

# Условные конструкции. Циклы. Функции

Язык программирования R

Иркутский государственный университет

## Оператор if-else

Условный оператор **if-else** выполняет инструкцию, если заданное условие верно. Также есть возможность выполнить другую инструкцию, если условие не верно. Он имеет следующий синтаксис:

```
gene_expression <- 8.5
gene_name <- "TP53"

# Классификация с помощью if-else
if (gene_expression > 15) {
    category <- "высокая экспрессия"
} else if (gene_expression > 5) {
    category <- "умеренная экспрессия"
} else {
    category <- "низкая экспрессия"
}
```

## Оператор ifelse

Условный оператор **ifelse** – компактная и векторизованная версия оператора if-else.

```
# Вектор с уровнями экспрессии генов  
gene_names <- c("TP53", "BRCA1", "EGFR", "MYC", "AKT1", "PTEN")  
expression_levels <- c(8.5, 25.2, 15.8, 3.2, 12.1, 18.9)  
  
# Векторизованная классификация  
expression_category <- ifelse(expression_levels > 20, "высокая",  
                                ifelse(expression_levels > 10, "умеренная",  
                                     "низкая"))  
  
# Создаем таблицу результатов  
results <- data.frame(  
  gene = gene_names,  
  expression = expression_levels,  
  category = expression_category  
)
```

## Оператор switch

Оператор выбора **switch** выбирает инструкцию для выполнения в зависимости от значения выражения expr. Он имеет следующий синтаксис:

```
# Выбор анализа по типу данных
analysis_type <- "expression"

result <- switch(analysis_type,
  "expression" = "Анализ экспрессии генов",
  "methylation" = "Анализ метилирования",
  "Неизвестный тип анализа")
```

## Цикл for

Конструкция цикла **for** выполняет инструкцию для каждого значения в последовательности.

```
dna_sequence <- "ATGCCGTA"
nucleotides <- c("A", "T", "G", "C")

dna_vector <- strsplit(dna_sequence, "")[[1]]
counts <- c(A = 0, T = 0, G = 0, C = 0)

for (i in dna_vector) {
    counts[i] <- counts[i] + 1
}

print(counts)
```

## Цикл while

Конструкция цикла **while** повторно выполняет инструкцию, пока заданное условие остается истинным.

```
dna_sequence <- "ATGCCGTA"
counts <- c(A = 0, T = 0, G = 0, C = 0)
i <- 1

while(i <= nchar(dna_sequence)) {
  nucleotide <- substr(dna_sequence, i, i)
  counts[nucleotide] <- counts[nucleotide] + 1
  i <- i + 1
}

print(counts)
```

## Цикл repeat

В **repeat** цикл повторяется до тех пор, пока в явном виде не будет вызван оператор **break**. Также можно использовать оператор **next** внутри цикла **repeat** для пропуска итерации

```
# Поиск старт-кодона ATG
position <- 1
dna <- "ACATGCTAGCATG"

repeat {
  if (position > nchar(dna) - 2) break

  codon <- substr(dna, position, position + 2)

  if (codon == "ATG") {
    cat("Найден старт-кодон на позиции", position, "\n")
    break
  }

  position <- position + 1
  next
}
```

## Циклы и векторизация

Векторизация в R работает быстрее циклов, поскольку использует оптимизированные низкоуровневые функции для обработки целых векторов данных за одну операцию.

```
# Подсчет суммы через цикл
sum_loop <- function(x) {
    total <- 0
    for(i in x) {
        total <- total + i
    }
    return(total)
}

x <- 1:100000000
system.time(sum_loop(x))
# >   user   system elapsed
# >  0.258   0.004   0.262

# Подсчет суммы через векторизацию
system.time(sum(x))
# >   user   system elapsed
# >  0.001   0.000   0.001
```

# Числовые и текстовые функции

## Математические функции:

<code>abs(x)</code>	Модуль
<code>sqrt(x)</code>	Квадратный корень
<code>ceiling(x)</code>	Ближайшее целое число, не меньшее, чем x
<code>floor(x)</code>	Ближайшее целое число, не большее, чем x
<code>trunk(x)</code>	Целое число, полученное округлением x в сторону нуля
<code>round(x, digits=n)</code>	Округляет x до заданного числа знаков n после запятой
<code>signif(x, digits=n)</code>	Округляет x до заданного числа n значащих цифр

# Числовые и текстовые функции

## Статистические функции:

mean(x)	Среднее арифметическое
median(x)	Медиана
sd(x)	Стандартное отклонение
var(x)	Дисперсия
quantile(x, probs)	Квантили, где x – числовой вектор, для которого вычисляются квантили, а probs – числовой вектор с вероятностями в диапазоне [0; 1]
range(x)	Размах значений
sum(x)	Сумма

