Визуализация и анализ географических данных на языке R

Тимофей Самсонов

2017-09-21

Contents

Введение

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

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

- Язык R
- Среда разработки RStudio

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

Форматы представления материалов

Прадлагаемый вашему вниманию курс состоит из ряда лекций (модулей), каждый из которых представлен в двух форматах: **HTML** и **R**.

Файл HTML

Файл **HTML** доступен в каждом модуле под названием *Лекция 1*, *Лекция 2* и т.д. Его удобно использовать в качестве справочника при выполнении заданий и подготовки к занятиям. Помимо текста и специальным образом форматированных фрагментов исходного кода, он содержит результаты генерации графиков и карт, а также навигацию по разделам.

Фрагменты исходного кода в тексте выглядят следующим образом:

```
a <- 5
sin(sqrt(a))
## [1] 0.7867491
sqrt(sin(a) + 2)
## [1] 1.020331
```

Обычным шрифтом `sin(sqrt(a))` в этих фрагментах показан исходный код, а бледным курсивом *`## [1] 0.7867491`* — результат выполнения этого кода, который выводится в консоль среды разработки (в нашем случае это RStudio).

> Если вы хотите сохранить файл HTML себе и просматривать локально, достаточно после его открытия щелкнуть внутри страницы в любом месте правой кнопкой мыши (сокращенно ПКМ) и выбрать команду *Сохранить*, *Сохранить как* или *Сохранить фрейм как* (в зависимости от браузера формулировка может меняться).

Файл R

Файл **R** доступен в каждом модуле под названием *Исходный код лекции 1*, *Исходный код лекции 2* и так далее. Файл **R** также можно использовать при подготовке к лекциям и выполнении заданий. Содержание файла исходного кода полностью соответствует лекции. Отличие заключается в том, что файл **R** - это полноценно работающий скрипт. Все команды, приведенные в лекции, можно последовательно выполнить, а не только посмотреть на них, как это представлено в версии **HTML**. Исходный текст лекции в файле R преобразован в комментарии, так что вы можете последовательно читать и выполнять команды.

Весьма полезно также *создать копию* файла **R** и в процессе знакомства с новой темой попробовать поменять различные параметры и входные данные, чтобы посмотреть как меняется результат. В любом случае, даже если вы что-то испортите, исходный файл всегда доступен через систему электронного обучения, в которой вы в настоящий момент находитесь.

> Пользователи операционной системы Windows должны скачивать себе файл с суффиксом *CP1251* в названии, а OS X — с суффиксом *UTF8*. Отличие связано с тем, что в данных операционных системах используются разные кодировки для представления символов (текста). Если у вас при открытии файла отображаются кракозябры или знаки вопросов — это значит, что кодировка файла не соответствует вашей операционной системе или стандартным установкам RStudio. Проблему можно решить, скачав файл в нужной кодировке или воспользовавшись командой меню *File* > *Open With Encoding...*

Как работать с кодом

Существует несколько способов выполнения исходного кода:

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

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

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

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

Комментарии

Комментарии — это фрагменты текста программы, начинающиеся с символа **#**. Комментарии не воспринимаются как исполняемый код и служат для документирования программы. При выполнении программы содержимое комментария

в зависимости от настроек среды может выводиться или не выводиться в консоль, однако их содержание никак не влияет на результаты выполнения программы.

