Основы анализа больших данных

Лекция 5. Управляющие конструкции в R.
Создание собственных функций в R.

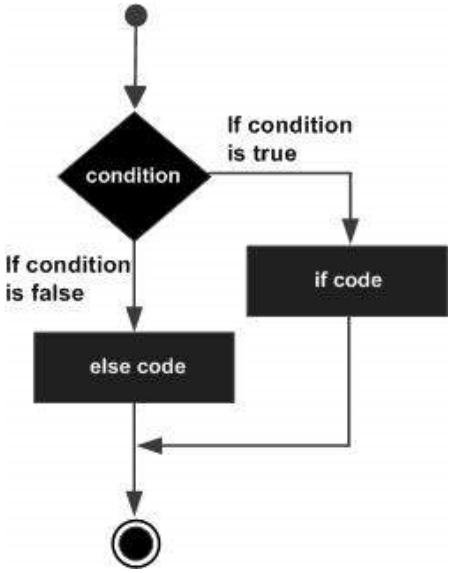
Основы анализа больших данных

Лекция 5. Управляющие конструкции в R.
Создание собственных функций в R.

Условные операторы if и else

if (<условие>**) {**<выполнить что-то>**} else {**<выполнить что-то другое>**}**

- <условие> – выражение, результатом которого будет логический вектор длины 1 (TRUE или FALSE);
- <выполнить что-то> и <выполнить что-то другое> - произвольные команды;
- при переносе на новую строку слово else должно находиться на той же строке, что и закрывающая фигурная скобка предыдущего оператора;
- фигурные скобки после else нужны, если нужно выполнить несколько команд.



Пример:

```
# Проверим, является ли корень из 10 больше числа пи.
if (sqrt(10) > pi) {print("Больше")} else {print("Меньше")}
```

Тот же пример с правильным переносом строк:

```
if (sqrt(10) > pi) {
  print("Больше")
} else {
print("Меньше")
}
```

Если попробовать поставить в <условие> вектор длины больше 1, получим ошибку:

```
if (c(F,T)) print("test")
Ошибка в if (c(F, T)) print("test") :условие длиной > 1
```

Условный оператор ifelse

ifelse (<условие>, yes, no**)**

- <условие> является логическим вектором любой заданной размерности;
- если yes или no слишком короткие, их элементы реплицируются;
- «yes» будет выполняться тогда и только тогда, когда какой-либо элемент <условия> истинен. Аналогично для «no»;
- отсутствующие значения в <условии> приводят к отсутствию значений в результате.

Пример 1:

```
> score=51; ifelse(score > 50, "Зачет", "Незачет")
[1] "Зачет"
```

Пример 2:

```
> x=c(1,3,1,5,1,7,1,9)
> y=c(2,3,4,5,2,7,1,8)
> z = ifelse(x==y,"+","-"); z
[1] "-" "+" "-" "+" "-" "+" "+" "-«
```

Пример 3: (ветвление путём вложенных операторов ifelse):

```
x <- runif(5)
ifelse(x>2/3, "Камень",
       ifelse(x>1/3, "Ножницы", "Бумага"))
[1] "Ножницы" "Камень"  "Бумага"  "Камень"  "Ножницы"
```

