Введение в программирование на R

Дмитрий Храмов

Содержание

Глава 1

Знакомство с R

В этой главе мы установим R и научимся пользоваться средой разработки RGui. По пути решим простую, но весьма распространённую задачу – построим график функции.

Установка

Рассмотрим способ установки R, который подойдёт для любого типа операционных систем, поддерживаемых пакетом: Linux, Mac или Windows. В случае Windows этот путь – единственный, для Linux и Mac проще воспользоваться менеджером пакетов соответствующей системы.

Чтобы установить R, зайдём на его официальный сайт и выберем интересующую версию программы. Сам R, документация к нему и дополнительные пакеты распространяются через сеть ftp- и веб-серверов, называемую CRAN (Comprehensive R Archive Network). Поэтому следующим шагом будет выбор одного из более чем шести десятков зеркал CRAN. После этого, указав тип операционной системы, скачиваем дистрибутив R.

Запустим инсталляционный файл и будем следовать указаниям программы-установщика. Единственный момент, который потребует вашего внимания, – выбор разрядности операционной системы (рис. 1.1).

После установки запустим программу. В Linux и Mac, по умолчанию, R работает в консоли. Кроме того, вместе с пакетом поставляется графический интерфейс. В Linux он основан на связке Tcl/Tk и запускается командой `R --gui=Tk`. В Mac встроенный графический интерфейс для R называется R.app.

Версия R для Windows поставляется с графической оболочкой RGui (рис. 10.5), которую мы и рассмотрим в дальнейшем.

Заметим, что работа со всеми указанными выше графическими оболочками выглядит примерно одинаково.

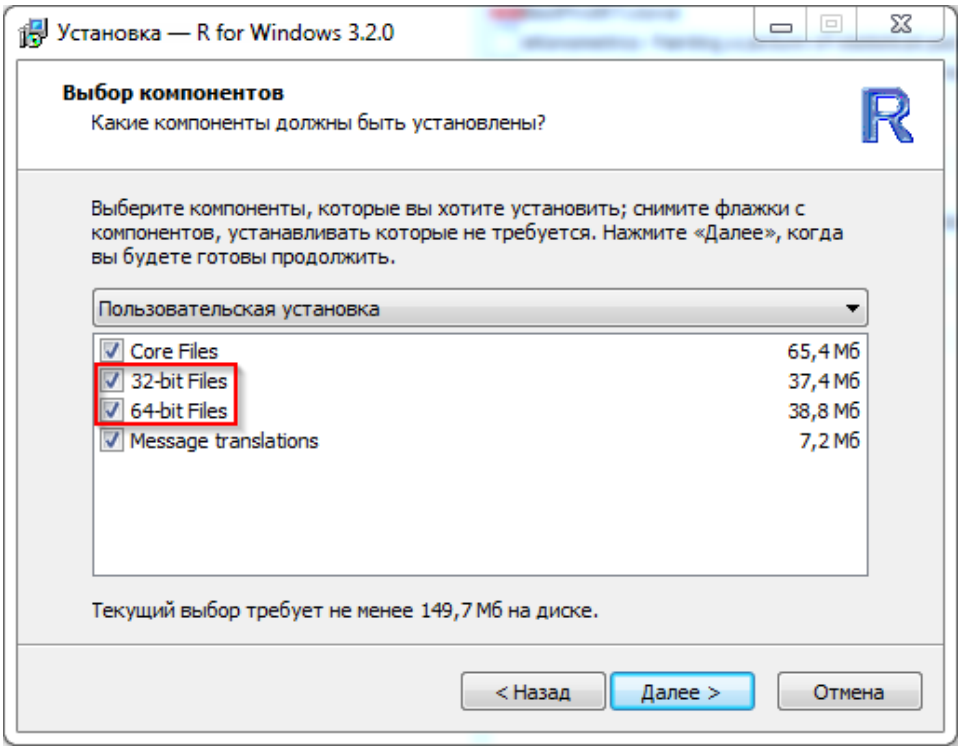


Рис. 1.1 – Выбор компонентов в ходе установки R под Windows

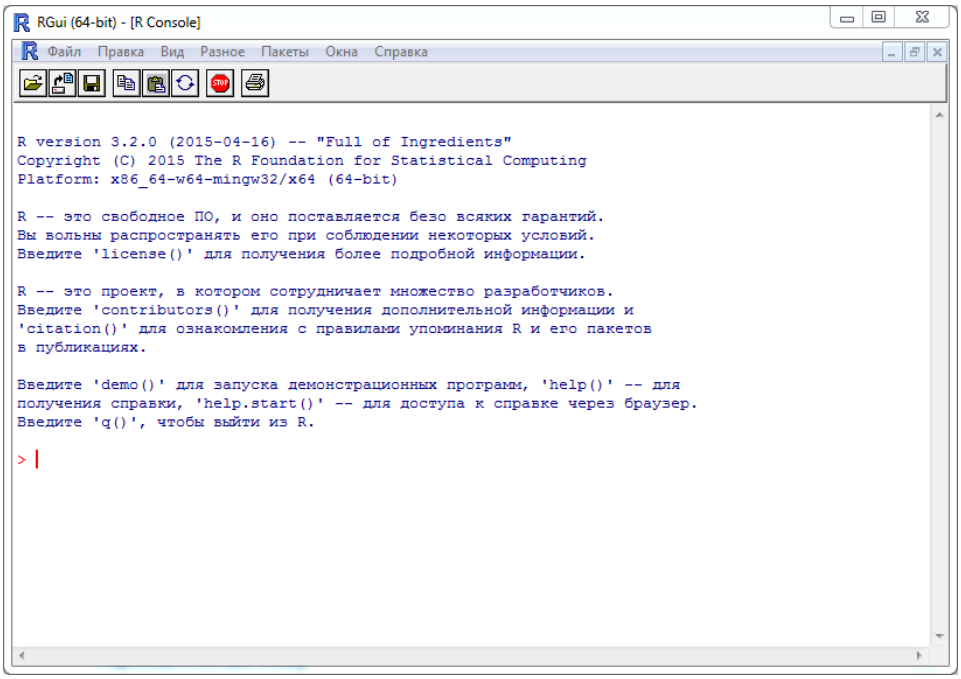


Рис. 1.2 – Консоль R в графической оболочке RGui (Windows)

Работа в среде RGui

RGui – это стандартная графическая оболочка R под Windows, являющаяся простейшей средой разработки. Она быстро загружается и достаточно удобна в использовании. В RGui есть три вида окон:

- консоль;
- редактор скриптов;
- окно графического устройства.

Команды R вводятся в консоли (рис. 10.5) после приглашения пользователя (значка '>') и отправляются на выполнение нажатием *Enter*.

Управление консолью:

- дополнение команды – *Tab*;
- перемещение по истории команд – клавиши со стрелками;
- прекращение выполнения команды – *Esc*;
- переключение в другое окно – *Ctrl+Tab*;
- очистка консоли – *Ctrl+L*.

Для создания собственных программ (скриптов)[1] удобнее использовать не консоль, а редактор (рис. 1.3).

Открыть его можно в меню Файл/Новый скрипт. Первое окно открывается с помощью меню, последующие – так же или комбинацией клавиш *Ctrl+N*.

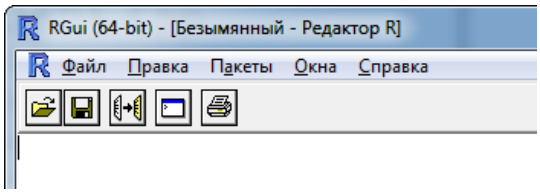


Рис. 1.3 – Интерфейс редактора кода в RGui

Обратите внимание на то, как изменилась панель инструментов по сравнению с консолью (рис. 10.5).

В качестве примера построим график синусоиды:

```
x <- seq(-pi,pi,.1)
y <- sin(x)
plot(x,y)
```

В первой строке формируется одномерный массив (вектор) x-координат, значения которых изменяются от $-\pi$ до π с шагом $0{,}1$. Этот массив сохраняется в переменной x. Комбинация символов <- обозначает операцию присваивания.

[1] Мы используем термины «скрипт» и «программа» как синонимы.

Во второй строке создаётся вектор `y`, состоящий из синусов элементов вектора `x`.

Наконец, в третьей строке функция `plot` строит требуемый график.

В редакторе можно набирать команды и выполнять их как по одной, так и целыми блоками, с помощью комбинации *Ctrl+R* (на панели инструментов также есть соответствующая кнопка). Например, можно выделить весь скрипт – *Ctrl+A* и отправить его на выполнение *Ctrl+R* (рис. 1.4).

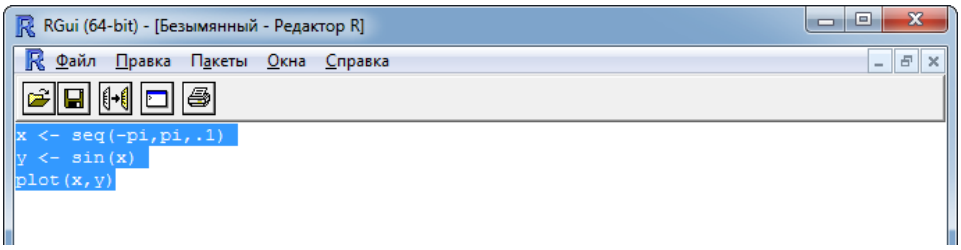


Рис. 1.4 – Выделенную часть кода можно отправить на выполнение комбинацией клавиш *Ctrl+R*

В результате получим графическое окно с построенным в нём графиком синусоиды (рис. 1.5). По терминологии R окна, в которых строятся графики, и файлы, в которых эти графики сохраняются, вместе называются *графическими устройствами*.

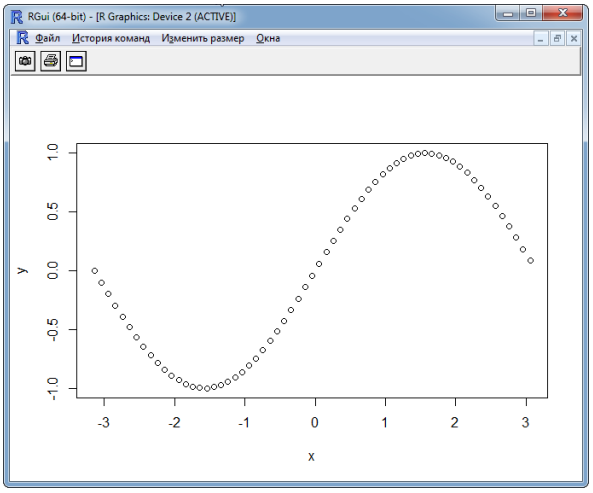


Рис. 1.5 – Графическое устройство в RGui

Вернуться к консоли можно нажатием кнопки на панели инструментов (рис. 1.6) или при помощи *Ctrl+Tab*.

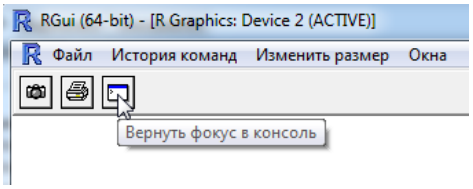


Рис. 1.6 – Кнопка **Вернуть фокус в консоль** на панели инструментов графического устройства

Управление окнами при помощи меню Окна показано на рис. 1.7.

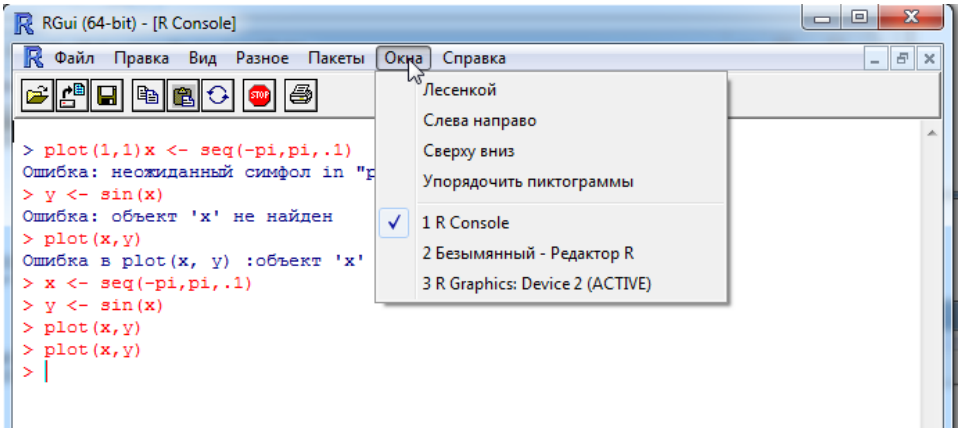


Рис. 1.7 – Управление окнами с помощью меню Окна

Сохранить скрипт можно при помощи команд Сохранить или Сохранить как... меню Файл редактора или соответствующей кнопки на его панели управления.

Заметим, что скрипты R, объединённые общей темой, удобно держать в отдельном каталоге.

Выйти из R можно, вызвав в консоли `q()` или воспользовавшись меню Файл/Выйти.

При выходе из RGui среда предложит сохранить рабочее пространство (рис. 1.8).

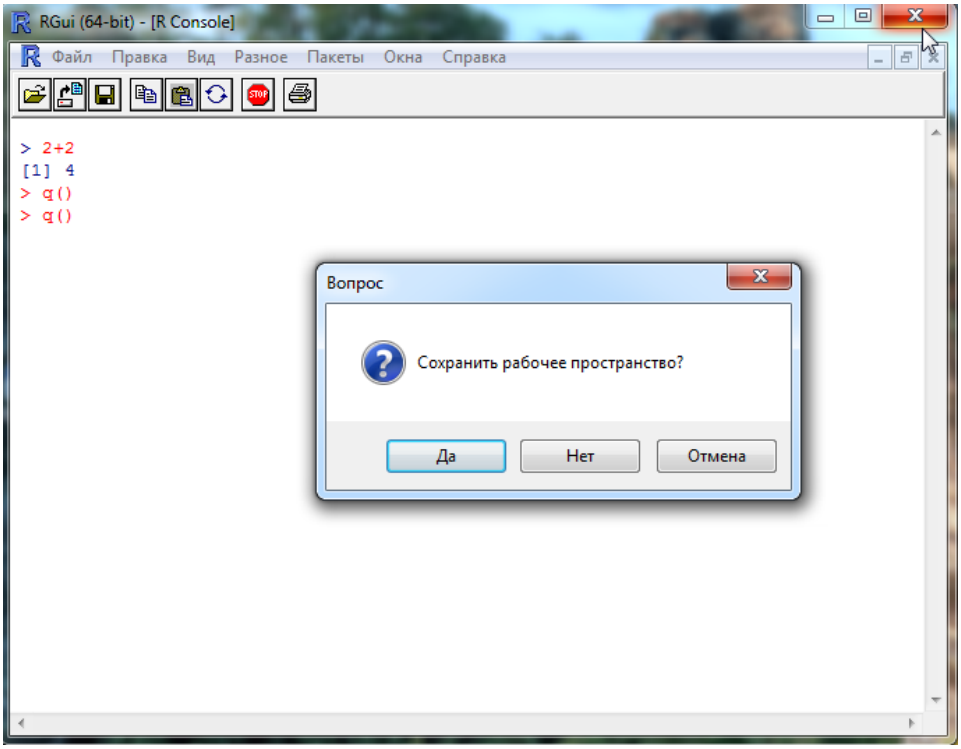


Рис. 1.8 – Сохранение рабочего пространства по выходе из RGui

Рабочее пространство (workspace) – это область оперативной памяти, в которой хранятся все созданные пользователем объекты (векторы, матрицы, таблицы, списки, функции и т. п.). Сохранение рабочего пространства в файле и его последующая загрузка в новом сеансе работы с R (Файл/Загрузить рабочее пространство...) позволяют продолжить работу с того места, где она была прервана. При этом сохраняются значения всех переменных, вычисленные на прошлом сеансе работы.

Просмотреть список объектов в рабочем пространстве можно при помощи функции `ls`. Вызовем её в консоли

```
> ls()
```

и получим в результате

```
## [1] "x" "y"
```

Действительно, в памяти сейчас хранятся векторы координат синусоиды.

Два символа решётки (##) указывают на то, что далее идёт результат выполнения программы (в консоли они не отображаются, см. рис. 1.9). Смысл единицы (`[1]`) станет ясен в следующей главе.

Другие команды управления рабочим пространством:

```
# Сохранить рабочее пространство в файл .RData в текущем рабочем каталоге
save.image()
# Сохранить заданные объекты в файле
save(object_list, file="myfile.RData")
# Загрузить образ рабочего пространства
load("myfile.RData")
# Очистить рабочее пространство
rm(list=ls())
```

Символ # открывает строку комментария.

Сохранить или загрузить образ рабочего пространства можно при помощи меню Файл. Следует иметь в виду, что это меню, как и любое другое в RGui, просто запускает на выполнение соответствующие функции R.

Например, команда меню Файл/Сохранить рабочее пространство... вызывает функцию `save.image`, и результат этого вызова можно увидеть в консоли (рис. 1.9).

```
> x <- seq(-pi, pi, .1)
> y <- sin(x)
> plot(x,y)
> ls()
[1] "x" "y"
> save.image("C:\\Users\\Admin\\Documents\\myimage")
>
```

Рис. 1.9 – Команда меню Файл/Сохранить рабочее пространство... представляет собой вызов функции `save.image`

Клавиши со стрелками вверх и вниз позволяют перемещаться по списку выполненных ранее команд – *истории команд*. Историю команд также можно сохранить в файл на диске.

```
# Вывод истории команд
history()                 # выводит список 25 последних выполненных команд
history(max.show=Inf)     # выводит все выполненные в сессии команды
# Сохранить историю команд
savehistory(file="myfile") # по умолчанию ".Rhistory"
# Загрузить историю команд
loadhistory(file="myfile") # по умолчанию ".Rhistory"
```

Помимо RGui, для R существуют другие, более продвинутые среды разработки, например R Commander или RStudio. Так же, как и сам R, они являются кросс-платформенными и распространяются свободно. Заметим, что хотя возможностей у этих сред гораздо больше, они дополняют возможности RGui, но не перечёркивают их. Так что навыки работы с RGui пригодятся и при работе в более совершенной среде.

Справка

Верхний пункт меню Справка (в консоли это Консоль, в редакторе скриптов, соответственно, Редактор) даёт короткую подсказку по работе с соответствующим окном. Справка для редактора показана на рис. 1.10.

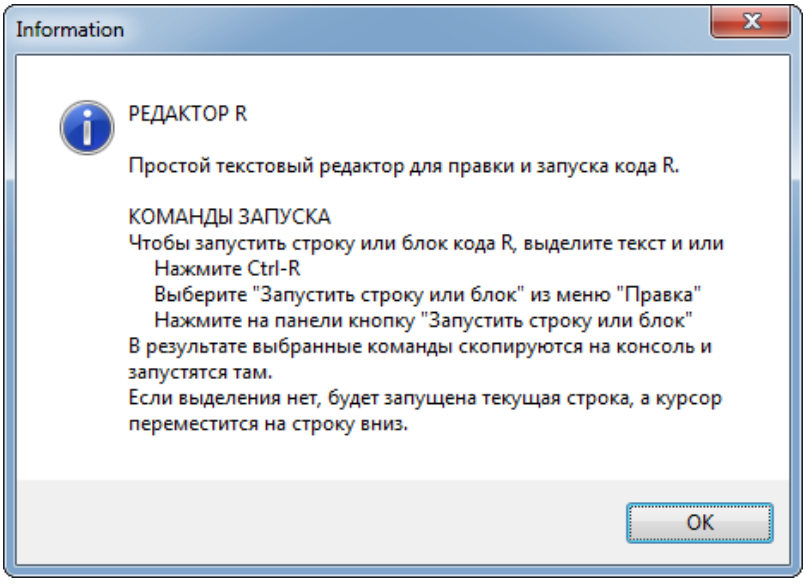


Рис. 1.10 – Справка по работе в редакторе скриптов RGui

Справку по функции с именем `fun` можно получить, набрав в консоли `?fun`. Например, для получения справки по функции создания последовательностей `seq` наберём:

```
?seq
# или
help(seq)
```

В ответ на выполнение этой команды откроется браузер со справочной информацией (рис. 1.11).

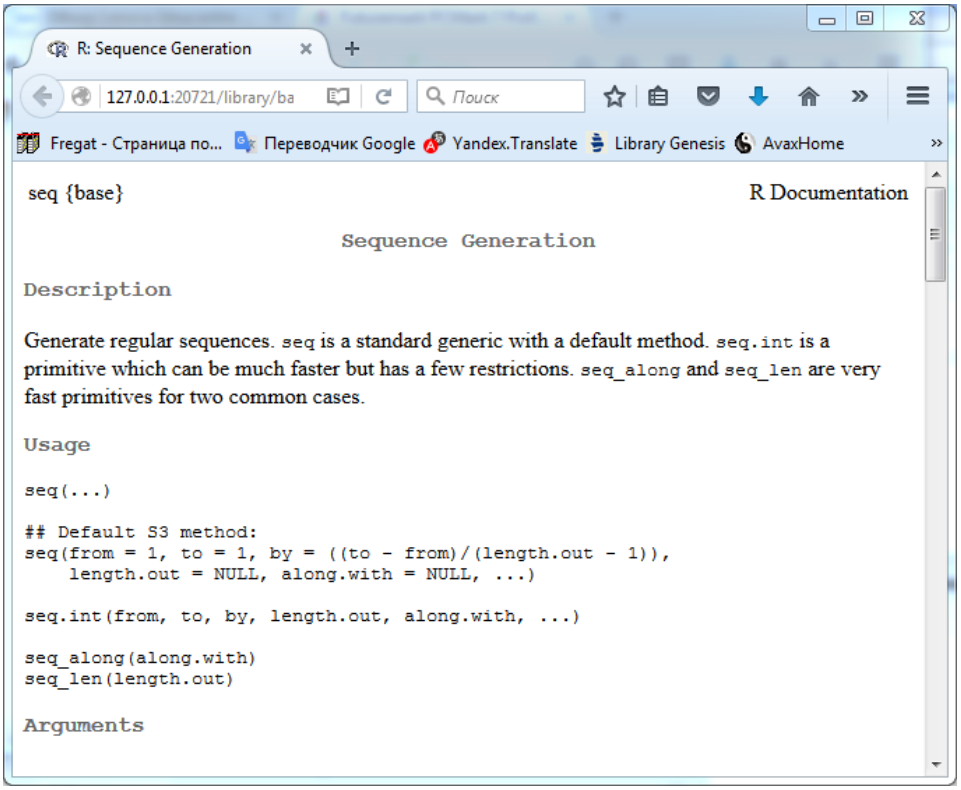


Рис. 1.11 – Справка по функции seq

Глава 2

Скаляры, векторы и матрицы

В этой главе мы научимся создавать переменные и выполнять над ними арифметические и логические операции. Познакомимся с основными типами данных: числовыми, символьными и логическими, а затем узнаем, что все они представляют собой векторы, которые в R вовсе не составной тип данных, а наоборот – простейший. Научимся создавать из векторов матрицы и в несколько строк кода вычислять интегралы.

Арифметические операции и присваивание

Арифметические операции в R выглядят так же, как и во многих других языках программирования:

```
3+2
2*(4-1)/6
3^2; sqrt(4) # ';' служит для разделения выражений в строке
```

```
## [1] 5
## [1] 1
## [1] 9
## [1] 2
```

А вот запись операции присваивания имеет особенности. Оно может обозначаться знаком равенства '=' или «стрелками», позволяющими присваивать как влево (`a <- b`), так и вправо (`b -> a`).

```
a <- 2   # присвоим значение a - традиционная форма записи для R
3 -> b   # присвоим значение b
c = a+b  # и это - то же присваивание
c        # напечатаем значение c
```

```
## [1] 5
```

Мы будем использовать обозначение '`<-`' как из соображений совместимости со старыми версиями пакета, так и чтобы отличать присваивание от других операций, например от задания имён столбцов в таблицах – в последнем случае всегда используется '='.

Имена

При назначении имён переменных и функций придерживаются следующих правил:

1. Использовать для имён только латинские буквы, символ подчёркивания '`_`', цифры и точку '`.`'; при этом имя не должно начинаться с цифры или точки: `plot_new2` и `plot.new` – это правильно, а `.plot` и `2plot` – нет.
2. Помнить, что R чувствителен к регистру: `Var`, `var` и `VAR` – это разные имена.
3. Не давать объектам имена, занятые распространёнными функциями (например, не следует создавать функцию с именем `c()`) или ключевыми словами (особенно это касается `T`, `F`, `NA`, `NaN`, `Inf`).

Одна из особенностей R состоит в том, что в именах объектов допустимо использовать точку. Её нередко применяют для разделения смысловых частей имени, подобно тому как в других языках используют символ подчёркивания (который в ранних версиях R использовать было нельзя). Так, функция `install.packages()` служит для установки новых пакетов расширений, но это именно функция, а не метод `packages()` объекта `install`.

Проверить, существует ли переменная или функция с заданным именем, можно при помощи функции `exists`:

```
exists('a')    # переменная 'a' уже определена,
```

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

```
exists('d')    # ...а `d` - ещё нет
```

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

```
exists('all')  # существует функция с именем 'all'
```

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

Параметр `mode` позволяет указать вид объектов, среди которых выполняется поиск. Так, переменная `a` существует, а вот функция `a` – нет:

```
exists("a", mode="function")
```

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

Простые типы данных

Простые типы данных – это такие данные, которые нельзя разделить на составляющие элементы. Их ещё называют скалярными данными, или *скалярами*, в отличие от составных данных – векторов, матриц, списков и т. п.

В R существуют три вида простых типов данных: числовые, символьные и логические.

Числа

R различает три вида чисел:

- `numeric` – вещественные;
- `integer` – целые;
- `complex` – комплексные.

```
a <- 1 # или a <- 1.0
class(a)
```

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

```
str(a)
```

```
##  num 1
```

Функция `class` позволяет ответить на вопрос, к какому типу данных (классу) относится объект. Функция `str` возвращает структуру объекта: тип данных, к которому относится объект (`num` – сокращение от `numeric`), и его значение (в нашем случае – `1`).

