Минимальное введение в R

Установка R

Базовый R

Вы можете скачать последнюю версию R с официального сайта: https://www.r-project.org/.
В этом курсе мы будем пользоваться лишь базовыми возможностями R, но, из-за потенциальных проблем с совместимостью, укажем что мы будем пользоваться версией 3.6.
Если у Вас уже установлен R, но вы не знаете какой именно версии, - введите 'R.version.string' в командной строке R.

R studio

Мы также настоятельно рекомендуем установить R studio:
https://www.rstudio.com/products/rstudio/.

Базовые понятия

Познакомимся с основами работы R. Для начала работы запустите R studio, - Вам сразу же должна открыться командная строка. Это всё что нам пока понадобится.

R как калькулятор

Самый простой способ использования R это произведение простых вычислений. Например введите следующее:

```
2 + 2 * 2
```

В ответ R просто напечатает 6.
Другие распространённые операции это `-`, `/` и `^` (`x^2` означает x^2).
Чуть более продвинутой функцией является возможность запоминать результаты вычислений.
Как и во всех других языках программирования, в R нам для этого понадобится создать переменную. И создание переменной, и присвоение ей значения делается с помощью "оператора присвоения" (`<-`).
О нём можно думать как о стрелке, присваивающей значение справа от неё,

переменной слева от неё. Например, чтобы запомнить в переменной `x` значение нашего предыдущего вычисления мы должны ввести:

```
x <- 2 + 2 * 2
```

Как Вы могли заметить, в этот раз R не вывел нам результат вычисления. Если Вам хочется узнать записанное в переменную значение, - просто введите `x` в консоль.

Записав значение в переменной мы можем использовать его в последующих вычислениях. Например, если нам хочется узнать чему равна половина `x` мы может ввести следующее:

```
y <- x / 2
y
```

В ответ R выведет 3.

> **Способы печати на экран**
>
> 1. Здесь мы видели что для вывода значения переменной в R достаточно просто напечатать её имя `x`.
> 2. Другой способ - использовать функцию `print(x)`. Хотя в большинстве случаев её использование может показаться лишним, она может сделать код более выразительным. Кроме того, она предоставляет ряд дополнительных возможностей, - сравните сами вывод `print(3.14159265359)` и `print(3.14159265359, digits=3)` (введите `?print` в консоли RStudio для более подробной информации).
> 3. Кроме того, небольшой трюк, - заключив операцию присваивания в скобки мы сделаем её выражением, которое необходимо вычислить и чьё значение будет равно присвоенному значению. Так, выше мы могли бы написать просто `(y <- x / 2)`, чтобы и присвоить `y` значение и одновременно вывести его в консоль.

Простейшие типы данных

Числа - не единственный тип данных, который может хранится в переменных. Рассмотрим ещё несколько простых примеров.

Логический тип данных (`Logical`)

Этот тип данных стандартно используется для хранения информации об истинности утверждений и принимает 2 значения - `TRUE` и `FALSE` (формально он также принимает значение `NA`, но об этом мы поговорим несколько позже).

Простейший пример получения этого типа значений - использование операций сравнения:

Оператор	Значение
`<`	строго меньше
`>`	строго больше
`<=`	меньше или равно
`>=`	больше или равно
`==`	значения совпадают
`!=`	значения различаются

```
> x <- 4
> x < 10
[1] TRUE
> (isSeven <- x == 7)
[1] FALSE
```

Если у Вас уже есть две логические переменные (или 2 логических выражения) `A` и `B`, то к ним можно применять стандартные логические операции:

Оператор	Смысл
`A & B`	логическое "и"
`A | B`	логическое "или"
`!A`	логическое отрицание

```
> x <- 16
> (greaterThanTen <- x > 10)
[1] TRUE
> (lessThanFifteen <- x < 15)
[1] FALSE
> greaterThanTen & !lessThanFifteen
[1] TRUE
```

Сделаем важное замечание - чтобы избежать досадных ошибок, - всегда окружайте знаки ваших операторов пробелами. Сравните выводы двух команд, различающихся только пробелами:

```
> (x<-2)
[1] 2
> (x < -2)
[1] FALSE
```

Символьный тип данных (`Character`)

Другой распространенный тип данных - это символьный тип, задаваемый с помощью двойных кавычек `"` :

```
my_name <- "Иванов Иван Иванович"
```

Пока укажем лишь на один способ модификации символьных данных - конкатенация, т.е. склеивание/сцепление строк. Эта операция может быть произведена с помощью функции `paste` :

```
paste("Hello,", "world!", "It's", "a", "nice", "day!")
```

Приведенная команда выдаст нам в ответ одну строку `"Hello, world! It's a nice day!"` . Нетрудно видеть, что `paste` просто объединяет вместе переданные ей строки, попутно вставляя между ними знаки пробела.

`paste` может принимает на вход объекты и не символьного типа, автоматически конвертируя их в строки:

```
> paste("Score:", 4576)
[1] "Score: 4576"
```

Кроме этого Вы можете заменить разделитель строк с пробела на любую другую строку, указав её в дополнительном параметре `sep` :

```
> paste(1, 3, 5, 2, 4, "pi", sep = " + ")
[1] "1 + 3 + 5 + 2 + 4 + pi"
```

Другие численные типы данных (`Numeric`)

Регулярные случаи (`Double` , `Integer` , `Complex`)

Любое введенное число в R по-умолчанию трактуется как число двойной точности (`Double`). Это значит, что когда Вы вводите `1` или `2` в R, то, хотя они выглядят как целые числа, в реальности они являются числами с плавающей запятой.

