Expressions and Conditionals

Programming Structures

Gaston Sanchez

STAT 33B, Fall 2025

About

Before describing some of the common programming structures in R, we need to talk about a basic concept called **Expressions**.

You've been using simple expressions so far, but we need to introduce the notion of a compound expression.

R Expressions

Simple Expressions

R code is composed of a series of *expressions*. So far, we've been using **simple expressions** like the following ones:

```r
# assignment statement
a <- 12345

# arithmetic expression
525 + 34 - 280

# function call
median(1:10)
```

Grouping Expressions

It is also possible to group several simple expressions

Constructs for grouping together expressions in R:

- semicolons: `;`
- curly braces: `{ }`

Grouping Simple Expressions

Simple expressions separated with new lines:

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

Grouping simple expressions with semicolons (within a single line of text):

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

Although this is a perfectly valid expression, we recommend avoiding semicolons, since they make code harder to review.

Grouping Expressions

Another way to group expressions is by wrapping them within braces:

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

R will treat this as one “unit” or “block” of code

Note: this piece of code is a perfectly valid expression, but I’m just using it for illustration purposes (useRs don’t write code like this!)

Grouping Expressions

Multiple expressions in one line within braces:

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

Note: again, this piece of code is just for illustration purposes (don't write code like this!)

Expressions

So far:

```
# Expressions can be simple statements:
5 + 3
```

```
[1] 8
```

```
# Expressions can also be compound:
{5 + 3; 4 * 2; 1 + 1}
```

```
[1] 2
```

Compound Expressions

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

Expressions

In summary:

- 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**

Every expression has a value

Expressions

The value of an expression is the last evaluated statement:

```r
# value of an expression
{5 + 3; 4 * 2; 1 + 1}
```

```
[1] 2
```

The result has the visibility of the last evaluation

Compound Expressions

What happens when R executes this code?

```r
{
  a <- "hi"
  print(2 + 2)
  mean(1:10)
}
```

```
[1] 4
```

```
[1] 5.5
```

The variables inside the braces can be used in later expressions.

Compound Expressions

What about this code:

```
x <- {
  a <- "hi"
  print(2 + 2)
  mean(1:10)
}
```

```
[1] 4
```

```
a
```

```
[1] "hi"
```

The variables inside the braces can be used in later expressions

Compound Expressions

```
# simple expressions in newlines
z <- {
  x <- 4
  y <- x^2
  x + y}
x
```

```
[1] 4
```

```
y
```

```
[1] 16
```

```
z
```

```
[1] 20
```

Repeat this Mantra

Every expression in R has a value: **the value of the last statement that was evaluated**

Every expression in R has a value: **the value of the last statement that was evaluated**

Every expression in R has a value: **the value of the last statement that wasevaluated**

Using Compound Expressions

So when do you use (compound) expressions?

We use compound expressions (i.e. single expressions wrapped within braces) in programming structures like:

- conditionals (if-else statements)
- iterations (loops)
- functions

Parenthesis, Brackets, and Braces

`()`	functions	`mean(1:10)`

`[]`	objects	`vec[3]`
		`mat[2,4]`

`{}`	compound expressions	`{`
		`  a <- 3`
		`  b <- a^2`
		`}`

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')
```

Conditionals

Conditionals

If-else or **if-then-else**

As you know, this class of statements make it possible to choose between two (possibly compound) expressions depending on the value of a **logical condition**.

Generate a random Normal number

```
x <- rnorm(1)
```

?

Is it positive or negative?

If `x > 0`

↓

positive

If `x < 0`

↓

negative

Toy Example

```r
x <- rnorm(1)

if (x > 0) {
    print("positive")
} else {
    print("negative")
}
```

Equivalently

```r
if (x < 0) {
    print("negative")
} else {
    print("positive")
}
```

Anatomy of if-else

```
x <- rnorm(1)

if (x > 0) {
 print("positive")
} else {
 print("negative")
}
```

Anatomy of if-else