Если вещественные числа создаются непосредственно, то для создания целых служит функция `as.integer`:

```
b <- as.integer(a)
class(b)
```

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

Для проверки, относится ли объект к тому или иному типу данных, используют функции семейства `is.*`, например `is.integer`, `is.numeric` и т. п.

```
is.integer(b)
```

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

```
is.integer(a)
```

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

```
is.numeric(a)
```

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

Добавляя к целому или вещественному числу мнимую часть, обозначаемую символом `i`, получаем комплексное число:

```
c <- sqrt(-1)     # так не пойдёт!
```

```
## Warning in sqrt(-1): созданы NaN
```

```
str(c)
```

```
##  num NaN
```

```
c <- sqrt(-1+0i)  # правильный способ
class(c)
```

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

```
str(c)
```

```
##  cplx 0+1i
```

```
is.numeric(c)
```

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

```
is.complex(c)
```

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

Неявное приведение числовых типов данных (type coercion) выполняется по следующей схеме:

```
integer -> numeric -> complex
```

Это означает, например, что складывая целое и вещественное числа, мы получим в результате вещественное число, а добавляя в выражение число комплексное – получим комплексный же результат. Для введенных ранее переменных имеем:

```
ab <- a+b
class(ab)
```

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

```
abc <- ab/c
class(abc)
```

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

Явное приведение типов данных выполняется при помощи функций семейства `as.*`:

```
as.numeric(1.5+1i) # при потере мнимой части выдается предупреждение
```

```
## Warning: мнимые части убраны при преобразовании
```

```
## [1] 1.5
```

```
dn <- as.numeric(1.5+0i)
str(dn)
```

```
##  num 1.5
```

```
di <- as.integer(1.5+0i)
str(di)
```

```
##  int 1
```

```
as.complex(di)
```

```
## [1] 1+0i
```

Символьный тип

Данные символьного типа `character` представляют собой строки, состоящие из символов, заключённых в одинарные или двойные кавычки.

```
s1 <- "Text string."
s2 <- 'Another string.'
s12 <- paste(s1,s2)
str(s12)
```

```
##  chr "Text string. Another string."
```

Соединение строк – *конкатенация* – выполняется функцией `paste`.

Кавычки одного типа могут находиться внутри кавычек другого типа:

```
"First 'attempt'"
'Second "attempt"'
```

Если типы внутренних и внешних кавычек совпадают, то внутренние кавычки нужно заэкранировать при помощи '\' (бэкслэш):

```
"Third \"attempt\""
```

Иногда удобней создать строку с помощью функции `sprintf`, которая похожа на одноимённую функцию языка C:

```
sprintf("%s was founded in AD %d.", "London", 43)
```

```
## [1] "London was founded in AD 43."
```

Выделить подстроку позволяет функция `substr`):

```
substr("abcdefg",3,5) # выделить подстроку с 3-го по 5-й элемент
```

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

Функции `sub` и `gsub` позволяют заменить часть строки, соответствующую заданному образцу. При этом `sub` заменяет только первое вхождение, а `gsub` выполняет глобальную замену подстроки.

```
#     что менять # на что заменить  # где искать
# sub("a",        "A",                "rama")
sub("a","A","rama")  # sub = substitute
```

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

```
gsub("a","A","rama") # gsub = global substitute
```

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

```
sub("a","A",c("mama","rama","banana"))
```

```
## [1] "mAma"   "rAma"   "bAnana"
```

Числа преобразуются в строки «сами собой». Обратное преобразование выполняется с помощью функций `as.*`.

```
paste("the value of PI is", 3.14)
```

```
## [1] "the value of PI is 3.14"
```

```
as.numeric("3.14")
```

```
## [1] 3.14
```

```
as.numeric("a")
```

```
## Warning: в результате преобразования созданы NA
```

```
## [1] NA
```

```
as.character(3.14)
```

```
## [1] "3.14"
```

Вот как можно преобразовать число с десятичной запятой из символьной формы записи (`"9,1"`) в числовую `9.1`:

```
r <- "9,1"
as.numeric( gsub(",", ".", r) )
```

```
## [1] 9.1
```

Логический тип

Данные, относящиеся к логическому типу `logical`, имеют всего два значения: `TRUE` («истина») и `FALSE` («ложь»).

Чаще всего данные этого типа возникают при сравнении переменных:

```
x = 2; y = 5
z = x > y       # x больше y?
z               # напечатаем полученное логическое значение
```

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

```
class(z)
```

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

Основные логические операции: `&` (И), `|` (ИЛИ) и `!` (НЕ):

```
a = TRUE; b = FALSE # Запись можно сократить: a = T; b = F
a & b           # и И v
```

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

```
a | b           # и ИЛИ v
```

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

```
!a              # НЕ-u
```

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

Находясь в одном выражении с числами, данные логического типа приводятся к соответствующему числовому типу данных. При этом `FALSE` представляется как `0`, а `TRUE` – как `1`:

```
1 + FALSE
```

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

```
1 + TRUE
```

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

```
class(as.integer(1) + TRUE)
```

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

Общая схема неявного приведения типов в R выглядит так:

```
logical -> числовые типы -> character
```

Например:

```
paste(FALSE, 1.0, "test")
```

```
## [1] "FALSE 1 test"
```

Напомним, что явное приведение типов выполняется функциями семейства `as.*`, а проверку типа можно выполнить с помощью функций `is.*`.

Векторы

Основным типом данных в R является вектор. *Вектор* – это последовательность элементов *одного* типа:

```
# числовой вектор
x <- c(1.5, 6, 8.3, 9, 6, .6, 2e-4)
# символьный вектор
s <- c("s", "t", "r", "i", "n", "g", "another string")
# логический вектор
b <- c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE)
```

Функция `c()` служит для создания вектора. Её название происходит от английского ‘concatenate’ – собирать, то есть она как бы собирает отдельные элементы в вектор.

Элементы векторов нумеруются, начиная с единицы. Чтобы выбрать тот или иной элемент вектора, нужно указать его номер в квадратных скобках:

```
x[1]
```

```
## [1] 1.5
```

Векторы в R играют особую роль: из них строятся все остальные типы данных. Да, выше мы изучали простые типы данных, однако технически эти типы реализуются в R как векторы единичной длины. Например, число – это числовой вектор единичной длины.

```
a = 3
a[1]
```

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

```
# но:
a[2]
```

```
## [1] NA
```

(`NA` – ‘Not Available’ – означает недоступность данных, пропуск в данных).

Возможно, вас интересовало, почему выводу результатов в консоли R всякий раз предшествует `[1]`. Так вот, это номер элемента вектора, с которого начинается строка вывода:

```
c("s", "t", "r", "i", "n", "g", "another string")
```

```
## [1] "s"              "t"              "r"              "i"
## [5] "n"              "g"              "another string"
```

Просто до сих пор мы работали с весьма короткими векторами.

Отрицательный индекс в квадратных скобках означает: выбрать все элементы, кроме указанных:

```
v <- c(1.1, 2.2, 3.3, 4.4, 5.5)
v[1:4]
```

```
## [1] 1.1 2.2 3.3 4.4
```

```
v[-5]
```

```
## [1] 1.1 2.2 3.3 4.4
```

Тип данных, составляющих вектор и его структуру, как и раньше, можно определить с помощью функций `class` и `str` соотвественно:

```
class(v)
```

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

```
str(v)
```

```
##  num [1:5] 1.1 2.2 3.3 4.4 5.5
```

Частным случаем векторов являются последовательности элементов, изменяющихся по определённому закону. Последовательность с шагом `1` можно создать при помощи двоеточия `':'`

```
v = -5:5
v
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

Значительно больше возможностей даёт функция `seq`. Она позволяет создавать последовательности с заданным шагом

```
seq(5,1,by=-.5)
```

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

и заданной длины

```
seq(1,10,length=6)
```

```
## [1]  1.0  2.8  4.6  6.4  8.2 10.0
```

Как и во многих других функциях R, в `seq` есть необязательные аргументы. Их можно не указывать явно, и тогда будут использованы значения, заданные для этих аргументов по умолчанию. Узнать, что это за аргументы и какие значения по умолчанию они принимают, можно из справки по функции `seq`:

```
?seq # help(seq)
```

Функция `rep` позволяет повторить объект заданное число раз:

```
rep(1:3,5)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

А вот вариант похитрее:

```
rep(1:3,c(5,5,5))
```

```
##  [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Эту команду можно сократить, используя ещё один `rep`:

```
rep(1:3,rep(5,3))
```

```
##  [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Арифметические операции над векторами выполняются поэлементно:

```
u <- c(1,2,3)
v <- c(4,5,6)
u+v
```

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

```
u*v
```

```
## [1]  4 10 18
```

```
u/v
```

```
## [1] 0.25 0.40 0.50
```

Скалярное произведение векторов записывается как

```
u %*% v
```

```
##      [,1]
## [1,]   32
```

Деление на ноль даёт в результате `Inf` – бесконечность

```
w <- c(v[1:2],0) # добавляем элемент к фрагменту вектора v
u/w
```

```
## [1] 0.25 0.40  Inf
```

которая при последующих операциях «поглощает» все конечные значения:

```
u+u/w
```

```
## [1] 1.25 2.40  Inf
```

Сложим два вектора разной длины. «Будет ошибка», – скажете вы. А вот и нет:

```
u <- c(1,2,3)
v <- c(4,5,6,7,8,9,10)
u+v
```

```
## Warning in u + v: длина большего объекта не является
## произведением длины меньшего объекта
```

```
## [1]  5  7  9  8 10 12 11
```

Операция будет выполнена, но R предупредит, что длина векторов-операндов различается.

При этом сложение и другие подобные операции, требующие в «обычном» состоянии равной длины операндов, выполняются так: более короткий вектор u повторяется столько раз, сколько нужно, чтобы его длина сравнялась с длиной v, после чего выполняется заданная операция. Фактически складываются векторы:

```
u <- c(1,2,3,1,2,3,1)
v <- c(4,5,6,7,8,9,10)
u+v
```

```
## [1]  5  7  9  8 10 12 11
```

Добавление элементов в вектор осуществляется функциями c и append (”добавить”):

```
vec = c('a','b')
vec = c(vec,'c','d')
vec
```

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

```
values = c('e','f','g')
vec <- append(vec, values)
vec
```

```
## [1] "a" "b" "c" "d" "e" "f" "g"
```

Удаление элементов из вектора выполняется следующим образом:

```
a <- sample(1:10)  # генерируем случайные целые числа от 1 до 10
a
remove <- c(3,5,7) # выберем для удаления 3-й, 5-й и 7-й элементы
a <- a[-remove]    # удалим выбранные элементы
a
```

```
##  [1]  5  6  4  7  3  1  8  9 10  2
## [1]  5  6  7  1  9 10  2
```

Длина вектора, то есть число его элементов, вычисляется функцией length():

```
length(a)
```

```
## [1] 7
```

Указать последний элемент вектора можно так:

```
a[length(a)]
```

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

Векторизация и логическая индексация

Векторизация – подход к программированию, позволяющий выполнять операции над вектором в целом, а не над отдельными его элементами (скалярами).

```
a <- c(1,2,3); b <- c(4,5,6)
c1 <- vector() # создаём пустой вектор
# Вместо того чтобы делать так:
for (i in 1:3) {
  c1[i] <- a[i] + b[i] # c1 обязательно нужно заранее создать
}
# Проще сделать так:
c2 <- a + b
```

Векторизация используется не только в R, но и в других языках программирования, например в MATLAB и Python (при использовании последнего совместно с библиотекой NumPy).

Для векторизации расчётов широко используется логическая индексация:

```
a <- c(6,-2,1,8,0,9)
ind_a <- a > 0
ind_a
```

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

Логический индекс (`ind_a`) – вектор, длиной равный исходному (`a`), элементы которого равны `TRUE`, если соответствующий элемент исходного вектора удовлетворяет логическому условию (`a > 0`), и `FALSE` – в противном случае.

Логическая индексация позволяет заменить связку «цикл + условный оператор». Например, чтобы выбрать положительные элементы `a`, не нужно организовывать цикл с проверкой в его теле условия `a[i] > 0`, а можно поступить так:

```
# Выбрать положительные элементы a
a[a > 0] # или a[ind_a]
```

```
## [1] 6 1 8 9
```

```
# Подсчитать их количество...
length(a[ind_a])
```

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

```
# ... или сумму
sum(a[a > 0])
```

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

Таблица 2.1 – Логические операции

Обозначение	Значение
`a == b`	`a` равно `b`
`a != b`	`a` не равно `b`
`&`, `|`	И, ИЛИ (поэлементные)
`&&`, `||`	И, ИЛИ (сравниваются левые крайние элементы векторов)
`xor(a,b)`	Исключающее ИЛИ
`any(a)`	`TRUE`, если хотя бы один из элементов `a` истинен
`all(a)`	`TRUE`, если все элементы `a` истинны

Примеры использования логических операций:

```
a <- c(6,-2,1,8,0,9)
a > 0 & a < 9
```

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

```
a < 2 | a > 8
```

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

```
any(a>0)
```

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

```
all(a>0)
```

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

Если данные содержат пропуски (`NA`), это может повлиять на результат вычислений. Проверка таких случаев реализуется с помощью функции `is.na`:

```
# Данные с пропусками:
a <- c(6,-2,NA,1,8,0,NA,9)
# Их сумма даёт:
sum(a)
```

```
## [1] NA
```

```
# Является ли элемент пропуском в данных?
is.na(a)
```

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

```
# Вычисление суммы с учётом проверки на пропуски в данных
sum( a[!is.na(a)] )
```

```
## [1] 22
```

Покажем, как с помощью векторизации можно легко вычислить определённый интеграл.

Рассмотрим в качестве примера интеграл $\int_1^2 x^2 dx$. Вычислим его приближённо, воспользовавшись методами прямоугольников и трапеций (для проверки: интеграл равен $7/3 = 2{,}333(3)$)[2].

```
a <- 1; b <- 2             # границы промежутка интегрирования
n <- 1000                  # число узлов интегрирования
x <- seq(a,b,length.out=n) # координаты узлов сетки
h <- x[2]-x[1]             # шаг сетки
y <- x^2                   # значения подынтегральной функции в узлах сетки
```

Значения нижней и верхней интегральных сумм дают оценки величины интеграла снизу и сверху соответственно. Любая их этих сумм даёт приближённое значение интеграла, вычисленное методом прямоугольников:

```
sd<- h*sum(y[-length(y)])  # Нижняя интегральная сумма
sd
```

```
## [1] 2.331832
```

```
su<- h*sum(y[-1])          # Верхняя интегральная сумма
su
```

```
## [1] 2.334835
```

Метод трапеций даёт более точный результат:

```
(su+sd)/2
```

```
## [1] 2.333334
```

Матрицы и массивы

Матрица (`matrix`) в R – это специальный тип вектора, обладающий дополнительными атрибутами, которые позволяют трактовать его как совокупность строк и столбцов.

```
v1 = 1:4
v2 = -2:1
# Создадим из элементов v1 матрицу размера 2x2
m1 = matrix(v1, nrow=2, ncol=2)
m1
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
# Создать матрицу можно, объединяя векторы по строкам
m2 = rbind(v1,v2)
m2
```

```
##    [,1] [,2] [,3] [,4]
## v1    1    2    3    4
## v2   -2   -1    0    1
```

```
# ... или по столбцам
m3 = cbind(v1,v2)
m3
```

```
##      v1 v2
## [1,]  1 -2
## [2,]  2 -1
## [3,]  3  0
## [4,]  4  1
```

При выборе элементов матрицы указывают индексы строк и столбцов:

[2] Вспомнить их можно по книге: *Турчак Л. И., Плотников П. В.* Основы численных методов. М.: Физматлит, 2003.

```
# Элемент 4-й строки и 2-го столбца матрицы m3
m3[4,2]
```

```
## v2
##  1
```

Кроме того, у строк и столбцов матриц могут быть заданы имена. Например, у матрицы `m2` имена строк (а у `m3` – имена столбцов) равны `v1` и `v2`, так что к её элементам можно обратиться так:

```
m2["v2",1] # эквивалентно m2[2,1]
```

```
## v2
## -2
```

Размерность вектора и матрицы можно задать функцией `dim`:

```
A <- 1:9
dim(A) <- c(3,3) # формируем из вектора матрицу размерности 3x3
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Транспонирование матрицы выполняется функцией `t()`:

```
t(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Многомерные массивы в R создаются при помощи функции `array`. Размерность массива задаётся атрибутом `dim`:

```
arr <- array(1:24, dim=c(2,4,3))
arr
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   19   21   23
## [2,]   18   20   22   24
```

Резюме

Синтаксис R имеет две очевидные особенности: в именах переменных можно использовать точку `'.'`, а операция присваивания обозначается стрелкой `->`.

Особую роль в R играют векторы – числовые, символьное и логические. Из них строятся все остальные типы данных. Даже скаляры представляют собой лишь векторы единичной длины. Это приводит к тому, что все операции осуществляются над векторами в целом, а не над отдельными их элементами. В результате в R сравнительно редко используются циклы с перебором по элементам.

Глава 3

Списки и таблицы

Векторы и матрицы состоят из данных одного типа. Чтобы сохранять наборы данных, относящихся к различным типам, в R существуют списки и таблицы, с которыми мы познакомимся в этой главе.

Списки

Список (`list`) напоминает вектор, но может содержать объекты разных типов. Создаётся список функцией `list`:

```
lst <- list(n=1:3, s="test", b=c(T,F,F,F,T))
# или
n <- 1:3
s <- "test"
b <- c(T,F,F,F,T)
lst <- list(n,s,b) # состоит из копий векторов n,s,b
lst
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "test"
##
## [[3]]
## [1]  TRUE FALSE FALSE FALSE  TRUE
```

Структура созданного списка имеет вид:

```
str(lst)
```

```
## List of 3
##  $ : int [1:3] 1 2 3
##  $ : chr "test"
##  $ : logi [1:5] TRUE FALSE FALSE FALSE TRUE
```

Список `lst` состоит из трёх элементов, каждый из которых является вектором: числовым, символьным и логическим соответственно.

Элементы списков, так же как элементы векторов и матриц, могут иметь имена. Узнать или присвоить эти имена можно при помощи функции `names`:

```
names(lst)
```

```
## NULL
```

В настоящий момент имена элементов `lst` не заданы.

Присвоим элементам списка `lst` имена `num`, `cha` и `log`:

```
names(lst) <- c("num","cha","log")
```

и посмотрим, что получилось:

```
str(lst)
```

```
## List of 3
##  $ num: int [1:3] 1 2 3
##  $ cha: chr "test"
##  $ log: logi [1:5] TRUE FALSE FALSE FALSE TRUE
```

Элементы списка можно выбрать одним из следующих способов:

1) с помощью квадратных скобок `[,]`. Полученный в результате объект, в свою очередь, будет являться списком, даже если длина его окажется единичной:

```
lst[1]  # или lst["num"]
```

```
## $num
## [1] 1 2 3
```

```
class(lst[1])
```

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

2) с помощью двойных квадратных скобок `[[,]]`. Возвращённый объект будет того же типа, каким он был до включения в список:

```
lst[[1]] # или lst[["num"]]
```

```
## [1] 1 2 3
```

```
class(lst[[1]])
```

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

3) использовать имена элементов списка и знак доллара `$`:

```
lst$num
```

```
## [1] 1 2 3
```

```
class(lst$num)
```

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

Удалять имена элементов можно следующим образом:

```
names(lst) <- NULL
```

но мы пока делать этого не будем.

Добавить элемент в список можно так же, как и в вектор, – при помощи функций `c()` или `append()`:

```
v <- 5:10            # создадим числовой вектор
l1 <- c(lst,list(v))  # преобразуем его и добавим в список
str(l1)
```

```
## List of 4
##  $ num: int [1:3] 1 2 3
##  $ cha: chr "test"
##  $ log: logi [1:5] TRUE FALSE FALSE FALSE TRUE
##  $    : int [1:6] 5 6 7 8 9 10
```

Заметим, что если вектор не преобразовать при помощи `list()`, то к списку будет добавлен не один элемент-вектор, а шесть элементов-чисел:

```
l2 <- c(lst,v)
str(l2)
```

```
## List of 9
##  $ num: int [1:3] 1 2 3
##  $ cha: chr "test"
##  $ log: logi [1:5] TRUE FALSE FALSE FALSE TRUE
##  $    : int 5
##  $    : int 6
##  $    : int 7
##  $    : int 8
##  $    : int 9
##  $    : int 10
```

Функции `c` и `append`, помимо векторов, позволяют объединять и списки:

```
l12 <- c(l1,l2)
```

Удаляются элементы списка так же, как элементы вектора:

```
remove <- c(2,3)      # удалить 2-й и 3-й элементы
lst <- lst[-remove]
str(lst)
```

```
## List of 1
##  $ num: int [1:3] 1 2 3
```

Определим количество элементов в списке `lst`:

```
length(lst)
```

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

Обратите внимание, что функция `length` возвращает нам число элементов списка. В нашем случае такой элемент один. Другое дело, что сам этот элемент является числовым вектором. Чтобы найти длину этого последнего, нужно применить двойные квадратные скобки

```
length(lst[[1]])
```

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

или преобразовать список в вектор при помощи `unlist()`:

```
vec_from_lst <- unlist(lst)
class(vec_from_lst)
```

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

```
length(vec_from_lst)
```

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

Таблицы

Таблицы (data frames) хранят двумерные данные, но, в отличие от матриц, позволяют находиться в разных колонках данным различных типов. По своему внутреннему устройству таблица – это список колонок равной длины. Можно сказать и так: если вы когда-нибудь видели таблицу в Microsoft Excel, то вам уже знакомы таблицы в R.

Создаются таблицы с помощью функции `data.frame`:

```
df1 <- data.frame(a=1:5,b=6:10,c=letters[1:5])
str(df1)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ a: int  1 2 3 4 5
##  $ b: int  6 7 8 9 10
##  $ c: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
```

Таблицы можно объединять по строкам:

```
df2 <- data.frame(a=6:10,b=11:15,c=letters[6:10])
rbind(df1,df2)
```

```
##     a  b c
## 1   1  6 a
## 2   2  7 b
## 3   3  8 c
## 4   4  9 d
## 5   5 10 e
## 6   6 11 f
## 7   7 12 g
## 8   8 13 h
## 9   9 14 i
## 10 10 15 j
```

или по столбцам:

```
df2 <- data.frame(c=6:10,d=11:15,e=letters[6:10])
cbind(df1,df2)
```

```
##   a  b c  c  d e
## 1 1  6 a  6 11 f
## 2 2  7 b  7 12 g
## 3 3  8 c  8 13 h
## 4 4  9 d  9 14 i
## 5 5 10 e 10 15 j
```

Ещё один способ объединения таблиц даёт в результате список:

```
l3 <- c(df1,df2)
str(l3)
```

```
## List of 6
##  $ a: int [1:5] 1 2 3 4 5
##  $ b: int [1:5] 6 7 8 9 10
##  $ c: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
##  $ c: int [1:5] 6 7 8 9 10
##  $ d: int [1:5] 11 12 13 14 15
##  $ e: Factor w/ 5 levels "f","g","h","i",..: 1 2 3 4 5
```

Если нужно объединить таблицы, имеющие общие колонки, то применяется функция `merge`. Как правило, она используется для объединения таблиц с общими ключами:

```
first <- data.frame(id=1:5, COL=LETTERS[1:5], col=letters[1:5])
secnd <- data.frame(id=1:5, col=letters[1:5])
# объединяем таблицы по колонке id
total <- merge(first,secnd,by="id")
# объединяем таблицы по колонкам id и col
total <- merge(first,secnd,by=c("id","col"))
```

Размеры таблицы определяются следующим образом:

```
length(df1) # длиной таблицы является длина списка, состоящего из колонок
```

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

```
ncol(df1)   # число колонок
```

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

```
nrow(df1)   # число строк
```

```
## [1] 5
```

Выбор элементов таблицы рассмотрим на примерах:

```
df <- data.frame(a=1:5,b=6:10,c=letters[1:5])
df              # вся таблица
```

```
##   a  b c
## 1 1  6 a
## 2 2  7 b
## 3 3  8 c
## 4 4  9 d
## 5 5 10 e
```

```
df[1,]          # 1-я строка
```

```
##   a b c
## 1 1 6 a
```

```
df[,2]          # 2-я колонка
```

```
## [1]  6  7  8  9 10
```

```
df[4,3]         # элемент 4-й строки 3-й колонки
```

```
## [1] d
## Levels: a b c d e
```

```
df[4,c(1,3)]    # элемент 4-й строки из 1-й и 3-й колонок
```

```
##   a c
## 4 4 d
```

Иногда для выбора нужного фрагмента таблицы удобно использовать логические условия:

```
df[df$c == 'd', ] # 4-я строка
```

```
##   a b c
## 4 4 9 d
```

```
df[,df$c == 'a']  # 1-я колонка
```

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

```
df[,df$c == 'd']  # Нет такой колонки!
```

```
## Error in `[.data.frame`(df, , df$c == "d"): undefined
## columns selected
```

Другой способ, позволяющий выделить фрагмент таблицы, – использование функции `subset`.

