

Control Flow Structures

STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

`gastonsanchez.com`

`github.com/gastonstat/stat133`

Course web: `gastonsanchez.com/teaching/stat133`

Expressions

Expressions

R code is composed of a series of **expressions**

- ▶ assignment statements
- ▶ arithmetic expressions
- ▶ function calls
- ▶ conditional statements
- ▶ etc

Simple Expressions

assignment statement

`a <- 12345`

arithmetic expression

`525 + 34 - 280`

function call

`median(1:10)`

Expressions

One way to separate expressions is with new lines:

```
a <- 12345  
525 + 34 - 280  
median(1:10)
```

Grouping Expressions

Constructs for grouping together expressions

- ▶ semicolons
- ▶ curly braces

Separating Expressions

Simple expressions separated with new lines:

```
a <- 10  
b <- 20  
d <- 30
```

Grouping Expressions

Grouping expressions with semicolons:

```
a <- 10; b <- 20; d <- 30
```


Grouping Expressions

Grouping expressions with braces:

```
{  
  a <- 10  
  b <- 20  
  d <- 30  
}
```

Grouping Expressions

Multiple expressions in one line within braces:

```
{a <- 10; b <- 20; d <- 30}
```

Brackets and Braces in R

Symbol	Use
[]	brackets
()	parentheses
{ }	braces

Brackets and Braces

```
# brackets for objects  
dataset[1:10]
```

Brackets and Braces

```
# brackets for objects  
dataset[1:10]
```

```
# parentheses for functions  
some_function(dataset)
```

Brackets and Braces

```
# brackets for objects  
dataset[1:10]
```

```
# parentheses for functions  
some_function(dataset)
```

```
# brackets for expressions  
{  
  1 + 1  
  mean(1:5)  
  tbl <- read.csv('datafile.csv')  
}
```

Test yourself

Which is NOT a valid option for indexing data.frames

- A) Numeric vectors
- B) Logical vectors
- C) Missing values
- D) Empty indices
- E) Blank spaces

Expressions

- ▶ A program is a set of instructions
- ▶ Programs are made up of expressions
- ▶ R expressions can be simple or compound
- ▶ Every expression in R has a value

Expressions

```
# Expressions can be simple statements:
```

```
5 + 3
```

```
## [1] 8
```

```
# Expressions can also be compound:
```

```
{5 + 3; 4 * 2; 1 + 1}
```

```
## [1] 2
```

Expressions

The value of an expression is the last evaluated statement:

```
# value of an expression
```

```
{5 + 3; 4 * 2; 1 + 1}
```

```
## [1] 2
```

The result has the visibility of the last evaluation

Simple Expressions

We use braces `{ }` to group the statements of an expression:

```
# simple expression
```

```
{5 + 3}
```

```
## [1] 8
```

For simple expressions there is really no need to use braces

Compound Expressions

- ▶ Compound expressions consist of multiple simple expressions
- ▶ Compound expressions require braces
- ▶ Simple expressions in a compound expression can be separated by semicolons or newlines

Compound Expressions

Simple expressions in a compound expression separated by semicolons:

```
# compound expression  
{mean(1:10); '3'; print("hello"); c(1, 3, 4)}  
  
## [1] "hello"  
## [1] 1 3 4
```

Compound Expressions

Simple expressions in a compound expression separated by newlines:

```
# compound expression
{
  mean(1:10)
  '3'
  print("hello")
  c(1, 3, 4)
}

## [1] "hello"
## [1] 1 3 4
```

Compound Expressions

It is possible to have assignments within compound expressions:

```
{  
  x <- 4  
  y <- x^2  
  x + y  
}
```

```
## [1] 20
```

Compound Expressions

The variables inside the braces can be used in later expressions

```
z <- {x <- 4; y <- x^2; x + y}
```

```
x
```

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

```
y
```

```
## [1] 16
```

```
z
```

```
## [1] 20
```


Compound Expressions

```
# simple expressions in newlines
```

```
z <- {  
  x <- 4  
  y <- x^2  
  x + y}
```

```
x
```

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

```
y
```

```
## [1] 16
```

```
z
```

```
## [1] 20
```

Using Expressions

Expressions are typically used in

- ▶ Flow control structures (e.g. `for` loop)
- ▶ Functions

Compound Expressions

Do not confuse a function call (having arguments in multiple lines) with a compound expression

```
# this is NOT a compound expression  
plot(x = runif(10), y = rnorm(10),  
     pch = 19, col = "#89F39A", cex = 2,  
     main = "some plot",  
     xlab = 'x', ylab = 'y')
```

Control Flow Structures

Control Flow

There are many times where you don't just want to execute one statement after another: you need to control the flow of execution.