Если Вы хотите явно задать целое число (`Integer`), то Вам требуется добавить к нему суффикс "L". Т.е. `1` это число с плавающей запятой (`Double`), а `1L` это уже именно целое число (`Integer`).
Целые числа обычно используются если есть либо необходимость ограничить возможный диапазон значений, либо для сохранения места в памяти. `Integer` автоматически конвертируется в `Double` при необходимости.

R также предоставляет встроенный тип данных для работы с комплексными числами `Complex` , который определяется через чистую мнимую часть `i` :

```
> 1 + 2i
[1] 1+2i
```

Помимо удобства важно помнить, что поведение ряда функций может меняться в зависимости от того, передаете Вы им `Integer` или `Complex` значение. Например вызов `sqrt(-1)` приведет к ошибке. С другой стороны `sqrt(-1+0i)` выдаст нам `0+1i` , т.е. i.

`Inf` И `NaN`

Результаты не всех арифметических операций могут быть представлены в виде чисел. Иногда результатом условно подразумевается "бесконечность" $(1/0)$, а иногда результат и вовсе не определен $(0/0)$. Для таких случаев в R зарезервированы два специальных значения: `Inf` (*infinity*) и `NaN` (*not-a-number*).
Эти значения могут быть записаны в переменные и участвовать в арифметических операциях:

Expression	Result
`1 / 0`	`Inf`
`0/0`	`NaN`
`1/0 + 1/0`	`Inf`
`1/0 - 1/0`	`NaN`
`sqrt(-1)`	`NaN`
`NaN^0`	`1`
`0^NaN`	`NaN`
`0^0`	`1`
`0*Inf`	`NaN`

Expression	Result
`0*NaN`	`NaN`

Как правило, любые вычисления в которых участвует `NaN` возвращают `NaN` . Исключениями обычно являются лишь операции, которые всегда возвращают один и тот же результат, поэтому `NaN^0` возвращает `1` .

Отсутствующие значения

Отсутствующие значения играют огромную роль в статистике и часто требует изучение само их наличие. В R для того, чтобы представить значение, "которого нет", "которое отсутствует", используется `NA` .
`NA` отличается от `NaN` тем, что R все же трактует `NaN` как число (неудачный результат каких-либо вычислений), а `NA` это просто пропущенный тип данных, потенциально любого типа, включая символьный (`Character`) и логический (`Logical`). Аналогично `NaN` , большинство операций, включающих в себя `NA` снова производит `NA` :

```
> var <- NA
[1] NA
> var * 2
[1] NA
> NA^0
[1] 1
> NA | TRUE
[1] TRUE
```

Один из способов произвести значение `NA` "естественным образом" это операции сравнения:

```
> 1 == NaN
[1] NA
> NaN == NaN
[1] NA
> 1 == NA
[1] NA
> NA == NA
[1] NA
> NaN == NaN
[1] NA
```

Интересным здесь является то, что результат сравнения двух `NA` или `NaN` не является `TRUE` . Это связано с тем, что `NA` не является конкретным значением,а лишь представляет собой "затычку" для отсутствующего значения. Поэтому простым сравнением мы не сможем узнать

содержит ли переменная одно из этих значений. Вместо этого в R предусмотрены специальные функции `is.na()` и `is.nan()`:

```
> x <- NA
> y <- NaN
> x == NA
[1] NA
> is.na(x)
[1] TRUE
> x == NaN
[1] NA
> is.nan(x)
[1] FALSE
> y == NA
[1] NA
> is.na(y)
[1] TRUE
> y == NaN
[1] NA
> is.nan(y)
[1] TRUE
```

Из приведенного листинга видно, что `is.na` возвращает `TRUE` не только для `NA`, но и для `is.nan`.

В приведенной ниже таблице указаны результаты этих функций для разных типов аргументов (включенные также функции `is.finite()` и `is.infinite()`, имеющие очевидный смысл проверки чисел на конечность/бесконечность):

	3	**1/0, Inf**	**NaN**	**NA**
is.finite()	TRUE	FALSE	FALSE	FALSE
is.infinite()	FALSE	TRUE	FALSE	FALSE
is.na()	FALSE	FALSE	TRUE	TRUE
is.nan()	FALSE	FALSE	TRUE	FALSE

Отличный способ сравнения подобных значений это функция `identical()`, являющейся простым и безопасным путем сравнения двух значений на *точное* равенство:

```
> identical(NA, NA)
[1] TRUE
> identical(NA, NaN)
[1] FALSE
> identical(NaN, NaN)
[1] TRUE
> identical(1, NA)
[1] FALSE
> identical(1/0, Inf)
[1] TRUE
```

Немного о `NULL`

Ради устранения возможной путаницы скажем пару слов про `NULL`.
`NULL` используется для обозначения пустого (не обязательно численный) объекта и может быть возвращен некоторыми функциями для обозначения незаданных значений.
Не погружаясь в технические детали пока лишь отметим, что общая идея примерно такова:

- `NaN` - означает $0/0$, результат вычисления не являющийся ни числом, ни бесконечностью;
- `NA` - отсутствующее, несуществующее значение;
- `NULL` - пустой объект.

Проверить является ли значение `NULL` можно с помощью функции `is.null()`.