Для работы с именами колонок, помимо функции `names`, существует отдельная функция:

```
colnames(df)
```

```
## [1] "a" "b" "c"
```

Можно также задать имена строк таблицы. Делается это с помощью функции `rownames()`:

```
rownames(df) <- LETTERS[1:nrow(df)]
```

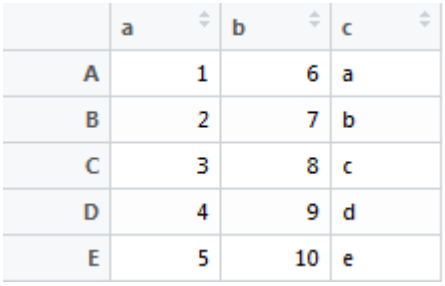
Теперь выбирать элемент таблицы `df` можно так:

```
df["A","b"] # df[1,2]
```

```
## [1] 6
```

Функция `View` отображает таблицу в Excel-подобном виде (рис. 3.1):

```
View(df)
```



	a	b	c
A	1	6	a
B	2	7	b
C	3	8	c
D	4	9	d
E	5	10	e

Рис. 3.1 – Представление таблицы `df` с помощью функции `View` в среде разработки RStudio

Большую таблицу отображать на экране нецелесообразно. Чтобы понять, как она устроена, достаточно рассмотреть её «голову» (`head`) или «хвост» (`tail`). Если данные в таблице упорядочены по времени, это позволит увидеть наиболее свежие или, наоборот, самые старые данные. Число выводимых строк таблицы регулируется параметром `n` и по умолчанию равно 6:

```
df <- data.frame(col1=1:20, col2=letters[1:20])
head(df)
```

```
##   col1 col2
## 1    1    a
## 2    2    b
## 3    3    c
## 4    4    d
## 5    5    e
## 6    6    f
```

```
tail(df, n=3)
```

```
##    col1 col2
## 18   18    r
## 19   19    s
## 20   20    t
```

Функции `head` и `tail` можно также применять для вывода частей векторов, матриц, списков и даже функций.

Функции, применяемые к составным данным

Весьма распространённой причиной использования циклов является необходимость применить один и тот же набор операций (тело цикла) к элементам вектора, списка и т. п. Для решения этой задачи в R существует множество функций, большинство из которых относится к семейству `*apply` (”применить”). Благодаря этим функциям во многих случаях удаётся избежать использования циклов, что делает код более компактным.

Мы рассмотрим три функции из семейства `*apply` (об остальных можно узнать в справке: `?lapply`) и дополняющую их функцию `do.call`.

apply

`apply()` применяет функцию `fun` к строкам или столбцам матрицы и имеет вид:

```
apply(матрица, размерность, fun)
```

Например:

```
# Создадим матрицу M размера 4x4
M <- matrix(1:16,4,4)
M
```

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

```
# Найдём минимальные элементы среди строк M
apply(M,1,min)
```

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

```
# Найдём среднее арифметическое столбцов M
apply(M,2,mean)
```

```
## [1]  2.5  6.5 10.5 14.5
```

```
# Проверим, являются ли столбцы M векторами
apply(M,2,is.vector)
```

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

lapply

`lapply()` (от ‘list apply’) применяет функцию `fun` к элементам списка или таблицы:

```
lapply(список, fun)
```

Пример:

```
x <- list(a = 1, b = 1:3, c = 10:50)
# Вычислим длину каждого элемента
lapply(x, length)
```

```
## $a
## [1] 1
##
## $b
## [1] 3
##
## $c
## [1] 41
```

```
# ... и сумму его значений
lapply(x, sum)
```

```
## $a
## [1] 1
##
## $b
## [1] 6
##
## $c
## [1] 1230
```

Функции можно задавать самому:

```
lapply(c(1,2,4), function(x) x^2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 16
```

Функция `fun` может зависеть и от нескольких аргументов:

```
lapply(c(1,2,4), function(x,y)
                    x^2+y,
                        3 # значение, передаваемое y
       )
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] 19
```

sapply

Функция `sapply` работает аналогично `lapply`, но старается упростить тип результата. Например, если `lapply` возвращает список:

```
li <- lapply(c(1,2,4), function(x) x^2)
str(li)
```

```
## List of 3
##  $ : num 1
##  $ : num 4
##  $ : num 16
```

то `sapply` удаётся получить вектор:

```
ve <- sapply(c(1,2,4), function(x) x^2)
str(ve)
```

```
##  num [1:3] 1 4 16
```

или матрицу:

```
x <- list(1:4, 5:8)
sapply(x, function(x) x^2)
```

```
##      [,1] [,2]
## [1,]    1   25
## [2,]    4   36
## [3,]    9   49
## [4,]   16   64
```

А вот как с помощью `sapply` можно преобразовать хранящиеся в таблице данные к числовому типу:

```
# Преобразуем 1-ю и 3-ю колонки таблицы data:
data[, c(1,3)] <- sapply(data[, c(1,3)], as.numeric)
```

do.call

Если *`apply` применяют заданную функцию к каждому элементу, выполняя таким образом множество вызовов функции, то `do.call` применяет функцию один раз ко всему списку аргументов:

```
do.call(fun, список_аргументов)
```

Примеры:

```
do.call(sum, list(1,2,4,1,2))
# Без do.call понадобилось бы избавиться от списка:
# li <- list(1,2,4,1,2)
# sum(unlist(li))
```

```
## [1] 10
```

```
A <- list(1:3,4:6,7:9)
do.call(rbind,A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Резюме

Списки представляют собой одномерные наборы разнотипных данных. На их основе создаются таблицы, представляющие собой списки колонок. Как правило, данные, которые нам предстоит собирать, будут сохраняться именно в виде таблиц.

Функции семейства `*apply` позволяют применить один и тот же набор операций к элементам матрицы, списка, таблицы и т. п., что позволяет во многих случаях избежать использования циклов.

Глава 4

Управление процессом вычислений

Говоря об управлении вычислительным процессом, обычно имеют в виду условные операторы и циклы.

Циклы

Циклы в R используются сравнительно редко. «Винить» в этом следует векторизацию вычислений. Тем не менее иногда именно циклы являются наиболее простым путём решения задачи.

Синтаксис циклов R очень похож на синтаксис циклов C-подобных языков программирования. Циклы в R могут вкладываться друг друга в соответствии с обычными правилами вложения циклов.

Цикл со счётчиком

Цикл со счётчиком записывается следующим образом:

```
for (i in start:stop) {
  <тело цикла>
}
```

Его удобно применять, когда количество повторений команд, помещённых в тело цикла, известно заранее. Предположим, например, что в банк положена сумма в 1000 долларов сроком на 5 лет с процентной ставкой 12% годовых, и нужно узнать, сколько денег окажется на счету вкладчика по окончании срока вклада.

```
sum0 <- 1000     # исходная сумма
rate <- .12      # процент
num_years <- 5   # срок хранения
sum <- sum0      # текущая сумма
for (i in 1:num_years) {
  sum <- sum + rate*sum0
  print(c(i,sum)) # выводим на печать номер года и накопленную сумму
}
```

```
## [1]    1 1120
## [1]    2 1240
## [1]    3 1360
## [1]    4 1480
## [1]    5 1600
```

Заметим, что «эхо» внутри цикла не работает и для вывода промежуточных данных используется функция `print`.

Решение при помощи цикла – простое и прямолинейное, но не самое короткое. Немного подумав, можно прийти к формуле:

```
sum0*(1+num_years*rate)
```

```
## [1] 1600
```

Оценим время, затраченное на заполнение при помощи цикла элементов динамического массива (вектора `x`):

```
n <- 50000             # число элементов
x <- c()               # создадим пустой вектор
system.time(
  for (i in 1:n) {
    x <- c(x,i)        # дополним вектор x очередным элементом
  }
)
```

```
##   user system elapsed
##  5.348  0.072   5.426
```

Обратите внимание на то, как вкладывается цикл в функцию оценки времени `system.time`.

Аргумент `user` функции `system.time` возвращает процессорное время в секундах, затраченное текущим процессом (сеансом работы с R). `system` – это процессорное время, затраченное операционной системой на выполнение работ, связанных с текущим процессом (открытие файлов, операции ввода/вывода, запуск других процессов и т. п.). Наконец, `elapsed = user+system` содержит полное время работы скрипта.

Использованный нами способ инициализации вектора, очевидно, неэффективен. В первую очередь желательно зарезервировать место под элементы вектора:

```
x <- vector(length = n) # создадим вектор заданной длины
system.time(
  for (i in 1:n) {
    x[i] <- i
  }
)
```

```
##   user system elapsed
##  0.096  0.008   0.103
```

Экономия времени налицо. Однако лучший результат достигается при использовании векторизации:

```
system.time(x <- 1:n)
```

```
##   user system elapsed
##      0      0       0
```

Такая ситуация типична для R: циклы работают медленнее, чем векторизованные функции.

Всё происходит так быстро, что `system.time` не успевает отследить изменения. Для более точной оценки времени выполнения скрипта используем пакет tictoc:

```
library(tictoc) # загрузим пакет tictoc
tic()           # засечём время
x <- 1:n
toc()           # остановим таймер и выведем результат
```

```
## 0.004 sec elapsed
```

О работе с пакетами расширений R мы поговорим в главе 8. Сейчас же, завершая разговор об оценке времени работы скрипта, покажем, как получить информацию об используемой версии R, операционной системе и загруженных пакетах, без которой данные о времени выполнения не имеют смысла:

```
sessionInfo()
```

```
## R version 3.2.3 (2015-12-10)
## Platform: i686-pc-linux-gnu (32-bit)
## Running under: Ubuntu 16.04 LTS
##
## locale:
## [1] LC_CTYPE=ru_RU.UTF-8 LC_NUMERIC=C
## [3] LC_TIME=ru_RU.UTF-8 LC_COLLATE=ru_RU.UTF-8
## [5] LC_MONETARY=ru_RU.UTF-8 LC_MESSAGES=ru_RU.UTF-8
## [7] LC_PAPER=ru_RU.UTF-8 LC_NAME=C
## [9] LC_ADDRESS=C LC_TELEPHONE=C
## [11] LC_MEASUREMENT=ru_RU.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats graphics grDevices utils datasets methods base
##
## loaded via a namespace (and not attached):
## [1] magrittr_1.5 formatR_1.4 tools_3.2.3 htmltools_0.3.5
## [5] yaml_2.1.13 Rcpp_0.12.5 stringi_1.1.1 rmarkdown_0.9.6
## [9] knitr_1.13 stringr_1.0.0 digest_0.6.9 evaluate_0.9
```

Цикл с предусловием

Цикл с предварительным условием используется тогда, когда число повторений тела цикла неизвестно, зато известно условие, которое должно быть выполнено в результате этих повторений.

Тело цикла выполняется до тех пор, пока истинно `логическое условие`:

```
while (логическое_условие) {
 <тело цикла>
}
```

Вновь рассмотрим задачу о банковском вкладе. Только теперь процент будет не простой, а сложный, то есть будет добавляться к исходной сумме вклада. Определим, через сколько лет сумма, накапливаемая таким способом, превысит ту, что удалось скопить с помощью простого процента (`1600`), и сколько именно денег удастся скопить.

```
sum0 <- 1000
rate <- .12
i <- 0
sum <- sum0
while (sum < 1600) {
 sum <- sum + rate*sum
 i <- i + 1
 print(c(i,sum))
}
```

```
## [1]    1 1120
## [1]    2.0 1254.4
## [1]    3.000 1404.928
## [1]    4.000 1573.519
## [1]    5.000 1762.342
```

Естественно, что и в этом случае можно было записать формулу:

```
sum0*(1+rate)^num_years
```

```
## [1] 1762.342
```

Для преждевременного выхода из цикла используется команда `break`. После неё управление передаётся во внешний цикл, команде, следующей за циклом, из которого мы вышли:

```
for (i in 1:3) {
  for (j in 1:3) {
    if (j==2) break
    cat(sprintf(' loop: i=%d j=%d\n',i,j))
  }
  cat(sprintf('break: i=%d j=%d\n',i,j))
}
```

```
##  loop: i=1 j=1
## break: i=1 j=2
##  loop: i=2 j=1
## break: i=2 j=2
##  loop: i=3 j=1
## break: i=3 j=2
```

Команда `next` выполняет переход к следующей итерации цикла.

Условные операторы

Скалярный условный оператор `if`, или, проще говоря, обычный `if`, записывается так:

```
if (логическое_условие) {
  операторы выполняются
  если логическое_условие истинно
} else {
  операторы выполняются
  если логическое_условие ложно
}
```

Конструкции `else-if` в R нет.

Обратите внимание, что если в коде записано:

```
if (condition == TRUE) x <- TRUE
else x <- FALSE
```

то R воспринимает первую строку как закончившуюся. Вторая строка начинается с `else`, который будет истолкован как самостоятельный оператор. Поскольку такого оператора не существует, то появится сообщение об ошибке:

```
##    Error: unexpected 'else' in " else"
```

Чтобы R интерпретировал `else` как часть предшествующего `if`'а, нужно указать ему при помощи фигурных скобок, что `if` ещё не закончился:

```
if (condition == TRUE) {
  x <- TRUE
} else {x <- FALSE}
```

Есть в R и скалярный `switch`, но используется он редко.

Векторный условный оператор `ifelse` выполняет проверку условия сразу для всего вектора и возвращает логический индекс, в соответствии с которым выполняется то или иное действие:

```
ifelse(логическое_условие, выполнить_если_ИСТИНА, выполнить_если_ЛОЖЬ)
```

Пример:

```
x <- c(6:-4)
sqrt(x) # выдаёт предупреждение из-за наличия отрицательных аргументов
```

```
## Warning in sqrt(x): созданы NaN
```

```
##  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000
##  [8]      NaN      NaN      NaN      NaN
```

`NaN` означает "Not-a-Number", то есть результат не является числом.

Значения аргументов вначале необходимо отфильтровать, заменив отрицательные числа какими-нибудь другими значениями (в нашем случае удобно использовать `NA`), и лишь затем приступать к вычислению корня.

```
x1 <- ifelse(x>=0,x,NA) # фильтр для данных
x1
```

```
##  [1]  6  5  4  3  2  1  0 NA NA NA NA
```

```
sqrt(x1)
```

```
##  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000
##  [8]       NA       NA       NA       NA
```

Однако мы снова получим предупреждение в ситуации:

```
ifelse(x>0,sqrt(x),NA)
```

```
## Warning in sqrt(x): созданы NaN
```

```
##  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000       NA
##  [8]       NA       NA       NA       NA
```

– потому что, несмотря на условие, `sqrt()` выполняет операцию над исходным «нефильтрованным» вектором.

Резюме

Циклы в R используются сравнительно редко, поскольку они, как правило, работают существенно медленнее функций `*apply`, выполняющих групповые операции над элементами массивов, списков и таблиц. Синтаксис циклов R мало чем отличается от большинства C-подобных языков программирования.

Особенностью R является наличие векторного условного оператора `ifelse`.

Глава 5

Базовая графика

В этой главе мы рассмотрим основы работы с базовой двумерной графикой R. «Базовой» она названа потому, что реализована в пакете `graphics`, который поставляется вместе с R и подключается автоматически.

Графические возможности R весьма богаты, и их описанию посвящён ряд монографий. Так, в R существуют пакеты `lattice` и `ggplot2`, возможности которых даже шире, чем у `graphics`, и которые используют другую идеологию построения графиков. Список книг с описанием графических возможностей R вы найдёте в конце этой главы в разделе «Резюме и ссылки».

Напомним, что графики в R выводятся в различные графические устройства: окна, если вывод осуществляется на экран, или файлы, если данные сохраняются на диске.

Функции низкого и высокого уровней

Построим график синусоиды, координаты которой хранятся в векторах `x` и `y`:

```
x <- seq(-pi,pi,.1)
y <- sin(x)
```

График можно составить из отдельных элементов (рис. 5.1):

```
plot.new()
plot.window(xlim = c(-pi,pi), ylim = c(-1,1))
points(x,y)
axis(1)
axis(2)
box()
title(xlab = "x")
title(ylab = "y")
```

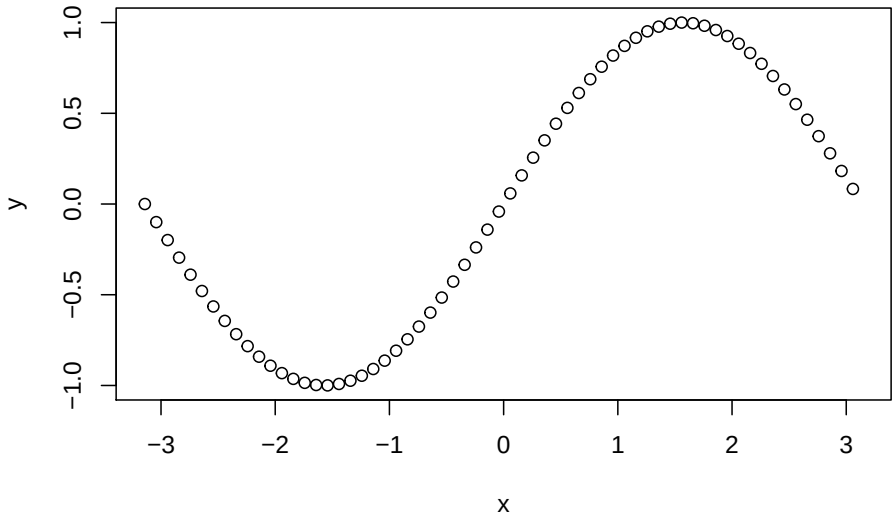


Рис. 5.1 – График синусоиды, построенный из отдельных элементов

Функции, реализующие тот или иной элемент графика, называются функциями низкого уровня, или *низкоуровневыми*. В данном случае это:

- `plot.new`, которая создаёт новое графическое окно;
- `plot.window` – задаёт пределы изменения x- и y-координат на графике и другие графические параметры (те же, что и в функции `par()` – см. далее);

- `points` – строит график в виде точек;
- `axis` – добавляет на график оси координат: 1 – снизу, 2 – слева, 3 – сверху, 4 – справа;
- `box` – строит вокруг графика рамку;
- `title` – выводит пояснения к графикам. Например: параметры `xlab`, `ylab` задают обозначения осей координат, а `main` – заголовок графика.

Тот же график можно построить при помощи всего лишь одной функции:

```
plot(x,y)
```

`plot` представляет собой пример функций высокого уровня, которые нацелены на реализацию наиболее распространённых видов графиков.

Как правило, в функциях высокого и низкого уровней используются одни и те же графические параметры. Вот, например, как можно оформить пояснения к графику при помощи параметров `main`, `xlab`, `ylab` функции `plot` (рис. 5.2):

```
plot(x, y,
     main = "2d-Graph",
     xlab = "Axis X",
     ylab = "Axis Y"
     )
```

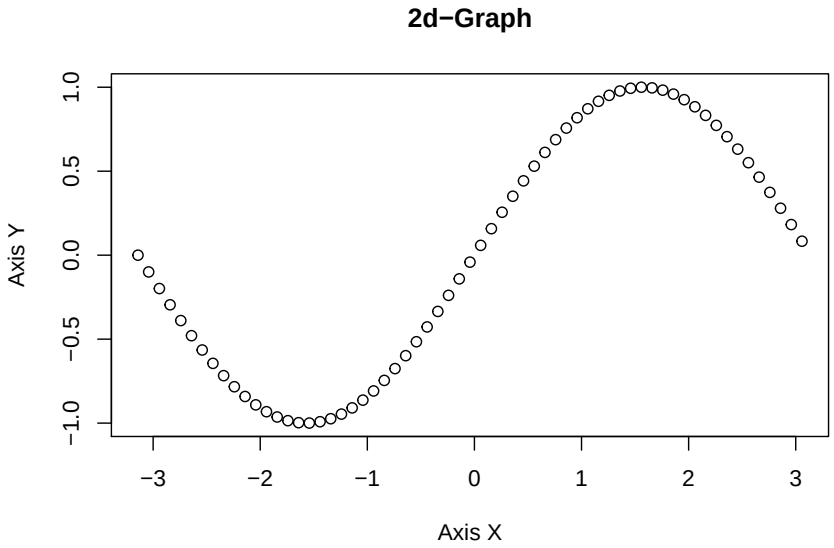


Рис. 5.2 – График синусоиды, построенный функцией `plot`

По умолчанию точки данных изображаются в виде незакрашенных кружков чёрного цвета. Символы для отображения точек (маркеры) настраиваются параметром `pch` (plot character), цвет – параметром `col`.

Пусть, например, точки данных будут отображаться красными «звёздочками» (рис. 5.3):

```
plot(x,y,pch="*",col="red")
```

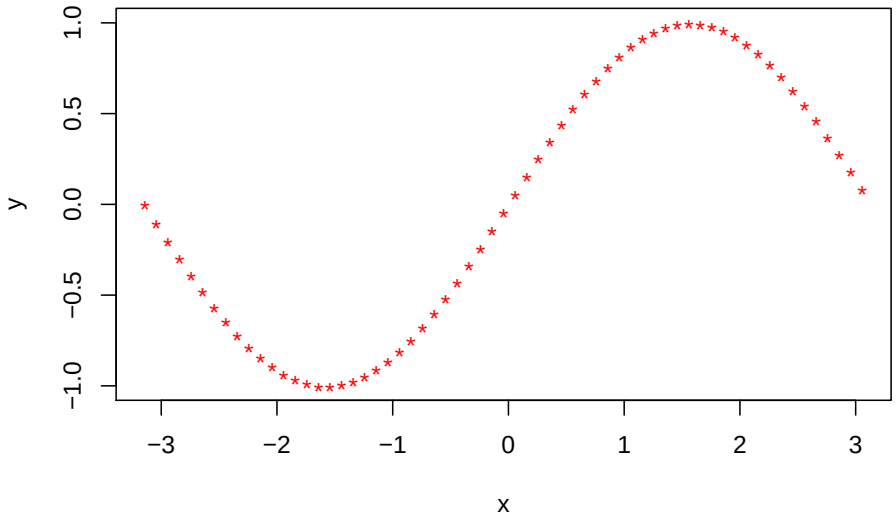


Рис. 5.3

Маркеры можно задавать не только при помощи строк, но и по номерам. Первые двадцать видов маркеров показаны на рис. 5.4.

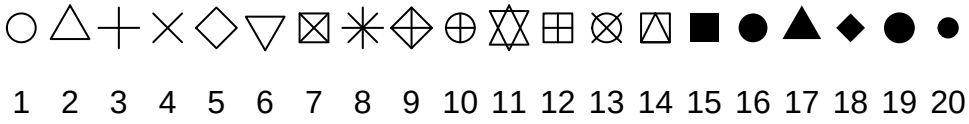


Рис. 5.4 – Маркеры

Помимо вида маркера в целом, можно настраивать отдельно цвет заполнения и цвет границы маркера.

Цвета можно задавать по номерам (рис. 5.5), строкам названий цветов (вроде «red») и шестнадцатеричным строкам вида #RRGGBB (как в языке HTML).

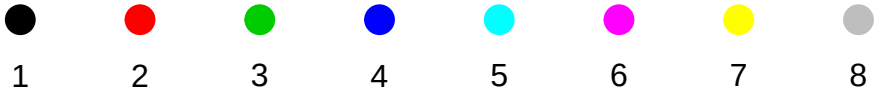


Рис. 5.5 – Цвета и их номера

Названия цветов, поддерживаемых R, возвращает функция `colors`. В настоящее время таких цветов 657.

Функция `col2rgb` преобразует название цвета в его RGB-представление, а функция `rgb` вычисляет по этому представлению строку #RRGGBB.

Строить ли кривые на графике при помощи точек, линий или и так, и этак, определяет параметр `type`. Кроме того, для этой цели существуют специальные низкоуровневые функции, например `lines` (рис. 5.6):

```
x <- seq(-pi,pi,.3)
y <- sin(x)
plot(x,y,
     col="#00FF00",
     type="b"      # тип отображения кривой
                   #  "p" - точки (по умолчанию),
                   #  "l" - линия,
                   #  "b" - точки и линия (both))
    )
lines(x+.3, y, col="#0000FF")
```

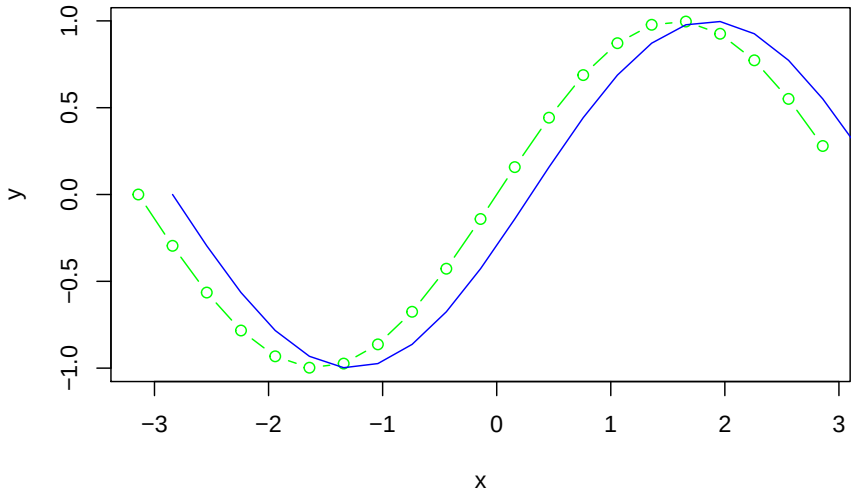


Рис. 5.6

Тип линии (сплошная, штриховая, пунктирная и т. п.) определяется параметром `lty` (`solid`, `dashed`, `dotted`… или по номерам). Параметр `lwd` задёт толщину линии в пунктах (рис. 5.7).

```
x <- seq(-pi,pi,.1)
y <- sin(x)
plot(x,y,
     type="l",      # линия
     lty="dashed", # штриховая
     lwd=2         # толщиной 2 пт.
     )
```

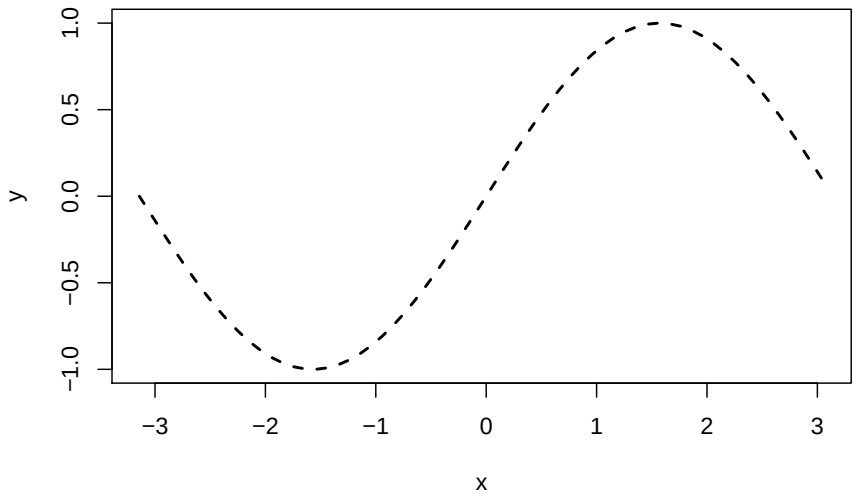


Рис. 5.7

Значение параметра `type="n"` позволяет создать пустой график (рис. 5.8).