Main Idea

Execute some code when
a condition is fulfilled

Control Flow Structures

- ▶ if-then-else
- ▶ switch cases
- ▶ repeat loop
- ▶ while loop
- ▶ for loop

If-Then-Else

If-Then-Else

If-then-else statements make it possible to choose between two (possibly compound) expressions depending on the value of a (logical) condition.

```
# if-then-else  
if (condition) expression1 else expression2
```

If condition is true then expression1 is evaluated otherwise expression2 is executed.

If-Then-Else

If-then-else with **simple** expressions (equivalent forms):

```
# no braces  
if (condition) expression1 else expression2
```

```
# with braces  
if (condition) {  
    expression1  
} else {  
    expression2  
}
```

If-Then-Else

If-then-else with **compound** expressions:

```
# compound expressions require braces
if (condition) {
    expression1
    expression2
    ...
} else {
    expression3
    expression4
    ...
}
```

Example: If-Then-Else

Equivalent forms:

```
# simple if-then-else  
if (5 > 2) 5 * 2 else 5 / 2
```

```
## [1] 10
```

```
# simple if-then-else  
if (5 > 2) {  
  5 * 2  
} else {  
  5 / 2  
}
```

```
## [1] 10
```

Example: If-Then-Else

Equivalent forms:

```
# simple if-then-else  
if (x > 0) y <- sqrt(x) else y <- abs(x)
```

```
# simple if-then-else  
if (x > 0) {  
  y <- sqrt(x)  
} else {  
  y <- abs(x)  
}
```

If-Then-Else

- ▶ `if()` takes a **logical** condition
- ▶ the condition must be a logical value **of length one**
- ▶ it executes the next statement if the condition is true
- ▶ if the condition is false, then it executes the false expression

2 types of If conditions

if-then-else

```
# if and else  
if (condition) {  
    expression_true  
} else {  
    expression_false  
}
```

Just if

```
# simple if  
if (condition) {  
    expression_true  
}
```

It is also possible to just have the if clause (without else)

Just If

Just if

```
# just if  
if (condition) {  
    expression1  
    ...  
}
```

Equivalent to:

```
# just if (else NULL)  
if (condition) {  
    expression1  
    ...  
} else NULL
```


If and Else

- ▶ `if()` takes a **logical** condition
- ▶ the condition must be a logical value of length one
- ▶ it executes the next statement if the condition is true
- ▶ if the condition is false, and there is no `else`, then it stops
- ▶ if the condition is false, and there is an `else`, then it executes the false expression

Reminder of Comparison Operators

operation	usage
less than	<code>x < y</code>
greater than	<code>x > y</code>
less than or equal	<code>x <= y</code>
greater than or equal	<code>x >= y</code>
equality	<code>x == y</code>
different	<code>x != y</code>

Comparison operators produce logical values

Reminder of Logical Operators

operation	usage
NOT	<code>!x</code>
AND (elementwise)	<code>x & y</code>
AND (1st element)	<code>x && y</code>
OR (elementwise)	<code>x y</code>
OR (1st element)	<code>x y</code>
exclusive OR	<code>xor(x, y)</code>

Logical operators produce logical values

Just if's behavior

```
# this prints  
if (TRUE) {  
    print("It is true")  
}
```

```
# this does not print  
if (FALSE) {  
    print("It is false")  
}
```

```
# this does not print  
if (!TRUE) {  
    print("It is not true")  
}
```

```
# this prints  
if (!FALSE) {  
    print("It is not false")  
}
```

Just if

```
x <- 7

if (x >= 0) {
  print("it is positive")
}

## [1] "it is positive"

if (is.numeric(x)) {
  print("it is numeric")
}

## [1] "it is numeric"
```

If and Else

```
y <- -5

if (y >= 0) {
  print("it is positive")
} else {
  print("it is negative")
}

## [1] "it is negative"
```

The else statement must occur on the same line as the closing brace from the if clause!

If and Else

The logical condition must be of length one!

```
if (c(TRUE, TRUE)) {  
  print("it is positive")  
} else {  
  print("it is negative")  
}
```

```
## Warning in if (c(TRUE, TRUE)) {: the condition has  
length > 1 and only the first element will be used
```

```
## [1] "it is positive"
```

If and Else

What's the length of the logical condition?

```
x <- 3
y <- 4

if (x > 0 & y > 0) {
  print("they are not negative")
} else {
  print("they may be negative")
}

## [1] "they are not negative"
```


If and Else

If there's is a single statement, you can omit the braces:

```
if (TRUE) { print("It is true") }
```

```
if (TRUE) print("It is true")
```

```
# valid but not recommended
```

```
if (TRUE)  
    print("It is true")
```

Equivalent ways

No braces:

```
# ok  
if (TRUE) print("It's true")
```

```
# valid but not recommended  
if (TRUE)  
    print("It's true")
```

