Минимальное введение в R

Дата фреймы

Дата фреймы (`dataframe`) это один из самых важных типов данных в R, который практически отождествляется с табличными данными. Он является особым видом списка, в котором каждый элемент представляет собой столбец таблицы и имеет одну и ту же длину (т.е. является прямоугольным списком).

Для создания дата фрейма используется функция `data.frame()` , которой просто передаются желаемые содержимые столбцов нашего дата фрейма:

```
> (data <- data.frame(letters[1:10], 1:10, 11:20))
   letters.1.10. X1.10 X11.20
1              a     1     11
2              b     2     12
3              c     3     13
4              d     4     14
5              e     5     15
6              f     6     16
7              g     7     17
8              h     8     18
9              i     9     19
10             j    10     20
> is.list(data)
[1] TRUE
```

Если мы нарушим требование одинаковой длины столбцов, то эта функция вернет ошибку:

```
> data.frame(letters[1:10], 1:10, 11:21)
Error in data.frame(letters[1:10], 1:10, 11:21) :
  arguments imply differing number of rows: 10, 11
```

Как можно видеть из успешного вывода `data.frame()` , строкам автоматически приписываются нумерующие их имена. Столбцы также получают какие-то, возможно не очень красивые, имена. И те, и другие можно задать самостоятельно ещё при создании дата фрейма:

```
> (data <- data.frame(id = letters[1:10], x = 1:10, y = 11:20, row.names = letters[26:17]))
  id  x  y
z  a  1 11
y  b  2 12
x  c  3 13
w  d  4 14
v  e  5 15
u  f  6 16
t  g  7 17
s  h  8 18
r  i  9 19
q  j 10 20
```

Или же можно поменять их после создания:

```
> names(data) <- c("first", "second", "third")
> rownames(data) <- 101:110
> data
    first second third
101     a      1    11
102     b      2    12
103     c      3    13
104     d      4    14
105     e      5    15
106     f      6    16
107     g      7    17
108     h      8    18
109     i      9    19
110     j     10    20
```

Ряд полезных функций над дата фреймами:

Функция	Описание
`head()`, `tail()`	Первые/последние 6 строк
`dim()`, `nrow()`, `ncol()`	Функции, возвращающие измерения дата фрейма
`str()`	Описывает структуру дата фрейма: количество строк, столбцов, а также имя, тип и несколько значений для каждой колонки

Индексация

Первый способ индексации `[ ]` с вектором в качестве параметра вернет нам дата фрейм с соответствующими столбцами:

```
> data[2]
    second
101      1
102      2
103      3
104      4
105      5
106      6
107      7
108      8
109      9
110     10
> data[c(1,3)]
    first third
101     a    11
102     b    12
103     c    13
104     d    14
105     e    15
106     f    16
107     g    17
108     h    18
109     i    19
110     j    20
> data[-3]
    first second
101     a      1
102     b      2
103     c      3
104     d      4
105     e      5
106     f      6
107     g      7
108     h      8
109     i      9
110     j     10
> class(data[-2])
[1] "data.frame"
```

Указание двух параметров позволяет нам извлекать отдельные элементы:

```
> data[7, 2]
[1] 7
```

Или же целые пересечения выбранных столбцов и строк:

```
> data[4:7, c("second", "third")]
    second third
104      4    14
105      5    15
106      6    16
107      7    17
```

Так как дата фреймы также являются и списками, то мы можем также извлекать отдельные столбцы как векторы:

```
> data
    first second third
101     a      1    11
102     b      2    12
103     c      3    13
104     d      4    14
105     e      5    15
106     f      6    16
107     g      7    17
108     h      8    18
109     i      9    19
110     j     10    20
> data[[1]]
 [1] a b c d e f g h i j
Levels: a b c d e f g h i j
> data[[2]]
 [1]  1  2  3  4  5  6  7  8  9 10
> data[[3]]
 [1] 11 12 13 14 15 16 17 18 19 20
```

Вывод команды `data[[1]]` для нас ещё не знаком: опишем его в следующей секции.

Факторы

Факторы это объекты данных, используемые для категоризации данных, сохраняя их в качестве уровней. Они полезны в случаях, когда необходимо описать данные, принимающие ограниченное количество значений.

