ТИПЫ ДАННЫХ, УСЛОВИЯ, ВВОД И ВЫВОД

1 Типы данных

Тип данных – это класс данных, характеризуемый членами класса и операциями, которые могут быть к ним применены. С помощью типов данных мы можем представлять привычные нам сущности, такие как числа, строки и т.д. В языке R существует 5 базовых типов данных:

Название	Тип данных
`complex`	комплексные числа
`character`	строки
`integer`	целые числа
`logical`	логические (булевы)
`numeric`	числа с плавающей точкой

Помимо этого, есть тип `Date`, который позволяет работать с датами. Рассмотрим использование каждого из перечисленных типов.

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. Однако рекомендуется все же 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)`, округления `round(x, digits)`, натурального логарифма `abs(x)`, тригонометрические функции `sin(x)`, `cos(x)`, `tan(x)`, обратные к ним `asin(y)`, `acos(y)`, `atan(y)` и многие другие. Основные математические функции содержатся в пакете `base`, который по умолчанию доступен в среде R и не требует подключения.

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 <- "В историю трудно войти, но легко вляпаться (М.Жванецкий)"
s
## [1] "В историю трудно войти, но легко вляпаться (М.Жванецкий)"
```

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

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

Строки можно складывать также, 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 <- "В историю трудно войти,"
s2 <- "но легко вляпаться"
s3 <- "(М.Жванецкий)"
```

Посмотрим содержимое подстрок:

```
s1
## [1] "В историю трудно войти,"
s2
## [1] "но легко вляпаться"
s3
## [1] "(М.Жванецкий)"
```

А теперь объединим их в одну:

```
s <- paste(s1, s2)
s
## [1] "В историю трудно войти, но легко вляпаться"
s <- paste(s1, s2, s3)
s
## [1] "В историю трудно войти, но легко вляпаться (М.Жванецкий)"
```

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 <- "Максимальная численность населения в Детройте пришлась на"
s2 <- "год и составила"
s3 <- "тыс. чел"
s <- paste(s1, year, s2, pop, s3)
s
## [1] "Максимальная численность населения в Детройте пришлась на 1950 год и составила 1850
тыс. чел"
```

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. 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] "2018-11-12"
```

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 11955 days
```

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

```
current + 40
## [1] "2018-12-22"
```

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`.

2 Манипуляции с типами

2.1 Определение типа данных

Определение типа данных осуществляется с помощью функции `class()`

```
## [1] "numeric"
class(0.5)
## [1] "numeric"
class(1 + 2i)
## [1] "complex"
class("sample")
## [1] "character"
class(TRUE)
## [1] "logical"
class(as.Date('1986-02-18'))
## [1] "Date"
```

В вышеприведенном примере видно, что R по умолчанию «повышает» ранг целочисленных данных до более общего типа чисел с плавающей точкой, тем самым закладываясь на возможность точного деления без остатка. Если вы хотите, чтобы данные в явном виде интерпретировались как целочисленные, их нужно принудительно привести к этому типу. Операторы преобразования типов рассмотрены ниже.

2.2 Преобразование типов данных

Преобразование типов данных осуществляется с помощью функций семейства `as(d, type)`, где `d` – это входная переменная, а `type` – название типа данных, к которому эти данные надо преобразовать (см. таблицу в начале главы). Несколько примеров:

```
k <- 1
print(k)
## [1] 1
class(k)
## [1] "numeric"

l <- as(k, "integer")
print(l)
## [1] 1
class(l)
## [1] "integer"

m <- as(l, "character")
print(m)
## [1] "1"
class(m)
## [1] "character"

n <- as(m, "numeric")
print(n)
## [1] 1
class(n)
## [1] "numeric"
```

Для функции `as()` существуют обертки (*wrappers*), которые позволяют записывать такие преобразования более компактно и выглядят как `as.<dataype>(d)`, где `datatype` – название типа данных:

```
k <- 1
l <- as.integer(k)
print(l)
## [1] 1
class(l)
## [1] "integer"

m <- as.character(l)
print(m)
## [1] "1"
class(m)
## [1] "character"

n <- as.numeric(m)
print(n)
## [1] 1
class(n)
## [1] "numeric"

d <- as.Date('1986-02-18')
print(d)
## [1] "1986-02-18"
class(d)
## [1] "Date"
```

Если преобразовать число с плавающей точкой до целого, то дробная часть будет отброшена:

```
as.integer(2.7)
## [1] 2
```

После преобразования типа данных, разумеется, к переменной будут применимы только те функции, которые определены для данного типа данных:

```
a <- 2.5
b <- as.character(a)
b + 2
## Error in b + 2: non-numeric argument to binary operator
nchar(b)
## [1] 3
```

2.3 Проверка типов данных и пустых значений

Для проверки типа данных можно использовать функции семейства `is.<datatype>`:

```
is.integer(2.7)
## [1] FALSE
is.numeric(2.7)
## [1] TRUE
is.character('Привет!')
## [1] TRUE
```

Особое значение имеют функции проверки пустых переменных (имеющих значение `NA` – not available), которые могут получаться в результате несовместимых преобразований или соответствовать пропускам в исходных данных:

```
as.integer('Привет!')
## [1] NA
is.na(as.integer('Привет!'))
## [1] TRUE
```

3 Ввод и вывод данных в консоли

3.1 Ввод данных

Для ввода данных через консоль можно воспользоваться функцией `readline()`, которая будет ожидать пользовательский ввод и нажатие клавиши `Enter`, после чего вернет введенные данные в виде строки. Предположим, пользователь вызывает эту функцию и вводит с клавиатуры `1024`:

```
a <- readline()
```

Выведем результат на экран:

```
a
## [1] "1024"
```

Функция `readline()` всегда возвращает строку, поэтому если вы ожидаете ввод числа, полученное значение необходимо явным образом преобразовать к числовому типу.

3.2 Ввод данных

Для вывода данных в консоль можно воспользоваться тремя способами:

- Просто напечатать название переменной с новой строки (*не работает при запуске программы командой* `Source`)
- Вызвать функцию `print()`
- Вызвать функцию `cat()`
- Заключить выражение в круглые скобки `()`

Первый способ мы уже регулярно использовали ранее в настоящей главе. Следует обратить внимание на то, что он хорош для отладки программы, но выглядит некрасиво в рабочих программах, поскольку просто печатая название переменной с новой строки вы как бы явно не говорите о том, что хотите вывести ее значение в консоль, а лишь подразумеваете это. Более того, если скрипт запускается командой `Source`, данный метод вывода переменной просто не сработает, интерпретатор его проигнорирует.

Поэтому после отладки следует убрать из программы все лишние выводы в консоль, а оставшиеся (действительно нужные) оформить с помощью функций `print()` или `cat()`.

Функция `print()` работает точно так же, как и просто название переменной с новой строки, отличаясь лишь двумя особенностями:

- `print()` явным образом говорит о том, что вы хотите вывести в консоль некую информацию
- `print()` работает при любых методах запуска программы, в том числе методом `Source`.

Например:

```
a <- 1024
a
## [1] 1024
print(a)
## [1] 1024

b <- "Fourty winks in progress"
b
## [1] "Fourty winks in progress"
print(b)
## [1] "Fourty winks in progress"

print(paste("2 в степени 10 равно", 2^10))
## [1] "2 в степени 10 равно 1024"

print(paste("Сегодняшняя дата - ", Sys.Date()))
## [1] "Сегодняшняя дата -  2018-11-12"
```

Функция `cat()` отличается от `print()` следующими особенностями:

- `cat()` выводит значение переменной, и не печатает ее измерения и внешние атрибуты типа двойных кавычек вокруг строки. Это означает, что `cat()` можно использовать и для записи данных в файл (на практике этим мало кто пользуется, но знать такую возможность надо);
- `cat()` принимает множество аргументов и может осуществлять конкатенацию строк аналогично функции `paste()`;
- `cat()` не возвращает никакого значений, в то время как `print()` возвращает значение, переданное ей в качестве аргумента;

- `cat()` можно использовать только для атомарных типов данных. Для классов (таких как Date) она будет выводит содержимое объекта, которое может не совпадать с тем, что пользователь ожидает вывести.

Например:

```
cat(a)
## 1024
cat(b)
## Fourty winks in progress

cat("2 в степени 10 равно", 2^10)
## 2 в степени 10 равно 1024

cat("Сегодняшнаяя дата -", Sys.Date())
## Сегодняшнаяя дата - 17847
```

Можно видеть, что в последнем случае `cat()` напечатала отнюдь не дату в ее привычном представлении, а некое число, которое является внутренним представлением даты в типе данных `Date`. Такие типы данных являются классами объектов в R, и у них есть своя функция `print()`, которая и выдает содержимое объекта в виде, который ожидается пользователем. Поэтому пользоваться функцией `cat()` надо с некоторой осторожностью.

Заключительная возможность – вывод с помощью заключения выражения в круглые скобки – очень удобна на стадии отладки программы. При этом переменная, которая создается в выражении, остается доступной в программе:

```
(a <- rnorm(5)) # сгенерируем 5 случайных чисел, запишем их в переменную a и выведем на экран
## [1] -0.6772907 0.4603677 -0.2430490 1.2022944 -0.2197313
(b <- 2 * a) # переменная a доступна, ее можно использовать и далее для вычислений
## [1] -1.3545814 0.9207354 -0.4860980 2.4045889 -0.4394625
```

4 Условный оператор

Проверка условий позволяет осуществлять так называемое ветвление в программе. Ветвление означает, что при определенных условиях (значениях переменных) будет выполнен один программный код, а при других условиях – другой. В R для проверки условий используется условный оператор **if** – **else if** – **else** следующего вида:

```
if (condition) {
  statement1
} else if (condition) {
  statement2
} else {
  statement3
}
```

Сначала проверяется условие в выражении `if (condition)`, и если оно истинно, то выполнится вложенный в фигурные скобки программный код `statement1`, после чего оставшиеся условия не будут проверяться. Если первое условие ложно, программа перейдет к проверке следующего условия `else if (condition)`. Далее, если оно истинно, то выполнится вложенный код `statement2`, если нет — проверка переключится на следующее условие и так далее. Заключительный код `statement3`, следующий за словом `else`, выполнится только если ложными окажутся все предыдущие условия.

Конструкций `else if` может быть произвольное количество, конструкции `if` и `else` могут встречаться в условном операторе только один раз, в начале и конце соответственно. При этом условный оператор может состоять только из конструкции `if`, а `else if` и `else` не являются обязательными.

Например, сгенерируем случайное число, округлим его до одного знака после запятой и проверим относительно нуля:

```
(a <- round(rnorm(1), 1))
## [1] 0.2

if (a < 0){
  cat('Получилось отрицательное число!')
} else if (a > 0) {
  cat('Получилось положительное число!')
} else {
  cat('Получился нуль!')
}
## Получилось положительное число!
```

Условия можно использовать, в частности, для того чтобы обрабатывать пользовательский ввод в программе. Например, охарактеризуем положение точки относительно Полярного круга:

```
cat('Введите широту вашей точки:')
phi <- as.numeric(readline())
```

Пользователь вводит `68`, а мы оцениваем результат:

```
if (!is.na(phi)) { # проверяем, является ли введенное значение числом

  if (abs(phi) >= 66.562 && abs(phi) <= 90) { # выполняем проверку на заполярность
    cat('Точка находится в Заполярье')
  } else {
    cat('Точка не находится в Заполярье')
  }

} else {
  cat('Необходимо ввести число!') # оповещаем о некорректном вводе
}
## Точка находится в Заполярье
```

5 Оператор переключения

Оператор переключения (switch) является удобной заменой условному оператору в тех случаях, когда надо вычислить значение переменной в зависимости от значения другой переменной, которая может принимать *ограниченное (заранее известное)* число значений. Например:

```
cat('Введите название федерального округа:')
name <- readline()
```

Пользователь вводит:

```
Приволжский
# Определим центр в зависимости от названия:
capital <- switch(name,
                  'Центральный' = 'Москва',
                  'Северо-Западный' = 'Санкт-Петербург',
                  'Южный' = 'Ростов-на-Дону',
                  'Северо-Кавказский' = 'Пятигорск',
                  'Приволжский' = 'Нижний Новгород',
                  'Уральский' = 'Екатеринбург',
                  'Сибирский' = 'Новосибирск',
                  'Дальневосточный' = 'Хабаровск')
print(capital)
## [1] "Нижний Новгород"
```