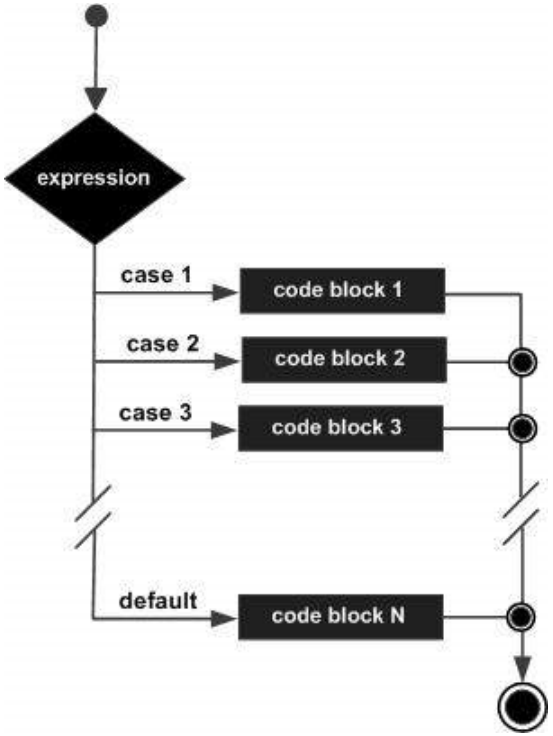
Множественный выбор: оператор switch

switch(управляющее выражение, ...**)**

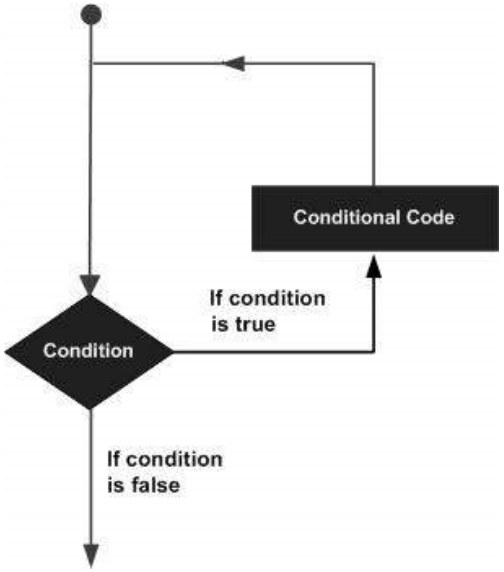
- ... – альтернативные действия (команды), одно из которых будет выполнено в зависимости от результатов управляющего выражения;
- управляющее выражение – выражение, возвращающее либо целое число (от 1 до числа альтернатив), либо символьную переменную, соответствующую имени выполняемой операции;
- если совпадений управляющего выражения с альтернативной командой более одного, то возвращается первый совпавший элемент.

Пример:

```
switch("case2",
  case1 = "case 1",
  case2 = "case 2",
  case3 = "case 3",
  case4 = "case 4")
[1] "case 2"
```



Циклы в R: repeat, while, for



Язык программирования R поддерживает следующие виды циклов:

- цикл **`repeat -`** выполняет последовательность операций несколько раз и сокращает код, управляющий переменной цикла
- цикл **`while -`** повторяет операцию или группу операций, пока заданное условие истинно. Он проверяет условие перед выполнением тела цикла
- цикл **`for –`** аналогичен оператору while, за исключением того, что он проверяет условие в конце тела цикла

а также следующие операторы управления циклом (изменяют выполнение цикла по сравнению с его обычной последовательностью:

- оператор **`break -`** завершает цикл и передает выполнение операции, следующей сразу за циклом
- оператор **`next –`** позволяет пропустить текущую итерацию цикла, не завершая цикл

Цикл repeat

Цикл **repeat** выполняет один и тот же код снова и снова, пока не будет выполнено условие остановки (критерий останова).

Базовый синтаксис для цикла repeat:

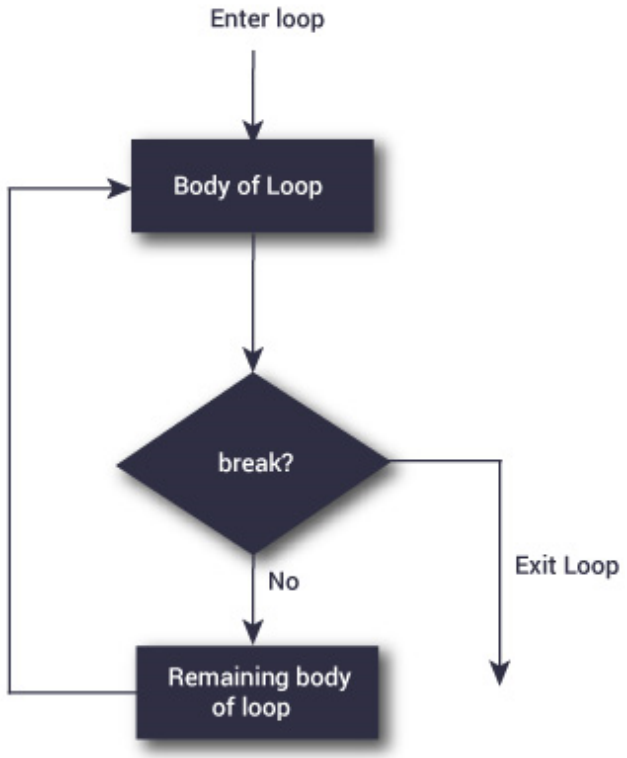
```
repeat {
   команды
   if(условие) {
      break
   }
}
```

Пример 1:

```
text <- c("Hello","loop")
count <- 2
repeat{
  print(text)
  count <- count+1
  if(count > 5){
    break
  }
}
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

Пример 2:

```
i <- 0; set.seed(66)
repeat{
  i <- i + runif(1)
  print(i)
  if(i>5) break
}
[1] 0.9899366
[1] 1.819629
[1] 2.405516
[1] 2.822562
[1] 3.484659
[1] 3.863961
[1] 4.288227
[1] 5.130739
```



Цикл while

Цикл `while` выполняет один и тот же код снова и снова, пока не будет выполнено услови остановки, однако в отличие от repeat, цикл может быть не выполнен ни разу.

Базовый синтаксис для цикла `while`:

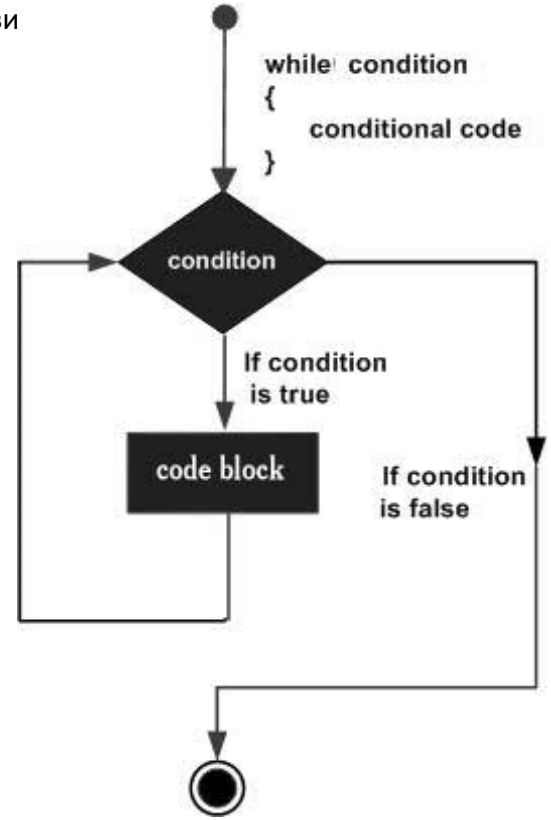
```
while(test_expression){
   statement
}
```

Пример 1:

```
text <- c("Hello","while loop")
count <- 2
while (count < 6){
  print(text)
  count = count + 1
}
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
```

Пример 2:

```
i <- 0; set.seed(66)
while(i <= 5){
  i <- i + runif(1)
  print(i)
}
[1] 0.9899366
[1] 1.819629
[1] 2.405516
[1] 2.822562
[1] 3.484659
[1] 3.863961
[1] 4.288227
[1] 5.130739
```

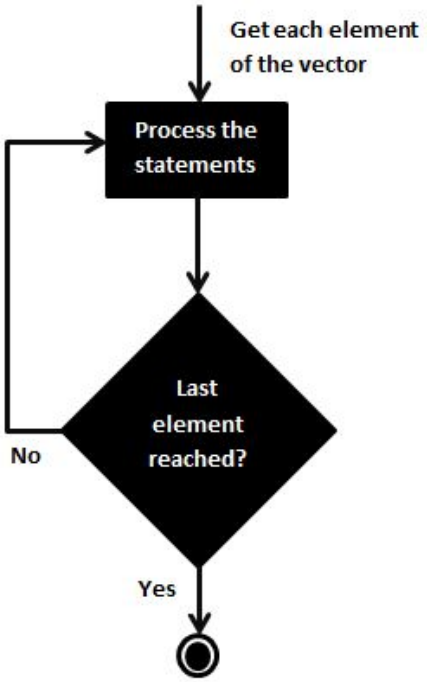


Цикл for

Цикл `for` — это структура управления повторением операций, которая позволяет эффективно писать цикл, который необходимо выполнить определенное количество раз.

Базовый синтаксис для цикла `for`:

```
for (value in vector) {
   statements
}
```



Пример 1:

```
text <- LETTERS[1:4]
for (i in text){
   print(i)
}
[1] "A"
[1] "B"
[1] "C"
[1] "D"
```

Пример 2:

```
i <- 0; set.seed(66)
while(i <= 5){
  i <- i + runif(1)
  print(i)
}
[1] 0.9899366
[1] 1.819629
[1] 2.405516
[1] 2.822562
[1] 3.484659
[1] 3.863961
[1] 4.288227
[1] 5.130739
```

Сравнение: цикл for против векторизации

Пример 1:

```
v <- 1:1e6
system.time({
  x <- 0
  for(i in v) x[i] <- sqrt(v[i])
})

пользователь     система      прошло
        0.30        0.06        0.36


system.time({
  y <- sqrt(v)
})

пользователь     система      прошло
        0.04        0.03        0.08
```

Пример 2:

```
set.seed(5)
scores <- round(runif(10,0,100))

results <- rep(NA,10)
for(i in 1:length(scores)){
  if (scores[i] > 51){
    results[i] <- "зачёт"
  } else results[i] <- "незачёт"}

print(scores)
[1] 20 69 92 28 10 70 53 81 96 11
print(results)
[1] "незачёт" "зачёт"   "зачёт"   "незачёт"
"незачёт" "зачёт"   "зачёт"   "зачёт"   "зачёт"
"незачёт"
```

Решение через ifelse в одну строчку:

```
results <- ifelse(scores > 51, "зачёт", "незачёт")
```

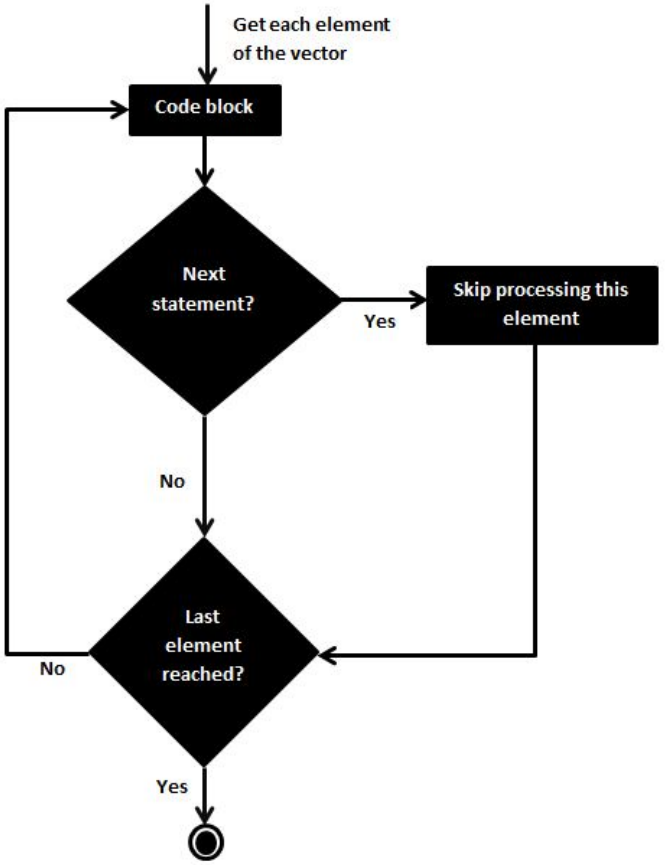
Оператор управления циклом next

Оператор `next` полезен, когда мы хотим пропустить текущую итерацию цикла, не завершая сам цикл. При обнаружении `next` анализатор R пропускает дальнейшие вычисления и начинает следующую итерацию цикла.

Пример:

```
text <- LETTERS[1:6]
for (i in text){
   if(i == "D"){
      next
   }
   print(i)
}
```

```
[1] "A"
[1] "B"
[1] "C"
[1] "E"
[1] "F"
```



Создание собственных функций в R

Стандартная форма задания функции в R:

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

Компоненты функции:

- `function_name` – это фактическое имя функции. Оно хранится в среде R как объект с этим именем.
- `function` – ключевое слово, сообщающее R о том, что будет создана функция.
- Аргументы функции — список формальных аргументов, разделенных запятой. При вызове функции вы передаёте значение аргументу. Аргументы не являются обязательными. Также аргументы могут иметь значения по умолчанию. Формальным аргументом может быть: символ, выражение вида имя = выражение, специальный формальный аргумент – троеточие.
- Тело функции – представляет собой команду или блок команд, которые определяют, что делает функция. Как правило, этот набор команд зависит от определённых ранее аргументов функции. Отдельные команды в блоке пишутся с новой строки.
- Возвращаемое значение функции — это последнее выражение в теле функции, которое должно быть выполнено.

Обращение к функции: `function_name(arg1, arg2,…)`. Здесь значения arg1, arg2, … являются фактическими аргументами, которые подставляются вместо соответствующих формальных аргументов, определённых при задании функции.

Создание собственных функций в R: примеры

Пример 1. (функция одного аргумента)

```
function_one <- function(x){x^2}
> function_one(3)
[1] 9
```

Пример 2. (функция двух аргументов)

```
my_function <- function(x, y){
  s <- x + y
  r <- x - y
  return(c(s,r))
}
> my_function(3,7)
[1] 10 -4
```

Пример 3. (функция без аргументов)

```
hello_function <- function(){
  text <- "Hello, world!"
  return(text)}
> hello_function()
[1] "Hello, world!"
```

Пример 3. (функция с произвольным числом аргументов)

```
fun <- function (...) {
  data = list(...)
  n = length(data)
  maxs <- numeric(n)
  mins <- numeric(n)
  means <- numeric(n)
  for (i in 1:n) {
    maxs[i] <- max(data[[i]])
    mins[i] <- min(data[[i]])
    means[i] <- mean(data[[i]])
  }
  print(maxs)
  print(mins)
  print(means)
}

> x=rnorm(100); y=rnorm(200); z=rnorm(300)
> fun(x,y,z)
[1] 2.387233 2.600142 3.401872
[1] -2.183967 -3.498059 -3.034946
[1] 0.020602626 0.017320416 0.001538235
```

Формальные аргументы, локальные и свободные переменные

- Формальные аргументы – это аргументы, перечисленные в заголовке функции
- Локальные переменные – это переменные, не являющиеся формальными аргументами, значения которых определяются во время выполнения функции.
- Свободные (глобальные) переменные – это переменные, не являющиеся формальными аргументами и локальными переменными.

Пример:

```
f = function(x){
  y = 2*x
  print(x); print(y); print(z)
}

> z <- 3
> f(1)
[1] 1
[1] 2
[1] 3
```

Сильное присваивание в R

```
arg <<- выражение
```