно пропущенное значение. Для этого служит функция *na.omit*().

```
> na.omit(nk)
                k
```

```
[1,] 0 0.000000
[2,] 1 1.000000
[3,] 2 1.414214
[4,] 3 1.732051
[5,] 4 2.000000
attr(,"na.action")
[1] 1 2
attr(,"class")
[1] "omit"
> apply(na.omit(nk),2,mean)
                  k
2.000000 1.229253
```

Как мы видим, значение среднего по первому столбцу изменилось.

Преобразование объектов

Для преобразования объектов в другой тип существуют команды вида *vector*(), *factor*(), *list*() и т.п., каждая из которых имеет дополнительные аргументы. Сокращенный вариант такой команды называется *as.vector*(), *as.factor*(), *as.list*()… , он не содержит дополнительных аргументов.

Нужно помнить, что не всякое преобразование объектов в другой тип допустимо, т.к. они имеют разные требования к составу компонент.

Если вектор или фактор содержит повторяющиеся элементы, их можно «убрать» с помощью команды *unique*()

```
> x
[1] 1 3 2 3 4
> unique(x)
[1] 1 3 2 4
> f
 [1] e a c a a e e b a e
Levels: a b c e
> unique(f)
[1] e a c b
Levels: a b c e
```

Объединение данных

Если данные из разных объектов нужно объединить в один, можно использовать команды *cbind*() для столбцов и *rbind*() для строк. При объединении векторов результат будет матрицей:

```
> cbind(2:7,letters[1:6])
     [,1] [,2]
[1,] "2"  "a"
[2,] "3"  "b"
[3,] "4"  "c"
[4,] "5"  "d"
[5,] "6"  "e"
[6,] "7"  "f"
```

Как мы видим, результат имеет тип *char* хотя первый столбец был числовым. Если же хотя бы один объект является таблицей (*data.frame*), то и результат будет таблицей:

```
> cbind(data.frame(2:7),letters[1:6])-> dl
> dl
  X2.7 letters[1:6]
1    2            a
2    3            b
```

```
3     4            c
4     5            d
5     6            e
6     7            f
> str(dl)
'data.frame':    6 obs. of  2 variables:
 $ X2.7        : int  2 3 4 5 6 7
 $ letters[1:6]: Factor w/ 6 levels "a","b","c","d",..: 1 2 3 4 5 6
```

В этом случае тип данных в столбце сохранился.

Иногда нужно объединять таблицы, в которых часть столбцов повторяется, но в результат повторяющиеся столбцы включать не надо. Для этого используется команда *merge*()

```
> merge(dl,dl, by="X2.7")
  X2.7 letters[1:6].x letters[1:6].y
1    2              a              a
2    3              b              b
3    4              c              c
4    5              d              d
5    6              e              e
6    7              f              f
```

В качестве общих столбцов в параметре *by* можно указать вектор из имен столбцов.

Транспонирование матриц и таблиц производится командой *t*().

```
> t(dl)
              [,1] [,2] [,3] [,4] [,5] [,6]
X2.7          "2"  "3"  "4"  "5"  "6"  "7"
letters[1:6] "a"  "b"  "c"  "d"  "e"  "f"
> str(t(dl))
 chr [1:2, 1:6] "2" "a" "3" "b" "4" "c" "5" "d" "6" "e" "7" "f"
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:2] "X2.7" "letters[1:6]"
  ..$ : NULL
```

Как мы видим, результат является не таблицей, а матрицей.

Сортировка

Для сортировки данных используются разные команды. Например, команда *sort*() может упорядочить вектор (как по убыванию, так и по возрастанию).

```
> x<-strsplit("представление",split=NULL)
> x
[[1]]
 [1] "п" "р" "е" "д" "с" "т" "а" "в" "л" "е" "н" "и" "е"
> sort(x)
Ошибка в sort.int(x, na.last = na.last, decreasing = decreasing, ...) :
  'x' должен быть элементарным
> sort(x[[1]])
 [1] "а" "в" "д" "е" "е" "е" "и" "л" "н" "п" "р" "с" "т"
```

Как мы видим, этот оператор применим только к векторам. По умолчанию упорядочение производится по возрастанию (в данном случае – по алфавиту).

```
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> sort(m)
[1] 1 2 3 4 5 6
```

Если сортируется матрица, она воспринимается как вектор и теряет свою структуру.

Можно применить операцию *sort*() и к факторам. Например, вспомним созданную ранее таблицу данных (*data.frame*) *rasp*. У такого объекта любой символьный столбец воспринимается как фактор.

```
> sort(rasp[[1]]) # Для чего здесь двойные прямые скобки?
[1] вс вт пн пт сб ср чт
Levels: вс вт пн пт сб ср чт
```

Чтобы отсортировать матрицу или таблицу по отдельному столбцу/строке необходимо использовать другой оператор, *order*(). Если применить ее к вектору, она выдает номера компонент в порядке возрастания/убывания. Если потом использовать эти номера в качестве индекса, элементы объекта будут переставлены по порядку.

```
> rasp
 NA NA NA
1 пн 12 14
2 вт 13 15
3 ср 12 14
4 чт 12 14
5 пт 13 15
6 сб 14 16
7 вс 12 14
> (i<-order(rasp[1]))
[1] 7 2 1 5 6 3 4
> rasp[i,]
 NA NA NA
7 вс 12 14
2 вт 13 15
1 пн 12 14
5 пт 13 15
6 сб 14 16
3 ср 12 14
4 чт 12 14
```

Приведенный набор команд упорядочивает таблицу по возрастанию первого аргумента (в данном случае – по алфавиту).