Определение типа переменных

R это язык с динамическими типами и одной и той же переменной можно присваивать разные типы значений:

```
> (x <- "I am string")
[1] "I am string"
> (x <- 7)
[1] 7
> (x <- TRUE)
[1] TRUE
> (x <- "And now I am string again")
[1] "And now I am string again"
```

Это может быть как полезным свойством языка, так и источником головной боли.
Это значит, по крайней мере, что кроме тестов на "вырожденные" значения нам нужны тесты на принадлежность к конкретному типу: `is.logical()`, `is.character()`, `is.integer()`, `is.double()`, `is.complex()`:

```
> is.character("Text")
[1] TRUE
> is.double(2)
[1] TRUE
> is.logical(1 == 2)
[1] TRUE
> is.complex(1+3i)
[1] TRUE
```

Отметим, что тесты `is.integer()`, `is.double()`, `is.complex()` требуют от переменной иметь в точности такой тип:

```
> is.integer(2)
[1] FALSE
> is.complex(2)
[1] FALSE
> is.double(1L)
[1] FALSE
> is.complex(1L)
[1] FALSE
```

Если же Вы просто хотите узнать является ли переменная каким-нибудь числом, что используйте общий тест `is.numeric`:

```
> is.numeric(1)
[1] TRUE
> is.numeric(3L)
[1] TRUE
> is.numeric(4+5i)
[1] FALSE
```

Вектора

Простейшей структурой данных в R является вектор. Даже переменные `x` и `y`, созданные нами для хранения чисел, считаются векторами длины 1.

Создание векторов

Функция c()

Простейший способ создать вектор состоит в использовании функции `c()` (название происходит от слов `concatenate` и `combine`).
Например чтобы создать вектор из чисел 2.17, 3 и 4.57 достаточно ввести `c(2.17, 3, 4.57)`.
Запишем этот вектор в переменную `z` и попросим вывести её содержание:

```
> z <- c(2.17, 3, 4.57)
> z
[1] 2.17 3.00 4.57
```

Как мы видим, R выдал нам `[1] 2.17 3.00 4.57` , без каких-либо запятых.

Раз аргументы 2.17, 3 и 4.57, с которыми мы вызывали `c()` , это вектора длины 1, то может быть мы можем использовать её и для объединения векторов большей длины? Ответ - да. Следующая команда создаст вектор длины 6:

```
> c(5, z, c(7.6, 4))
[1] 5.00 2.17 3.00 4.57 7.60 4.00
```

Оператор `:` , функция `seq()`

Другой способ создать вектор - это порождение последовательности чисел по некоторому правилу. Выражение `a:b` наполняет вектор числовой последовательностью от `a` до `b` с шагом ± 1, первый элемент которой равен `a` :

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> pi:10
[1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
> 1:1
[1] 1
```

Во многих ситуациях нам хотелось бы иметь больше контроля над создаваемой последовательностью, чем нам предоставляет оператор `:` . В таких случаях мы может использовать функцию `seq()` .
Передав ей два числовых аргумента мы получим тот же результат, что и при использовании `:` :

```
> seq(1, 10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(pi, 10)
[1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593
> seq(10, 1)
 [1] 10  9  8  7  6  5  4  3  2  1
> seq(1, 1)
[1] 1
```

Но она имеет и ряд дополнительных возможностей. Например, используя дополнительный аргумент `by` , мы может изменить инкремент последовательности:

```
> seq(1, 5, by = 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Вместо инкремента мы можем фиксировать длину конечной последовательности и `seq()` сама выберет требуемую длину шага:

```
> seq(0, 1, length = 5)
[1] 0.00 0.25 0.50 0.75 1.00
> seq(0, 1, length = 6)
[1] 0.0 0.2 0.4 0.6 0.8 1.0
> seq(0, 1, length = 7)
[1] 0.0000000 0.1666667 0.3333333 0.5000000 0.6666667 0.8333333 1.0000000
```

Функция `rep()`

Укажем также полезную функцию `rep()`. Её простейшее использование - повторить переданный ей вектор определенное количество раз:

```
> rep(1, 5)
[1] 1 1 1 1 1
> rep(c(1,2,3), 3)
[1] 1 2 3 1 2 3 1 2 3
> rep("SPAM", 10)
 [1] "SPAM" "SPAM" "SPAM" "SPAM" "SPAM" "SPAM" "SPAM" "SPAM" "SPAM" "SPAM"
```

Арифметические операции над векторами

Арифметические операции применяются к векторам покомпонентно. То есть, если вы хотите удвоить каждую компоненту вектора `z`, а затем прибавить к ним сотню, достаточно ввести:

```
z * 2 + 100
```

На что R ответит: `[1] 104.34 106.00 109.14`.

Это же правило применимо и ко многим функциям. Например, для вычисления квадратного корня можно применять функцию `sqrt()`, которая также будет действовать покомпонентно:

```
sqrt(z)
```

и выведет `[1] 1.473092 1.732051 2.137756`.

Повторное использование значений векторов

Если Вы примените арифметические операции к векторам одинаковой длины, то R просто применит их покомпонентно. Так

```
> c(1, 2, 3) + c(5, 7, 5)
[1] 6 9 8
```

выдаёт нам `[1] 6 9 8` .

Но что, если вы применяете арифметические операции к векторам разной длины? В таком случае R повторно использует значения более короткого вектора.
Когда мы просили R вычислить `z * 2 + 100` мы комбинировали вектор `z` длины 3 и вектора `2` и `100` длины 1. Поэтому в реальности R вычислял следующее:
`z * c(2, 2, 2) + c(100, 100, 100)` .
Другой хорошей иллюстрацией этого процесса служит следующий пример:

```
c(1, 2, 3, 4) + c(0, 100)
```

выдаст `[1] 1 102 3 104` . Здесь R просто поочередно прибавлял 0 и 100 к значениям `c(1, 2, 3, 4)` .