# Числовые и текстовые функции

## Текстовые функции:

<code>nchar(x)</code>	Подсчитывает число элементов в x
<code>toupper(x)</code>	Преобразует строку x в верхний регистр
<code>tolower(x)</code>	Преобразует строку x в нижний регистр
<code>paste(..., sep=)</code>	Объединяет строки, разделяя их заданной строкой sep
<code>substr(x, start, stop)</code>	Извлекает или замещает часть текстового вектора
<code>strsplit(x, sep, fixed)</code>	Разделяет текстовый вектор x на элементы, используя разделитель sep. При fixed=FALSE разделитель воспринимается как регулярное выражение, при fixed=TRUE — как обычная текстовая строка

## Пользовательские функции

```
classify_expression <- function(expression_value)
  if (expression_value > 100) {
    return("очень высокая")
  } else if (expression_value > 50) {
    return("высокая")
  } else if (expression_value > 10) {
    return("умеренная")
  } else {
    return("низкая")
  }
}
```

## Область видимости

```
x <- "глобальная"

outer_func <- function() {
  x <- "внешняя"

  inner_func <- function() {
    x <- "локальная"
    print(x) # Найдет "локальную"
  }
  inner_func()
}
```

## Дата и время

```
as.Date(x, "input_format")
```

где x – это дата в текстовом формате, а input\_format определяет формат представления даты

## Факторы

**Фактор** - специальный тип данных для представления порядковых и номинальных переменных с фиксированным набором возможных значений

```
expr_lvl <- factor(  
  c("low", "high", "medium", "low", "high"),  
  levels = c("low", "medium", "high"),  
  ordered = TRUE  
)
```

где `levels` - это уникальные категории или возможные значения, которые может принимать фактор, `ordered` - флаг порядка категорий

## Операции над факторами

```
# Определить порядок уровней
response <- factor(c("yes", "no", "yes", "maybe"))
response <- factor(response, levels = c("no", "maybe", "yes"))

# Поменять порядок
response_rev <- factor(response, levels = rev(levels(response)))

# Объединение факторов
factor1 <- factor(c("A", "B", "A"))
factor2 <- factor(c("B", "C", "A"))
combined <- c(factor1, factor2)
```

# Матрицы

**Матрица** – это двумерный массив данных, в котором все элементы имеют один и тот же тип (числовой, текстовый или логический).  
Матрицы создаются при помощи функции `matrix`. Общий синтаксис этой функции:

```
my_matrix <- matrix(  
  вектор,  
  nrow=число строк,  
  ncol=число столбцов,  
  byrow=логическое значение,  
  dimnames=list(текстовый вектор с названиями строк,  
  текстовый вектор с названиями столбцов)  
)
```

## Массивы

Массивы — это объекты данных, которые могут хранить данные в более чем двух измерениях. Массивы создаются при помощи функции `array`:

```
my_array <- array(vector, dimensions, dimnames)
```

где `vector` – это вектор с данными, `dimensions` – числовой вектор, определяющий размеры измерений, а `dimnames` – необязательный список названий измерений.

## Таблицы данных

Таблицы данных – это основной класс объектов R, используемых для хранения данных. Таблица данных создается при помощи функции `data.frame()`:

```
my_data <- data.frame(col1, col2, col3, ...)
```

где – `col1, col2, col3, ...` это векторы любого типа (текстового, числового или логического), которые станут столбцами таблицы.

## Пример таблицы данных

```
genome_data <- data.frame(  
  gene_id = c("BRCA1", "TP53", "EGFR", "HBB"),  
  chromosome = c("17", "17", "7", "11"),  
  start_position = c(43044295, 7668402, 55019017, 5248232),  
  end_position = c(43170245, 7687550, 55211628, 5249264),  
  strand = c("+", "-", "+", "-"),  
  expression = c(15.2, 8.7, 23.4, 150.2)  
)
```

# Операции с таблицами данных

`rbind()` - объединяет два набора данных по вертикали (по строкам):

```
# Первый набор данных - экспрессия генов
gene_expression <- data.frame(
  gene_id = c("TP53", "BRCA1", "EGFR"),
  sample_A = c(15.2, 8.7, 25.4),
  sample_B = c(18.9, 7.3, 28.1)
)

# Второй набор данных - дополнительная экспрессия
dop_expression <- data.frame(
  gene_id = c("AKT1", "PTEN"),
  sample_A = c(22.3, 9.8),
  sample_B = c(24.1, 8.9)
)

# Объединяем два набора данных по экспрессии генов
all_expression <- rbind(gene_expression, dop_expression)
```

	gene_id	sample_A	sample_B
1	TP53	15.2	18.9
2	BRCA1	8.7	7.3
3	EGFR	25.4	28.1
4	AKT1	22.3	24.1
5	PTEN	9.8	8.9