```
plot(x, y, type="n")
```

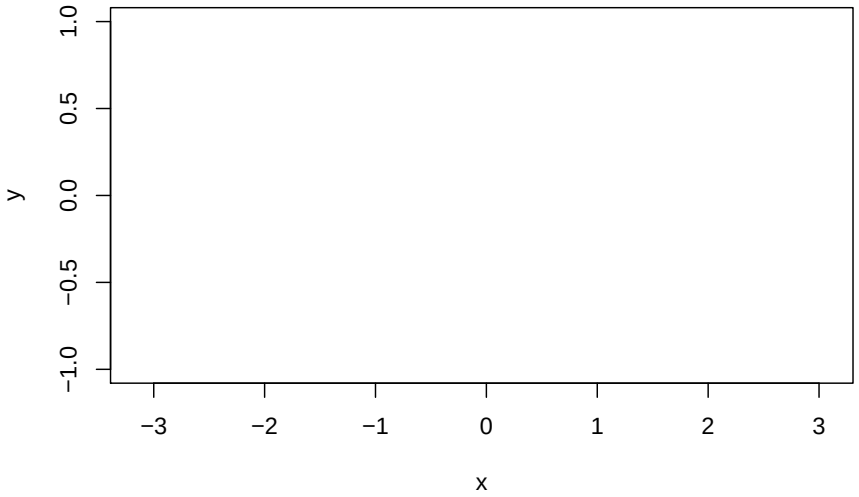


Рис. 5.8

Глобальные и локальные параметры графиков

Для глобальной настройки параметров графиков используется функция `par`. Вызванная без аргументов, она позволяет узнать текущие значения всех графических параметров (в R версии 3.3.1 их 72).

Указав в качестве аргумента строку с именем графического параметра, получим значение одного этого параметра:

```
par("col")
```

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

По умолчанию для рисования используется чёрный цвет.

Покажем, чем отличаются локальные параметры, которые задаются в конкретной функции построения графика (`plot`, `points`, `lines`), от глобальных параметров, устанавливаемых в `par` (рис. 5.9).

```
x <- seq(-pi,pi,.3)
y <- sin(x)
par(col="red")
plot(x,y,type="b")
lines(x+.3, y)
points(x-.3, y, col="blue")
```

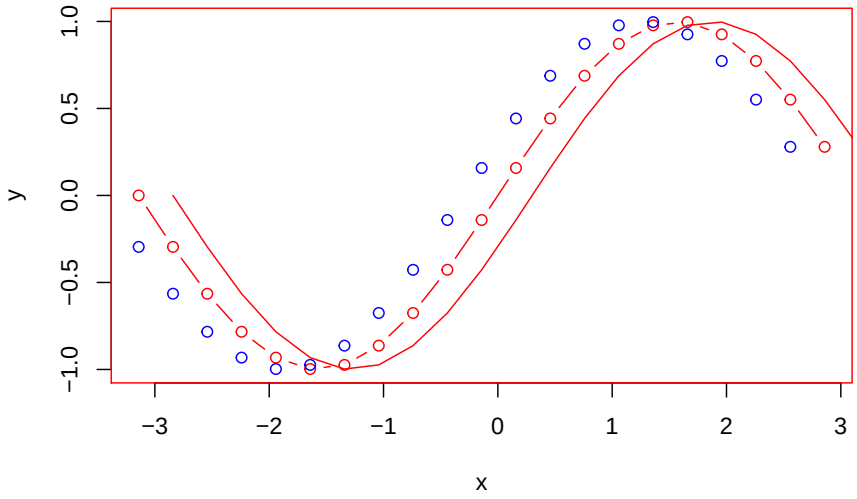


Рис. 5.9

Функция `par` устанавливает единый для всех графиков цвет вывода кривых и осей координат (красный). Локальное значение параметра (синий цвет последнего графика) ожидаемо перекрывает действие глобального.

Действие настроек, сделанных в `par`, продолжается до конца сеанса работы с R. Чтобы восстановить предыдущие значения графических параметров, используют следующий приём:

```
par()              # узнать текущие настройки
old_par <- par()   # сделать их копию
par(...)           # создать новые настройки
<plot>             # построить необходимые графики
par(old_par)       # восстановить прежние настройки
```

Список наиболее употребительных графических параметров приведен в табл. 5.1.

Таблица 5.1 – Наиболее употребительные графические параметры

Обозначение	Значение
`main`	Заголовок графика
`xlab`, `ylab`	Подписи к осям x и y
`xlim`, `ylim`	Диапазон изменения данных по осям x и y
`pch`	Маркер, которым отображается точка данных
`cex`	Размер маркера
`col`	Цвет
`lty`	Тип линии (`solid`, `dashed`, `dotted`...)
`lwd`	Толщина линии

Легенда

Функция `legend` добавляет к текущему графику легенду – список пояснений, касающихся построенных на графике кривых. Рассмотрим построение легенды к графику на следующем примере (рис. 5.10):

```
# Строим три графика
x <- seq(-pi, pi, len=20)
plot(x, sin(x), type="l", ylim=c(-1.2, 2.1), col=2, lty=4, lwd=2)
points(x, cos(x), pch=8, col=3)
lines(x, tan(x), type="b", lty=1, pch=4, col=4)
# Строим легенду
legend(-1.5, 2.1,                # координаты левого верхнего угла
       c("sin", "cos", "tan"), # подписи
       col = c(2, 3, 4),       # цвет линии в легенде
       text.col = "green4",    # цвет подписи
       lty = c(4, -1, 1),      # тип линии в легенде
       pch = c(NA, 8, 4),      # маркер в легенде
       bg = "gray90"           # цвет фона в легенде
       )
```

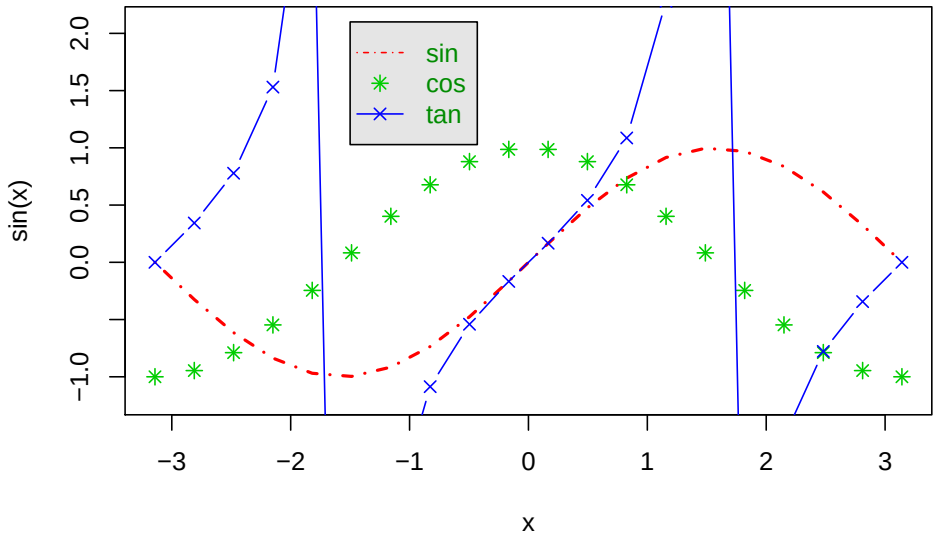


Рис. 5.10

Параметр `ylim` в `plot` нужен, чтобы качественно отобразить график тангенса. Иначе диапазон изменения y-координаты будет рассчитываться автоматически по значениям синуса и график тангенса окажется сплюснут.

Каждое из значений `-1` и `NA` в параметрах `lty` и `pch` легенды означает отсутствие соответствующего элемента: для `lty` – это отсутствие линий у графика, построенного точками; для `pch` – отсутствие маркеров у кривой, построенной с помощью линий.

Комбинации графиков

Построим два графика в общих осях координат (рис. 5.11):

```
x <- seq(-pi,pi,.1)
y1 <- sin(x)
y2 <- dnorm(x)
plot(x, y1, type="l", col="red")
lines(x, 2*y2, col="green")
grid()
```

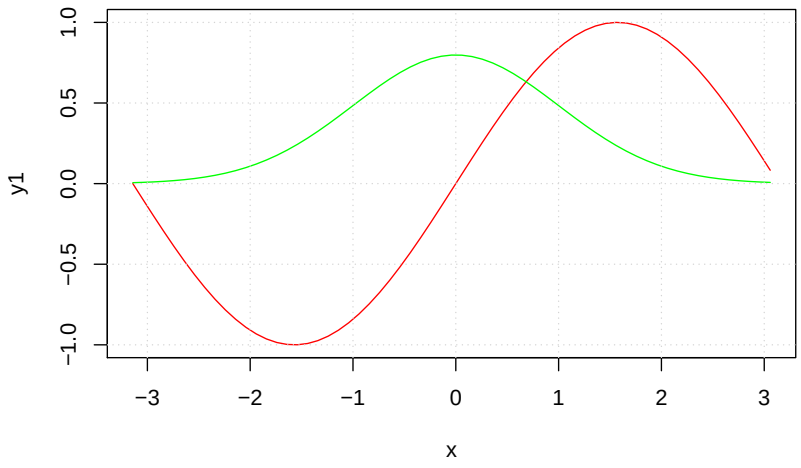


Рис. 5.11

Функции `lines` и `points`, в отличие от `plot`, не перерисовывают содержимое графического окна, а просто добавляют в него график.

Функция `grid` строит на графике сетку.

Построить несколько графиков в общем окне, но разных осях координат, можно при помощи параметра `mfrow` (или `mfcol`):

```
mfrow = c(число_строк,число_столбцов)
```

Расположение графиков в графическом окне можно представить в виде матрицы. Допустим, что в одной строке мы хотим разместить два графика. Выглядеть это будет так (рис. 5.12):

```
par(mfrow=c(1,2))
plot(x,y1,mfrow=c(1,2))
plot(x,y2)
```

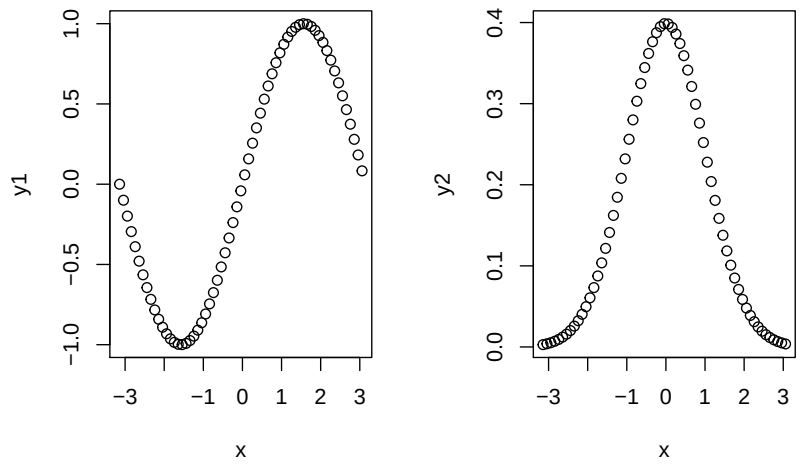


Рис. 5.12

Заметим, что поскольку `mfrow` является глобальной настройкой, то все последующие графики в текущем сеансе работы будут отображаться по два в строке. Чтобы вернуться к обычному способу отображения графиков, нужно восстановить с помощью `par` настройки, принятые по умолчанию:

```
par(mfrow=c(1,1))
```

Чтобы построить каждый график в отдельном окне, нужно перед очередным вызовом `plot` создавать новое графическое окно (устройство).

```
# dev.new() - при первом вызове plot создаётся новое графическое устройство
plot(x,y1) # ... и дополнительно вызывать dev.new() не нужно
dev.new()  # Создать новое графическое устройство
plot(x,y2) # Вывести в него график
```

Кроме того:

- `dev.off()` – закрывает текущее графическое окно;
- `graphics.off()` – закрывает все графические окна в сессии.

Графики функций

Графики функциональных зависимостей вида $y = f(x)$ строятся при помощи функции `curve` (рис. 5.13):

```
curve(x^2, from=-1, to=1, main="sqr(x)", ylab="y = x^2")
```

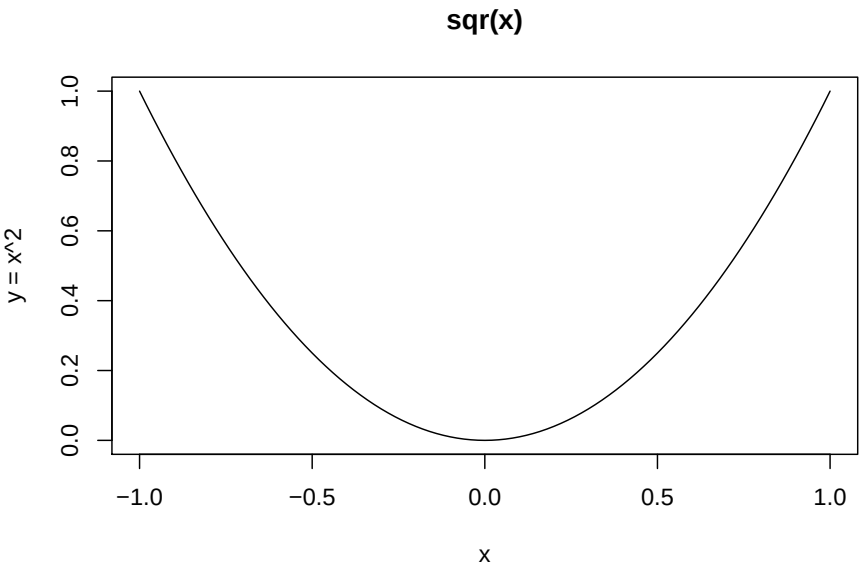


Рис. 5.13

Экспорт в файлы

Справка по доступным графическим устройствам:

```
?Devices
```

Узнать, в каких форматах можно сохранить график, позволяет также функция `capabilities`:

```
capabilities()
```

```
##        jpeg         png        tiff       tcltk         X11        aqua
##        TRUE        TRUE        TRUE        TRUE       FALSE       FALSE
##    http/ftp     sockets      libxml        fifo      cledit       iconv
##        TRUE        TRUE        TRUE        TRUE       FALSE        TRUE
##         NLS     profmem       cairo         ICU long.double     libcurl
##        TRUE        TRUE        TRUE        TRUE        TRUE        TRUE
```

Она, в частности, возвращает имена функций, настраивающих параметры вывода в тот или иной графический формат. Вот как происходит сохранение графика в файл формата PNG:

```
# Описание устройства вывода
png("myfile.png", width=420, height=340, units="px")
# Вывод графика в устройство
plot(x,y1)
# Закрыть устройство
dev.off()
# Теперь можете посмотреть полученный файл
```

Резюме и ссылки

Мы лишь прикоснулись к графическим возможностям R. Посмотрим, что можно почитать, чтобы узнать о них больше.

- *Кабаков Р. И.* R в действии. Анализ и визуализация данных в программе R. М.: ДМК Пресс, 2014.

Глава 3 этой книги дополнит и расширит материалы главы. Тема графики развивается в главах 6 и 11, а глава 17 представляет собой краткое знакомство с пакетами `lattice` и `ggplot2`.

- *Зарядов И. С.* Введение в статистический пакет R: типы переменных, структуры данных, чтение и запись информации, графика. М.: Изд-во РУДН, 2010.

Глава 7 посвящена базовой графике.

Всего понемногу: базовая графика, `lattice` и `ggplot2` в следующих двух книгах:

- *Murrel P.* R Graphics. Chapman & Hall/CRC, 2005.
- *Hilfiger Jh. J.* Graphing Data with R. O'Reilly Media, 2016.

Только по `lattice`:

- От разработчика пакета: *Sarkar D.* Lattice: Multivariate Data Visualization with R. Springer-Verlag New York, 2008.

По `ggplot2`:

- От разработчика: *Wickham H.* ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2009.
- Сборник рецептов: *Chang W.R.* R Graphics Cookbook. O'Reilly Media, 2013.

Глава 6

Функции

Функция – это фрагмент кода, предназначенный для решения какой-либо конкретной задачи. Мы уже знакомы с большим количеством готовых функций R и теперь научимся создавать функции сами, чтобы расширить возможности этого языка.

Создание функций

Синтаксис функции имеет вид:

```
function(список_аргументов) {
  тело функции
}
```

Например:

```
function(x) {
  x^2
}
```

или короче:

```
function(x) x^2
```

У функции должно быть имя, чтобы её можно было вызвать:

```
sqr <- function(x) x^2
sqr(2)
```

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

Но можно использовать и анонимные функции:

```
(function(x) x^2)(3)
```

```
## [1] 9
```

Обратите внимание, что присваивание имени функции выглядит так же, как присваивание значения переменной. И действительно, переменная `sqr` теперь относится к классу функций:

```
class(sqr)
```

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

Вы можете проанализировать структуру функции, используя:

- `имя_функции` – без круглых скобок оно возвращает код функции;
- `body()` – возвращает код, помещённый в теле функции;
- `formals()` – возвращает список формальных параметров.

```
sqr
```

```
## function(x) x^2
```

```
body(sqr)
```

```
## x^2
```

```
formals(sqr)
```

```
## $x
```

По умолчанию функция в R возвращает то значение, которое возвращается последней строкой её кода. Но для определённости лучше использовать `return()`:

```
sqr <- function(x) {
  return(x^2)
}
```

Список аргументов функции может быть пуст, и тогда после имени функции указываются лишь круглые скобки `()`. В списке можно задавать значения аргументов по умолчанию в виде `x = a`, а также формировать список аргументов переменной длины, добавляя `...` в конце списка.

Таким образом, открывается широкий простор для создания функций-«обёрток». Например, нам нужна функция, строящая графики линиями красного цвета. Все остальные её параметры – такие же, как у стандартной `plot`.

Создадим функцию-обёртку над `plot`. То есть функцию, основу которой составляет `plot`, но с некоторыми доработками – в данном случае с фиксированным значением параметра цвета (`col`):

```
plot.red <- function(x, y, ...) {
  plot(x, y, col = "red", ...)
}
```

Вот как это работает (рис. 6.1):

```
x <- seq(-pi,pi,.1)
y <- sin(x)
plot.red(x,y)
```

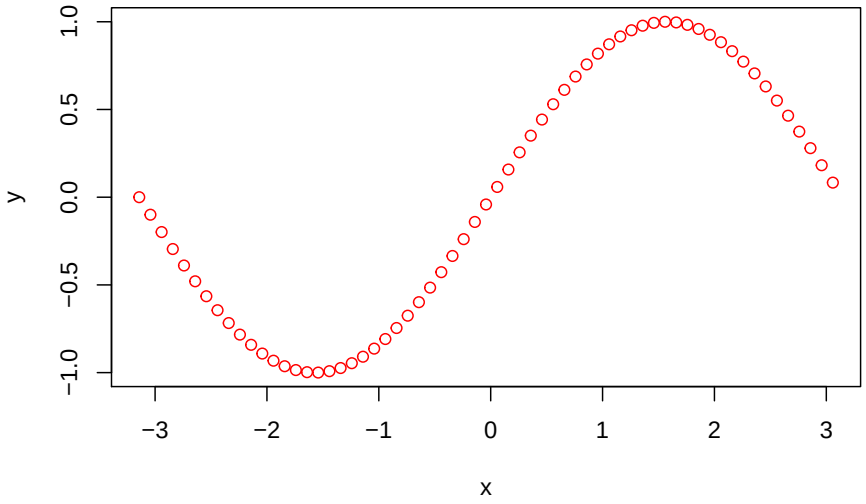


Рис. 6.1

Функции в R могут возвращать несколько объектов, относящихся к разным типам данных. Для этого такие объекты нужно предварительно поместить в список:

```
sqr <- function(x) {
  val <- x^2
  txt <- "Squaring"
  ret <- list(value=val,text=txt)
  return(ret)
}
sqr(2)
```

```
## $value
## [1] 4
##
## $text
## [1] "Squaring"
```

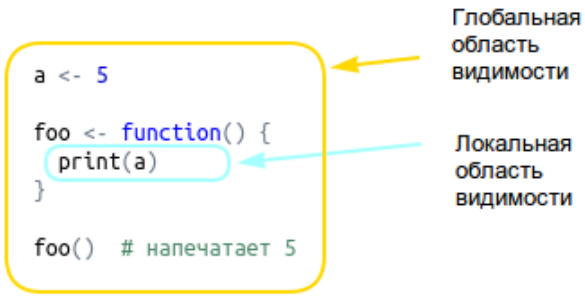


Рис. 6.2 – Глобальная и локальная области видимости переменных

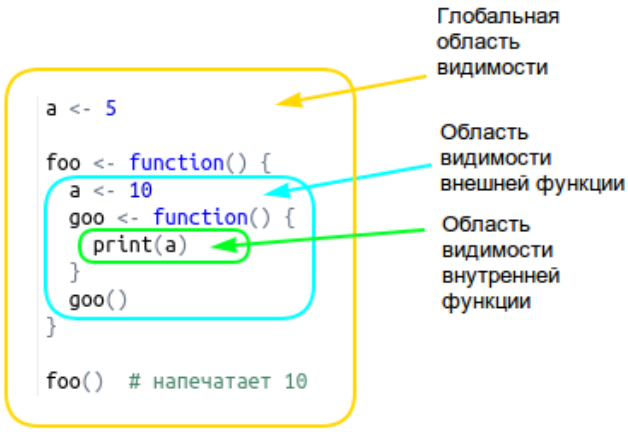


Рис. 6.3 – Вложение областей видимости

Локальные и глобальные переменные. Области видимости

Переменные, созданные внутри функции, являются *локальными*, то есть их значения доступны (видимы) только внутри данной функции или вложенных в неё функций. Соответственно, и функция «видит» переменные, заданные вне её пределов:

```
a <- 5
foo <- function() {
  b <- 10  # локальная переменная foo
  print(a) # foo видит a
  print(b) # foo видит b
}
foo()
```

```
## [1] 5
## [1] 10
```

Хотя переменная `a` задана вне функции и не передаётся в неё через входные параметры, проблем с ней не возникает (рис. 6.2). А вот локальная переменная `b` в основной программе не видна:

```
print(b)   # b в основной программе не определена
```

```
## Error in print(b): объект 'b' не найден
```

Области видимости вкладываются друг в друга (рис. 6.3).

Глобальные переменные внутри функции можно задать при помощи оператора присваивания `<<-`:

```
a <- 5
foo <- function () {
   b <<- 10
   # другой вариант:
   # assign("b", "new", envir = .GlobalEnv)
}
foo()
b  # теперь b видима из основной программы
```

```
## [1] 10
```

Диагностические сообщения

Для вывода диагностических сообщений используются функции:

- `message` – выводит сообщение;
- `warning` – выводит сообщение об ошибке, не останавливая выполнения команд;
- `stop` – выводит сообщение об ошибке и останавливает выполнение.

Функции в качестве аргументов

В R функцию можно передавать в качестве аргумента в другие функции.

Допустим, у вас есть функция `toPercent`, преобразующая вектор чисел – процентов – в вектор строк со знаком '%'. В ней используется округление, выполняемое с помощью функции `round`:

```
toPercent <- function(x) {
  percent <- round(x * 100)
  paste(percent, "%", sep = "")
}
a <- c(0.15, 1.26, 0.98, 1.02)
toPercent(a)
```

```
## [1] "15%"  "126%" "98%"  "102%"
```

Вдруг, неизвестно по какой причине, вам понадобилось выполнить округление при помощи другой функции – `signif`. Эта функция позволяет указывать, после какого знака следует округлять число.

Что делать? Писать новую функцию `toPercent` не слишком разумно, так как она будет почти точной копией предыдущей. Вместо этого слегка изменим `toPercent`:

```
toPercent <- function(x, FUN = round, ...) {
  percent <- FUN(x * 100, ...)
  paste(percent, "%", sep = "")
}
```

Если не указывать явно аргумент `FUN`, то новая функция будет работать так же, как прежняя, – ведь по умолчанию используется всё тот же `round`. Покажем это:

```
a <- c(0.15, 1.26, 0.98, 1.02)
toPercent(a)
```

```
## [1] "15%"  "126%" "98%"  "102%"
```

Вместо многоточия `'...'` можно указывать дополнительные аргументы, которые будут передаваться функции `FUN`. Например:

```
toPercent(a, FUN = signif, digits = 3)
```

```
## [1] "15%"  "126%" "98%"  "102%"
```

Здесь:

- R присваивает код функции `signif` аргументу `FUN`, и функция `FUN()` становится точной копией `signif()`;
- `toPercent` принимает аргумент `digits` (число знаков, после которых `signif` выполняет округление) и передает его в `FUN()`.

Обратите внимание, что при передаче функции в качестве аргумента используется только имя функции, но не скобки – `signif`, а не `signif()`. Последний вариант R трактует как вызов функции, к тому же ошибочный из-за отсутствия аргументов.

Функция может быть неявным аргументом другой функции. Так, если в вызове функции `g()`

```
g <- function(x, FUN = NULL) {
  if (missing(FUN))
    FUN <- function(x) x^2
  FUN(x)
}
```

используется только первый аргумент `x`, без которого никак не обойтись, то «скрытая» функция `FUN` возводит значение `x` в квадрат

```
g(2)
```

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

`missing` проверяет, не пропущен ли аргумент при вызове функции, и если это так, то запускает `FUN`.

Функцию `FUN` можно задать явно. Например, в виде анонимной функции:

```
g(2, FUN = function(x) x^3)
```

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

Функциональное программирование

По сути, R является языком функционального программирования. Поэтому практически всё в нём делается при помощи функций. Например, элементы векторов выбираются при помощи функции-квадратной скобки:

```
x <- c(1,2,3)
x[1]
```

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

Такая форма записи привычна, но функцию-квадратную скобку можно переписать и в более «функциональном» виде:

```
`[`(x,1)
```

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

За следующей записью:

```
`+`(1,`*`(2,3))
```

кроется обычное

```
1+2*3
```

а операцию присваивания можно выполнить и так:

```
assign("a",1)
a
```

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

Создадим и мы какой-нибудь оператор. Так, в R нет готового оператора +=. Ликвидируем этот пробел:

```
`%+=%` <- function(o1,o2) eval.parent(substitute(o1 <- o1 + o2))
x <- 1
x %+=% 2
x
```

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

Резюме

В R вы можете присваивать функции как значения переменных, передавать их в виде аргументов в другие функции, хранить функции в списках, создавать функции внутри функций и даже возвращать в качестве результата выполнения функции.

К сожалению, рассказ об этом уведёт нас слишком далеко от темы сбора данных. Поэтому адресую вас к книге Хэдли Уикэма:

- *Wickam H.* Advanced R. CRC Press, 2015.

Уикэм разработал для R несколько весьма популярных пакетов, в частности `ggplot2` и `rvest`. Последний мы планируем использовать для извлечения данных из веб-страниц.

Функциональному программированию посвящена вторая часть книги, но и кроме него там есть много интересного.

Глава 7

Факторы и даты

Помимо количественных данных, мы будем иметь дело с данными иного рода, которые нельзя представить в виде чисел. Например, цвета светофора – «красный», «жёлтый» и «зелёный» – следуют в определённом порядке, отличном от алфавитного[3]. Это сближает их с числами, однако сложение или вычитание названий цветов лишено всякого смысла. Для таких данных в R есть специальный тип данных – факторы.

Кроме того, нам предстоит научиться работать с датами. Это малозаметный, но весьма полезный тип данных, поскольку многие ценные сведения, например финансовые отчёты и биржевые сводки, привязаны к конкретным моментам времени.

Категориальные данные

Большинство видов данных, которые мы рассматривали до сих пор, – числовые и логические – так или иначе можно представить при помощи чисел. Числа можно упорядочивать и выполнять над ними арифметические операции. Но есть данные, с которыми так поступить не получится.

Рассмотрим, например, список товаров в магазине. Число наименований товаров (категорий) всегда ограничено. При этом каждый товар относится только к одной категории. Мы можем отличить `"йогурт"` от `"велосипед"`, можем определить количество йогуртов и велосипедов в нашем магазине. Мы даже можем упорядочить наименования товаров по алфавиту. Но нет смысла складывать йогурты и велосипеды. И даже если каждой категории товара присвоить числовой идентификатор, то складывать такие числа всё равно будет бессмысленно.

Итак, существуют данные, которые обычно представляют собой строковые значения из ограниченного набора категорий (названия городов, ФИО сотрудников, марки автомобилей и т. п.) и для которых нельзя ввести алгебраические операции. Такие данные называются *категориальными данными*, или *факторами*.

Но если данные представляются в виде строк, то почему бы и не обойтись строками, не вводя нового типа данных? Посмотрим, какие преимущества даёт нам использование факторов.

Пусть у нас есть три вида фруктов и мы хотим подсчитать количество фруктов каждого вида и построить график.