Как вы видите, этот оператор задает не номера компонент, а их сами. Но переставляются те же компоненты, которые упорядочиваются.

Для случая повторяющихся данных можно вместо *order*() использовать команду *rank*(), которая присваивает компонентам ранги (возможно, дробные: если число 3 стоит на 3 и 4 местах, оба вхождения получат ранг 3,5).

Существует и другие сортирующие команды, например, *sort.list*().

Группировка данных

Команду *hist*() можно использовать для группировки элементов вектора. Она сыграет такую же роль, как команда *table*() для факторов.

```
> h.x$counts
[1]  2 13 40 27 16  1  1
> h.x$breaks
[1] -3 -2 -1  0  1  2  3  4
```

Эти данные можно использовать в методах, для которых нужна предварительная группировка значений, например, хи-квадрат.

Впрочем для группировки значений есть и специальная команда *cut*().

```
> table(cut(x,7))
```

```
(-2.91,-2.04] (-2.04,-1.18] (-1.18,-0.31] (-0.31,0.555] (0.555,1.42] (1.42,2.28] (2.28,3.16]
            2            10            30            32            15            10             1
```

Сама команда *cut*() выдает в качестве результата фактор, элементами которого являются промежутки, к которым относится то или иное значение. Вот несколько первых значений:

```
> head(cut(x,7))
[1] (-1.18,-0.31] (1.42,2.28] (-1.18,-0.31] (-0.31,0.555] (-0.31,0.555] (-0.31,0.555]
Levels: (-2.91,-2.04] (-2.04,-1.18] (-1.18,-0.31] (-0.31,0.555] (0.555,1.42] (1.42,2.28]
(2.28,3.16]
```

Как мы видим, команда сама подбирает границы промежутков, на которые разбивается множество значений вектора. Однако мы можем задать не только их количество, но и границы:

```
> table(cut(x,-3:4))
(-3,-2] (-2,-1]  (-1,0]   (0,1]   (1,2]   (2,3]   (3,4]
      2      13      40      27      16       1       1
```

Результат получился тот же, что и с помощью команды *hist*(), но с б'ольшими усилиями (самостоятельным подбором границ).

С помощью команды *cut*() можно также построить таблицу сопряженности признаков, заданных числовыми значениями.

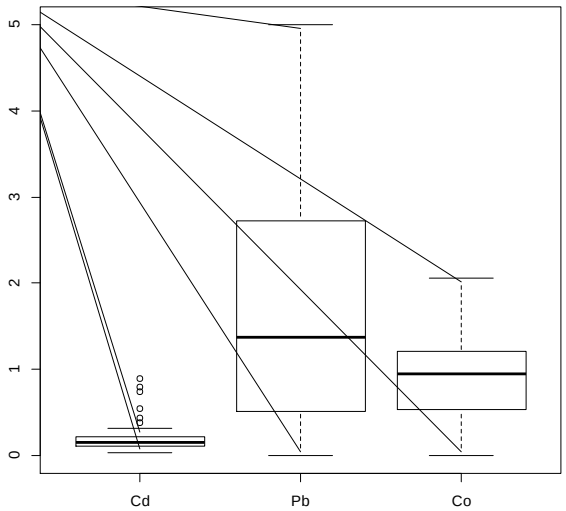
```
> x<-rnorm(50); y<-x+rnorm(50)
> table(cut(x,4),cut(y,3))
                 (-2.15,-0.458] (-0.458,1.23] (1.23,2.92]
  (-1.75,-0.908]              6             1           0
  (-0.908,-0.069]             6             7           0
  (-0.069,0.77]               4            15           3
  (0.77,1.61]                 0             2           6
```

Визуализация данных

Графические команды мы разберем далее в специальном разделе. Здесь же приведем только самые простые из них.

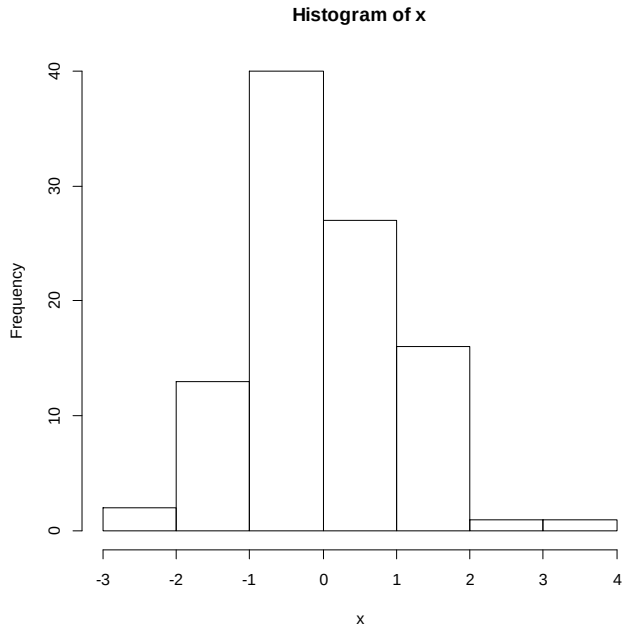
Сводку основных данных можно вывести в наглядном, графическом виде командой *boxplot*(). Он строит так называемые «ящики с усами».

```
> boxplot(dann[3:5])
```



Здесь каждый прямоугольник соответствует одному показателю. Ширина прямоугольника не имеет значения. Жирная линия показывает положение медианы, границы прямоугольника – квартили. «Усы» указывают минимум и максимум, если те превосходят не более, чем в 1,5 раза соответствующие квартили. Значения, не поместившиеся в эти пределы (выбросы) показаны отдельными точками.