Заметим ещё, что если длина короткого вектора не делит без остатка длину большего вектора, то, хотя R по-прежнему повторно использовать значения более короткого, мы получим предупреждение о том, что что-то может быть не так:

```
> c(1, 2, 3, 4) + c(0, 10, 100)
[1]   1  12 103   4
Warning message:
In c(1, 2, 3, 4) + c(0, 10, 100) :
  longer object length is not a multiple of shorter object length
```

Извлечение значений из векторов

Для нужд этой подсекции создадим вектор `x` , содержащий как случайные числа, так и ряд пропущенных (`NA`) значений:

```
> rand <- rnorm(20)
> NAs <- rep(NA, 20)
> (x <- sample(c(rand, NAs), 20))
 [1] -2.91857697 -2.65096792          NA -0.08976424          NA          NA          NA
[10]          NA  0.88669572 -0.64943055 -0.73531225          NA  0.19626571 -1.05730103
[19] -0.70417490          NA
```

Первая строка создает вектор `rand`, содержащий 20 случайных значений, сгенерированных согласно стандартному нормальному распределению с помощью функции (`rnorm`).
Вторая строка создает вектор из 20 пропущенных значений.
Наконец, третья строка наугад выбирает 20 элементов из обоих векторов с помощью функции `sample()`.
Разумеется, после выполнения этих операций, содержание вектора `x` у Вас будет отличаться.

Начнем с того, что извлечем первые 10 элементов нашего вектора:

```
> x[1:10]
 [1] -2.91857697 -2.65096792          NA -0.08976424          NA          NA          NA
[10]          NA
```

Переданный в квадратных скобках аргумент - это индексный вектор, который может иметь 4 разных типа:

1. Логический вектор;
2. Вектор из положительных чисел;
3. Вектор из отрицательных чисел;
4. Символьный вектор.

Логический индексный вектор

Одна из самых распространенных задач - избавится ото всех пропущенных значений в наших данных. Мы сможем добиться этого с помощью логического индексного вектора.
Если мы подставляем вектор из логических значений в качестве индексного вектора в выражении `x[]`, то будут выбраны лишь те значения вектора `x`, которым соответствуют значения `TRUE`. В обще случае этот процесс может быть проиллюстрирован следующей таблицей:

Вектор `v`	a_1	a_2	a_3	a_4	a_5	...	a_n
Индексный вектор `i`	TRUE	TRUE	FALSE	TRUE	FALSE	...	TRUE
	a_1	a_2		a_4		...	a_n
`v[i]`	a_1	a_2	a_4	...	a_n		

Таким образом, для того, чтобы отфильтровать все пропущенные значения нам необходимо построить логический вектор, содержащий `FALSE` на местах, соответствующих `NA` и `TRUE` иначе. Этого легко добиться с помощью функции `is.na()`:

```
> is.na(x)
 [1] FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE F
[20]  TRUE
> !is.na(x)
 [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
[20] FALSE
```

Запишем полученный результат в вектор `y`:

```
> (y <- x[!is.na(x)])
 [1] -2.91857697 -2.65096792 -0.08976424  0.88669572 -0.64943055 -0.73531225  0.19626571 -1.0573
[10] -0.70417490
```

Индексный вектор из положительных чисел

Ранее мы извлекли первые 10 элементов из вектора `x` передав ему список интересующих нас индексов:

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

Заметьте, что, в отличии от многих других языков программирования, индексирование элементов здесь начинается не с нуля, а с единицы.

Таким образом с помощью данной конструкции мы может извлекать любые индексы и в любом порядке:

```
> temp <- 1:100
> temp[c(45, 65, 89)]
[1] 45 65 89
> temp[c(59, 37, 92, 1)]
[1] 59 37 92  1
> temp[7]
[1] 7
```

Но необходимо быть осторожными, так как R не попытается нас остановить, если мы вылезем за пределы доступных индексов:

```
> temp[0]
integer(0)
> temp[1000]
[1] NA
```

Индексный вектор из отрицательных чисел

Что если мы хотим извлечь все элементы кроме 5-ого и 19-ого?
Создавать вектор перечисляющий все такие индексы достаточно неудобно. К счастью, достаточно указать нежелательные индексы со знаком минус и мы извлечем все элементы кроме них:

```
> x[c(-5, -19)]
 [1] -2.91857697 -2.65096792          NA -0.08976424          NA          NA          NA
[10]  0.88669572 -0.64943055 -0.73531225          NA  0.19626571 -1.05730103          NA -0.1422
> temp <- 1:20
> temp[c(-5, -19)]
 [1]  1  2  3  4  6  7  8  9 10 11 12 13 14 15 16 17 18 20
```

Для тех же целей можно использовать и вектор с положительными индексами, если поместить перед ним знак минуса:

```
> x[-c(5, 19)]
 [1] -2.91857697 -2.65096792          NA -0.08976424          NA          NA          NA
[10]  0.88669572 -0.64943055 -0.73531225          NA  0.19626571 -1.05730103          NA -0.1422
> temp[-c(5, 19)]
 [1]  1  2  3  4  6  7  8  9 10 11 12 13 14 15 16 17 18 20
```

Смешение положительных и отрицательных значений приведет к ошибке:

```
> x[c(1, 2, -5)]
Error in x[c(1, 2, -5)] : only 0's may be mixed with negative subscripts
```

Символьный индексный вектор

Здесь нам понадобится концепция именованных элементов. Присвоить имена элементам вектора достаточно просто:

```
> (vect <- c(foo = 11, bar = 2, norf = NA))
 foo  bar norf
  11    2   NA
> names(vect)
[1] "foo"  "bar"  "norf"
```

Как нетрудно догадаться, приведенная функция `names` возвращает вектор имен.