```
fruits <- c("apple", "orange", "orange", "banana", "apple", "orange")
is.character(fruits)
```

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

```
is.vector(fruits)
```

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

Пока всё идёт хорошо. Но если мы захотим узнать количество фруктов одного вида, то нам придётся считать количество вхождений строки с наименованием этого вида фруктов в вектор `fruits`. Сделать это несложно, но всё же… И если понадобится построить график, например столбчатую диаграмму «вид фруктов – количество», то нам также придётся повозиться.

А что факторы? Для начала построим из вектора `fruits` одноимённый фактор:

```
fruits <- as.factor(fruits)
```

Теперь это больше не строковый вектор – это фактор:

```
is.character(fruits)
```

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

[3] Строки упорядочиваются в алфавитном или *лексикографическом* порядке.

```
is.vector(fruits)
```

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

```
is.factor(fruits)
```

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

Создавать факторы «с нуля» можно функцией `factor`.

Вернёмся к фруктам. Их у нас три категории:

```
levels(fruits)
```

```
## [1] "apple"  "banana" "orange"
```

Функция `levels` позволяет узнать или установить число категорий, или *уровней*, данного фактора. Сделать это можно и при создании фактора, управляя параметром `levels` в `factor()`. Есть в этой функции и возможность упорядочить категории (`ordered`), а также ввести для них специальные метки (`labels`).

«Всё это чудесно, – скажет нетерпеливый читатель, – но как насчёт решения задачи?» А она уже решена.

Вот количество фруктов каждого вида:

```
table(fruits)
```

```
## fruits
##  apple banana orange
##      2      1      3
```

А вот и диаграмма (рис. 7.1):

```
plot(fruits)
```

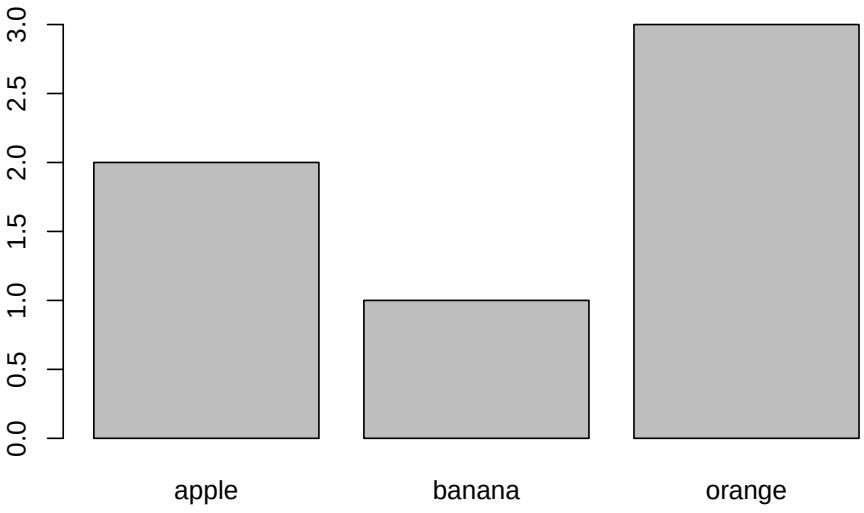


Рис. 7.1

Функция `which.max` позволяет определить уровень, встречающийся чаще всего:

```
which.max(table(fruits))
```

```
## orange
##      3
```

Дата и время

Работать с датами… непривычно. Даты нельзя складывать, но их можно вычитать, отвечая на вопрос: «Через сколько дней (часов, минут, …) после одного события произошло другое событие?» При этом необходимо учитывать возможные различия в часовых поясах. К датам можно добавлять и вычитать из них целые числа. Даты могут быть представлены в разных форматах… В общем, для работы с датами существует специальный тип данных – `Date`.

Создадим переменную типа `Date` при помощи функции `as.Date`, указав в качестве даты День космонавтики:

```
cd <- as.Date("1961-04-12")
str(cd)
```

```
##  Date[1:1], format: "1961-04-12"
```

Какой день недели это был?

```
weekdays(cd)
```

```
## [1] "среда"
```

Какой месяц?

```
months(cd)
```

```
## [1] "Апрель"
```

А какой квартал года?

```
quarters(cd)
```

```
## [1] "Q2"
```

Какая дата наступила через 7 дней?

```
cd + 7
```

```
## [1] "1961-04-19"
```

Сколько дней оставалось до старта Германа Титова?

```
as.Date("1961-08-6") - cd
```

```
## Time difference of 116 days
```

Функция `Sys.Date` возвращает текущую дату, а функция `Sys.time` – текущие дату и время.

```
Sys.Date()
```

```
## [1] "2016-08-12"
```

```
Sys.time()
```

```
## [1] "2016-08-12 11:06:36 EEST"
```

Здесь `EEST` – восточно-европейское летнее время (Eastern European Summer Time, EEST). Оно соответствует времени Гринвичского меридиана (GMT) плюс 3 часа: EEST = GMT + 3.

`Sys.timezone()` возвращает часовой пояс, установленный в локальных настройках операционной системы (локали).

По умолчанию число, месяц и год разделяются в строке даты символами `'-'`, иначе выражение не будет истолковано как дата. Формой представления даты можно управлять при помощи параметра `format`. Виды форматов даты и времени приведены в табл. 7.1 и 7.2.

Таблица 7.1 – Форматы представления даты

Формат	Описание
`%y`	Год без указания столетия
`%Y`	Год с указанием столетия
`%m`	Месяц в виде числа (от 01 до 12)
`%b`	Сокращённое название месяца в соответствии с текущей локалью
`%B`	Полное название месяца в текущей локали
`%d`	День месяца (от 1 до 31)
`%a`	Сокращённое название дня недели, в соответствии с текущей локалью
`%A`	Полное название дня недели в текущей локали
`%w`	День недели в виде числа (от 0 до 6, где Sunday соотвествует 0)

Таблица 7.2 – Форматы представления времени

Формат	Описание
`%I`	Часы (от 01 до 12)
`%H`	Часы (от 00 до 23)
`%M`	Минуты (от 00 до 59)
`%S`	Секунды (от 00 до 61)
`%p`	индикатор AM/PM

Пусть число, месяц и год в исходных данных разделены пробелами:

```
as.Date("12 4 1961", format="%d %m %Y")
```

```
## [1] "1961-04-12"
```

Если бы мы не указали нужный формат, то получили бы сообщение об ошибке:

```
as.Date("12 4 1961")
```

```
## Error in charToDate(x): character string is not in a
## standard unambiguous format
```

Добавим к дате время запуска космического корабля «Восток-1» – 9 ч. 7 мин. по московскому времени:

```
# Запишем строку с датой и временем
cd_datetime <- "12 Апрель 1961 09:07:00"
# Зададим формат представления даты
cd_format <- "%d %B %Y %H:%M:%S"
# Выясним название часового пояса (оно зависит от операционной системы)
tz_names <- OlsonNames() # Выберем нужный часовой пояс из tz_names
# Воспользуемся функцией as.POSIXct, которая похожа на as.Date,
# но позволяет дополнительно задавать часовой пояс tz (time zone)
cd <- as.POSIXct(cd_datetime, format = cd_format, tz = "Europe/Moscow")
cd
```

```
## [1] "1961-04-12 09:07:00 MSK"
```

Сменим формат представления даты на `Число/Месяц/Год`, для чего используем функцию `format`:

```
format(cd, "%d/%m/%y")
```

```
## [1] "12/04/61"
```

А теперь вернёмся к `cd_format`:

```
format(cd, cd_format)
```

```
## [1] "12 Апрель 1961 09:07:00"
```

Иногда может понадобиться извлечь из даты какой-то определённый элемент, например минуты. В этом случае дату необходимо вначале преобразовать в тип данных `POSIXlt`, а затем извлечь из неё минуты (`min`):

```
# преобразуем дату к типу POSIXlt
cd <- as.POSIXlt(cd)
str(cd)
```

```
##  POSIXlt[1:1], format: "1961-04-12 09:07:00"
```

```
# и извлечём из неё минуты
cd$min
```

```
## [1] 7
```

Резюме

Категориальные данные – это данные, значения которых делятся на категории. Такие данные отображают качественную информацию об объекте, к которой не применимы арифметические операции. Для хранения категориальных данных в R существует специальный тип данных – факторы.

R позволяет манипулировать датами, представленными в разных форматах, с учётом часовых поясов. Для работы с датами существуют типы данных `Date`, `POSIXct` и `POSIXlt`.

Глава 8

Пакеты

Привлекательной стороной R является наличие у него многочисленных пакетов, расширяющих возможности языка. Среди этих пакетов есть как библиотеки, созданные специально для R, так и интерфейсы к сторонним библиотекам и сервисам (например, к API социальных сетей). Таким образом, пользователь R, занимающийся сбором и анализом данных, может, не покидая R, применять лучшие (или наиболее известные) программные инструменты, которые только существуют в этой области. При этом зачастую одна и та же задача может решаться разными способами и при помощи разных пакетов.

Чтобы понять, насколько велико разнообразие средств, оказавшихся в нашем распоряжении, достаточно сказать, что в основном хранилище R – CRAN – доступно для загрузки более 10000 пакетов.

Оборотной стороной такого разнообразия является проблема выбора нужного пакета. К счастью, в R существует несколько средств, позволяющих решить и эту задачу.

Установка и загрузка

Пакет (package) в R представляет собой набор функций, данных и скомпилированного кода, предназначенный для решения какой-то определённой задачи.

Для установки пакета наберём в командном окне R:

```
install.packages("имяПакета")
```

Чтобы использовать возможности пакета, его нужно загрузить в память:

```
library(имяПакета)
```

В принципе, этой информации достаточно, чтобы начать пользоваться пакетом. Но аппетит приходит во время еды, так что, скорее всего, вам понадобится информация о том, какие именно пакеты у вас установлены и где они находятся:

```
library()   # вывод списка установленных пакетов
search()    # вывод списка пакетов, загруженных в память
.libPaths() # возвращает каталог, в который установлены пакеты
```

Установлен ли интересующий вас пакет?

```
any(grepl("имяПакета", installed.packages()))
```

Возможно, потребуется установить сразу несколько пакетов:

```
install.packages( c("пакет1","пакет2",...) )
```

И наверняка понадобится периодически их обновлять:

```
update.packages() # обновляет все установленные пакеты в диалоговом режиме
```

Устанавливать пакеты удобно при помощи графического интерфейса. Так, в RStudio команды для работы с пакетами сгруппированы в меню Tools. Однако даже если в своей ежедневной практике вы будете пользоваться только графическим интерфейсом, помните, что его команды реализованы при помощи вызовов всё тех же функций R, с которыми мы познакомились выше.

Перейдём теперь к загрузке пакетов. Кроме функции `library`, для этого используется очень похожая на неё функция `require`. Отличие между ними в том, что в случае, когда загружаемый пакет не установлен, `library` выдаёт сообщение об ошибке, а `require` возвращает `FALSE`, и потому её удобно использовать в условиях вида:

```
if (!require(package)) {
    install.packages(package)
    library(package)
}
```

которые обеспечивают загрузку пакета в случае его необходимости.

В ходе загрузки пакет может сообщать различную диагностическую информацию. Подавить вывод этой информации можно функцией `suppressPackageStartupMessages`. Вот как с её помощью загрузить пакет `rvest`:

```
suppressPackageStartupMessages(library(rvest))
```

Иногда нам нужен не весь пакет, а лишь одна функция из него. В этом случае функцию можно вызвать в виде: `имя_пакета::имя_функции()`, не используя `library` или `require`. Например, вызвать `sum()`, принадлежащую пакету `base`, можно следующим образом:

```
base::sum( c(1,2,3) )
```

```
## [1] 6
```

Другое дело, что пакет `base` устанавливается вместе с R и загружается в память автоматически, так что подобное обращение к `sum` избыточно.

Выбор пакета

Если вы не знаете, какой именно пакет может помочь в решении стоящей перед вами задачи, то путь ваш лежит на CRAN Task Views – это настоящий справочник по пакетам R.

Вначале выберем интересующую тему. Это могут быть, в частности:

- Web Technologies and Services – доступ к данным по HTTP, разбор содержимого веб-страниц, интерфейсы к API социальных сетей и т. п.;
- Open Data – пакеты для доступа к источникам открытых данных, а также к средствам хранения (Dropbox, Amazon Simple Storage Service и т. п.) и распространения данных (Google Sheets, Scribd, …).

Прелесть Task Views состоит в том, что по каждой теме в нём приведен обзор соответствующих пакетов, короткие аннотации к ним и сравнение реализаций той или иной технологии (алгоритма) в разных пакетах. Поэтому, скорее всего, вы покинете Task Views, «присмотрев» для себя один или несколько пакетов.

Примеры использования пакетов удобно искать на R-Bloggers – агрегаторе блогов, посвящённых R. Все наиболее значимые статьи из мира R, по крайней мере на английском языке, рано или поздно оказываются на этом сайте.

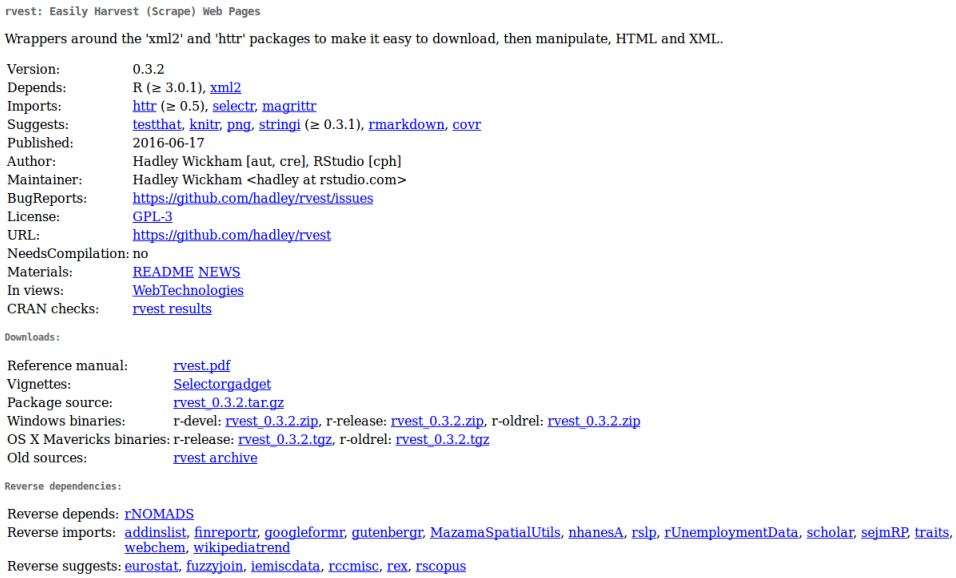
Для решения более узких вопросов по работе с пакетом существует сервис StackOverflow.

Наконец, R Documentation представляет собой поисковик документации по пакетам и отдельным функциям R, которые хранятся как в CRAN, так и в других репозиториях.

Страница пакета на CRAN находится в поисковике по запросу: `CRAN имя_пакета`. Посмотрим, какую информацию о пакете она может нам дать.

Справка и её разновидности

На странице пакета в CRAN содержится его подробное описание (рис. 8.1), с указанием названия, номера версии, пакетов, от которых зависит его работа, имён разработчиков и т. п.



rvest: Easily Harvest (Scrape) Web Pages

Wrappers around the 'xml2' and 'httr' packages to make it easy to download, then manipulate, HTML and XML.

Version:	0.3.2
Depends:	R (≥ 3.0.1), xml2
Imports:	httr (≥ 0.5), selectr, magrittr
Suggests:	testthat, knitr, png, stringi (≥ 0.3.1), rmarkdown, covr
Published:	2016-06-17
Author:	Hadley Wickham [aut, cre], RStudio [cph]
Maintainer:	Hadley Wickham <hadley at rstudio.com>
BugReports:	https://github.com/hadley/rvest/issues
License:	GPL-3
URL:	https://github.com/hadley/rvest
NeedsCompilation:	no
Materials:	README NEWS
In views:	WebTechnologies
CRAN checks:	rvest results

Downloads:

Reference manual:	rvest.pdf
Vignettes:	Selectorgadget
Package source:	rvest_0.3.2.tar.gz
Windows binaries:	r-devel: rvest_0.3.2.zip, r-release: rvest_0.3.2.zip, r-oldrel: rvest_0.3.2.zip
OS X Mavericks binaries:	r-release: rvest_0.3.2.tgz, r-oldrel: rvest_0.3.2.tgz
Old sources:	rvest archive

Reverse dependencies:

Reverse depends:	rNOMADS
Reverse imports:	addinslist, finreportr, googleformr, gutenbergr, MazamaSpatialUtils, nhanesA, rslp, rUnemploymentData, scholar, sejmRP, traits, webchem, wikipediatrend
Reverse suggests:	eurostat, fuzzyjoin, iemiscdata, rccmisc, rex, rscopus

Рис. 8.1 – Страница пакета rvest в CRAN

С точки зрения получения информации по использованию пакета, нам наиболее интересны: его аннотация, справочное руководство (reference manual) и виньетки (vignettes).

Справочные руководства к пакетам R выглядят стандартно: описание пакета, названия функций и их назначение, описания параметров, примеры применения.

Другое дело – виньетки – это особый вид справки, существующий в R. Виньетка больше похожа на статью, чем на обычную справку: в ней описана проблема и показано, как её можно решить при помощи данного пакета. Виньетка показывает, на какие категории можно разделить функции пакета и как эти функции взаимодействуют друг с другом в ходе решения задачи.

Виньетки есть у многих пакетов. Увидеть, какие виньетки установлены в данный момент на вашем компьютере, можно с помощью функции `browseVignettes`. Чтобы увидеть виньетки для конкретного пакета, используйте `browseVignettes("имя_пакета")` (рис. 8.2).

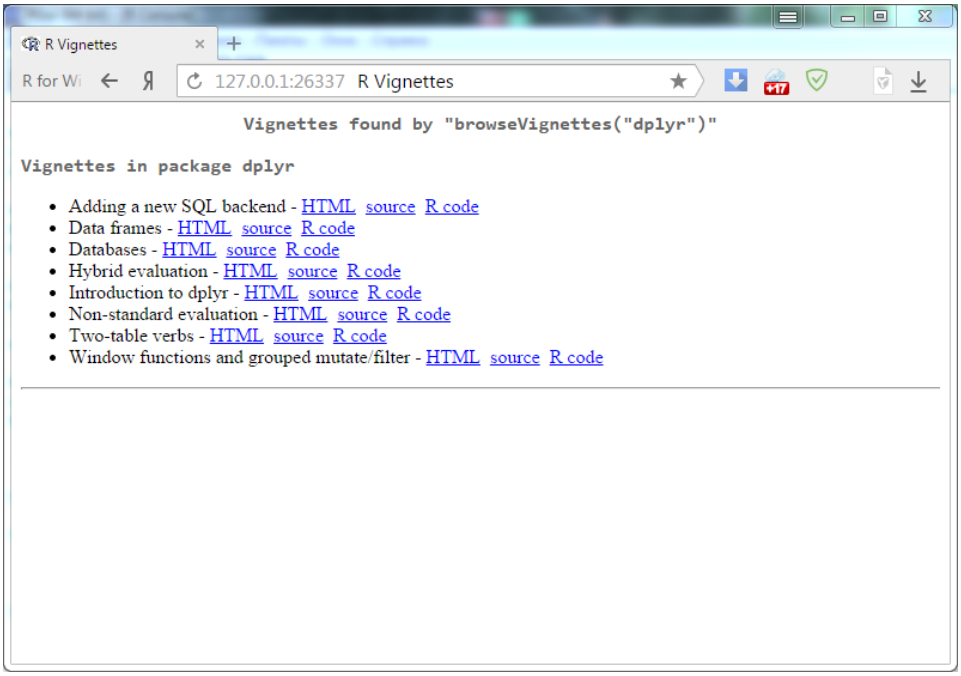


Рис. 8.2 – Список виньеток пакета dplyr

Каждая виньетка состоит из трёх обязательных компонент: исходного файла, HTML-страницы или PDF-файла, которые, собственно, и читает пользователь, и файла исходного кода на R. Таким образом, можно не только познакомиться с примером решения задачи, но и воспользоваться готовым кодом этого решения.

Вы можете прочитать виньетку при помощи функции `vignette` и посмотреть её код, используя `edit(vignette(имя))`.

Как самому создать пакет R?

Такой вопрос возникает рано или поздно. С ростом числа созданных вами функций R их придётся как-то упорядочивать, например размещать тематически связанные функции в отдельные папки. Потом в такие папки начинают добавляться используемые данные, потом – пояснения к функциям. Наконец, возникает мысль: не создать ли из всего этого пакет?

Создание пакета – лучший способ поделиться с кем-нибудь решённой вами задачей. Ведь в пакете есть код, реализующий решение; тест, поясняющий, как этим решением воспользоваться (виньетка и справочное руководство), и данные, которые при этом использовались, – нужно только взять всё это и установить.

Разумеется, для этого придётся пойти на некоторые затраты, например оформить пакет в соответствии со стандартами R. С другой стороны, задача может иметь продолжение, а в лице заинтересованных пользователей пакета вы можете получить неожиданных соратников.

Наиболее ёмким руководством по разработке пакетов в R является книга уже знакомого нам Хэдли Уикэма

- *Wickham H.* R Packages. O'Reilly Media, 2015.

и его же одноимённый сайт.

И в завершение главы познакомимся с одним очень полезным пакетом. Он делает более наглядным представление операций в R.

Пакет magrittr: конвейер операций

Предположим, что нам нужно применить к переменной `x` функцию `f`. Всё просто: делаем `f(x)`. А если теперь к полученному результату применить функцию `g`, а потом ещё и функцию `h`? Получится `h(g(f(x))` – выглядит уже не так просто и красиво.

Теперь представим себе, что существует оператор «применить», который позволяет записать эти операции в том порядке, в котором они выполняются:

```
к x применить f
```

а затем:

```
к x применить f применить g применить h
```

то есть к `x` применить функцию `f`, к полученному результату применить функцию `g` и, наконец, применить функцию `h`.

Именно таким оператором и снабжает нас пакет `magrittr`. Записывается этот оператор так: `%>%`, напоминая этим конвейеры (pipes) в Linux.

Теперь наши примеры можно оформить следующим образом:

```
x %>% f
x %>% f %>% g %>% h
```

Итак, оператор `%>%` позволяет:

- записывать последовательность действий слева направо (`x %>% f %>% h`), а не изнутри наружу (`h(f(x))`);
- легко добавлять в последовательность (конвейер) новое действие;
- избавиться от использования вложенных функций.

Рассмотрим два простых примера:

```
library(magrittr)
x <- seq(-pi,pi,.1)
y <- x %>% sin       # эквивалентно y <- sin(x)
Sys.Date() %>%
  as.POSIXct %>%
  seq(by = "15 mins", length.out = 9) %>%
  data.frame(timestamp = .)
```

```
##             timestamp
## 1 2016-09-06 03:00:00
## 2 2016-09-06 03:15:00
## 3 2016-09-06 03:30:00
## 4 2016-09-06 03:45:00
## 5 2016-09-06 04:00:00
## 6 2016-09-06 04:15:00
## 7 2016-09-06 04:30:00
## 8 2016-09-06 04:45:00
## 9 2016-09-06 05:00:00
```

Точка в последней строке кода (`timestamp = .`) является заменителем результата выполнения предыдущей операции конвейера.

Кроме `magrittr`, сходные возможности реализованы в пакете `pipeR`.

Глава 9

Ввод и вывод данных. Работа с файлами

Рабочий каталог пользователя

Перед тем как приступить к чтению или к записи данных, пользователь должен указать имя файла, с которым ему предстоит работать. По умолчанию поиск файла происходит в текущем рабочем каталоге пользователя.

Узнать текущий рабочий каталог можно с помощью функции `getwd` (*get working directory* – "узнать рабочий каталог"):

```
getwd()
```

```
## [1] "C:/Users/Admin/Documents/WebMining2"
```

а установить – при помощи `setwd`:

```
setwd("/path/to/dir")    # слэши '/' - как в Linux
setwd("\\path\\to\\dir") # или так
```

Вначале рассмотрим функции R для записи данных. С их помощью мы создадим файлы, которые затем будем считывать.

Запись данных в стандартное устройство вывода

По умолчанию стандартным устройством вывода является консоль. Мы уже знакомы с одной из функций, которая выводит данные в консоль, – это функция `print`. Но сейчас речь пойдёт о функции `cat`, обладающей гораздо большими возможностями.

Однако начнём с вывода данных в консоль:

```
cat("Hello,\n", "world!\n")
```

Этот код можно улучшить, поскольку `cat` позволяет указать символ-разделитель строк в параметре `sep`:

```
cat("Hello,", "world!", sep="\n")
```

Вот что получится в результате:

```
## Hello,
## world!
```

`cat` может выводить данные не только в консоль, но и в файл. Имя файла вывода служит значением параметра `file`.