With braces:

```
# ok  
if (TRUE) {print("It's true")}
```

```
# recommended  
if (TRUE) {  
    print("It's true")  
}
```

If and Else

If there are multiple statements, you must use braces:

```
if (x > 0) {  
  a <- x^(2)  
  b <- 3 * a + 34.8 - exp(2)  
}
```

Multiple If's

Multiple conditions can be defined by combining `if` and `else` repeatedly:

```
set.seed(9)
x <- round(rnorm(1), 1)

if (x > 0) {
  print("x is positive")
} else if (x < 0) {
  print("x is negative")
} else if (x == 0) {
  print("x is zero")
}

## [1] "x is negative"
```

Vectorized ifelse()

`if()` takes a single logical value. If you want to pass a logical vector of conditions, you can use `ifelse()`:

```
true_false <- c(TRUE, FALSE)

ifelse(true_false, "true", "false")

## [1] "true"  "false"
```

Vectorized If

```
# some numbers
numbers <- c(1, 0, -4, 9, -0.9)

# are they non-negative or negative?
ifelse(numbers >= 0, "non-neg", "neg")

## [1] "non-neg" "non-neg" "neg"      "non-neg" "neg"
```

Test yourself

Which option will cause an error:

```
# A
if (is.numeric(1:5)) {
  print('ok')
}
```

```
# B
if ("TRUE") {
  print('ok')
}
```

```
# C
if (0) print('ok')
```

```
# D
if (NA) print('ok')
```

```
# E
if ('yes') {
  print('ok')
}
```

Switch

Multiple Selection with `switch()`

When a condition has multiple options, combining several `if` and `else` can become cumbersome

```
first_name <- "harry"

if (first_name == "harry") {
  last_name <- "potter"
} else {
  if (first_name == "ron") {
    last_name <- "weasley"
  } else {
    if (first_name == "hermione") {
      last_name <- "granger"
    } else {
      last_name <- "not available"
    }
  }
}

last_name

## [1] "potter"
```

Multiple Selection with switch()

```
first_name <- "ron"

last_name <- switch(
  first_name,
  harry = "potter",
  ron = "weasley",
  hermione = "granger",
  "not available")

last_name

## [1] "weasley"
```

Multiple Selection with `switch()`

- ▶ the `switch()` function makes it possible to choose between various alternatives
- ▶ `switch()` takes a character string
- ▶ followed by several named arguments
- ▶ `switch()` will match the input string with the provided arguments
- ▶ a default value can be given when there's no match
- ▶ multiple expression can be enclosed by braces

Multiple Selection with `switch()`

```
switch(expr,  
  tag1 = rcode_block1,  
  tag2 = rcode_block2,  
  ...  
)
```

`switch()` selects one of the code blocks, depending on the value of `expr`

Multiple Selection with switch()

```
operation <- "add"

result <- switch(
  operation,
  add = 2 + 3,
  product = 2 * 3,
  division = 2 / 3,
  other = {
    a <- 2 + 3
    exp(1 / sqrt(a))
  }
)

result

## [1] 5
```

Multiple Selection with switch()

- ▶ switch() can also take an integer as first argument
- ▶ in this case the remaining arguments do not need names
- ▶ instead, they will have associated integers

```
switch(  
  4,  
  "one",  
  "two",  
  "three",  
  "four")  
  
## [1] "four"
```

Empty code blocks in switch()

Empty code blocks can be used to make several tags match the same code block:

```
student <- "ron"

house <- switch(
  student,
  harry = ,
  ron = ,
  hermione = "gryffindor",
  draco = "slytherin")
```

In this case a value of "harry", "ron" or "hermione" will cause "gryffindor"

Loops

About Loops

- ▶ Many times we need to perform a procedure several times
- ▶ The main idea is that of **iteration**
- ▶ For this purpose we use loops
- ▶ We perform operations as long as some condition is fulfilled
- ▶ R provides three basic paradigms:
 - `for`, `repeat`, `while`

Repeat Loops

Repeat Loops

repeat executes the same code over and over until a stop condition is met.

```
repeat {  
  keep_doing_something  
  if (stop_condition) break  
}
```

The break statement stops the loops

Repeat Loops

```
value <- 2

repeat {
  value <- value * 2
  print(value)
  if (value >= 40) break
}
```

```
## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
```

If you enter an infinite loop, break it by pressing ESC key

Repeat Loops

To skip a current iteration, use `next`

```
value <- 2

repeat {
  value <- value * 2
  print(value)
  if (value == 16) {
    value <- value * 2
    next
  }
  if (value > 80) break
}
```

```
## [1] 4
## [1] 8
## [1] 16
## [1] 64
## [1] 128
```