Альтернативно мы можем сначала создать вектор с некоторыми значениями, а затем присвоить его значениям имена, снова используя `names` :

```
> (vect2 <- c(11, 2, NA))
[1] 11  2 NA
> (names(vect2) <- c("foo", "bar", "norf"))
[1] "foo"  "bar"  "norf"
> vect2
 foo  bar norf
  11    2   NA
```

> **Замечание**
> Обсуждение того, почему мы может так вольно обращаться с `names()` и что это такое выходит сильно за рамки этого курса. Либо просто примите как данность, либо самостоятельно почитайте:
> http://adv-r.had.co.nz/Data-structures.html

С помощью функции `identical()` можно убедиться, что полученные вектора оказались идентичными:

```
> identical(vect, vect2)
[1] TRUE
```

После создания вектором с именованными элементами, ты можем извлекать необходимые нам значения по их именам:

```
> vect["foo"]
foo
 11
> vect["bar"]
bar
  2
> vect[c("foo", "norf")]
 foo norf
  11   NA
```

Проверка принадлежности вектору

Отметим также способ проверить что некоторый элемент принадлежит данному вектору. Это можно сделать с помощью оператора `%in%` :

```
> 3 %in% c(1,3,5,2,4)
[1] TRUE
> 3 %in% c(5,7,8)
[1] FALSE
```

Проверить принадлежность всего вектора некоторому другому вектору можно с помощью функции `all()`, возвращающей `TRUE` только если переданный ей логический вектор не содержит `FALSE`:

```
> c(1,5,3) %in% c(1,2,3,4,5)
[1] TRUE TRUE TRUE
> all(c(1,5,3) %in% c(1,2,3,4,5))
[1] TRUE
> c(1,7,5,3) %in% c(1,2,3,4,5)
[1]  TRUE FALSE  TRUE  TRUE
> all(c(1,7,5,3) %in% c(1,2,3,4,5))
[1] FALSE
```

Атомарные вектора

На данный момент мы рассматривали лишь числовые вектора. Но вектор могут содержать и другие типы данных - логические, символьные, целые, комплексные и т.д.. Все их будет объединять то, что они будут содержать один и только один тип данных, т.е. они будут так называемыми *атомарными векторами* (atomic vectors).

Создадим простой числовой вектор и применим к нему операции сравнения `<` (строго меньше), `>=` (больше или равно). Как и в случае арифметических операций, операторы `<` и `>=` будет применяться покомпонентно и породит логический вектор:

```
> num_vect <- c(2, 0.3, -7, 7)
> num_vect < 1
[1] FALSE  TRUE  TRUE FALSE
> num_vect >= 6
[1] FALSE FALSE FALSE  TRUE
```

При необходимости вектор можно создавать и напрямую:
`log_vect <- c(TRUE, FALSE, FALSE, TRUE)`.
Точно также можно создать и символьный вектор:

```
> (my_char <- c("This", "is", "vector"))
[1] "This"   "is"     "vector"
```

Приведение типов

Так как все элементы должны иметь один и тот же тип, то при попытке комбинирования объектов нескольких разных типов, они будут *приведены* к единому типу.
Так, комбинирование строк и чисел в одном векторе дает нам список строк:

```
> c("A text", 7L, 19.3)
[1] "A text" "7"      "19.3"
```

А комбинирование чисел и логических значений переводит `TRUE` в 1 и `FALSE` в 0:

```
> c(12, TRUE, FALSE, TRUE)
[1] 12  1  0  1
```

Порядок приведения базовых типов таков: `logical > integer > double > character` .

Приведение значений вектора может произойти и автоматически, например при передаче его как аргумента функции.
Это может быть весьма удобно. Например, мы можем использовать функцию `sum` , находящую сумму значений

```
> sum(2, 5, c(7, 1, 2))
[1] 17
```

чтобы найти количество истинных или ложных значений в логическом векторе:

```
> booleanVector <- c(F, T, T, T, F, F, T)
> v
[1] 4
> sum(!booleanVector)
[1] 3
```

Но автоматическое приведение типов может также быть и источником проблем, так как может привести к неожиданным результатам:

```
> 1 == "1"
[1] TRUE
> identical(1, "1")
[1] FALSE
```

Узнать текущий тип вектора можно с помощью функции `typeof`

```
> typeof(c(1,2L,3))
[1] "double"
> typeof(c(1L,2L))
[1] "integer"
> typeof(booleanVector)
[1] "logical"
```

Списки

Вектора, которые мы создавали выше были так называемыми "атомарными векторами" (atomic vectors), способными содержать лишь один тип данных: числа, строки, логические значения и т.д.. Списки специальный тип векторов, способных содержать элементы различных типов. Списки могут быть созданы с помощью функции `list()` , принимающей на вход произвольное количество аргументов:

```
> (x <- list(1, "a", TRUE, 1 + 4i))
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

Вывод содержания списка выглядит заметно сложнее такого вывода для атомарных векторов. Мы более детально изучим его ниже, пока же просто примите что индексы элементов указаны в двойных квадратных скобках.

Как и вектора, списки можно объединять в один при помощи функции `c()` :

```
> c(list(1, "yes!"), list(3, "no!"))
[[1]]
[1] 1

[[2]]
[1] "yes!"

[[3]]
[1] 3

[[4]]
[1] "no!"
```

Индексация списков

Списки можно индексировать так же как и атомарные вектора:

```
> x[c(1,3)]
[[1]]
[1] 1

[[2]]
[1] TRUE

> x[4]
[[1]]
[1] 1+4i

> x[-1]
[[1]]
[1] "a"

[[2]]
[1] TRUE

[[3]]
[1] 1+4i
```

Также для списков сохраняется возможность символьной индексации:

```
> (y <- list(foo="word", FALSE, bar=7))
$foo
[1] "word"

[[2]]
[1] FALSE

$bar
[1] 7

> y[c("foo", "bar")]
$foo
[1] "word"

$bar
[1] 7

> y[c(1, 2)]
$foo
[1] "word"