Их можно создавать с помощью функции `factor()` (а с помощью `is.factor()` проверять что в):

```
> (test <- c("up", "up", "down", "down", "left", "right", "left", "right"))
[1] "up"    "up"    "down"  "down"  "left"  "right" "left"  "right"
> (factor_test <- factor(test))
[1] up    up    down  down  left  right left  right
Levels: down left right up
```

R выделяет различные строковые значения, преданные в функцию `factor()`, сортирует их в алфавитном порядке и объявляет *уровнями* этого фактора.

Получить их список и количество можно с помощью функций `levels()` и `nlevels()`:

```
> levels(factor_test)
[1] "down"  "left"  "right" "up"
> nlevels(factor_test)
[1] 4
```

Значения факторов (и их порядок) можно изменить, задав параметр `levels` вручную:

```
> factor(factor_test, levels=c("up", "right", "down", "left"))
[1] up    up    down  down  left  right left  right
Levels: up right down left
> factor(factor_test, levels=c("up", "right", "down"))
[1] up    up    down  down  <NA>  right <NA>  right
Levels: up right down
```

Странный вывод

```
> data[[1]]
 [1] a b c d e f g h i j
Levels: a b c d e f g h i j
```

из секции выше получился потому, что `data.frame()` по-умолчанию конвертировал символьный вектор в фактор.
Если вы предпочитаете не использовать факторы, то при создании дата фрейма можете указать параметр `stringsAsFactors`):

```
> (data <- data.frame(id = letters[1:10], x = 1:10, y = 11:20, stringsAsFactors = F))
   id  x  y
1   a  1 11
2   b  2 12
3   c  3 13
4   d  4 14
5   e  5 15
6   f  6 16
7   g  7 17
8   h  8 18
9   i  9 19
10  j 10 20
> data[[1]]
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Сохранение и загрузка дата фреймов

Пересоздадим наш дата фрейм:

```
> (data <- data.frame(id = letters[1:10], x = 1:10, y = 11:20))
   id  x  y
1   a  1 11
2   b  2 12
3   c  3 13
4   d  4 14
5   e  5 15
6   f  6 16
7   g  7 17
8   h  8 18
9   i  9 19
10  j 10 20
```

Допустим, что мы сконструировали табличные данные, которые необходимо передать другим людям. Есть множество способов сохранить их в файл, но в данном курсе мы ограничимся сохранением и загрузкой в csv-файлы.

Сначала выберите рабочую директорию (в RStudio: Session > Set working directory > Choose Directory...), в которой вы хотите записать файл и выполните следующую команду: `write.csv(data, "my_data.csv")` .

В результате в выбранной директории должен появиться файл `my_data.csv` следующего содержания:

```
"","id","x","y"
"1","a",1,11
"2","b",2,12
"3","c",3,13
"4","d",4,14
"5","e",5,15
"6","f",6,16
"7","g",7,17
"8","h",8,18
"9","i",9,19
"10","j",10,20
```

Каждая строка этого файла, кроме первой, соответствует строке из нашей таблицы, а значения из разных столбцов разделены запятыми.

Загрузить эти данные назад в память почти столь же просто:

```
> (loaded <- read.csv("my_data.csv", row.names = 1))
   id  x  y
1   a  1 11
2   b  2 12
3   c  3 13
4   d  4 14
5   e  5 15
6   f  6 16
7   g  7 17
8   h  8 18
9   i  9 19
10  j 10 20
```

Дополнительный параметр `row.names = 1` нам понадобился чтобы указать, что первый столбец в нашем файле - имена столбцов, а не данные. Сравните с этим следующие листинги:

```
> read.csv("my_data.csv")
    X id  x  y
1   1  a  1 11
2   2  b  2 12
3   3  c  3 13
4   4  d  4 14
5   5  e  5 15
6   6  f  6 16
7   7  g  7 17
8   8  h  8 18
9   9  i  9 19
10 10  j 10 20
> read.csv("my_data.csv", row.names = 4)
    X id  x
11  1  a  1
12  2  b  2
13  3  c  3
14  4  d  4
15  5  e  5
16  6  f  6
17  7  g  7
18  8  h  8
19  9  i  9
20 10  j 10
> read.csv("my_data.csv", row.names = 2)
   X  x  y
a  1  1 11
b  2  2 12
c  3  3 13
d  4  4 14
e  5  5 15
f  6  6 16
g  7  7 17
h  8  8 18
i  9  9 19
j 10 10 20
```

Ещё о загрузке

Функции `read.csv()` и `write.csv()` имеют множество опций и мы не имеем возможности поговорить о них всех. Но приведем пример о паре из них, проведя загрузку данных из Unicode Character Database:

```
unicode <- read.csv("UnicodeData.txt")
```

Понять, что данные были загружены неверно можно выполнив:

```
> head(unicode)
                X0000..control..Cc.0.BN.....N.NULL....
1    0001;<control>;Cc;0;BN;;;;;N;START OF HEADING;;;;
2       0002;<control>;Cc;0;BN;;;;;N;START OF TEXT;;;;
3         0003;<control>;Cc;0;BN;;;;;N;END OF TEXT;;;;
4 0004;<control>;Cc;0;BN;;;;;N;END OF TRANSMISSION;;;;
5             0005;<control>;Cc;0;BN;;;;;N;ENQUIRY;;;;
6         0006;<control>;Cc;0;BN;;;;;N;ACKNOWLEDGE;;;;
> ncol(unicode)
[1] 1
```

Видимые проблемы:

1. Вместо разделения данных на столбцы, каждая строка файла была считана просто как последовательность символов, так как `read.csv()` ожидает что данные разделены запятыми.
2. В нашей таблице не было указано названий для столбцов и поэтому первая строка была воспринята как одно длинное название.

Чтобы исправить эти проблемы укажем два дополнительных параметра:

```
unicode <- read.csv("UnicodeData.txt", sep=";", header = F)
> head(unicode)
    V1        V2 V3 V4 V5 V6 V7 V8 V9 V10                 V11 V12 V13 V14 V15
1 0000 <control> Cc  0 BN    NA NA      N                NULL  NA
2 0001 <control> Cc  0 BN    NA NA      N    START OF HEADING  NA
3 0002 <control> Cc  0 BN    NA NA      N       START OF TEXT  NA
4 0003 <control> Cc  0 BN    NA NA      N         END OF TEXT  NA
5 0004 <control> Cc  0 BN    NA NA      N END OF TRANSMISSION  NA
6 0005 <control> Cc  0 BN    NA NA      N             ENQUIRY  NA
> ncol(unicode)
[1] 15
```

Стало лучше! Но проблемы остались:

```
> class(unicode[[1]])
[1] "factor"
> class(unicode[[11]])
[1] "factor"
```

Хотя это должны быть число и строка соответственно. Мы уже знаем как решить проблему с 11-ым столбцом:

```
> unicode <- read.csv("UnicodeData.txt", sep=";", header = F, stringsAsFactors = F)
> class(unicode[[11]])
[1] "character"
```

Но проблему первого столбца это всё ещё не решает:

```
> class(unicode[[1]])
[1] "character"
```

Мы можем решить эту проблему конвертировав строки в числа после загрузки:

```
> unicode[1] <- strtoi(unicode[[1]], base = 16)
> class(unicode[[1]])
[1] "integer"
> head(unicode, n = 20)
   V1        V2 V3 V4 V5 V6 V7 V8 V9 V10                 V11 V12 V13 V14 V15
1   0 <control> Cc  0 BN    NA NA      N                NULL  NA
2   1 <control> Cc  0 BN    NA NA      N     START OF HEADING  NA
3   2 <control> Cc  0 BN    NA NA      N        START OF TEXT  NA
4   3 <control> Cc  0 BN    NA NA      N          END OF TEXT  NA
5   4 <control> Cc  0 BN    NA NA      N  END OF TRANSMISSION  NA
6   5 <control> Cc  0 BN    NA NA      N              ENQUIRY  NA
7   6 <control> Cc  0 BN    NA NA      N          ACKNOWLEDGE  NA
8   7 <control> Cc  0 BN    NA NA      N                BELL  NA
9   8 <control> Cc  0 BN    NA NA      N            BACKSPACE  NA
10  9 <control> Cc  0  S    NA NA      N CHARACTER TABULATION  NA
11 10 <control> Cc  0  B    NA NA      N       LINE FEED (LF)  NA
12 11 <control> Cc  0  S    NA NA      N      LINE TABULATION  NA
13 12 <control> Cc  0 WS    NA NA      N       FORM FEED (FF)  NA
14 13 <control> Cc  0  B    NA NA      N CARRIAGE RETURN (CR)  NA
15 14 <control> Cc  0 BN    NA NA      N            SHIFT OUT  NA
16 15 <control> Cc  0 BN    NA NA      N             SHIFT IN  NA
17 16 <control> Cc  0 BN    NA NA      N     DATA LINK ESCAPE  NA
18 17 <control> Cc  0 BN    NA NA      N   DEVICE CONTROL ONE  NA
19 18 <control> Cc  0 BN    NA NA      N   DEVICE CONTROL TWO  NA
20 19 <control> Cc  0 BN    NA NA      N DEVICE CONTROL THREE  NA
```

Если бы числа не были указаны в шестнадцатеричном формате, то мы также могли бы использовать параметр `colClasses`, напрямую указывающий класс, ожидаемый в каждом столбце.

Скрипты

Разумеется R не принуждает нас пользоваться только консолью, мы также можем сохранять последовательность инструкций для выполнения в файлах с расширением `.R`. Создайте в

RStudio новый файл для хранения скриптов: File > New File > R Script. Вы можете редактировать эти файлы в любом текстовом редакторе, но RStudio удобна тем, что позволяет сразу выполнять любые выделенные строки нажатием ctrl + enter. Выполнить весь скрипт можно выделив все строки.

Условные конструкции и циклы

Эти концепции должны быть понятны вам из других языков программирования, поэтому просто приведем их синтаксис в R:

```
# Comment
# R does not support multi-line comments :(
if (test_expression) {
    statement1
}
else {
    statement2
}

while (test_expression) # Comment after some code
{
    statement
    # This one has a standart meaning of leaving the loop early
    break
    # This one functions as a continue: skips the current loop iteration, but does not break anc
    next
}

for (val in sequence)
{
    statement
}

function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

Чтобы вернуть значение в конце функции, достаточно просто указать его на последней строке:

```
returnSeven <- function (x) {
    7
}
> returnSeven()
[1] 7
```

Если необходим также ранний выход из функции, то тогда необходимо использоваться `return()`:

```
testFunc <- function(x) {
    if (x < 10) {
        return (10)
    }
    x
}
> testFunc(11)
[1] 11
> testFunc(5)
[1] 10
```

Упражнение 1

Напишите функции переводящие коды символов юникода в сами символы и обратно.

Цифры с 0 по 9 по-порядку занимают коды от 48 до 57. Латинские буквы в верхнем регистре - с 65 по 90, в нижнем - с 97 по 122.

Вам понадобятся функции `as.character`, `toupper()` и вектор `letters`.

Упражнение 2

Вспомните как мы считывали данные из базы данных юникода:

```
unicode <- read.csv("UnicodeData.txt", sep=";", header = F, stringsAsFactors = F)
unicode[1] <- strtoi(unicode[[1]], base = 16)
```

В первом столбце каждой строки указывается "код символа" (всё несколько сложнее, но для простоты пусть всё будет так), к которому относятся все остальные данные строки. В 13-ом столбце указывается код, соответствующего ему символа из верхнего регистра, в 14-ом - из нижнего. Опираясь на предыдущее упражнение, напишите программу, которая переводит символы в верхний регистр, используя загруженные данные юникода. Если версии в верхнем регистре нет, то верните сам символ. Протестируйте её на латинских буквах и цифрах.

Функции `lapply()`, `sapply()`, `mapply()`

При написании программ на R важно и полезно уметь работать с семейством функций `apply`. В рамках этого курса мы ограничимся `lapply()`, `sapply()` и `mapply()`, опустив рассмотрение `apply()`, `mapply()`, `vapply()` и `tapply()`.