# Операции с таблицами данных

`cbind()` - объединяет две матрицы или таблицы по горизонтали (по столбцам):

```
# Первый набор данных - экспрессия генов
gene_expression <- data.frame(
  gene_id = c("TP53", "BRCA1", "EGFR"),
  sample_A = c(15.2, 8.7, 25.4),
  sample_B = c(18.9, 7.3, 28.1)
)

# Добавляем данные по третьему образцу
sample_C_data <- data.frame(
  sample_C = c(16.8, 9.2, 26.7)
)

# Объединяем с исходными данными
expression_with_C <- cbind(gene_expression, sample_C_data)

gene_id sample_A sample_B sample_C
1      TP53     15.2     18.9     16.8
2      BRCA1     8.7      7.3      9.2
3      EGFR     25.4     28.1     26.7
```

# Операции с таблицами данных

merge() - объединяет таблицы по ключу:

```
# Таблица с экспрессией генов
expression_data <- data.frame(
  gene_id = c("TP53", "BRCA1", "EGFR", "MYC", "AKT1"),
  expression = c(15.2, 8.7, 25.4, 12.1, 18.9),
  sample = "sample_A"
)

# Таблица с мутациями
mutation_data <- data.frame(
  gene_id = c("TP53", "BRCA1", "EGFR", "PTEN"),
  mutation_count = c(3, 1, 2, 1),
  mutation_type = c("missense", "frameshift", "nonsense", "missense")
)

# Общие гены
inner_join <- merge(expression_data, mutation_data, by = "gene_id")
gene_id expression sample mutation_count mutation_type
1    BRCA1        8.7 sample_A            1   frameshift
2     EGFR       25.4 sample_A            2     nonsense
3     TP53       15.2 sample_A            3     missense
```

## Списки

**Список** – это упорядоченная коллекция объектов. Список может объединять разные объекты вне зависимости от их типов. К примеру, список может одновременно содержать векторы, матрицы, таблицы данных и другие списки.

```
my_list <- list(объект1, объект2, ...)
```

```
gene_info <- list(  
  gene_name = "TP53",  
  chromosome = 17,  
  position = c(7668402, 7687550),  
)
```

## Чтение табличных данных

Функция **read.table()** позволяет импортировать данные из текстовых файлов с разделителями. Некоторые параметры этой функции:

`file` - путь к файлу

`header` - логическое значение, указывает наличие строки с названиями колонок в начале файла

`nrows` - максимальное количество строк для чтения (-1 означает все строки)

`sep` - символ-разделитель колонок в файле

`dec` - символ, используемый как десятичный разделитель в числах

`skip` - количество строк в начале файла для пропуска перед чтением данных

`na.strings` - вектор строк, которые интерпретируются как пропущенные значения (NA)

`colClasses` - вектор, определяющий классы данных для каждой колонки

`stringsAsFactors` - логическое значение, указывает должны ли строковые колонки

преобразовываться в факторы

## Пример использования функции `read.table()`

```
data <- read.table(  
  "data.txt",  
  header = TRUE,           # есть заголовок  
  sep = "\t",              # разделитель - табуляция  
  stringsAsFactors = FALSE # не преобразовывать строки в факторы  
)  
  
# Параметры по умолчанию (для CSV)  
data <- read.table(  
  "data.csv",  
  header = TRUE,  
  sep = ",",  
  dec = "."  
)
```

## Подмножества

Квадратные скобки [ — сохраняют класс исходного объекта при извлечении подмножества и поддерживают выбор произвольного количества элементов.

Двойные квадратные скобки [[ — предназначены для извлечения единичного элемента из списка или таблицы данных, при этом возвращаемый объект может иметь другой класс.

Оператор \$ — обеспечивает доступ к элементам списка или таблицы данных по имени, аналогично работе оператора [[.