[[2]]
[1] FALSE

> y[c(1, "bar")]
$<NA>
NULL

$bar
[1] 7

> y[2]
[[1]]
[1] FALSE
```

Извлечение конкретных элементов

Можно заметить, что использованные способы индексации возвращают не сами элементы (атомарные вектора), а снова их списки. Это вполне естественно: списки могут содержать различные типы данных и мы просто не смогли бы представить `y[c(1, 2)]` в виде списка. `y[2]` могло бы вернуть нам логический вектор из одного элемента, но это сделало бы поведение оператора `[ ]` непоследовательным (более сложным и менее предсказуемым).

Поэтому для извлечения собственно значений из списка используется другой оператор - двойные квадратные скобки `[[ ]]` :

```
> x[[1]]
[1] 1
> x[[3]]
[1] TRUE
> y[[1]]
[1] "word"
> y[[2]]
[1] FALSE
```

Заметим, что он не пригоден для извлечения нескольких значений сразу, даже когда они имеют один и тот же тип:

```
Error in y[[c(1, 2)]] : subscript out of bounds
> y[[1,2]]
Error in y[[1, 2]] : incorrect number of subscripts
> list(1,2,3)[[c(1,2)]]
Error in list(1, 2, 3)[[c(1, 2)]] : subscript out of bounds
> list(1,2,3)[[1,2]]
Error in list(1, 2, 3)[[1, 2]] : incorrect number of subscripts
```

Операторы индексации можно сочетать друг с другом:

```
> list(c(1,2,3), c(4,5,6))[[2]][2]
[1] 5
```

Для извлечения одиночных именованных значений также можно использовать оператор `$` :

```
> y$foo
[1] "word"
> y$bar
[1] 7
> y$car
NULL
```

Его основное отличие от `[[ ]]` в том, что он не позволяет использовать "вычисленные" имена:

```
> name <- "foo"
> y[[name]]
[1] "word"
> y$name
NULL
```

> **Замечание**
>
> Оператор `[[ ]]` также можно использовать для извлечения одиночных элементов из

> атомарных векторов. В этом случае отличие от оператора `[ ]` будет состоять в том, что имя выбранного элемента будет забыто:

```
> (vect <- 1:10)
[1]  1  2  3  4  5  6  7  8  9 10
> (vect <- 1:3)
[1] 1 2 3
> (names(vect) <- c("do", "re", "mi"))
[1] "do" "re" "mi"
> vect
do re mi
1  2  3
> vect[2]
re
2
> vect[[2]]
[1] 2
```

Подробнее о выводе списков

Как мы уже видели выше, вывод списков отличается от вывода векторов, даже для содержащих всего один элемент:

```
> (v <- c(TRUE))
[1] TRUE
> (l <- list(TRUE))
[[1]]
[1] TRUE
```

Почему-то в списках наш элемент индексируется двумя единицами!

Происходящее поможет объяснить вывод следующего списка из списка действительных чисел, именованной строки и комплексного числа:

```
> (z <- list(1:100, first="not", 6-5i))
[[1]]
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  2
 [30]  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  5
 [59]  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  8
 [88]  88  89  90  91  92  93  94  95  96  97  98  99 100

$first
[1] "not"

[[3]]
[1] 6-5i
```

Как мы можем видеть из данного вывода, - именно число в двойных квадратных скобках указывает на номер элемента списка. А число в одиночных скобках - лишь часть стандартного вывода этого первого элемента, - атомарного вектора. Из вывода второго элемента списка мы видим что для именованных элементов индекс заменяется собственно именем.

Исследуйте самостоятельно вывод списков `x` и `y`, введенных выше как

```
> x <- list(1, "a", TRUE, 1 + 4i)
> y <- list(foo="word", FALSE, bar=7)
```

и убедитесь что понимаете какая часть к чему относится.

Списки из списков

Разберем ещё случай вывода списка, содержащего списки. Чтобы, в случае встречи с такой конструкцией, вы не испугались.

Рассмотрим следующее:

```
> list(list("first", "here"), list("next", "there"))
[[1]]
[[1]][[1]]
[1] "first"

[[1]][[2]]
[1] "here"


[[2]]
[[2]][[1]]
[1] "next"

[[2]][[2]]
[1] "there"
```

Теперь только перед выводом первой строки `"first"` нам показывают четыре индекса! Зачем и что они обозначают?

Строки с одним индексом в двойных скобках (`[[1]]` и `[[2]]`) обозначают, что после них будут выводиться списки-компоненты нашего внешнего списка: `list("first", "here")` и `list("next", "there")` . Как и раньше.

Остальные строки составляют вывод вложенных списков-компонент с тем отличием, что теперь в двойных скобках указывается два индекса: `[[индекс_вложенного_списка]][[индекс_его_элемента]]` . Происходящее снова становится понятнее, если мы добавим имена для наших списков:

```
> list(self=list("first", "here"), other=list("next", "there"))
$self
$self[[1]]
[1] "first"

$self[[2]]
[1] "here"


$other
$other[[1]]
[1] "next"

$other[[2]]
[1] "there"
```

О такой индексации можно думать как о структуре файловой системы: `$self[[2]]` сначала выбирает подсписок с именем `self`, а потом его второй элемент. Точно также `[[2]][[1]]` означает что мы сначала выбираем (во внешнем списке) второй список, а затем берем его первый элемент. Так же, как если бы мы хотели получить его значение:

```
> list(list("first", "here"), list("next", "there"))[[2]][[1]]
[1] "next"
```

Те же правила распространяются и на большую вложенность списков. Попробуйте сами предсказать, поиграть и разобраться с выводом списков на подобие:

```
folded_list <- list(list(1, TRUE), 7, list("yes!", list("no!", 8+3i, "oh my god!"), FALSE))
```

Массивы

До сих пор мы рассматривали лишь одномерные вектора. Массивы это специальный тип векторов, позволяющие хранить многомерные, "прямоугольные" данные. "Прямоугольность" означает что каждая строка, каждый столбец и все их многомерные аналоги имеют одинаковую длину.

Основной способ создания массивов - функция `array()`, которой требуется передать вектор с исходными данными и измерения (длины столбцов и строк) массива, например:

```
> array(1:20, dim = c(4, 5))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Как можно видеть, массив заполняется из данного вектора по колонкам.