Если набор данных рассматривается как выборка из некоторой с.в., то для наглядного представления о ее распределении можно использовать гистограмму. Для построения используется команда *hist*(). В ней есть много параметров, однако большинство из них можно не указывать, так как они рассчитываются по умолчанию, и вполне разумно.




```
> x<-rnorm(100)
> hist(x)-> h.x
```

После появления графического окна управление передается на него. Поэтому, чтобы вернуться в основную консоль, надо по ней щелкнуть мышкой.

Посмотрим, какой объект возвращает команда *hist*()

```
> str(h.x)
List of 6
 $ breaks  : num [1:8] -3 -2 -1 0 1 2 3 4
 $ counts  : int [1:7] 2 13 40 27 16 1 1
 $ density : num [1:7] 0.02 0.13 0.4 0.27 0.16 0.01 0.01
 $ mids    : num [1:7] -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
 $ xname   : chr "x"
 $ equidist: logi TRUE
 - attr(*, "class")= chr "histogram"
```

Как мы видим, это список из нескольких компонент:

breaks – точки деления (границы групп);

counts – число значений, попавших в каждую группу;

density – доля значений, попавших в каждую группу;

mids – середины промежутков;

equidist – логическая переменная, указывающая, является ли разбиение равномерным.

В нашем случае при вызове команды *hist*() не указывались никакие дополнительные параметры. Однако их можно задать. Например, самостоятельно выбрать точки разбиения *breaks* или хотя бы их количество. Есть также параметры, связанные с оформлением картинки (заголовки, метки осей и т.п.) Кроме того, аргумент *plot* указывает, следует ли вообще выводить гистограмму, или только рассчитать соответствующий объект.

Сохранение данных и скриптов

Вывод в файл данных и результатов расчетов

Для вывода символьных данных (на экран, в файл) служат операторы *cat*, *print*, *sprintf* и т.п. При выводе можно управлять переходом на новую строку. Например, \n, новая строка, \t, табуляция и \b, забой.

```
> cat("Первая строка","\n","Вторая строка","\t","Третья строка","\b")
Первая строка
 Вторая строка Третья строка>
```

Команда *print*(), в отличие от *cat*() выводит только один объект. Каждая команда *print*() заканчивает вывод концом абзаца, об этом можно не заботиться, как в случае с *cat*(). Если нужно напечатать данные из разных переменных в одной строке, их сначала можно объединить в одну строку. Например, с помощью команды *paste*().

Команда *print*() по умолчанию выводит строки в кавычках, но этим можно управлять с помощью параметра *quote*. При необходимости можно убрать кавычки из символьных строк параметром *quote* = *FALSE*.

Результаты вычислений обычно выводятся на экран, однако желательно иметь их и на диске. Для этого надо перенаправить вывод в файл. Это делается командой *sink*(), в которой указывается имя файла. При установке *R* каталогом по умолчанию является «Мои документы». В других случаях каталогом для вывода является текущая папка, из которой вы вызвали консоль. Вы можете во время сеанса работы поменять его. Кроме того, можно явно указать путь к файлу, если каталог не совпадает с текущим.

Пусть, например, вы создали в папке «Мои документы» папку «***R***». Тогда ссылку на файл можно записать в виде

```
> sink("R/rez.txt") # обратите внимание на slash.
```

Можно установить и другие расширения, кроме *.txt* (например, *.r*), все равно файл получится текстовым.

Чтобы вернуться к выводу на экран, запишите команду *sink*() без параметров. Если теперь снова назначить файл *rez.txt* для вывода, старая информация сотрется и будет записываться новая. Если же надо добавить информацию к существующей, придайте параметру *append* команды *sink*() значение *TRUE*.

Некоторые операторы сами управляют направлением вывода. Например, в оператор *cat*() можно включить название (путь) файла. А оператор *write*() требует этого аргумента обязательно. Эту возможность можно использовать, если вывод идет в разные файлы.

```
> write(dd,file="f.d",ncolumns=4) # сохраняет содержание матрицы dd в
файле f.d в 4 колонках
> write(dd,file="f.d")            # сохраняет содержание матрицы dd в
файле f.d в 5 колонках (по умолчанию)
```

В этой команде также можно использовать параметр *append*. В качестве разделителя по умолчанию используется пробел. Если записать *sep*="\t", то разделителем станет знак табуляции.