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

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

> Прервать комментарий и написать справа от него исполняемый код нельзя

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

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

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

#
d <- c / a

#
e <- d * b
```

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

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

Стандарт оформления кода на R

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

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

> Стандарт оформления кода иногда также называют стилем программирования. Мы не будем использовать этот термин, поскольку под стилем программирования традиционно также понимают фундаментальный

подход (*парадигму*) к построению программ: процедурный, функциональный, логический, объектно-ориентированный и некоторые другие.

Названия специальных символов

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

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

Установка и подключение пакетов

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

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

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

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

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

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

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

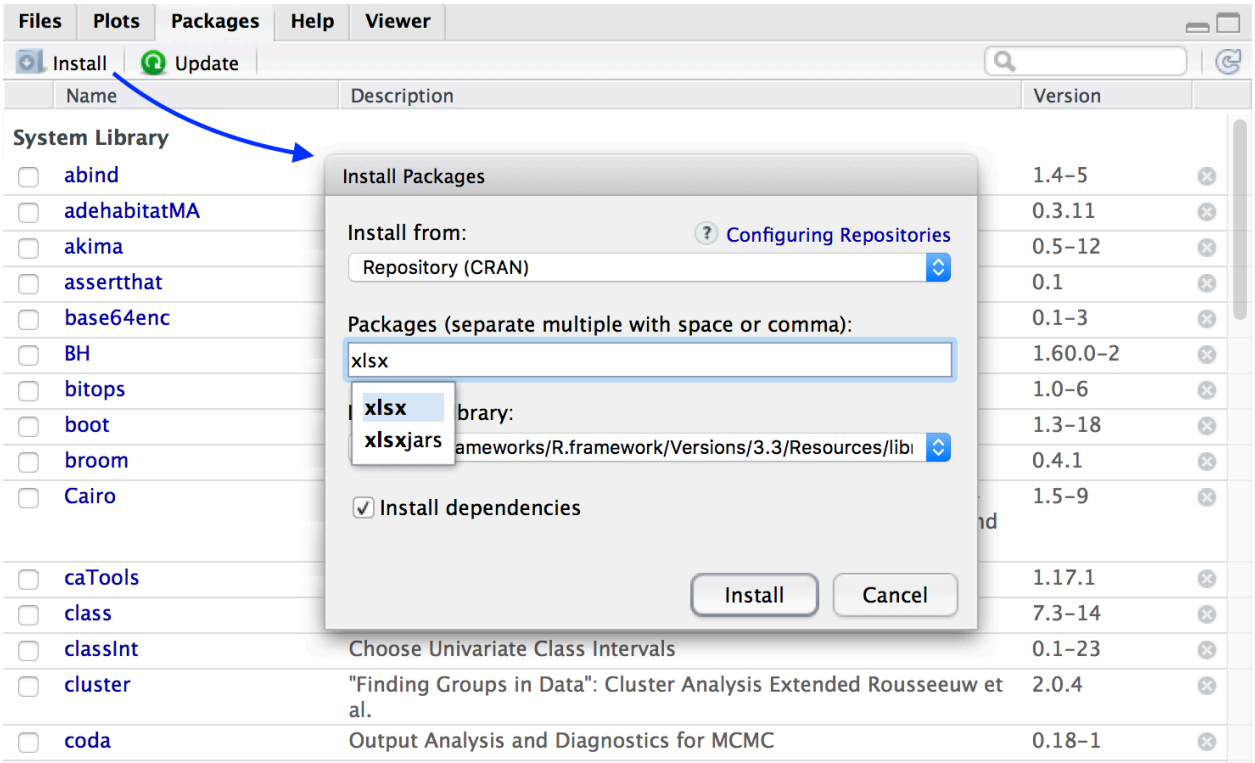

Figure 1: *Установка пакета*

```
library(openxlsx)
```

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

Как ссылаться

Если этот курс лекций оказался полезным для вас, и вы хотите процитировать его с списке литературы вашей работы, ссылку следует оформить как:

Самсонов Т.Е. **Визуализация и анализ географических данных на языке R.** М.: Географический факультет МГУ, 2017. DOI: 10.5281/zenodo.901911

Chapter 1

Атомарные типы данных

1.1 Числа

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

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

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

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

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

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

Результат деления по умолчанию имеет тип с плавающей точкой:

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

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

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

Остаток от деления можно получить с помощью оператора %%:

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

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

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

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

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

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

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

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

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

```
a + b
## [1] 8
a - b
## [1] 2
a / b
## [1] 1.666667
a %/% b
## [1] 1
a %% b
## [1] 2
```

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

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

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

Новую переменную можно создать на основе значений существующих переменных:

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

Посмотрим, что получилось:

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

Вы можете комбинировать переменные и заданные явным образом константы:

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

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

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

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

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

1.1.1 Числовые функции

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

Важнейшие примеры функций — математические. Это функции взятия корня `sqrt(x)`, модуля `abs(x)`, а также тригонометрические функции `sin(x)`, `cos(x)`, `tan(x)` и обратные к ним `asin(y)`, `acos(y)`, `atan(y)` и так далее. В качестве аргумента функции можно использовать переменную, константу, а также выражения:

```
sqrt(a)
## [1] 2.236068
sin(a)
## [1] -0.9589243
tan(1.5)
## [1] 14.10142
```

```
abs(a + b - 2.5)
## [1] 6.5
```

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

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

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

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

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

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

1.2 Строки

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

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

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

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

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

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

Посмотрим содержимое подстрок:

```
s1
## [1] "              ,"
s2
## [1] "           "
```

```
s3
## [1] "( .     )"
```

А теперь объединим их в одну:

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

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

Условно запишем значения переменных, как будто мы их знаем

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

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

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

1.3 Даты

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

```
birth <- as.Date('1986/02/18')
birth
## [1] "1986-02-18"
```

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

```
current <- Sys.Date()
current
## [1] "2017-09-21"
```

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

```
livedays <- current - birth
livedays
## Time difference of 11538 days
```

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