Синтаксис `dim =` означает, что мы передаём значение именно параметру с названием `dim`. Такая запись позволяет передавать значения параметрам в любом порядке и делает явным что именно значит каждое передаваемое функции значение.

Аналогично можно создать и массив большей размерности. При попытке вывести его значения в консоль вы увидите набор его двумерных срезов:

```
> array(1:80, dim = c(4, 5, 2, 2))
, , 1, 1

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

, , 2, 1

     [,1] [,2] [,3] [,4] [,5]
[1,]   21   25   29   33   37
[2,]   22   26   30   34   38
[3,]   23   27   31   35   39
[4,]   24   28   32   36   40

, , 1, 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   41   45   49   53   57
[2,]   42   46   50   54   58
[3,]   43   47   51   55   59
[4,]   44   48   52   56   60

, , 2, 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   61   65   69   73   77
[2,]   62   66   70   74   78
[3,]   63   67   71   75   79
[4,]   64   68   72   76   80
```

При необходимости можно также указать имена для отдельных строк и столбцов через список символьных векторов:

```
> array(1:20, dim = c(4, 5), dimnames = list(c("one", "two", "three", "four"), c("раз", "два", "
          раз  два  три
one         1    5    9
two         2    6   10
three       3    7   11
four        4    8   12
```

Заметим что следующий вызов приведет к тому же результату:

```
> array(1:20, dimnames = list(c("one", "two", "three", "four"), c("раз", "два", "три", "четыре",
...
```

Также вы можете преобразовать существующий одномерный вектор в массив, установив его размерности напрямую:

```
> vect <- 1:20
> dim(vect) <- c(4, 5)
> vect
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Существенное отличие этого способа в том, что в векторе должно быть в точности подходящее количество элементов, чтобы организовать их в массив с данными измерениями:

```
> vect <- 1:20
> dim(vect) <- c(4, 6)
Error in dim(vect) <- c(4, 6) :
  dims [product 24] do not match the length of object [20]
> vect
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> dim(vect) <- c(4, 4)
Error in dim(vect) <- c(4, 4) :
  dims [product 16] do not match the length of object [20]
> vect
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

При использовании же функции `array()` будет использовано только необходимое количество элементов из вектора, а при нехватке - переданный вектор будет повторен необходимое количество раз:

```
> array(1:1000000, dim = c(2, 2))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> array(1:3, dim = c(4, 5))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    1    2
[2,]    2    3    1    2    3
[3,]    3    1    2    3    1
[4,]    1    2    3    1    2
```

Матрицы

Матрицы это, по сути, двумерные массивы. Для их создания можно использовать функцию `matrix` , которой вместо вектора `dim` передается либо количество строк (`nrow`), либо количество столбцов (`ncol`):

```
> matrix(1:20, nrow = 4)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> matrix(1:20, ncol = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Как и массивы, матрицы заполняются из переданного вектора по колонкам. Но для функции `matrix()` это поведение можно изменить, указав `byrow = TRUE` :

```
> matrix(1:20, nrow = 4, byrow = TRUE)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
```

Заметим, что хотя функция `matrix()` также "подгонит" данные из переданного вектора под размерности матрицы, в отличии от функции `array()` она выдаст предупреждение, если сочтет что данные плохо подходят в матрицу:

```
> matrix(1:20, ncol = 2, nrow = 3)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
Warning message:
In matrix(1:20, ncol = 2, nrow = 3) :
  data length [20] is not a sub-multiple or multiple of the number of rows [3]
> matrix(1:20, ncol = 7, nrow = 3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    4    7   10   13   16   19
[2,]    2    5    8   11   14   17   20
[3,]    3    6    9   12   15   18    1
Warning message:
In matrix(1:20, ncol = 7, nrow = 3) :
  data length [20] is not a sub-multiple or multiple of the number of rows [3]
```

При этом:

```
> matrix(1:20, ncol = 8, nrow = 5)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    6   11   16    1    6   11   16
[2,]    2    7   12   17    2    7   12   17
[3,]    3    8   13   18    3    8   13   18
[4,]    4    9   14   19    4    9   14   19
[5,]    5   10   15   20    5   10   15   20
> matrix(1:20, ncol = 2, nrow = 2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

> **Замечание**
>
> При необходимости вы можете заставить R не выводить предупреждений, обернув виновное выражение в `suppressWarnings()`:
>
> ```
> > suppressWarnings(matrix(1:20, ncol = 7, nrow = 2))
>     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
> [1,]    1    3    5    7    9   11   13
> [2,]    2    4    6    8   10   12   14
> ```
>
> Но не рекомендуем этим злоупотреблять.