Заметьте, что при выводе объект рассматривается как вектор, причем его элементы располагаются «по вертикали». Чтобы повторить структуру матрицы при выводе, нужно не только указать число столбцов, но и транспонировать матрицу.

```
> y<-1:30;dim(y)<-c(10,3)
> y
 [,1] [,2] [,3]
 [1,]  1  11  21
```

```
 [2,]  2 12 22
 [3,]  3 13 23
 [4,]  4 14 24
 [5,]  5 15 25
 [6,]  6 16 26
 [7,]  7 17 27
 [8,]  8 18 28
 [9,]  9 19 29
[10,] 10 20 30
> write(y,file="tab.txt",ncol=ncol(y))
```

В файле будет такая запись:

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

Чтобы получить «правильный» результат, можно сделать так:

```
> write(t(y),file="tab.txt",ncol=ncol(y))
```

Для таблиц данных этот оператор не подходит. В этом случае удобнее использовать *print*().

В отличие от команды *write*(), которая выводит данные из векторов (таблиц) цепочкой в 5 (по умолчанию) колонок, *print*() оформляет вывод в соответствии с классом объекта.

Задание 1. Зададим вектора *x* и *y* (позаботьтесь, чтобы не все компоненты были целыми). Сравните разные способы вывода: 1) оператором *write*(); 2) оператором *print*() (в обоих случаях объедините вектора командой *paste*()); 3) командой *print*(), объединив вектора командой *data.frame*().

Вывод других данных

В файл можно выводить и другую информацию, например, пользовательскую функцию. Пусть вы создали функцию *f* и хотите сохранить ее. Напишите команду *dput*(*f* ,"Функция.*txt*"). Она создаст в текущем каталоге текстовый файл Функция.*txt* с таким, например, содержанием:

function (*x*, *y*)
{
 x * *log*(*x*^2 + *y*^2)
}

Так же можно сохранить информацию об объекте:

```
> y<-1:30; dim(y)<-c(5,6)
> dput(y,"Данные.txt")
```

В файле Данные появится запись *structure*(1:30, .*Dim* = 5:6). Если вы хотите, чтобы в файле сохранилось и имя переменной, используйте оператор *dump*:

```
> dump("y",file="Данные.txt")
```

Заметьте, что имя переменной записано в кавычках. Результатом будет запись в файле вида

y <-

structure(1:30, *.Dim* = 5:6)

Параметр *append* во всех случаях показывает, следует ли добавлять информацию (значение *TRUE*) или перезаписывать файл (*append* = *F*).

Как ни странно, количество столбцов в "печатном виде" таблиц зависит от ширины окна, причем эта же зависимость сохраняется и при выводе в файл!

Задание 2. Выполните следующий набор команд:

```
> dann<-seq(0,1,len=30);dim(dann)<-c(3,10)
> dann
> sink("proba.txt",append=T)
> print(dann)
> print(dann)
> sink()
```

Перед вторым применением оператора *print*() раскройте окно консоли на максимальную ширину. Сравните результаты записи в файл обоими способами (файл *proba.txt* окажется в текущем каталоге).

***Задание* 3**. Исследуйте с помощью справки и эксперимента оператор *write*.

Сохранение кодов программ (скриптов)

Команды, записанные на экране, существуют в момент их записи. При выходе из текущей сессии работы с ***R*** выводится запрос, следует ли сохранить рабочее пространство? Если да, сохраняется список введенных команд и набор созданных объектов. Они находятся в файлах *.Rhistory* и *.Rdata* текущего каталога. Заметьте, что эти файлы имеют только расширение, без имени. Сохраненные команды приписываются к предыдущим.

При желании можно самостоятельно придать имя сохраненной истории команд. Для этого существует команда *savehistory*(). С ее помощью все текущие команды сохраняются в файл. После этого получившийся файл можно редактировать в любом текстовом редакторе, убирая ненужные и ошибочные команды. Теперь, если Вы хотите повторить применение команд, можно сделать так:

```
> source("myscript.r", echo=T)
```

Впрочем, можно воспользоваться копкой «Файл» в меню и выбрать нужный программный код из подходящего каталога.

Если Ваши команды записаны в файл *myscript.r*, то все они будут выполнены (учтите, что любая ошибка прервет исполнение). Естественно, если Вы создавали какие-либо графические файлы, то они будут созданы заново.

(!) *Следите за открытыми направлениями вывода текстовой или графической информации! Если действие сценария прервалось до того, как была выполнена команда sink(), то файл вывода окажется незакрытым. При этом весь последующий вывод попадет не на экран, а в этот файл. Необходимо закрыть его из командной строки, применив оператор sink(). Возможно, это придется делать несколько раз, пока программа не укажет, что больше открытых файлов нет:*

```
> sink()
Предупреждение
In sink() : нет 'sink' для удаления
```

Заметьте, что вывод результатов вычислений при работе сценария отличается от обычного. При пошаговом выполнении команд (с экрана) достаточно записать переменную и нажать клавишу *Enter*, и на экране появится значение переменной (вектора, списка и т.п.). Если такую команду записать в файле *myscript.r*, ее выполнение никак не отразится на экране.

Для осуществления ввода/вывода надо записать в программу соответствующие команды:

```
> x<- readline("Введите номер. ")
```

В результате *x* примет значение символьной строки, которую Вы введёте после надписи на экране "Введите номер. "

Для вывода на экран результатов используются команды *print*() или *cat*(), о которых говорилось в соответствующем разделе.