Функции `lapply()`, `sapply()` и анонимные функции

`lapply()` позволяет применять одну и ту же функцию к элементам некоторого списка, выдавая в качестве результата новый список, содержащий результаты применения этой функции (аналогичные функции во многих языках программирования часто называются `map`). Простой пример:

```
increment <- function (x) {
  x + 1
}

lapply(0:3, increment)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

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

Здесь мы применяли `lapply()` не к списку, а к вектору `0:3`, но этот вектор был автоматически конвертирован в список с помощью `as.list()`:

```
> as.list(0:4)
[[1]]
[1] 0

[[2]]
[1] 1

[[3]]
[1] 2

[[4]]
[1] 3

[[5]]
[1] 4
```

В результате применения каждый элемент переданного вектора (списка) был увеличен на единицу и после был создан новый вектор, в котором на месте каждого элемента стоит новый результат.

Для второго примера рассмотрим функцию `mean()`:

```
> mean(c(1,2,3,4,5))
[1] 3
> mean(rnorm(100))
[1] 0.02977744
> x <- list(c(4,5,6,7,8,9,10), rnorm(1000), c(0, 4, 6, 7))
> lapply(x, mean)
[[1]]
[1] 7

[[2]]
[1] -0.04817959

[[3]]
[1] 4.25
```

В данном примере `mean()` возвращает лишь одно число на каждый элемент списка, - хранить такие простые данные в списке неудобно и мы бы предпочли использовать вектор. Здесь нам больше подходит функция `sapply()`, которая пытается автоматически упростить список до вектора:

```
> sapply(x, mean)
[1]  7.00000000 -0.04817959  4.25000000
```

Другой пример использует функцию `runif()` (генерация равномерно распределённых на отрезке случайных величин) для создания сложного списка из простого вектора:

```
> lapply(1:4, runif)
[[1]]
[1] 0.9371784

[[2]]
[1] 0.2663671 0.9135515

[[3]]
[1] 0.4207341 0.6614571 0.7826202

[[4]]
[1] 0.5778938 0.3605308 0.8295867 0.3073766
```

Заметим, что `sapply` в данной ситуации не приведет к упрощению:

```
> sapply(1:4, runif)
[[1]]
[1] 0.7903074

[[2]]
[1] 0.996361389 0.007793943

[[3]]
[1] 0.64337374 0.07052609 0.08256133

[[4]]
[1] 0.008710204 0.030272970 0.405761657 0.423068943
```

Это связано с тем, что вектора, полученные в результате, имеют разную длину, а упрощение происходит лишь в случае одинаковой длины (если длина содержащихся векторов 1 - то список упроститься до вектора, иначе до матрицы). Если же вы всё же хотите упростить этот список до вектора, то можно просто применить к результату функцию `unlist()` :

```
> unlist(lapply(1:4, runif))
 [1] 0.8703990 0.9634819 0.1745614 0.3840827 0.0658432 0.3271702 0.4894636 0.6234074 0.5776397 0
```

Что если мы хотим сгенерировать случайные величины из другого отрезка (по умолчанию от 0 до 1)? В таком случае мы можем воспользоваться тем, что `lapply()` перебрасывает последующие параметры в функцию, применяемую к списку:

```
> lapply(1:4, runif, min = 5, max = 10)
[[1]]
[1] 8.945708

[[2]]
[1] 5.740579 5.665440

[[3]]
[1] 5.082904 8.131413 8.714838

[[4]]
[1] 7.765466 7.672827 5.914276 5.247557
```

`mapply()`

Если функции `lapply()` и `sapply()` могли применять функцию только к элементам одного объекта: одного списка или вектора. `mapply()` же позволяет "сшивать" несколько списков или векторов одной функцией и выдавать результат в виде списка. Например:

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

`mapply()` берет функцию `rep()` (повторяющую одно и тоже значение несколько раз), которая принимает два аргумента и поочередно подставляет элементы вектора `1:4` в качестве значений, которые требуется повторить, а элементы `4:1` в качестве количества повторений: 1 повторяется 4 раза, 2 повторяется 3, 3 повторяется 2, а 4 - 1, т.е. делает работу, эквивалентную:

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

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

Упражнение 3

Рассмотрите функцию `noise` :

```
noise <- function (n, mean, sd) {
    rnorm(n, mean, sd)
}
```

генерирующую `n` значений числового "шума" с данными средним значением `mean` и среднеквадратичным значением разброса `sd` .

Используйте `lapply()` , `sapply()` или `mapply()` чтобы создать список, содержащий 10 векторов с шумом со средним значением 2 и среднеквадратичным разбросом 0.2. При этом первый вектор должен содержать 25 значений, а каждый последующий - на 2 меньше (23, 21, 19, ...).

Упражнение 4

Посвторите предыдущее упражнение, но так, чтобы теперь среднее значение шума в каждом векторе равнялось его индексу (у первого - 1, у второго - 2 и т.д.).