## Подмножества

Подмножества данных можно выбирать на основе любой структуры данных. Например, на основе векторов:

```
gene_expression <- c(15.2, 8.7, 23.4, 150.2, 5.3)
names(gene_expression) <- c("BRCA1", "TP53", "EGFR", "HBB", "CFTR")

# Выборка по индексам
gene_expression[3]           # EGFR: 23.4
gene_expression[c(2,4)]       # TP53: 8.7, HBB: 150.2

# Выборка по условиям
gene_expression[gene_expression > 20]    # гены с экспрессией > 20
gene_expression[gene_expression < 10]      # гены с экспрессией < 10

# Выборка по именам генов
gene_expression[c("BRCA1", "HBB")]        # конкретные гены
```

## Подмножества

Подмножества данных также можно выбирать на основе матриц:

```
expr_matrix <- matrix(  
  c(25.4, 18.9, 32.1, 45.2, 12.8, 8.7, 150.8, 95.3, 120.5),  
  nrow = 3,  
  byrow = TRUE,  
  dimnames = list(  
    c("SYP", "GAD1", "GFAP"),  
    c("Cortex", "Cerebellum", "Hippocampus")))  
  
marker_genes <- expr_matrix[c("SYP", "GFAP"), ]  
  
marker_genes <- expr_matrix["GAD1", , drop=FALSE]  
marker_genes <- expr_matrix["GAD1", ]  
  
custom_subset <- expr_matrix[c("GAD1", "GFAP"), c("Cerebellum", "Hippocampus")]  
high_gfap_regions <- expr_matrix["GFAP", expr_matrix["GFAP", ] > 100]
```

# Подмножества

## Подмножество на основе таблицы данных:

```
gene_data <- data.frame(  
  gene_name = c("TP53", "BRCA1", "EGFR", "MYC", "AKT1", "PTEN"),  
  expression = c(15.2, 8.7, 25.4, 12.1, 18.9, 9.5),  
  chr = c("chr17", "chr17", "chr7", "chr8", "chr14", "chr10"),  
  is_oncogene = c(FALSE, FALSE, TRUE, TRUE, TRUE, FALSE),  
  mutation_count = c(3, 1, 2, 0, 1, 2))  
  
# Подмножество по строкам и столбцам - Строки 2-4, колонки gene_name и expression  
subset1 <- gene_data[2:4, c("gene_name", "expression")]  
  
# Подмножество по условию (высокая экспрессия)  
high_expression <- gene_data[gene_data$expression > 15, ]  
  
# Подмножество по нескольким условиям (онкогены с мутациями)  
sample1 <- gene_data[gene_data$is_oncogene & gene_data$mutation_count > 0, ]  
  
# Подмножество с оператором ИЛИ  
sample2 <- gene_data[gene_data$chr == "chr17" | gene_data$expression > 20, ]  
  
# Подмножество по индексам строк  
selected_rows <- gene_data[c(1, 3, 5), ]
```

## Подмножества

Функция `subset()` – позволяет выбрать подвыборку данных на основе какого-либо условия.

```
chr17_high_expr <- subset(  
  genome_data,  
  chromosome == "17" & expression > 10)  
  
low_expr_genes <- subset(  
  genome_data,  
  expression < 10,  
  select = c(gene_id, chromosome, expression))
```

## Обработка пропущенных значений

**NA (Not Available)** означает отсутствие значения, тогда как **NaN (Not a Number)** указывает на неопределенный результат числовой операции, например,  $0/0$ .

`is.na()` - проверка на пропущенные значения

`is.nan()` - проверка на "не число"(Not a Number)

```
num_vector <- c(1, 2, NA, 4)
is.na(num_vector) # FALSE FALSE TRUE FALSE
```

```
char_vector <- c("a", "b", NA, "d")
is.na(char_vector) # FALSE FALSE TRUE FALSE
```

```
log_vector <- c(TRUE, FALSE, NA, TRUE)
is.na(log_vector) # FALSE FALSE TRUE FALSE
```

```
z <- c(1, 2, NA, NaN, 5)
print(is.na(z)) # FALSE FALSE TRUE TRUE FALSE
print(is.nan(z)) # FALSE FALSE FALSE TRUE FALSE
```

## Векторизованные операции

Векторизация — это возможность применять операции ко всему вектору сразу. Это одна из ключевых особенностей R, которая делает код эффективным и читаемым.

```
# Векторы
x <- 10:14
y <- 20:24

print(x + y)
print(x > 12)
print(y * 2 + x * 3)
print(x %% 2 == 0)

# Матрицы
mat1 <- matrix(1:9, nrow = 3)
mat2 <- matrix(10:18, nrow = 3)

print(mat1 + mat2)      # Поэлементное сложение
print(mat1 * 2 + mat2)  # Умножение на скаляр
print(mat1 * mat2)      # Поэлементное умножение (не матричное!)
```