Вот пример листинга, сохраненного в файле *R\big.r*.

```
readline("Введите число строк ")->m
as.numeric(m)->m
readline("Введите число столбцов ")->n
as.numeric(n)->n
x<-runif(m*n)                 # создаем случайный вектор
attr(x,"dim")<-c(m,n)         # превращаем его в матрицу
apply(x,1,sum)->y     # суммируем строки матрицы

print(y)              # выводим вектор на экран
hist(y)               # строим гистограмму для суммарного вектора
sink("R/rezult.r")    # задаем устройство для вывода
print(y)              # выводим вектор в файл
sink()                # восстанавливаем вывод на экран
```

Ход выполнения этой программы

```
> source("R/big.r")
Введите число строк 10
Введите число столбцов 5
 [1] 2.199060 2.881747 1.845853 2.065893 3.151794 1.239585 2.149884 1.794647
 [9] 2.230814 3.110016
```

Дальше вывод проходит в файл ***R****rezult.r*. Если есть необходимость следить за выполнением программы, можно настроить параметр *echo=TRUE*, тогда команды будут перед выполнением выводиться на экран. Или в файл вывода, если он назначен.

Можно управлять выводом с помощью параметров команды *print*().

Графика в *R*

В языке ***R*** можно создавать различные графические объекты. Заметим, что графические команды разделяются на команды высшего уровня (*hist*(), *plot*(), *image*(), *boxplot*(), …) и низшего (*lines*(), *points*(), *abline*(), *text*(), *legend*(), *title*(), …)

Первые открывают новое окно вывода и (вообще говоря) создают в нем рамку для данных. Команды низшего уровня выполняются только после команд высшего уровня и только добавляют элементы к уже существующей картинке

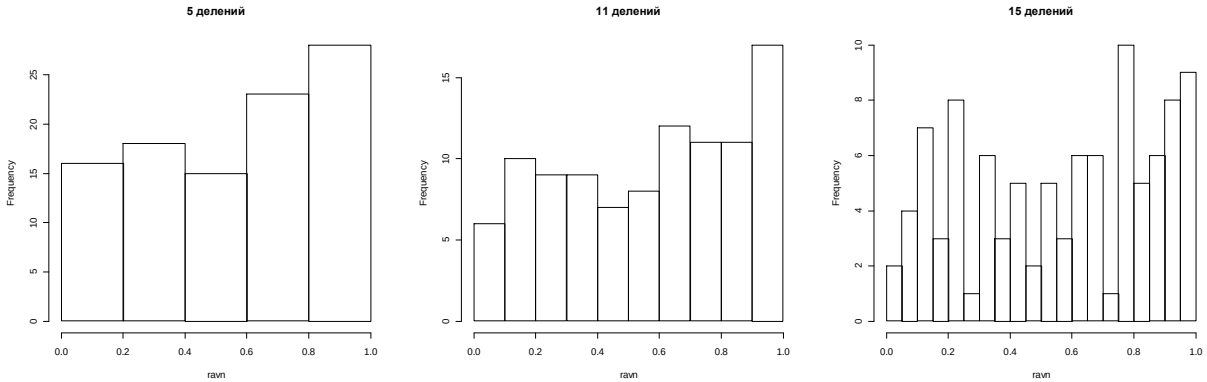
Построение гистограмм

Мы уже говорили о гистограммах в разделе первичной обработки данных. Рассмотрим теперь «графическую» составляющую.

Оператор имеет много параметров, которые, однако, можно не задавать самостоятельно, они вычисляются по умолчанию. Однако при желании можно, например, указать точки деления *breaks* самостоятельно. Если вы задаете не вектор всех значений, а только число, оно будет рассматриваться как количество (равных) промежутков. Однако оно имеет только рекомендательный характер. Например, сформируем выборку из 100 значений равномерно распределенной случайной величины и построим ее гистограммы с параметром *breaks* равным 5, 11 и 15:

```
> ravn<-runif(100)
> hist(ravn,breaks=5,main="5 делений")
> hist(ravn,breaks=11,main="11 делений")
> hist(ravn,breaks=15,main="15 делений")
```

Результаты вы видите на рисунках:



Если не указывать никакие параметры, гистограмма в данном случае будет состоять из 10 промежутков. Если вы хотите, чтобы программа рисовала все-таки 15, а не 20 интервалов, задайте *breaks* самостоятельно:

```
> hist(ravn,breaks=seq(0,1,len=16))
```

Для создания сетки равноотстоящих узлов можно также использовать функцию *pretty*. Однако она не дает точно заданное число узлов, а выбирает близкое количество так, чтобы промежутки между узлами были «красивыми», например, целыми или кратными 0.1 (0.2, 0.5 и т.п.)

```
> x<-runif(100)
> pretty(x)
[1] 0.0 0.2 0.4 0.6 0.8 1.0
> pretty(exp(x),n=3)) # то же получится, если не указывать n (n=5)
[1] 1.0 1.5 2.0 2.5 3.0
> pretty(exp(x),n=10)
```

```
 [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8
> pretty(exp(x),n=2)
[1] 1 2 3
```

Если не нужно выводить гистограмму на экран, нужно указать *plot* = *FALSE*.

Графический оператор *plot*

Этот оператор имеет многообразные варианты, в зависимости от класса того объекта, к которому применяется.