```
current + 40
## [1] "2017-10-31"
```

С другими примерами использования дат мы познакомимся в дальнейшем по мере работы с данными.

1.4 Логические

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

Следующие операторы приводят к возникновению логических переменных:

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

Посмотрим, как они работают:

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

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

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

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

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

Chapter 2

Векторы

В это модуле мы познакомимся с векторами – упорядоченными последовательностями объектов одного типа. Вектор является простейшей и одновременно базовой структурой данных в R. Понимание принципов работы с векторами необходимо для дальнейшего знакомства с более сложными структурами данных, такими как матрицы, фреймы данных, списки и массивы

2.1 Создание вектора

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

Существует множество способов создания векторов. Среди них наиболее употребительны:

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

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

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

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

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

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

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

Например:

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

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

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

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

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

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

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

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

Как видно, параметры функции `seq()` можно комбинировать различными способами и указывать в произвольном порядке (при условии, что вы используете полную форму (=). Главное, чтобы их совокупность *однозначно описывала последовательность*. Хотя, скажем, последний пример убывающей последовательности нельзя признать удачным с точки зрения наглядности.

Аналогичным образом можно создавать *последовательности дат*:

```
seq(from = as.Date('2016/09/01'), by = 1, length.out = 7) #                  2016/2017
## [1] "2016-09-01" "2016-09-02" "2016-09-03" "2016-09-04" "2016-09-05"
## [6] "2016-09-06" "2016-09-07"

seq(from = Sys.Date(), by = 7, length.out = 5) #             ,
## [1] "2017-09-21" "2017-09-28" "2017-10-05" "2017-10-12" "2017-10-19"
```

2.2 Работа с элементами вектора

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3 Анализ и преобразования векторов

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

```
min(lengths) #
## [1] 19
max(lengths) #
## [1] 45
range(lengths) #          =       -
```

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

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

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

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

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

2.4 Поиск и сортировка элементов

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

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

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

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

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

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

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

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

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

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

Chapter 3

Матрицы, фреймы данных и списки

В это модуле мы продвинемся дальше в изучении структур данных языка и рассмотрим такие важные его элементы как матрицы, фреймы данных и списки.

3.1 Матрицы

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

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

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

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

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

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

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

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

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

К матрицам можно применять операции, аналогичные операциям над векторами:

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

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

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

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

```
t(m)  #
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
m2<-matrix(-3:3,nrow = 3, ncol = 3)
## Warning in matrix(-3:3, nrow = 3, ncol = 3):         [7]
##                  [3]
m2
##      [,1] [,2] [,3]
## [1,]   -3    0    3
## [2,]   -2    1   -3
## [3,]   -1    2   -2
det(m2) #
## [1] -21
det(m)  #    !
## Error in determinant.matrix(x, logarithm = TRUE, ...): 'x'
```

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

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

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

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

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

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

Ура! Найденный элемент действительно равен 8.

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

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

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

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

Результат можно присоединить к уже созданной матрице:

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

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

Матрицы, однако, не дотягивают по функциональности до представления таблиц, и, в общем-то, не предназначены

для объединения разнородных данных в один набор (как мы это сделали). Если вы присоедините к матрице столбец с названиями веток метро, система не выдаст сообщение об ошибке, но преобразует матрицу в текстовую, так как текстовый тип данных способен представить любой другой тип данных:

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

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

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

3.2 Фреймы данных

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

> Мы специально не используем для перевода слова `data.frame` термин ‘таблица’, поскольку таблица — это достаточно общая категория, которая описывает концептуальный способ упорядочивания данных. В том же языке R для представления таблиц могут быть использованы как минимум две структуры данных: фрейм данных (data.frame) и тиббл (tibble), доступный в соответствующем пакете. Мы не будем использовать тибблы в настоящем курсе, но после его освоения вы вполне сможете ознакомиться с ними самостоятельною

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

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

К фреймам также можно пристыковывать новые столбцы:

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

Когда фрейм данных формируется посредством функции `data.frame()` и `cbind()`, названия столбцов берутся из

названий векторов. Обратите внимание на то, что листинге выше столбцы имеют заголовки, а строки — номера.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Обратимся по новому названию столбца:

```
t$
## [1] 28.0 40.0 45.0 19.0 38.0 40.5
t
##
## 1          28.0      20 0.7142857
## 2          40.0      21 0.5250000
## 3           45.0      22 0.4888889
```

```
## 4         19.0       12 0.6315789
## 5         38.0       24 0.6315789
## 6         40.5       22 0.4888889
```

3.3 Списки

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

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

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

Сооружаем список из трех элементов:

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

Можно дать элементам списка осмысленные названия при создании:

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

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

Теперь можно обратиться к элементу списка по его названию:

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

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

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

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

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

Использование *двойных скобок* отличает списки от векторов.

Bibliography