Получение размерностей

Размерности массивов и матриц можно получить в виде вектора просто использовав функцию `dim()`:

```
> dim(array(1:20, dim = c(4, 5)))
[1] 4 5
> dim(matrix(1:20, ncol = 5))
[1] 4 5
```

Если требуется получить именно количество строк или столбцов, то можно использовать функции `nrow()` и `ncol()`:

```
> ncol(mtr)
[1] 5
> nrow(mtr)
[1] 4
> ncol(arr)
[1] 5
> nrow(arr)
[1] 4
```

Функции `dim()`, `nrow()` и `ncol()` возвращают `NULL` для обычных, одномерных векторов. Но есть специальные функции `NCOL()` и `NROW()`, притворяющиеся что такие вектора это матрицы с одним столбцом:

```
> dim(c(1,2,3))
NULL
> nrow(c(1,2,3))
NULL
> ncol(c(1,2,3))
NULL
> NROW(c(1,2,3))
[1] 3
> NCOL(c(1,2,3))
[1] 1
> NROW(mtr)
[1] 4
> NCOL(arr)
[1] 5
```

Операции над матрицами и массивами

Операции `+`, `-`, `*`, `/` и `^` работают над матрицами и массивами поэлементно, как над обычными векторами:

```
> arr + mtr
     [,1] [,2] [,3] [,4] [,5]
[1,]    2   10   18   26   34
[2,]    4   12   20   28   36
[3,]    6   14   22   30   38
[4,]    8   16   24   32   40
> arr - mtr
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0
[4,]    0    0    0    0    0
> arr * mtr
     [,1] [,2] [,3] [,4] [,5]
[1,]    1   25   81  169  289
[2,]    4   36  100  196  324
[3,]    9   49  121  225  361
[4,]   16   64  144  256  400
> arr ^ 2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1   25   81  169  289
[2,]    4   36  100  196  324
[3,]    9   49  121  225  361
[4,]   16   64  144  256  400
> arr / mtr
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    1    1    1    1
[3,]    1    1    1    1    1
[4,]    1    1    1    1    1
```

В добавок к этому существует ряд операций, специфичных для матриц:

Операция	Описание
`%*%`	Матричное умножение
`t()`	Транспонирование
`solve()`	Обращение
`diag()`	Взятие диагональных элементов
`det()`	Найти детерминант
`eigen()`	Нахождение собственных значений и векторов

```
> (mtr <- matrix(1:20, nrow = 4))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> (m1 <- matrix(1:4, nrow = 2))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> (m2 <- matrix(1:2, nrow = 2))
     [,1]
[1,]    1
[2,]    2
> m1 %*% m2
     [,1]
[1,]    7
[2,]   10
> t(mtr)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
[5,]   17   18   19   20
> t(m1)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> t(m2)
     [,1] [,2]
[1,]    1    2
> diag(m1)
[1] 1 4
> diag(m2)
[1] 1
> diag(mtr)
[1]  1  6 11 16
> det(m1)
[1] -2
> det(m2)
Error in determinant.matrix(x, logarithm = TRUE, ...) :
  'x' must be a square matrix
> det(mtr)
Error in determinant.matrix(x, logarithm = TRUE, ...) :
  'x' must be a square matrix
> solve(m1)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

```
> solve(m2)
Error in solve.default(m2) : 'a' (2 x 1) must be square
```

Комбинирование матриц

Стандартная функция комбинации `c()` разбирает матрицы в одномерные вектора перед их соединением:

```
> (m1 <- matrix(1:8, nrow=2))
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> (m2 <- matrix(1:4, nrow=2))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> (m3 <- matrix(1:20, ncol = 4))
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
> c(m1, m2)
 [1] 1 2 3 4 5 6 7 8 1 2 3 4
```

Для соединения матриц по столбцам и по строкам стоит можно использовать функции `rbind()` и `cbind()` соответственно.

```
> cbind(m1, m2)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    1    3
[2,]    2    4    6    8    2    4
> rbind(m1, m3)
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
[3,]    1    6   11   16
[4,]    2    7   12   17
[5,]    3    8   13   18
[6,]    4    9   14   19
[7,]    5   10   15   20
```

Но соответствующие измерения матриц должны совпадать, иначе вы просто получите ошибку:

```
> cbind(m1, m3)
Error in cbind(m1, m3) :
  number of rows of matrices must match (see arg 2)
```

Упражнения

Упражнение 1

Задайте вектора `x` и `y` следующим образом:

```
rand <- rnorm(20)
NAs <- rep(NA, 20)
x <- sample(c(rand, NAs), 20)
y <- x[!is.na(x)]
```

Сравните результаты `x[x > 0]`, `y[y > 0]`. Объясните почему они не совпадают.
Получите вектор эквивалентный `y[y > 0]`, но без использования дополнительной переменной `y` (т.е. Вам дан `x` и Вы хотите извлечь из него все положительные числа не создавая новых переменных).

Упражнение 2

Предскажите тип полученного вектора в результате следующих операций:

```
c(1, FALSE)
c("a", 1)
c(TRUE, 1L)
```

Упражнение 3

Почему `1 == "1"` и `-1 < FALSE` истинны? А почему `"one" < 2` ложно?

Упражнение 4

n-ое треугольное число задается формулой $\frac{n(n+1)}{2}$.

Создайте последовательность 20 треугольных чисел. В R есть встроенный вектор `letters`, содержащий буквы английского алфавита. Назовите элементы созданного вами вектора первыми 20-ю буквами алфавита. Выберете только треугольные числа, названные гласными буквами.

Упражнение 5

Функция `diag()` имеет больше вариантов использования, чем нами было указано выше. Найдите и прочитайте по ней справку. После чего используйте её для создания матрицы 15-на-15 с нулевыми недиагональными элементами и со следующей диагональю: $7, \ldots, 1, 0, 1, \ldots, 7$.

Упражнение 6

Продолжая упражнение 5, создайте с помощью `diag()` новую матрицу, 14-на-15 у которой лишь на диагонали единицы. Затем добавьте её сверху строку, состоящую из нулей, чтобы получить матрицу с единицам, смещенными вниз от главной диагонали на одну позицию.

Создайте ещё одну такую матрицу с единицами смещенными на позицию вверх.

Сложите полученные матрицы с матрицей из упражнения 5. У вас должна была получиться матрица Вилкинсона.