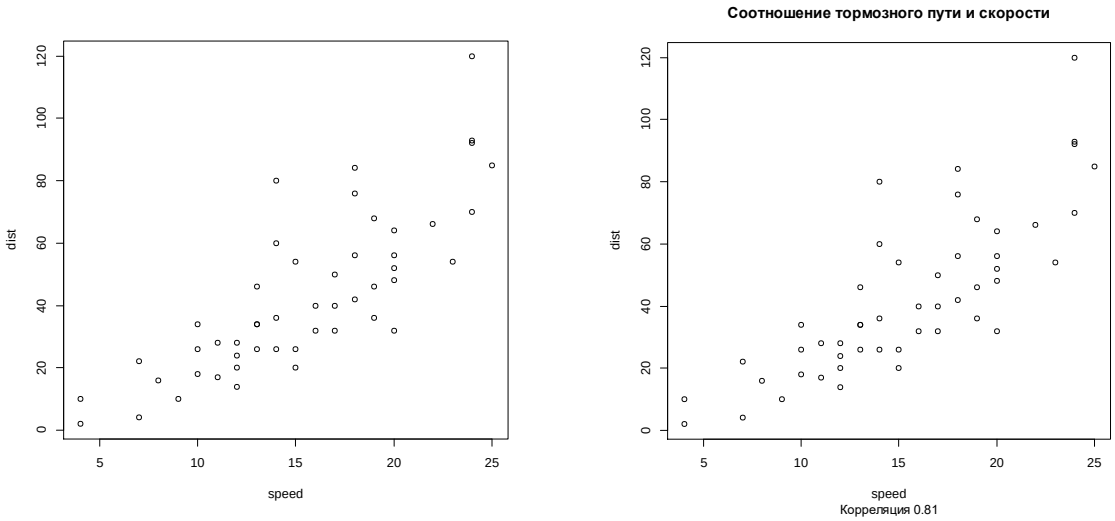
Самый простой вариант – указать два вектора одинаковой длины, они будут рассматриваться как координаты *x* и *y* точек на плоскости. Без дополнительных указаний каждая точка будет помечена кружком.

```
> plot(cars)     # см. левый рисунок
```

Этой командой мы выводим на экран данные из встроенного объекта cars. Указание: изучите этот объект с помощью справки, задав команду ?cars

При желании на рисунок можно добавить заголовок и подзаголовок

```
> plot(cars, main = "Соотношение тормозного пути и скорости",
       sub = paste("Корреляция", format(cor(cars)[1,2],dig=2)))
```

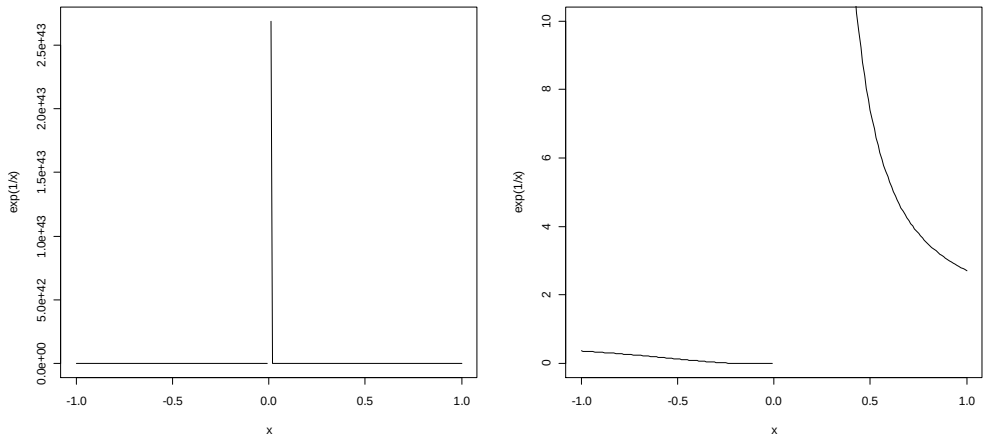


Разберитесь самостоятельно, как действуют все элементы данной команды.

Команда *plot*() имеет множество параметров, изучите их с помощью справки. Например, можно построить обычный график функции, если указать тип линии «*l*», т.е. *line*.

```
> x<-seq(-1,1,by=0.01)
> plot(x,exp(1/x),type="l")
```

Результат показан на рисунке слева.



Как мы видим, оператор не выдает ошибку при нарушениях в вычислении: для случая деления на 0 данные просто пропускаются. Недостатком в данном случае является то, что программа сама подобрала размер по оси *y*. Из-за этого детали поведения функции около 0 практически не распознаются. Чтобы избежать этого, можно задать размеры по оси *y* самостоятельно:

```
> plot(x,exp(1/x),type="l",ylim=c(0,5),col=2)
> abline(h=0,v=0)
```