```
cat("Hello,", "world!", sep="\n", file="hello.txt")
```

В этом случае данные выводятся в файл `hello.txt`, расположенный в рабочем каталоге.

Направление вывода данных можно изменить при помощи функции `sink`. Например, перенаправим вывод из консоли в файл, а затем снова вернём его в консоль:

```
sink("hello.txt")               # Перенаправляем вывод в файл hello.txt
cat("Hello,","world!",sep="\n") # Выводим данные в новую "консоль"
sink()                          # Возвращаем вывод в обычную консоль
```

Запись в текстовые файлы

Не вдаваясь в подробности, примем, что текстовые файлы содержат текст, то есть строки символов, которые могут быть прочитаны и поняты человеком. Напротив, бинарные файлы содержат «нечитаемые» данные: звук, видео, текст в закодированном виде и т. п.

Заметим, что одним из самых распространённых средств подготовки данных для их последующего анализа в R является Microsoft Excel. Чтобы сохранить таблицу Excel в виде txt- или csv-файла, в версиях Excel 2007 и более новых используют пункт Сохранить как (Save as), меню кнопки Office[4].

Ещё один простой способ экспорта данных из Excel: создать в редакторе Блокнот (Notepad) новый файл и перенести туда через буфер обмена всю таблицу Excel или её выделенную часть.

Вопросы чтения данных с помощью R непосредственно из таблиц Excel мы рассмотрим в главе 12.

Таблицы

Создадим таблицу данных и запишем её в файл `myfile.txt`:

```
height = c(180.5, 163.7, 177.2)
weight = c(75, 48, 72)
age = c(18, 17, 17)
df <- data.frame(height,weight,age)
names(df) <- c("Height","Weight","Age")
row.names(df) <- c("John","Mary","Sam")
write.table(df, file="myfile.txt")
df
```

```
##      Height Weight Age
## John  180.5     75  18
## Mary  163.7     48  17
## Sam   177.2     72  17
```

Самое важное в этом фрагменте – функция `write.table`, аргументами которой выступают имя таблицы данных и имя файла, куда эту таблицу предстоит записать.

Функция `write.table` насчитывает более десятка параметров. Например, параметр `append` определяет, будет ли файл `myfile.txt`, если он уже существует, создаваться заново (теряя прежнее содержимое) или только дополняться. По умолчанию `append = FALSE` и файл создаётся заново.

Параметр `sep` задаёт символ-разделитель между данными. По умолчанию разделителем служит пробел. Широко используются в качестве разделителей табуляция (`sep="\t"`) и запятая (`sep=","`). Последняя применяется настолько широко, что для неё существует специальная функция-обёртка над `write.table` – `write.csv` (CSV означает *comma separated values* – ‘значения, разделённые запятыми’).

Параметр `dec` устанавливает вид разделителя между целой и дробной частями числа. По умолчанию разделителем является точка: `dec="."`. Помимо точки, в этом качестве также используют запятую: `dec=","`. В частности, запятая применяется в странах СНГ и в континентальной Европе.

Для записи таблиц, содержащих числа с запятой в качестве разделителя целой и дробной частей, существует функция `write.csv2` – ещё одна обёртка `write.table`.

Заметим, что если запятая используется для разделения целой и дробной частей числа, то в качестве разделителя данных в файлах CSV применяется точка с запятой `';'`.

Записанный нами файл `myfile.txt` содержит в первой строке заголовки колонок таблицы. Точно так же поступают функции `write.csv` и `write.csv2`.

Чтобы записать данные, не помещая в файл заголовки колонок, воспользуемся параметром `col.names`:

```
write.table(df, sep=",",  col.names=FALSE)
```

Обратите внимание: мы создаём CSV-файл, используя `write.table`, а не `write.csv`, так как в последней параметра `col.names` просто нет.

Строки

Запись строк удобно выполнять функцией `writeLines`:

```
writeLines(text, con="myfile.txt", sep="\n")
```

которая в данном случае сохраняет вектор строк `text` в файле `myfile.txt`. Параметр `sep` задаёт строку символов, печатаемых в конце каждой строки из `text`.

[4] В более ранних версиях Excel – меню Файл (File).

Матрицы

Функция

```
write(A, file="mymatrix.txt", ncolumns=4)
```

сохраняет содержимое матрицы `A` в файле `mymatrix.txt` – в четырёх столбцах, разделённых пробелами.

Вывод матрицы в стандартное устройство вывода можно организовать следующим образом:

```
A <- matrix(1:10, ncol=5)
write(A, file="")
```

```
## 1 2 3 4 5
## 6 7 8 9 10
```

При этом данные записываются по строкам. Чтобы записать их по столбцам, транспонируем `A`:

```
write(t(A), file="")
```

```
## 1 3 5 7 9
## 2 4 6 8 10
```

Чтение из текстовых файлов

Элементы данных: scan

Функция `scan` считывает данные из текстового файла и возвращает в результате вектор или список. Создадим текстовый файл следующего вида:

```
24 1991
21 1993
53 1962
```

с помощью функции `cat`

```
cat("24 1991", "21 1993", "53 1962", file = "ex.data", sep = "\n")
```

Прочитав `ex.data` с помощью `scan`, получим вектор действительных чисел:

```
data <- scan("ex.data")
str(data)
```

```
##  num [1:6] 24 1991 21 1993 53 ...
```

Но ведь в файле явно хранилась таблица! Воспользуется тем, что `scan` возвращает вектор, и преобразуем этот вектор с помощью функции `matrix`:

```
data <- matrix(scan("ex.data"), nrow=3, byrow=TRUE)
data
```

```
##      [,1] [,2]
## [1,]   24 1991
## [2,]   21 1993
## [3,]   53 1962
```

Аргумент `n` определяет число считываемых элементов данных. Если `n = -1` (установлено по умолчанию) или равно любому отрицательному числу, то считывать данные нужно до конца файла. Считаем первое число:

```
data <- scan("ex.data", n=1)
data
```

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

По умолчанию `scan` выводит сообщения о том, сколько элементов данных она считала. Настройка `quiet=TRUE` отменяет вывод этих сообщений.

Если в первой строке файла находится текст заголовка, тогда как остальные строки содержат числовые данные, то `scan` выведет сообщение об ошибке. Этого можно избежать, указав, что именно (`what`) считывается. Например, так `scan` будет считывать строки:

```
data <- scan("myfile.txt", what = character(), quiet=TRUE)
str(data)
```

```
##  chr [1:15] "Height" "Weight" "Age" "John" "180.5" "75" ...
```

С другой стороны, первую строку можно при чтении пропустить. Число пропускаемых строк задаётся параметром `skip`. Считаем данные из файла `ex.data`, пропустив первую строку и сохранив результат в виде списка, состоящего из двух числовых векторов:

```
data <- scan("ex.data",
             what = list(Age = 0,
                         Birthyear = 0),
             skip=1,
             quiet=TRUE)
```

Параметр `what` определяет, что понимается под элементом данных, и косвенно влияет на другие настройки `scan`. Например, указав считать один элемент, мы теперь получим в результате весь список:

```
data <- scan("ex.data",
             what = list(Age = 0,
                         Birthyear = 0),
             n=1,
             quiet=TRUE)
str(data)
```

```
## List of 2
##  $ Age      : num [1:3] 24 21 53
##  $ Birthyear: num [1:3] 1991 1993 1962
```

Структура таблицы, хранящейся в `myfile.txt`, немного сложнее: в первом столбце находятся строки, а в остальных – числовые данные. Внесём необходимые изменения в список, заданный `what`:

```
data <- scan("myfile.txt",
             what = list(Name= "",
                         Heigth = 0,
                         Weight = 0,
                         Age = 0),
               skip=1,
               quiet=TRUE)
```

До получения таблицы остался один шаг:

```
df <- data.frame(data)
```

Строки: readLines

Для чтения строк текстового файла воспользуемся функцией `readLines`:

```
dat <- readLines("myfile.txt")
str(dat)
```

```
##  chr [1:4] "\"Height\" \"Weight\" \"Age\"" "\"John\" 180.5 75 18" ...
```

противоположной по смыслу уже знакомой нам `writeLines`.

`readLines` создаёт вектор `dat` из строк считанного ею файла `myfile.txt`. Число элементов `dat` равно числу прочитанных строк. В нашем случае их 4.

Число считываемых строк задаётся параметром `n`.

```
dat <- readLines("myfile.txt", n=1)
str(dat)
```

```
##  chr "\"Height\" \"Weight\" \"Age\""
```

Чтение строк удобно использовать, когда данные в файле не имеют структуры (как, например, в `hello.txt`). В нашем случае такая структура – таблица – существует, и ею нужно воспользоваться.

Таблицы

Чтение таблиц выполняется функцией `read.table`. Среди входных параметров этой функции наиболее часто используются:

- имя файла, из которого предстоит читать данные (`"myfile.txt"`);
- разделитель данных `sep` (в нашем случае – пробел);
- параметр `header`, указывающий, использовать ли первую строку файла в качестве заголовка таблицы (`TRUE` означает использовать).

```
dat <- read.table("myfile.txt", sep=" ", header=TRUE)
dat
```

```
##      Height Weight Age
## John  180.5     75  18
## Mary  163.7     48  17
## Sam   177.2     72  17
```

Если в строке файла `myfile.txt` обнаруживается символ комментария #, то остальная часть строки считается комментарием и при чтении игнорируется.

Для наиболее популярных разделителей данных – запятой и табуляции – существуют специальные функции (обёртки `read.table`):

- `read.csv`: `sep=","` или `sep="csv"`;
- `read.delim`: `sep="\t"`.

В обеих указанных выше функциях предполагается, что символом-разделителем целой и дробной частей числа является точка: `dec="."`.

Для чтения данных, в которых целая и дробная части числа разделены запятой, используются функции:

- `read.csv2`: `sep=","`, `dec=","`;
- `read.delim2`: `sep="\t"`, `dec=","`.

Очень полезным может оказаться аргумент `fill` (”заполнить”) функции `read.table`[5]. Если файл содержит строки разной длины, что означает пропуски в данных, то с `fill = TRUE` эти строки будут дополнены пробелами, а в полученной таблице на месте дополненных элементов будут стоять `NA`. В противном случае `read.table` выведет сообщение об ошибке чтения таблицы. В `read.csv*` и `read.delim*`, в отличие от `read.table`, `fill = TRUE` установлено по умолчанию.

Следует иметь в виду, что таким способом можно корректно заполнить только отсутствие элементов, стоящих в крайних правых колонках таблицы (то есть тех, что находятся в конце строки). Пропуски элементов из глубины таблицы, дополненные `fill = TRUE`, приведут к путанице элементов между колонками.

Отметим, что все рассмотренные нами к этому моменту функции чтения данных из текстовых файлов (`scan`, `readLines`, `read.*`) могут работать как с файлами, хранящимися на локальном компьютере, так и с файлами, размещёнными на Интернет-ресурсах.

Вот как выглядит загрузка данных wine quality из репозитория данных для машинного обучения UCI (рис. 9.1).

```
datafile <- "http://archive.ics.uci.edu/ml/
             machine-learning-databases/
             wine-quality/winequality-white.csv"
df <- read.table(file = datafile, sep=";", header=TRUE)
View(head(df))
```

	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.dioxide
1	7.0	0.27	0.36	20.7	0.045	45	17
2	6.3	0.30	0.34	1.6	0.049	14	13
3	8.1	0.28	0.40	6.9	0.050	30	9
4	7.2	0.23	0.32	8.5	0.058	47	18
5	7.2	0.23	0.32	8.5	0.058	47	18
6	8.1	0.28	0.40	6.9	0.050	30	9

Рис. 9.1 – Фрагмент набора данных wine quality из репозитория UCI

Обратите внимание, что хотя файл данных и имеет расширение `.csv`, что означает *comma separated values*, но данные в нём разделены точками с запятой. Поэтому используется не `read.csv`, а более общая функция `read.table`.

Завершающий штрих: проверка типа данных

После того как данные считаны и сохранены в R, стоит убедиться в том, что импорт был выполнен корректно. Например, если ожидалось получить таблицу, то сохранена именно таблица, а не список. Проверьте тип полученных данных с помощью функций `class` или `str`:

[5] Есть он и у `scan`.

```
# Импортируем таблицу.
df <- XLConnect::readWorksheetFromFile("hiv.xlsx",  sheet = 1)
# Проверяем тип полученного набора данных
# (str здесь не подойдёт, поскольку таблица слишком большая).
class(df)
```

```
## [1] "data.frame"
```

или выведите первые строки полученных данных при помощи `head`:

```
head(df, n = 2)
```

```
##   Estimated.HIV.Prevalence.....Ages.15.49. X1979 X1980 X1981 X1982 X1983
## 1                                 Abkhazia    NA    NA    NA    NA    NA
## 2                               Afghanistan    NA    NA    NA    NA    NA
##   X1984 X1985 X1986 X1987 X1988 X1989 X1990 X1991 X1992 X1993 X1994 X1995
## 1    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 2    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##   X1996 X1997 X1998 X1999 X2000 X2001 X2002 X2003 X2004 X2005 X2006 X2007
## 1    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 2    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##   X2008 X2009 X2010 X2011
## 1    NA  <NA>  <NA>  <NA>
## 2    NA  0,06  0,06  0,06
```

Убедитесь в том, что заголовки колонок таблицы импортированы корректно.

Работа с данными в бинарном формате

CSV и другие текстовые форматы очень удобны для хранения таблиц, но далеко не все объекты R являются таблицами. Чтобы сохранить произвольный объект R, используется функция `save`. Эта функция также может сохранить набор объектов:

```
df1 <- data.frame(a=1:5, b=6:10)
df2 <- data.frame(c=11:15, d=16:20)
save(df1, df2, file="myfile.RData") # сохраняем объекты
rm(df1, df2)                        # удаляем их из рабочего пространства

# Проверяем: существует ли объект?
df1
```

```
## Error in eval(expr, envir, enclos): объект 'df1' не найден
```

Для загрузки данных используется функция `load`:

```
load("myfile.RData")
df1
```

```
##   a  b
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

Скачивание файлов из Интернет

Скачать веб-страницу в R можно с помощью функции `download.file`:

```
# адрес документа
url <- "http://curl.haxx.se/"
# папка, где будет сохраняться скачанные документы
download_folder <- "c:/Downloads"
# скачать файл
download.file(url,paste(download_folder,"curl.html",sep=""))
```

Функция `paste` выполняет конкатенацию строк, результатом которой будет полный путь к файлу на диске.
Скачать медиафайлы или документы можно аналогично:

```
pdf_url <- "http://cran.r-project.org/web/packages/
            downloader/downloader.pdf"
download.file(pdf_url,
              paste(download_folder,"downloader.pdf",sep=""),
              mode="wb")
```

Обратите внимание на режим записи файла, заданный атрибутом `mode`. `"wb"` означает: записать (write) файл как бинарный (binary). По умолчанию используется значение `"w"` и файл считается текстовым. В результате, при работе под Windows будет возникать ошибка чтения PDF. Почему так происходит и о разнице между бинарными (двоичными) и текстовыми файлами см. Википедию.

Если одноимённый файл уже существует, то при скачивании в режимах `"w"` и `"wb"`, он будет переписан, а его прежнее содержимое – уничтожено. Запись в режимах `"a"` и `"ab"` означает: добавить (append) скачиваемый файл к уже существующему.

Удалить скачанный файл по окончании работы можно следующим образом:

```
file.remove(paste(download_folder,"downloader.pdf",sep=""))
```

Расширить возможности R по скачиванию документов можно с помощью пакета `downloader`. Заданная в нём функция `download` имеет те же опции, что и `download.file`, но в отличие от последней позволяет скачивать документы не только по протоколу HTTP, но и по HTTPS.

Excel

Для работы с файлами Microsoft Excel в R существует множество пакетов. Так, `readxl`, как ясно из названия, позволяет читать файлы Excel – как формате Excel '97 (XLS), так и в более современном OOXML (Excel 2007+, XLSX). Пакеты `xlsx` и `XLConnect` дают возможность не только читать, но также создавать и записывать указанные выше типы файлов. При этом все эти пакеты работают с файлами, расположенными на локальном компьютере. Пакет `gdata` позволяет одновременно скачивать и открывать файлы, но при этом требует установки интерпретатора Perl. В общем, мы будем пользоваться пакетом XLConnect, как наиболее распространённым.

Скачаем данные о заболеваемости ВИЧ из хранилища Gapminder и сохраним их в файле `hiv.xls`. Загрузить эту рабочую книгу и сохранить в таблице `df` содержимое листа `data`, помогут функции `loadWorkbook` и `readWorksheet` соотвественно:

```
library('XLConnect')
url_base <- "http://spreadsheets.google.com/pub"
key <- "pyj6tScZqmEfbZyl0qjbiRQ"
url <- paste0(url_base,"?key=",key,"&output=xls")
download.file(url, "hiv.xlsx", mode="wb")
wb = loadWorkbook("hiv.xlsx")
df = readWorksheet(wb, sheet = "data")
```

Функция `readWorksheetFromFile` объединяет возможности по загрузки рабочей книги и чтению нужного листа:

```
# Название листа знать необязательно, достаточно указать его номер:
df <- readWorksheetFromFile("hiv.xlsx",
                            sheet = 1)
```

Для более точного указания фрагмента данных служат параметры: `startRow`, `startCol`, `endRow`, `endCol`. Чтобы получить таким способом данные по Аргентине за 1990–1992 гг. нужно указать:

```
df <- readWorksheetFromFile("hiv.xlsx",
                            sheet=1,
                            startRow = 11, endRow   = 12,
                            startCol = 13, endCol   = 15)
```

Не слишком-то удобно… Параметр `region` позволит указать область данных в привычном для Excel стиле:

```
df <- readWorksheetFromFile("hiv.xlsx",
                            sheet=1,
                            region = "M11:O12")
head(df)
```

```
##   Col1 Col2 Col3
## 1  0.3  0.3  0.3
```

Google Spreadsheets

Функция `gs_read` из пакета `googlesheets` позволяет считывать данные из таблиц и сохранять их в R.

Работа с пакетом начинается с загрузки списка доступных вам таблиц[6]:

[6] Надеюсь вы не забыли зарегистрироваться в Google?

```
library(googlesheets)
gs_ls()
```

У меня получилось вот что:

```
Source: local data frame [6 x 10]

               sheet_title        author  perm version             updated
                     (chr)         (chr) (chr)   (chr)              (time)
1      Список інструментів       devrand     r     new 2016-05-17 21:06:10
2            Новая таблица     dakhramov    rw     new 2016-02-26 10:24:53
3 Comments for Import exc…        mart     r     new 2016-01-25 09:13:11
4 Программа Фестиваля урб…    auldnick     r     new 2014-11-28 20:16:40
5 Extract Twitter Data - …   matteoduo     r     new 2014-03-26 17:16:40
6             hcp_20120824   dakhramov    rw     new 2012-09-14 17:56:25
Variables not shown: sheet_key (chr), ws_feed (chr), alternate (chr), self
  (chr), alt_key (chr)
```

– у вас будет что-то своё.

Далее вы проходите аутентификацию (в браузере), после которой можно вернуться в R и выбрать интересующую таблицу по её названию (title) или по ключу (key):

```
data <- gs_title("имяТаблицы")
data <- gs_key()
```

Теперь можно считывать данные:

```
gs_read(data)
```

`gs_read` позволяет указать номер листа в таблице (`ws`) и диапазон считываемых ячеек (`range`), аналогично функциям для чтения таблиц Excel из пакета XLConnect.

Пакет googlesheets позволяет не только импортировать данные, но и создавать таблицы Google, загружать их на сервер и т.п.

JSON

Для работы с данными, хранящимися в формате JSON (JavaScript Object Notation), чаще всего используют три пакета: `rjson`, `RJSONIO` и `jsonlite`. Все они обладают функциями:

- `fromJSON` – для импорта данных JSON в R;
- `toJSON` – для экспорта в JSON.

Пакеты `rjson` и `jsonlite` импортируют данные как из локальных файлов, так и из удалённых ресурсов, заданных своим URL. `RJSONIO` работает только с локальными файлами, но разработчики обещают добавить возможность работы с URL в самом ближайшем будущем. В остальном возможности пакетов примерно одинаковы.

В настоящее время наиболее популярен пакет `jsonlite`. Причём это не голословное утверждение: узнать, какие пакеты чаще всего скачивают пользователи R, можно из данных CRAN package download logs. Сейчас мы этим и займёмся.

Какой из JSON-пакетов самый популярный?

Данные CRAN package download logs представляют собой логи скачиваний пакетов R, а также самого R, с одного из зеркал CRAN, поддерживаемого компанией RStudio. Здесь же приведен пример, как эти данные можно скачать с помощью R.

Наш анализ популярности JSON-пакетов будет состоять из следующих шагов:

1. скачаем файлы логов;
2. объединим данные со всех файлов в одну большую таблицу;
3. построим графики скачиваний основных пакетов, работающих с JSON.

В результате выполнения первого шага мы получим каталог `CRANlogs`, содержащий архивы с логами, созданный в рабочем каталоге пользователя.

```
## Шаг 1: Скачаем файлы логов.
# Составим вектор дат от начальной до конечной.
from <- as.Date('2016-07-01')
to   <- as.Date('2016-07-30')
days <- seq(from, to, by = 'day')
# Выделим из даты год (годы отсчитываются от 1900 г.).
year <- as.POSIXlt(days)$year + 1900
# Получим список файлов, которые нужно скачать.
```

```
urls <- paste0('http://cran-logs.rstudio.com/', year, '/', days, '.csv.gz')
# Создадим каталог для хранения данных
dir.create("CRANlogs")
# Скачаем файлы и сложим их в созданный каталог
for (i in 1:length(urls)) {
  download.file(urls[i], paste0('CRANlogs/', days[i], '.csv.gz'))
}
```

Второй шаг позволит нам получить список таблиц, импортированных из логов, а затем на их основе – одну большую таблицу.

```
## Шаг 2: Объединим данные из логов в одну большую таблицу (data.table).
# Составим список имён файлов из каталога CRANlogs.
file_list <- list.files("CRANlogs", full.names=TRUE)
# Считаем данные из файлов логов и составим из них список таблиц (data.frame).
logs <- list()
for (file in file_list) {
  print(paste("Reading", file, "..."))
  logs[[file]] <- read.table(file, header=TRUE, sep=",", quote ="\"",
                             dec=".", fill=TRUE, comment.char="",
                             as.is=TRUE)
}
# Загрузим пакет data.table для работы с большими таблицами
library(data.table)
# Объединим все таблицы списка logs в одну большую таблицу
# (data.table) по строкам
dt <- rbindlist(logs)
# Посмотрим, что у нас получилось.
str(dt)
```

```
Classes ‘data.table’ and 'data.frame':  15813383 obs. of  10 variables:
 $ date     : chr  "2016-07-01" "2016-07-01" "2016-07-01" "2016-07-01" ...
 $ time     : chr  "22:51:36" "22:51:37" "22:51:38" "22:51:32" ...
 $ size     : int  1085498 106715 35218 527 1568353 432844 368679 5169 467631 525 ...
 $ r_version: chr  "3.3.0" "3.3.0" "3.3.1" "3.3.1" ...
 $ r_arch   : chr  "x86_64" "x86_64" "x86_64" "x86_64" ...
 $ r_os     : chr  "linux-gnu" "linux-gnu" "linux-gnu" "darwin13.4.0" ...
 $ package  : chr  "git2r" "roxygen2" "reshape2" "ggplot2" ...
 $ version  : chr  "0.15.0" "5.0.1" "1.4.1" "2.1.0" ...
 $ country  : chr  "GB" "GB" "RO" "US" ...
 $ ip_id    : int  1 1 2 3 4 5 6 1 1 2 ...
 - attr(*, ".internal.selfref")=<externalptr>
```

```
# Список таблиц больше не нужен - удалим его.
rm(logs)
# Преобразуем типы данных в некоторых колонках.
dt <- dt[, date:=as.Date(date)]
dt <- dt[, package:=factor(package)]
dt <- dt[, country:=factor(country)]
# Посмотрим, как выглядит таблица теперь.
str(dt)
```

```
Classes ‘data.table’ and 'data.frame':  15813383 obs. of  10 variables:
 $ date     : Date, format: "2016-07-01" "2016-07-01" ...
 $ time     : chr  "22:51:36" "22:51:37" "22:51:38" "22:51:32" ...
 $ size     : int  1085498 106715 35218 527 1568353 432844 368679 5169 467631 525 ...
 $ r_version: chr  "3.3.0" "3.3.0" "3.3.1" "3.3.1" ...
 $ r_arch   : chr  "x86_64" "x86_64" "x86_64" "x86_64" ...
 $ r_os     : chr  "linux-gnu" "linux-gnu" "linux-gnu" "darwin13.4.0" ...
 $ package  : Factor w/ 10024 levels "A3","aaMI","abbyyR",..: 3253 7798 7394 3216 179 1883 8945 8040 18
 $ version  : chr  "0.15.0" "5.0.1" "1.4.1" "2.1.0" ...
 $ country  : Factor w/ 211 levels "A1","A2","AD",..: 71 71 167 200 99 200 200 71 71 167 ...
 $ ip_id    : int  1 1 2 3 4 5 6 1 1 2 ...
 - attr(*, ".internal.selfref")=<externalptr>
```

```
# Необходимо для работы с data.table
setkey(dt, package, date, country)
```

Этот шаг требует некоторых пояснений:

- `read.table` умеет читать архивы `*.gz`, так что дополнительных функций для работы с архивами не требуется. В архивах находятся CSV-файлы.