While Loops

While Loops

It can also be useful to repeat a computation until a condition is false. A `while` loop provides this form of control flow

```
while (condition) {  
    keep_doing_something  
}
```


While Loops

- ▶ `while` loops are like backward repeat loops;
- ▶ `while` checks first and then attempts to execute
- ▶ computations are carried out for as long as the condition is true
- ▶ the loop stops when the condition is false
- ▶ If you enter an infinite loop, break it by pressing ESC key

While Loops

```
value <- 2

while (value < 40) {
  value <- value * 2
  print(value)
}
```

```
## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
```

If you enter an infinite loop, break it by pressing ESC key

For Loops

For Loops

Often we want to repeatedly carry out some computation a fixed number of times. For instance, repeat an operation for each element of a vector. In R this is done with a `for` loop

For Loops

for loops are used when we know exactly how many times we want the code to repeat

```
for (iterator in times) {  
  do_something  
}
```

for takes an *iterator* variable and a vector of *times* to iterate through

For Loops

```
value <- 2

for (i in 1:5) {
  value <- value * 2
  print(value)
}

## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
```

For Loops

The vector of *times* does not have to be a numeric vector; it can be any vector

```
value <- 2
times <- c('1', '2', '3', '4', '5')

for (i in times) {
  value <- value * 2
  print(value)
}

## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
```

For Loops and Next statement

Sometimes we need to skip a loop iteration if a given condition is met, this can be done with a `next` statement

```
for (iterator in times) {  
    expr1  
    expr2  
    if (condition) {  
        next  
    }  
    expr3  
    expr4  
}
```


For Loops and Next statement

```
x <- 2
for (i in 1:5) {
  y <- x * i
  if (y == 8) {
    next
  }
  print(y)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 10
```

For Loops

For loop

```
# squaring values
x <- 1:5
y <- x

for (i in 1:5) {
  y[i] <- x[i]^2
}

y

## [1] 1 4 9 16 25
```

Vectorized computation

```
# squaring values
x <- 1:5
y <- x^2
y

## [1] 1 4 9 16 25
```

Nested Loops

It is common to have nested loops

```
for (iterator1 in times1) {  
  for (iterator2 in times2) {  
    expr1  
    expr2  
    ...  
  }  
}
```

Nested Loops

```
# some matrix
```

```
A <- matrix(1:12, nrow = 3, ncol = 4)
```

```
A
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    4    7   10  
## [2,]    2    5    8   11  
## [3,]    3    6    9   12
```

Nested Loops

```
# nested loops
for (i in 1:nrow(A)) {
  for (j in 1:ncol(A)) {
    if (A[i,j] < 6) A[i,j] <- 1 / A[i,j]
  }
}
```

A

```
##           [,1] [,2] [,3] [,4]
## [1,] 1.0000000 0.25   7    10
## [2,] 0.5000000 0.20   8    11
## [3,] 0.3333333 6.00   9    12
```

For Loops and Vectorized Computations

- ▶ R for loops have a bad reputation for being slow
- ▶ Experienced users will tell you to avoid for loops in R (me included)
- ▶ R provides a family of functions that tend to be more efficient than loops (i.e. `apply()` functions)

For Loops and Vectorized Computations

- ▶ For purposes of learning programming (and flow control structures in R), I won't demonize R loops
- ▶ You can start solving a problem using a for loop
- ▶ Once you solved it, try to see if you can find a vectorized alternative
- ▶ It takes practice and experience to find alternative solutions to for loops
- ▶ There are cases when using for loops is inevitable

Repeat, While, For

- ▶ If you don't know the number of times something will be done you can use either `repeat` or `while`
- ▶ `while` evaluates the condition at the beginning
- ▶ `repeat` executes operations until a stop condition is met
- ▶ If you know the number of times that something will be done, use `for`
- ▶ `for` needs an *iterator* and a vector of *times*

For Loop

This example is just for demo purposes (not recommended in R)

```
# empty numeric vector
x <- numeric(0)
x

## numeric(0)

# for loop to fill x
for (i in 1:5) {
  x[i] <- i
}
x

## [1] 1 2 3 4 5
```

For Loop

If you know the number of times

```
# empty numeric vector  
x <- numeric(5)  
x
```

```
## [1] 0 0 0 0 0
```

```
# for loop to fill x  
for (i in 1:5) {  
  x[i] <- i  
}  
x
```

```
## [1] 1 2 3 4 5
```

Quiz

- ▶ What happens if you pass `NA` as a condition to `if()`?
- ▶ What happens if you pass `NA` as a condition to `ifelse()`?
- ▶ What types of values can be passed as the first argument to the `switch()` function?
- ▶ How do you stop a repeat loop executing?
- ▶ How do you jump to the next iteration of a loop?