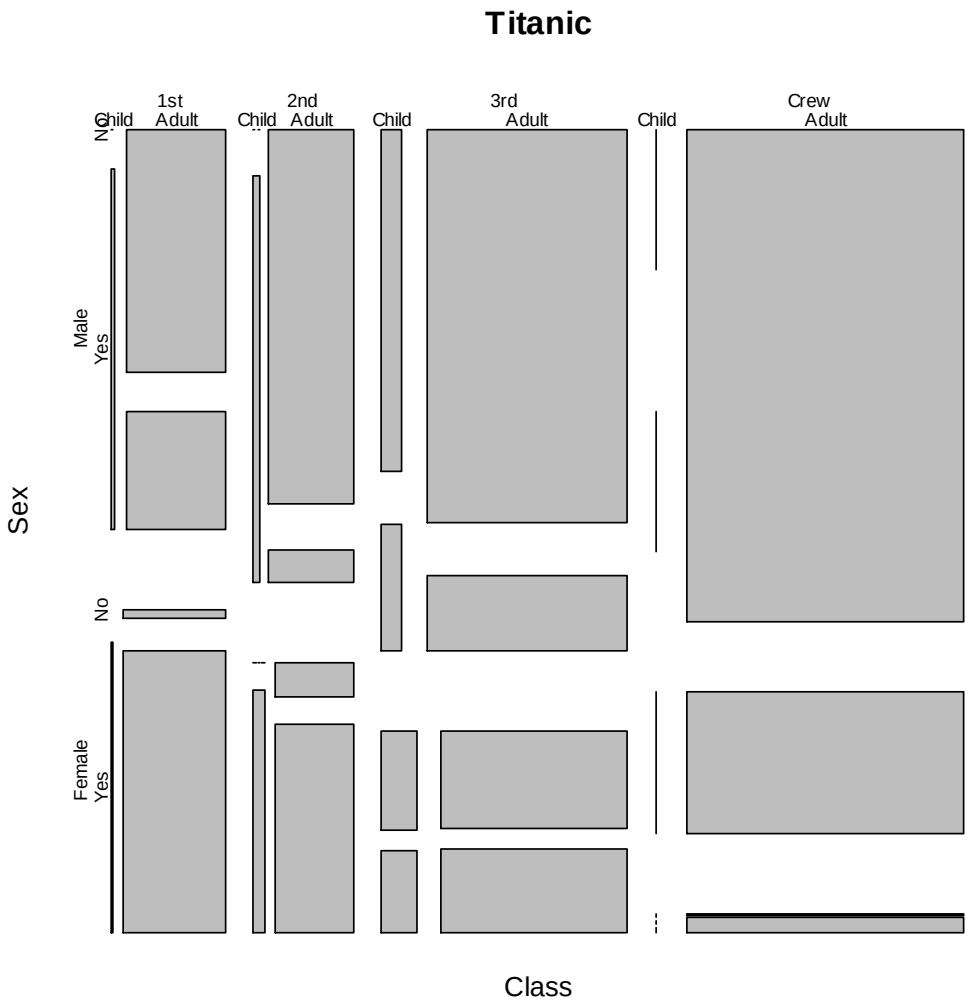
Результат показан на правом рисунке. Выясните, что делает команда *abline*().

Для объектов другого типа (класса) команда *plot*() будет выдавать совершенно непохожие результаты. Вот, например, встроенный объект *Titanic* имеет тип *table*.

Задание. Разберитесь самостоятельно в структуре этого объекта.

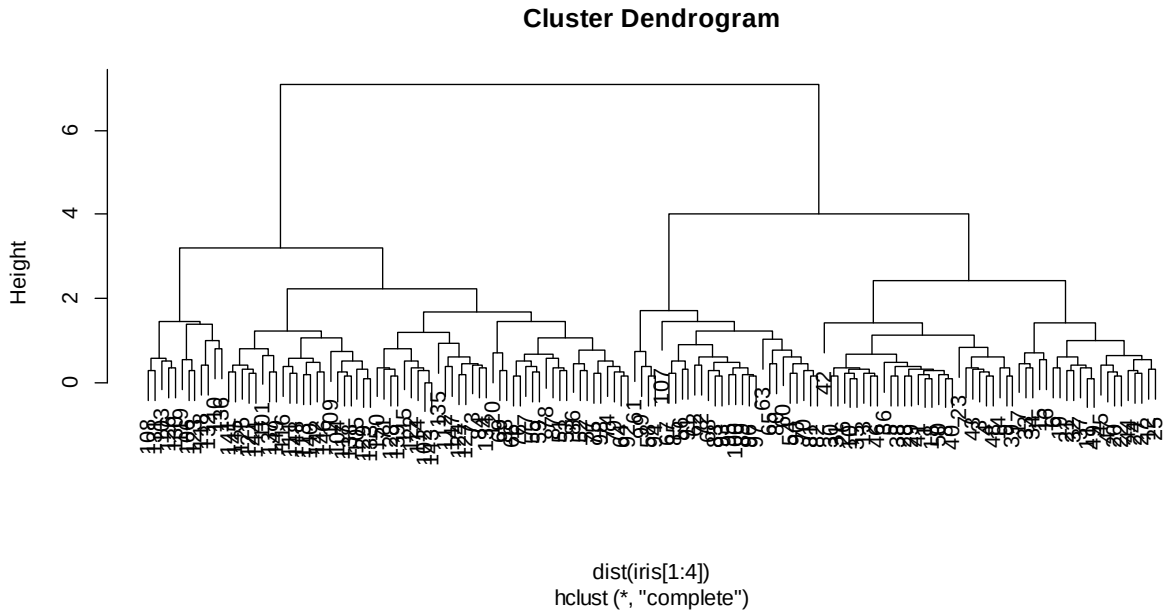
Вот как он выводится на экран:

```
> plot(Titanic)
```