- Чтение файлов может занять несколько минут, поэтому полезно выводить диагностические сообщения о ходе процесса (`print`).
- Пакет data.table позволяет создавать большие таблицы – так мы будем называть тип данных `data.table`, который унаследован от `data.frame`, и позволяет хранить большие наборы данных, не умещающиеся целиком в оперативной памяти компьютера.
- Функция `data.table::rbindlist` объединяет список таблиц в большую таблицу по строкам.
- `:=` позволяет модифицировать данные по ссылке, а не по значению. Относится к пакету data.table.
- `data.table::setkey` – сортирует данные в заданных колонках большой таблицы. Так же, как и `:=` работает по ссылке.

Для последующих сеансов работы с R можно сохранить объединённые данные `dt` в бинарном файле и загружать их при необходимости:

```
# Сохраним данные для последующего анализа
save(dt, file="CRANlogs/CRANlogs.RData")
# rm(dt)
# load("CRANlogs/CRANlogs.RData")
```

Осталось выбрать данные по интересующим пакетам и построить на их основе график.

```
## Шаг 3: Строим графики.
# Выберем данные по пакетам rjson, RJSONIO и jsonlite
packages <- dt[J(c("rjson", "RJSONIO", "jsonlite")),
               length(unique(ip_id)), by=c("date", "package")]
# Загрузим графическую библиотеку ggplot2
library(ggplot2)
# Построим график зависимости числа скачиваний от даты
# для выбранных пакетов
ggplot(packages, aes(x=date, y=V1, color=package, group=package)) +
  geom_line() + ylab("Downloads") + theme_bw()
# Сохраним последний график в файле.
# ggsave("json_pacs.png", device = "png")
```

Функция `J` относится к пакету `data.table`, позволяет выбрать нужные строки большой таблицы и сохранить результат в `data.table`.

Зависимость числа скачиваний JSON-пакетов от времени (июль 2016 г.) представлена на рис. 9.2. Из графика видно, что `jsonlite` значительно популярней двух других пакетов.

Взаимодействие с базами данных

Рассмотрим, как с помощью R:

- создать в базе данных таблицу на основе файла с данными (.csv);
- загрузить данные из таблицы базы данных в R.

Для работы с базами данных в R широко используются пакеты `DBI` и `sqldf`.

DBI + RSQLite

Пакет `DBI` обеспечивает единый интерфейс доступа ко множеству систем управления базами данных (СУБД), в частности к SQLite, MySQL и PostgreSQL.

Для работы с конкретной СУБД нужно установить её драйвер, реализованный в виде отдельного пакета. Для SQLite это пакет `RSQLite`, для PostgreSQL – `RPostgreSQL` и т. п.

Рассмотрим принципы работы с базами данных на примере SQLite.

```
library(DBI)
# Загружаем общий интерфейс и соединяемся с базой, используя её драйвер.
# Если базы не существует, то она будет создана.
db <- dbConnect(RSQLite::SQLite(), dbname="Test.sqlite")
# Скачиваем файл из репозитория и сохраняем в таблицу (iris)
fname <- "http://archive.ics.uci.edu/ml/machine-learning-databases/
          iris/iris.data"
iris <- read.table(file=fname, sep=",", header=FALSE)
# Запишем таблицу в базу.
dbWriteTable(conn = db, name = "Iris", value = iris,
             row.names = FALSE)
# Просмотрим список существующих в базе таблиц.
dbListTables(db)
# Извлечём содержимое таблицы в data.frame.
df1 <- dbGetQuery(db, "SELECT * FROM iris")
# Теперь с ней можно работать в R.
# Закрываем соединение.
dbDisconnect(db)
```

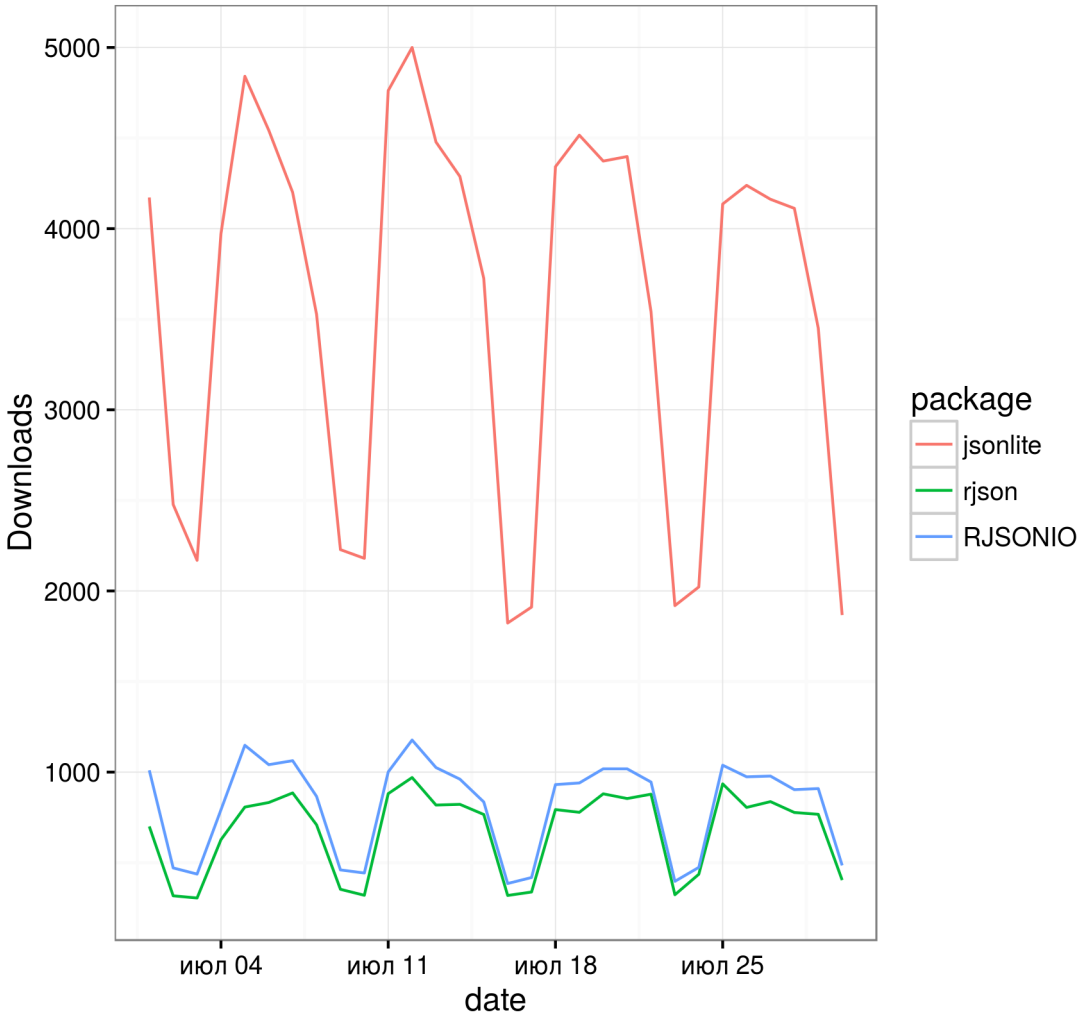


Рис. 9.2 – Зависимость числа скачиваний пакетов rjson, RJSONIO и jsonlite от времени (июль 2016 г.)

sqldf

Пакет `sqldf` импортирует DBI, но предлагает несколько иной способ работы с базами данных. В частности, он позволяет манипулировать с таблицами R (`data.frame`) при помощи языка SQL.

```
library(sqldf) # загружает также DBI и RSQLite
# Создаём новую базу данных.
sqldf("attach 'Test1.sqlite' as new")
# Скачиваем файл данных и сохраняем его на диск.
download.file("http://archive.ics.uci.edu/ml/
              machine-learning-databases/iris/iris.data",
              "iris.csv")
# Создаём в базе таблицу и заполняем её данными из файла.
read.csv.sql(file   = "iris.csv",
             header = FALSE,
             sql    = "CREATE TABLE Iris AS SELECT * FROM file",
             dbname = "Test1.sqlite"
             )
# Извлекаем содержимое таблицы базы данных для работы в R.
df2 <- sqldf("SELECT * FROM Iris", dbname = "Test1.sqlite")
```

Проверим идентичность созданных таблиц:

```
identical(df1,df2)
##[1] TRUE
```

Архивы

Архивы `*.z`, `*.gz` и `*.bz2` открываются функциями чтения данных, вроде `read.table` и `read.csv`, автоматически и никаких дополнительных функций для этого не требуется.

Несколько сложнее дела обстоят с архивами `.zip`. Функция `unz` распаковывает одиночные файлы, хранящиеся в ZIP-архивах:

```
unz(zipname, filename)
```

где `zipname` – имя ZIP-архива, а `filename` – запакованный в этот архив файл.

Предположим, что по адресу (выдуманному!) `http://www.xyz.org/data/` находится архив `data.zip`, в котором хранится файл данных `data.dat`. Алгоритм извлечения этих данных будет примерно следующим:

- Создать временный файл (`temp`). Например, с помощью `tempfile()`.
- Скачать `download.file` данные и поместить их во временный файл.
- С помощью `unz` извлечь данные из временного файла.
- Удалить временный файл (например, функцией `unlink`).

```
temp <- tempfile()
download.file("http://www.xyz.org/data/data.zip", temp, mode="wb")
data <- read.table(unz(temp, "data.dat"))
unlink(temp)
```

Напомним, что `mode="wb"` ни окажет никакого влияния при работе в Linux, но важен при работе в Windows, которая в противном случае будет рассматривать архив как текстовый файл.

Функция `unzip` позволяет распаковать ZIP-архив в заданный каталог на диске:

```
unzip("dataset.zip", exdir = ".")
```

В данном случае архив `dataset.zip` распаковывается в текущий каталог, что соотвествует настройкам функции по умолчанию.

Существует и обратная функция – `zip`, упаковывающая файлы в ZIP-архив.

Импортировать и распаковывать zip-архивы позволяет также функция `getZip` из пакета `Hmisc`. В следующем примере скачивается и распаковывается архив из репозитория данных ООН. Затем из распакованного файла считываются данные, хранящиеся в формате CSV, и сохраняются в таблице `unData`:

```
library(Hmisc)
 urlUN <- "http://data.un.org/Handlers/DownloadHandler.ashx?
           DataFilter=group_code:101;country_code:826&
           DataMartId=SNA&Format=csv&c=2,3,4,6,7,8,9,10,11,12,13&
           s=_cr_engNameOrderBy:asc,fiscal_year:desc,_grIt_code:asc"
 unData <- read.csv(getZip(urlUN))
```

В пакете `Hmisc` содержится множество других полезных функций, предназначенных для импорта данных.

Несколько слов об управлении файлами и каталогами

Работа с файлами и каталогами подробно описана в справке по R (`?files` и `?dir.create`). Мы рассмотрим всего один пример: создадим каталог, скачаем в него файл, проверим, что этот файл существует, а затем удалим и файл, и каталог.

```
# Создадим каталог
dir.create("test")
# Сделаем его рабочим
setwd("test")
# Скачаем в него файл
url <- "https://data.baltimorecity.gov/api/views/dz54-2aru/rows.csv?
       accessType=DOWNLOAD"
download.file(url, destfile="./rows.csv", method="curl")
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  9294  100  9294    0     0   1435      0  0:00:06  0:00:06 --:--:--  1439
```

```
# Посмотрим список файлов в текущем каталоге
list.files(".")
```

```
[1] "rows.csv"
```

```
# Удалим файл rows.csv
file.remove("rows.csv")
```

```
[1] TRUE
```

```
# Проверим, существует ли файл?
file.exists("rows.csv")
```

```
[1] FALSE
```

Ну, ещё бы.

```
# Сделаем рабочим родительский каталог для test
setwd("..")
# Удалим каталог test
unlink("test", recursive = TRUE)
```

Если указано `recursive = FALSE`, то даже пустой каталог не будет удалён.

Резюме

Функции R, предназначенные для чтения данных, позволяют считывать данные, хранящиеся как на локальных компьютерах, так и на интернет-ресурсах.

Более подробно об импорте/экспорте данных в R можно узнать из руководства R Data Import/Export.

Глава 10

Среда разработки RStudio

Окно среды разработки RStudio (рис. 10.1) делится на две части по вертикали. В левой половине код скриптов редактируется и отправляется на выполнение, а в правой отображаются результаты, рабочее пространство R и различная вспомогательная информация.

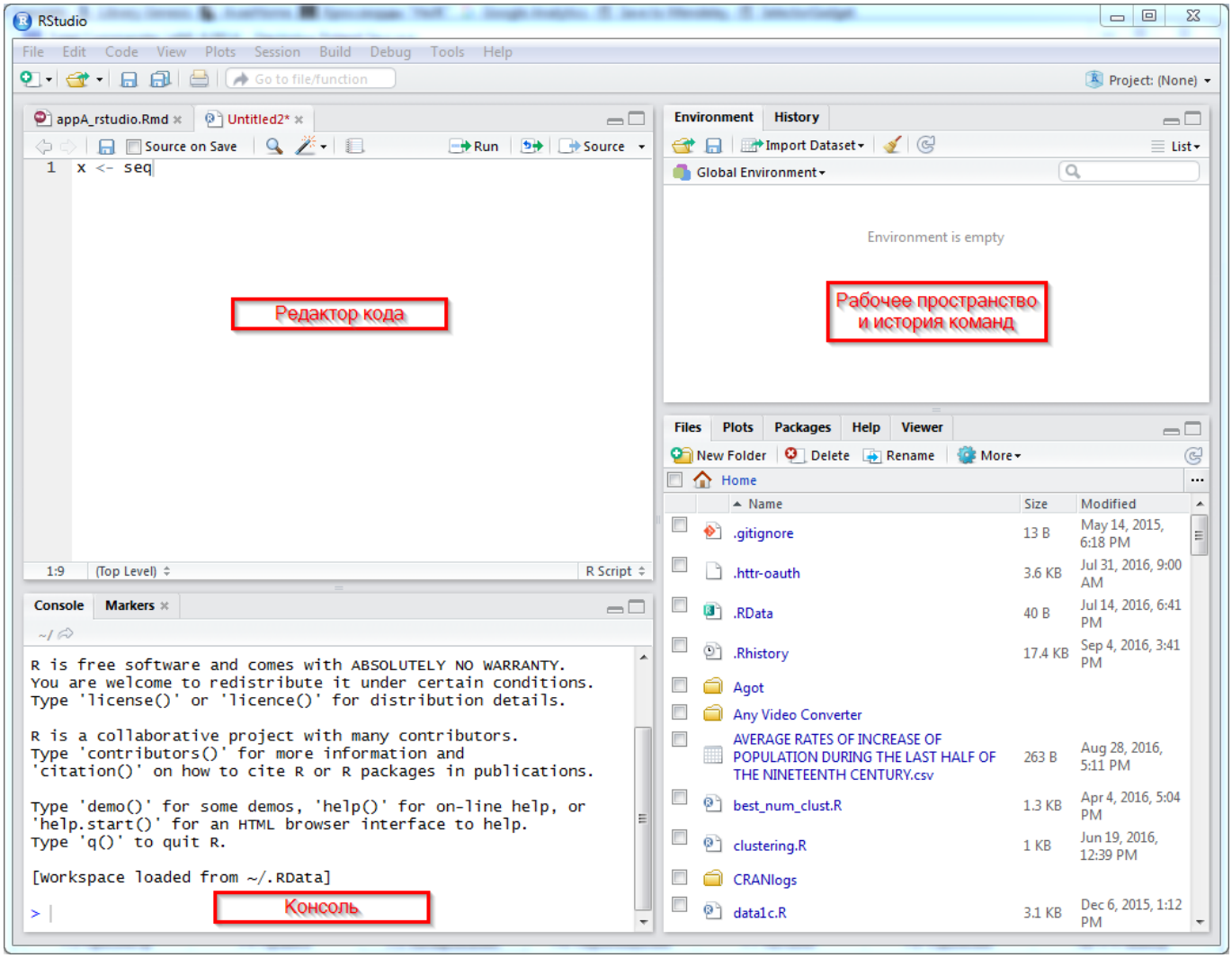


Рис. 10.1 – Окно среды разработки RStudio

Если у вас есть опыт работы с RGui, то все приобретенные при этом навыки пригодятся и в RStudio. Последний, однако, обладает гораздо большими возможностями, что влияет на стиль работы: вместо непосредственного вызова функций R чаще используются команды меню[7]. В частности, вместо функций управления рабочим пространством R удобнее использовать окно Environment (рис. 10.1, вверху справа), вместо функций установки пакетов – меню Tools и т. д.

[7] Которые, как и в RGui, в конечном счёте реализованы при помощи функций R.

Создание скрипта

Новый скрипт создаётся с помощью меню File/New File или кнопки New на панели инструментов (рис. 10.2). Таким образом можно создавать также документы Markdown, простые текстовые файлы, файлы исходного кода на языке C++ и ряд других файлов, работу с которыми поддерживает редактор кода RStudio. При этом обеспечиваются подсветка синтаксиса языка и соответствующее изменение меню редактора.

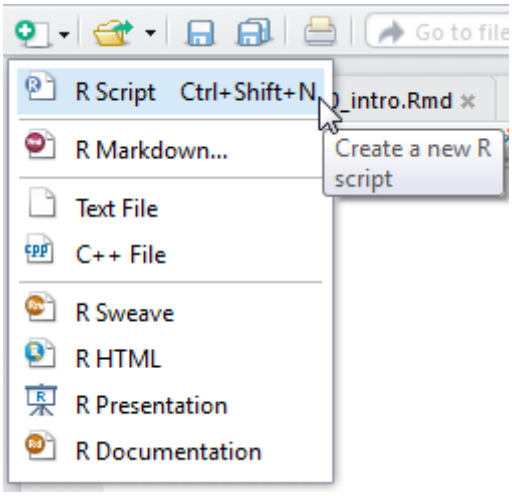


Рис. 10.2 – Создание скрипта с помощью кнопки **New** на панели инструментов

На рис. 10.3 показан внешний вид меню редактора кода при работе со скриптом R (рис. 10.3а) и с документом Markdown (рис. 10.3б).

Создадим простейший скрипт и рассмотрим на его примере, чем отличается работа в RStudio от работы в RGui.

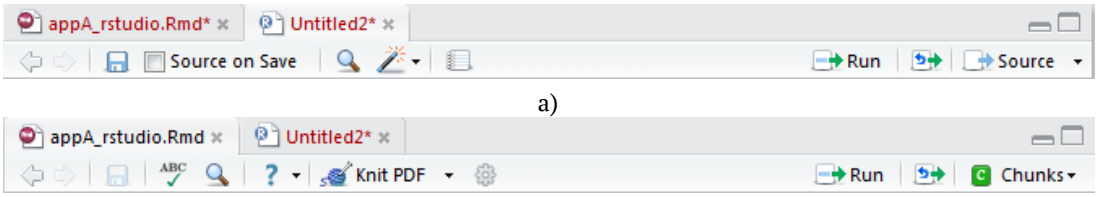


Рис. 10.3 – Меню редактора кода при работе со скриптом (а) и с документом Markdown (б)

Автодополнение имён объектов

Если в RGui имя функции можно дополнить во время набора, нажав клавишу *Tab*, то в RStudio возможности автодополнения гораздо шире. Редактор «угадывает» набираемую функцию (рис. 10.4), причём показывает пользователю не только имя функции, но и её сигнатуру, а также краткую справку. Дополняются не только имена встроенных функций, но и любых других объектов R, в частности переменных и пользовательских функций.

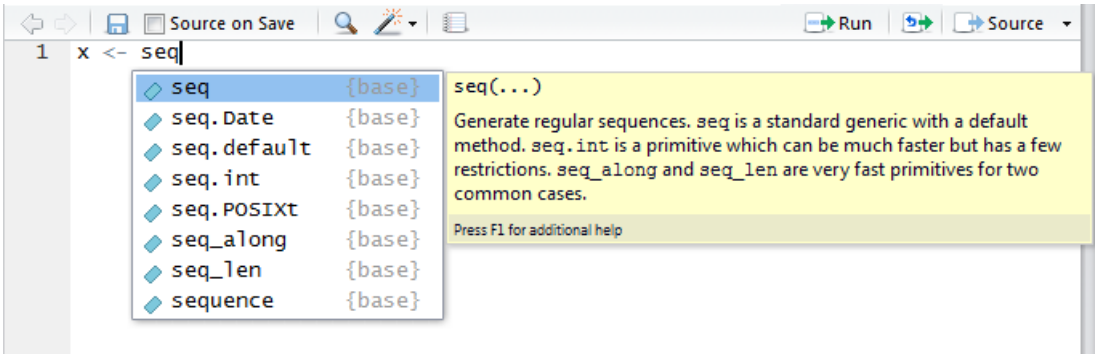


Рис. 10.4 – Редактор кода и консоль «подсказывают» имена функций и переменных

Выполнение

Выделив нужные строки кода, отправим их на исполнение с помощью кнопки Run (рис. 10.5).

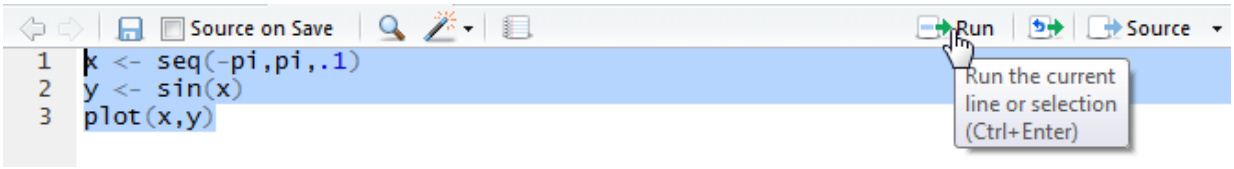


Рис. 10.5 – Кнопка Run отправляет на исполнение выделенные строки кода

При этом откроется графическое окно с построенным в нём графиком, а все выполненные команды будут отображены в консоли (рис. 10.6).

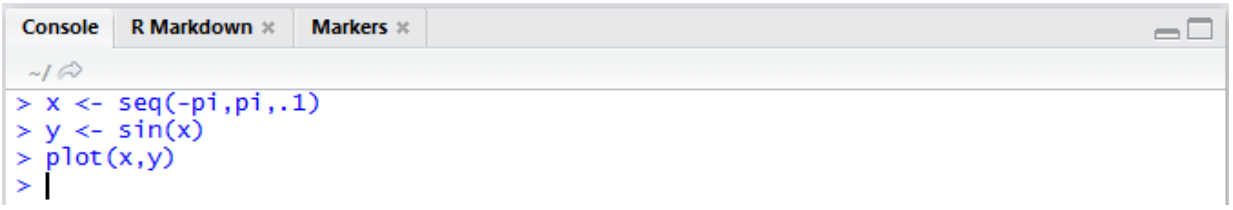


Рис. 10.6 – Выполненные команды скрипта отображаются в консоли. В заголовке консоли указан текущий рабочий каталог

Рабочее пространство

Значения переменных, помещённых в рабочее пространство, отображаются в окне Environment (рис. 10.7). Надпись «Global Environment» в заголовке окна говорит о том, что в окне отображаются глобальные переменные. Здесь же могут отображаться локальные переменные функций, а также функции пакетов-расширений, загруженные в текущем сеансе работы.

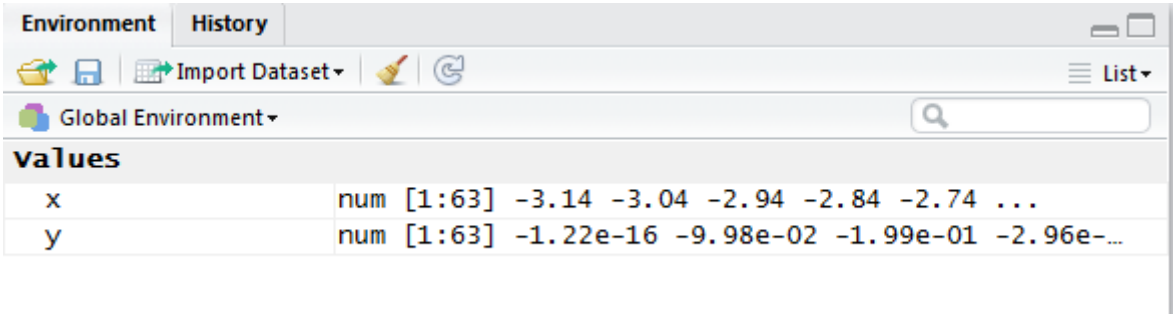


Рис. 10.7 – Значения переменных, помещённых в рабочее пространство, отображаются в окне **Environment**

Так как в окне Environment отображается структура переменных, нет особой необходимости использовать функции `str` и `class`. Кнопка с изображением метлы позволяет очистить рабочее пространство, что также быстрее использования функции или меню Session/Clear Workspace....

Особенно удобно в RStudio просматривать содержимое таблиц. Создадим таблицу из значений координат:

```
df <- data.frame(x=x,y=y)
```

Значение таблиц помещается в отдельный раздел (**Data**) на вкладке Environment (рис. 10.8).

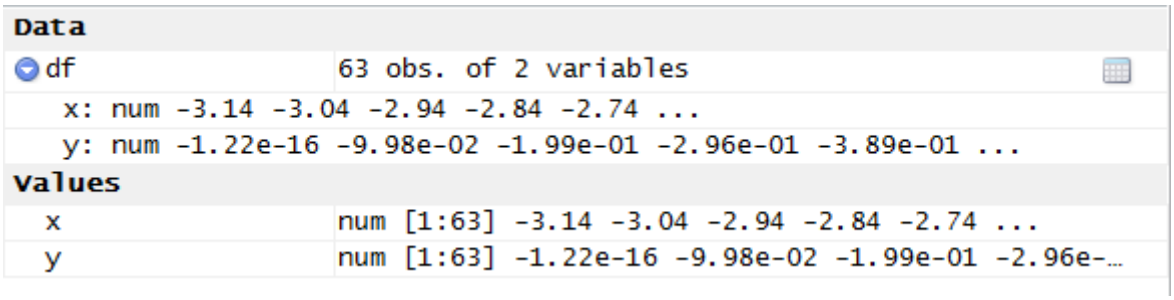
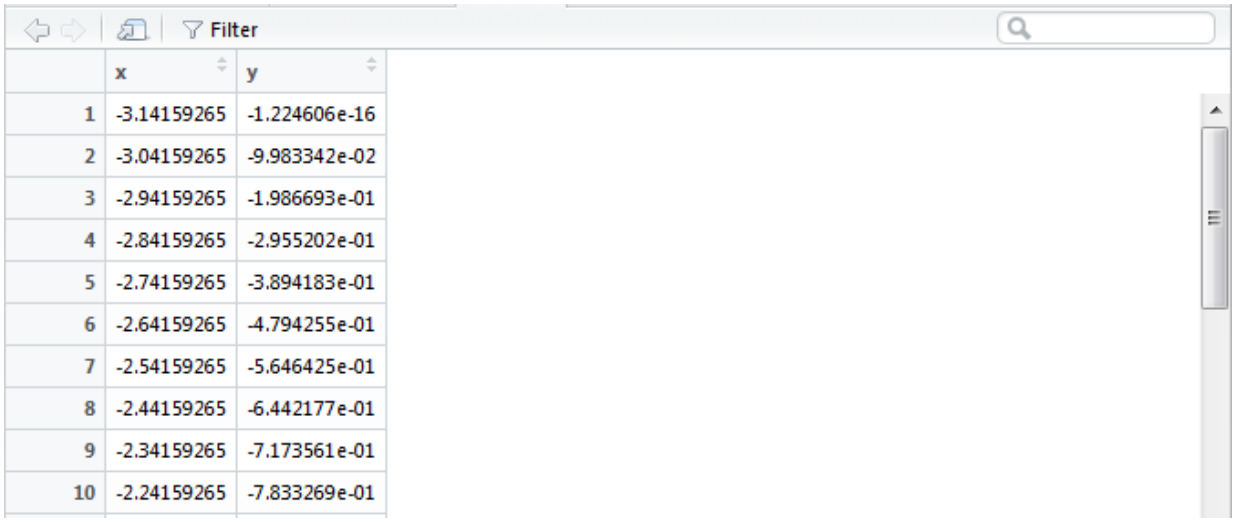


Рис. 10.8 – Отображение таблицы `df` в окне Environment

Значок таблицы, расположенный справа в строке с именем переменной, позволяет открыть таблицу на вкладке редактора (рис. 10.9). При этом появляется возможность сортировки, поиска элементов и фильтрации содержимого таблицы.

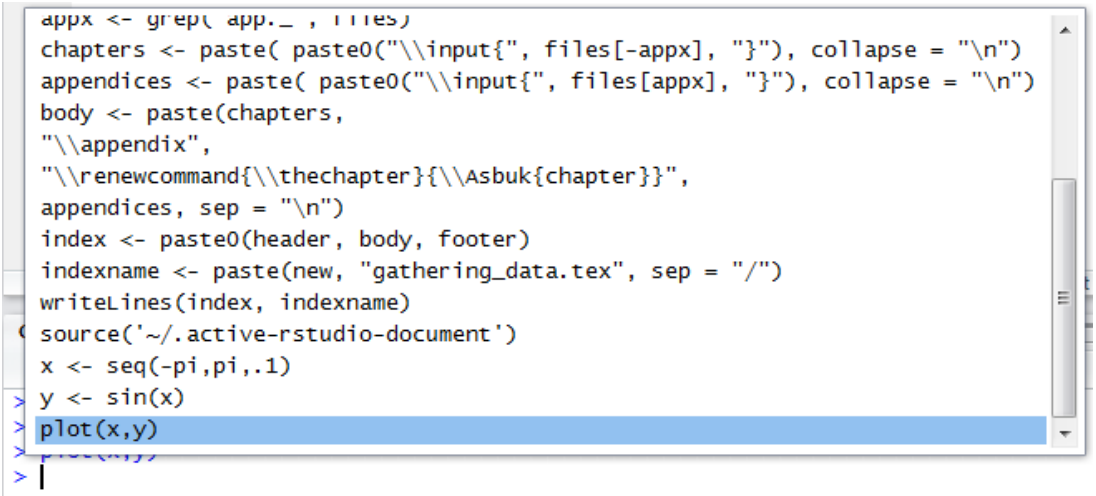


	x	y
1	-3.14159265	-1.224606e-16
2	-3.04159265	-9.983342e-02
3	-2.94159265	-1.986693e-01
4	-2.84159265	-2.955202e-01
5	-2.74159265	-3.894183e-01
6	-2.64159265	-4.794255e-01
7	-2.54159265	-5.646425e-01
8	-2.44159265	-6.442177e-01
9	-2.34159265	-7.173561e-01
10	-2.24159265	-7.833269e-01

Рис. 10.9 – Отображение таблицы `df` на вкладке редактора кода

История команд

Выполненные ранее команды можно использовать повторно, набрав в консоли *Ctrl+Стрелка вверх* (рис. 10.10). Это же сочетание клавиш подходит для быстрого поиска ранее вызванных функций. Например, для поиска функций, в имени которых содержится `plot`, следует просто ввести `plot` и нажать *Ctrl+Стрелка вверх*. При этом работает и обычная для R возможность перемещаться по истории команд при помощи клавиш со стрелками.



```
appx <- grep( app._ , files)
chapters <- paste( paste0("\\input{", files[-appx], "}"), collapse = "\n")
appendices <- paste( paste0("\\input{", files[appx], "}"), collapse = "\n")
body <- paste(chapters,
"\\appendix",
"\\renewcommand{\\thechapter}{\\Asbuk{chapter}}",
appendices, sep = "\n")
index <- paste0(header, body, footer)
indexname <- paste(new, "gathering_data.tex", sep = "/")
writeLines(index, indexname)
source('~/.active-rstudio-document')
x <- seq(-pi,pi,.1)
y <- sin(x)
plot(x,y)
```

Рис. 10.10 – Список набранных ранее команд, показанный в консоли

Ещё одним инструментом для работы с историей команд служит вкладка History (рис. 10.1, вверху справа). Команды в ней отображаются в порядке выполнения: последние выполненные находятся внизу списка.

На вкладке History команды можно выделять и затем отправлять в консоль (кнопка To Console) или в редактор кода (To Source) (рис. 10.11). Если в редакторе при этом нет открытых документов, то будет создан новый документ по имени Untitled с номером.

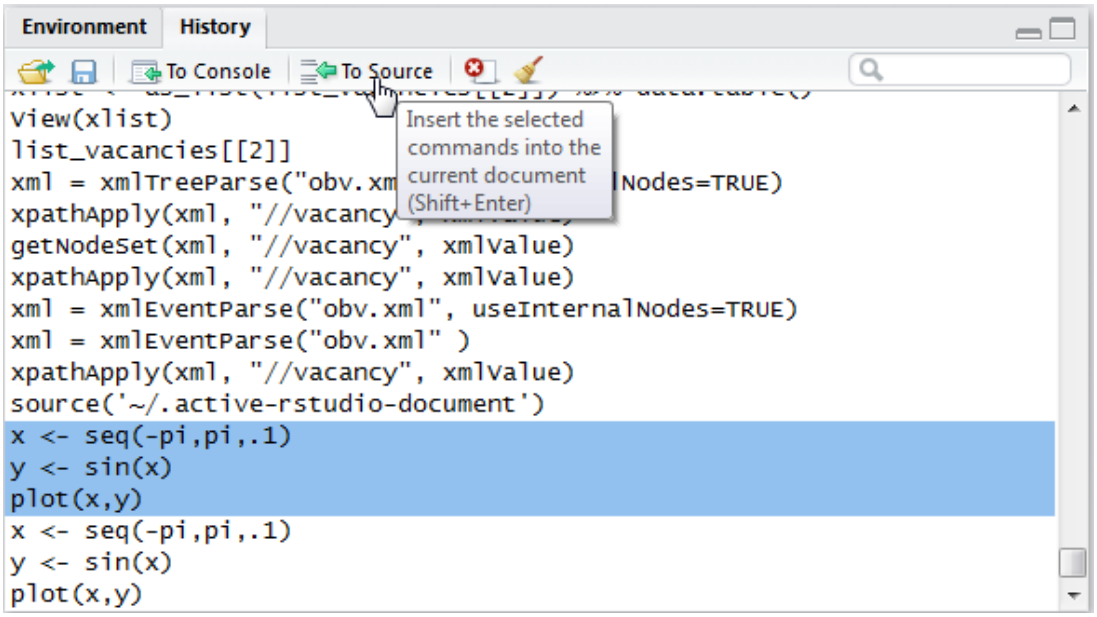


Рис. 10.11 – Выбор команд на вкладке History

Кроме этого, есть возможность очистить историю команд целиком или удалить из неё избранные команды.

Сохранение файлов

Файлы в RStudio при завершении работы с программой или аварийном останове сохраняются автоматически. Даже если они не были сохранены обычной командой сохранения (*Ctrl+S*).

Сохранить файл в заданной кодировке можно с помощью меню File/Save with Encoding...

Кодировки файлов

Возможность открыть файл в нужной кодировке даёт команда меню File/Reopen with Encoding.... В результате появится список вариантов кодировок (рис. 10.12).

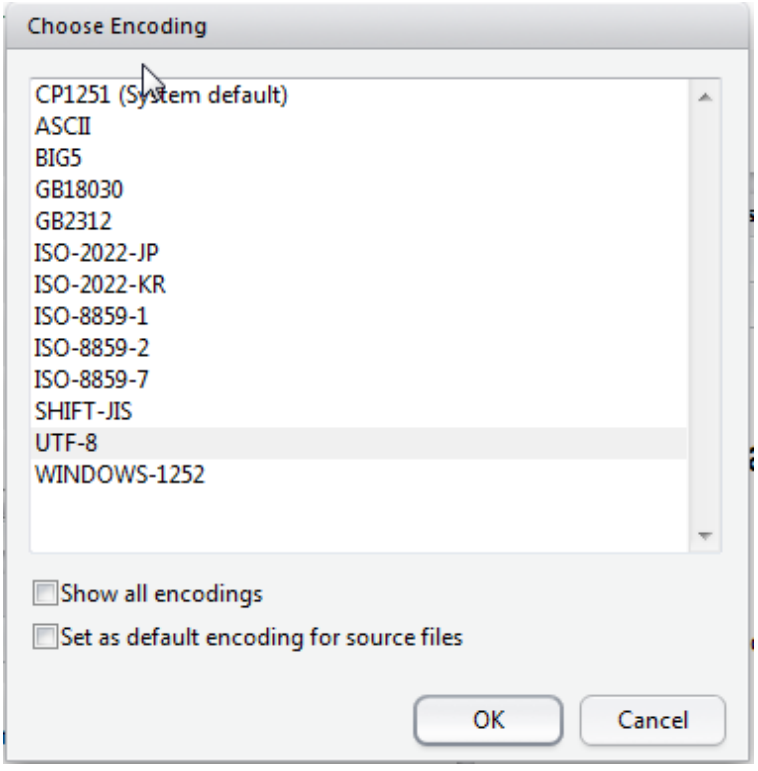


Рис. 10.12 – Выбор кодировок

Особенно часто к такому «переоткрытию» файлов приходится прибегать, работая в Windows с файлами в кодировке UTF-8. Дело в том, что русская кодировка в R под Windows по умолчанию устанавливается равной windows-1251, отчего кодированные в UTF-8 файлы отображаются «кракозябрами».

Чтобы задать кодировку файлов по умолчанию, воспользуемся меню Tools/Global Options..., раздел General, поле Default text encoding.

Управление файлами в рабочем каталоге

Вкладка Files, расположенная в нижнем правом окне среды RStudio, отображает файлы, хранящиеся в рабочем каталоге пользователя (рис. 10.13). Клик по имени файла открывает данный файл в редакторе. Здесь же вы можете создать, удалить, переименовать и проделать другие операции с файлами. Таким образом, вкладка Files представляет собой нечто вроде встроенного файлового менеджера.

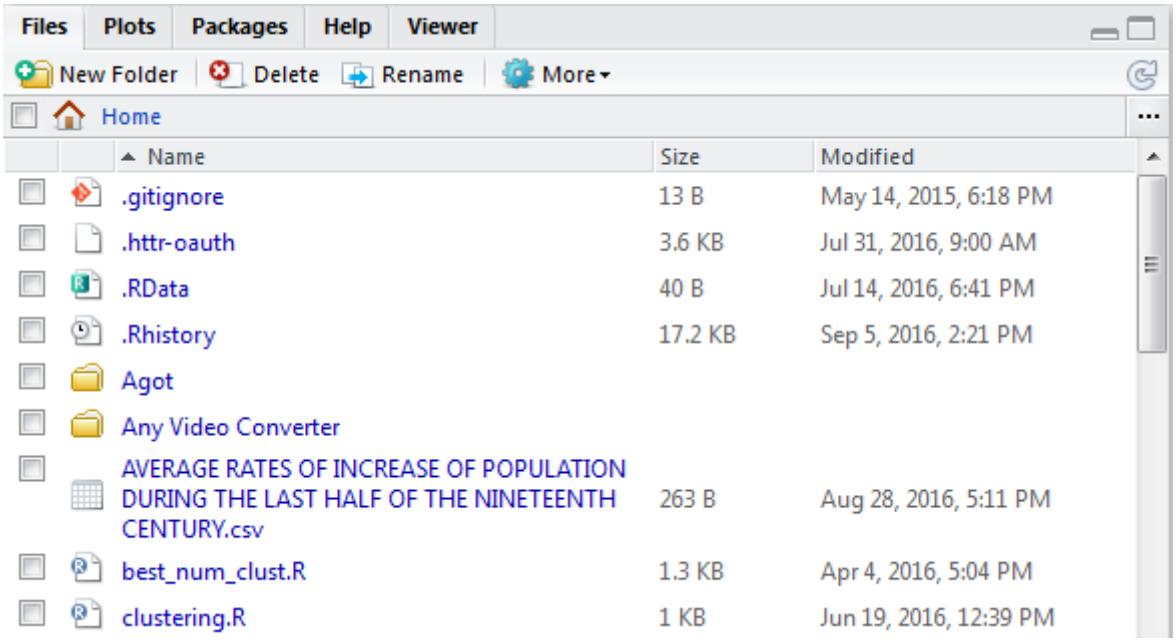


Рис. 10.13 – Вкладка **Files**: список файлов рабочего каталога пользователя

Управление пакетами

Установка и обновление пакетов выполняются при помощи команд меню Tools или вкладки Packages (рис. 10.14).

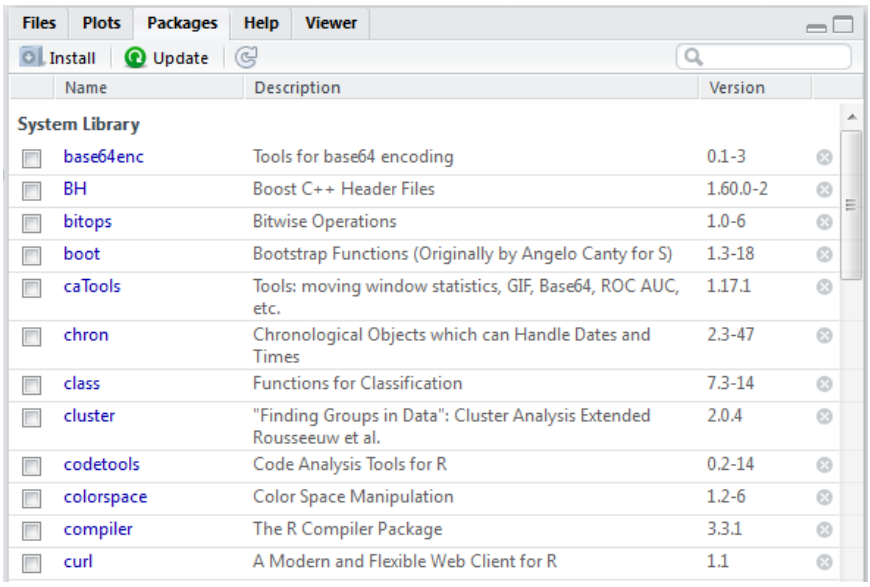


Рис. 10.14 – Вкладка Packages: управление пакетами расширений R

Поиск и замена

Поиск и замена текста внутри файла выполняются с помощью меню Edit. Кроме того, можно воспользоваться кнопкой (с изображением лупы) на панели редактора кода (рис. 10.15).

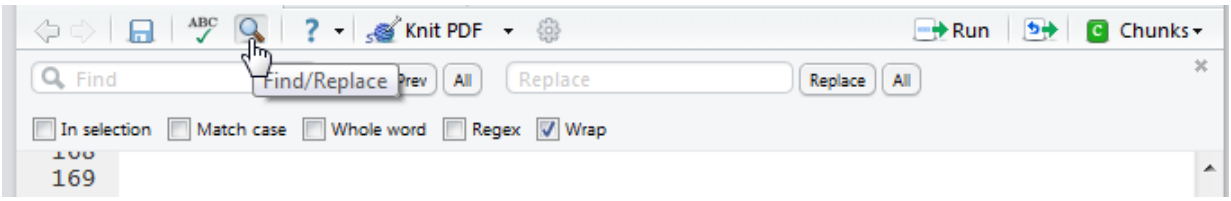


Рис. 10.15 – Панель поиска и замены в редакторе кода

Поиск в нескольких файлах осуществляется командой Edit/Find in Files...

Автоматическое создание функций

Редактор кода способен проанализировать выделенную часть кода и автоматически преобразовать её в функцию (рис. 10.16). Все «свободные» переменные в выделенном фрагменте[8] будут при этом преобразованы в аргументы функции.

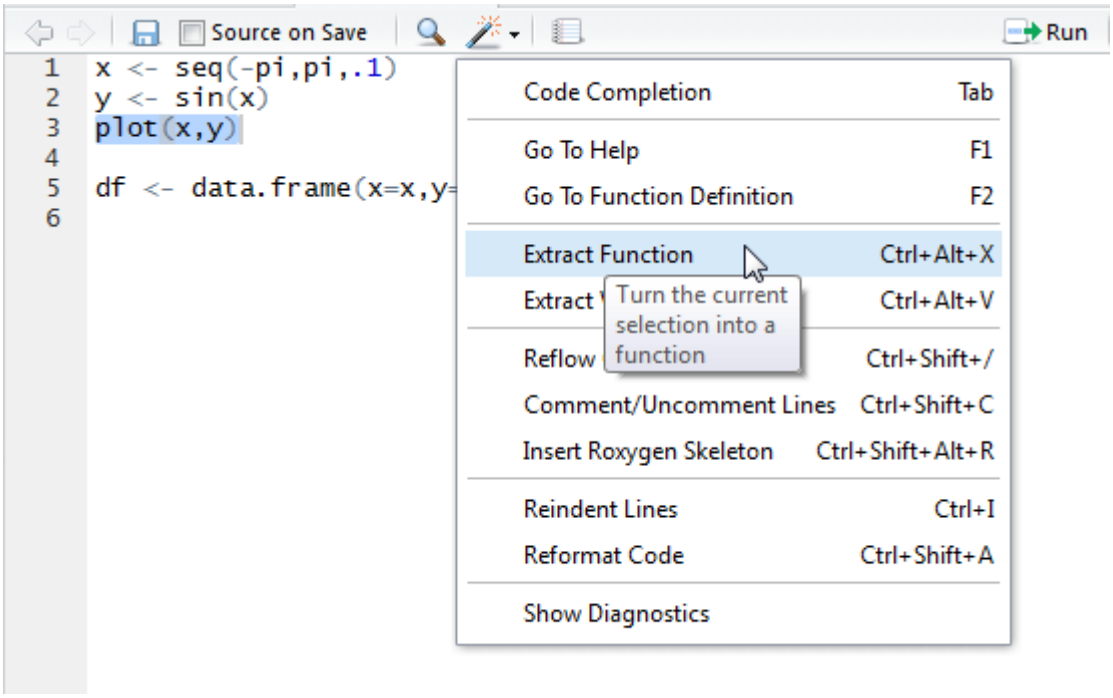


Рис. 10.16 – Автоматическое выделение функции из выбранного фрагмента кода

Например, выделенный на рис. 10.16 код

```
plot(x,y)
```

превратится в

```
myplot <- function (x, y) {
  plot(x,y)
}
```

Имя новой функции указывается пользователем.

Комментирование

Быстрее всего закомментировать или раскомментировать фрагмент кода можно при помощи комбинации клавиш *Ctrl+Shift+C*. Кроме того, можно использовать меню Code или клавишу с изображением волшебной палочки в редакторе кода (рис. 10.17).

[8] То есть объекты, на которые имеется ссылка, но которые не созданы внутри фрагмента.

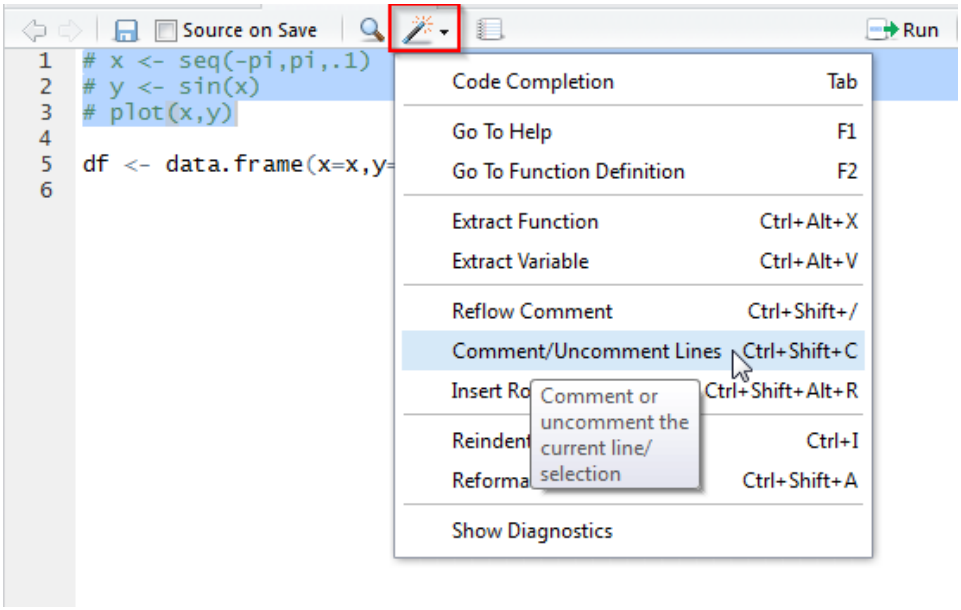


Рис. 10.17 – Комментирование выделенного фрагмента кода

Переход к определению функции

Функции, созданные пользователем, отображаются на вкладке Environment (рис. 10.18, раздел **Function**), а значок-свиток в правой части строки с именем функции позволяет посмотреть её код (рис. 10.19).

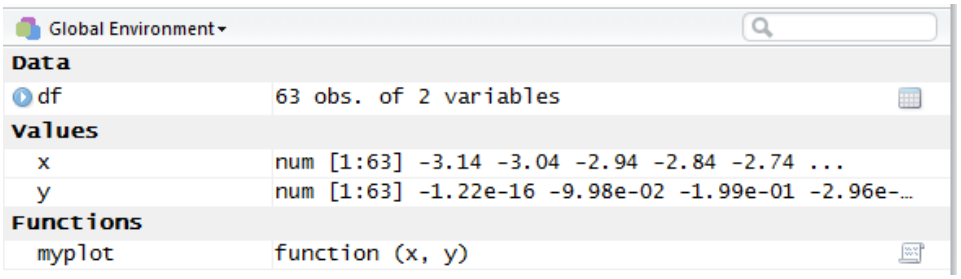


Рис. 10.18 – Пользовательская функция `myplot` на вкладке Environment

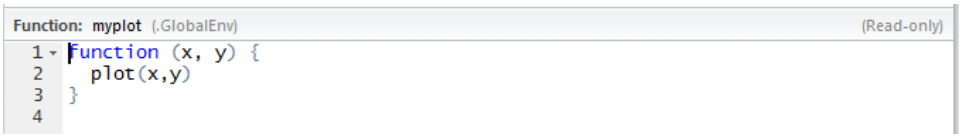


Рис. 10.19 – Код функции `myplot` на вкладке редактора кода

Клавиша *F2* позволяет, выделив упоминание функции в коде скрипта, перейти к её определению.

Ссылки

Подробную информацию по работе с RStudio IDE можно найти в разделе Support на официальном сайте компании RStudio (*support.rstudio.com*).

Предметный указатель