Вот как работает команда *plot*() с результатом иерархической кластеризации:

```
> plot(hclust(dist(iris[1:4])))
```



Команды низшего уровня

Для нанесения линий используется команда *lines*(), прямых линий – *abline*(), точек – *points*(), текста – *text*(). Разберитесь, как работают эти команды, запуская по очереди строки этого скрипта.

```
x<-seq(-2,2,0.01)*pi
plot(x,sin(x),asp=1,type="l")
abline(h=-1:1, v=-6:6,lty=2)
abline(h=0, v=0,col=2)
abline(c(0,1),col=3)
lines(x,cos(x),col=4)

plot(trees[1:2],type="n")
points(trees[1:2],pch=round(trees[,3]))
text(trees[,1],trees[,2],trees[,3])
```

Заметьте, что обращение к объекту trees происходит по-разному. Чем отличаются эти способоы?

Изучите также команды *mtext*(),*segments*(), *rect*(), *legend*(), *title*(). Какие ещё команды предлагает вам Справка?

Настройка параметров вывода

Еще одно соображение касается настройки. Почти все параметры выводимых графиков настраиваются, но это может потребовать больших усилий. К счастью, ***R***, как правило, предоставляет очень разумные подборки параметров.

Многие параметры можно указать прямо в команде (как высшего, так и низшего уровня). Другие надо задавать отдельно. Общая настройка происходит через команду *par*():

```
> oldpar <- par(mfrow=c(2,2), bg="white")
...
> par(oldpar)
```

Здесь мы поменяли цвет фона (это полезно для сохранения графики командой *dev.print*()) и количество графиков в окне. Они будут располагаться в 2 ряда и в две колонки.

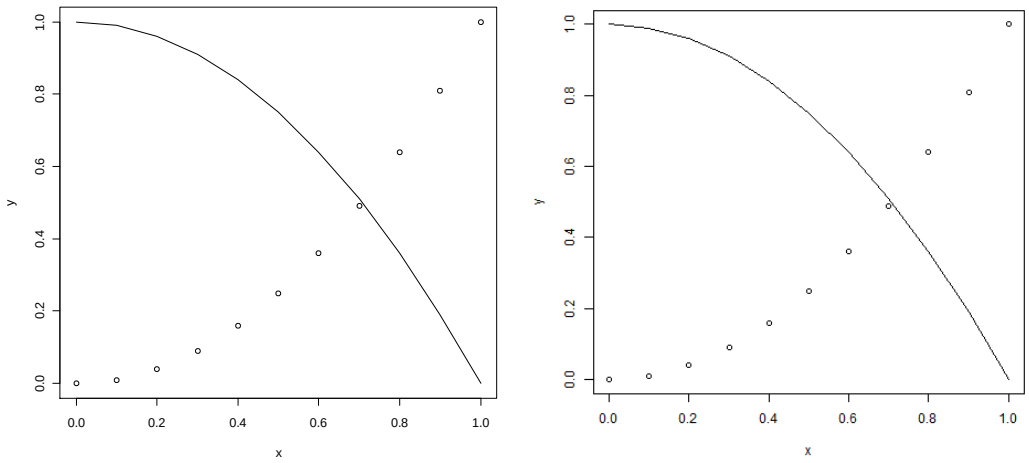
Заметьте, что команда *par*() работает и как функция. Возвращаемое ею значение – старые значения параметров (мы сохранили их в переменной *oldpar*).

```
> str(oldpar)
List of 2
 $ mfrow: int [1:2] 1 1
 $ bg   : chr "transparent"
```

В конце группы команд мы возвращаемся к старым значениям параметров.

Сохранение графики

Конечно, графические изображения тоже желательно сохранить в файл. Проще всего использовать копипаст: скопировать с помощью правой кнопки рисунок в буфер и вставить в нужный файл (так получен рисунок слева).



Однако этот процесс можно автоматизировать. Проще всего это делать через изменение *графического устройства* (*graphic device*). Вот как сохранить графики в файл с расширением *.png*:

```
> x<-seq(0, by=0.1); y<-x^2; z<-1-y
> png(file="grafik.png")
> plot(x,y)
> lines(x,z)
> dev.off()
```

Вторая команда "открывает" графическое устройство (в данном случае файл *grafik.png* в текущей папке), а последняя – закрывает его, записывая график в файл. Команды *plot*() и *lines*() ничего не выдают на экран, поскольку графическим устройством в этот момент является файл. В результате мы имеем в файле *grafik.png* такую картинку (выше, справа).

Аналогичные команды есть для форматов *.bmp*, *.jpeg*, *.tiff*, *.pdf* . Последняя команда позволяет сохранить в один файл несколько рисунков.

Другой способ сделать это – при помощи команды *dev.print*(). Преимущество последнего способа в том, что Вы можете сохранить в файл все содержимое текущего графического окна, в том числе и элементы, нанесенные при помощи интерактивных команд *locator*() и *identify*().

Содержимое текущего графического окна можно вывести в файл типа *.pdf* командой *dev.copy2pdf*(). Она поддерживает еще два типа файлов.