```
x <- rnorm(1)
```

if-else statement

```
if (x > 0) {
  print("positive")
} else {
  print("negative")
}
```

Anatomy of if-else

```
x <- rnorm(1)

if (x > 0) {
  print("positive")
} else {
  print("negative")
}
```

Logical condition

single TRUE

single FALSE

Anatomy of if-else

```
x <- rnorm(1)

if (x > 0) {
  print("positive")
} else {
  print("negative")
}
```

What to do if condition is TRUE

Anatomy of if-else

```
x <- rnorm(1)

if (x > 0) {
  print("positive")
} else {
  print("negative")
}
```

What to do if condition is FALSE

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 expressions in the `else` clause

The logical condition must be of **length one!**

```r
y <- rnorm(2)

if (y > 0) {
  print("positive")
} else {
  print("negative")
}
```

```
Error in if (y > 0) {: the condition has length > 1
```

Example

What if you don't care about the else clause?

```
x <- rnorm(1)

if (x < 0) {
    print("negative")
} else {             # don't care
    print("positive")  # don't care
}                    # don't care
```

Example

What if you don't care about the `else` clause?

```
x <- rnorm(1)

if (x < 0) {
    print("negative")
} else {             # don't care
    print("positive")  # don't care
}                    # don't care
```

If you don't care about the `else` clause, then don't use it:

```
x <- rnorm(1)

if (x < 0) {
    print("negative")
}
```

Example

When you don't care about the `else` clause, R is actually *nullifying* the `else` clause:

```r
x <- rnorm(1)

if (x < 0) {
    print("negative")
} else NULL
```

Minimalist if's (simple expressions, no else)

```r
# option 1
if (x > 0) print("positive")

# option 2
if (x > 0)
  print("positive")

# option 3
if (x > 0) {print("positive")}

# option 4 (I prefer this style)
if (x > 0) {
  print("positive")
}
```

Reminder of Comparison Operators

Operator	Description
`x == y`	equal
`x != y`	not equal
`x < y`	less than
`x > y`	greater than
`x <= y`	less than or equal
`x >= y`	greater than or equal

- recall that comparison operators produce logical values
- they are typically used in `if-else` statements

Reminder of Logical Operators

Operator	Description
`!x`	NOT
`x & y`	AND (elementwise)
`x && y`	AND (1st element)
`x | y`	OR (elementwise)
`x || y`	OR (1st element)
`xor(x, y)`	exclusive OR

- logical operators are also typically used in `if-else` statements

Multiple Nested If’s

Multiple Nested If’s

Generate a random Normal number. Is it positive? Is it negative? Or is it zero?

```r
x <- rnorm(1)

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

You can chain several if-else statements.

Multiple Nested If's

In the previous example, we can simplify the third condition as:

```r
x <- rnorm(1)

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

switch() function

Multiple If’s

Working with multiple chained if’s becomes cumbersome, for example:

```r
# Convert the day of the week into a number
day <- "Tuesday" # Change this value!

if (day == 'Sunday') {
  num_day <- 1
} else if (day == "Monday") {
  num_day <- 2
} else if (day == "Tuesday") {
  num_day <- 3
} else if (day == "Wednesday") {
  num_day <- 4
} else if (day == "Thursday") {
  num_day <- 5
} else if (day == "Friday") {
  num_day <- 6
} else if (day == "Saturday") {
  num_day <- 7
}
```

switch() function

If you find yourself using many if-else statements with identical structure for slightly different cases, you may want to consider a **switch** statement instead:

```r
# Convert the day of the week into a number
day <- "Tuesday" # Change this value!

switch(
  day, # The expression to be evaluated
  Sunday = 1,
  Monday = 2,
  Tuesday = 3,
  Wednesday = 4,
  Thursday = 5,
  Friday = 6,
  Saturday = 7,
  NA) # an (optional) default value if there are no matches
```

```
[1] 3
```

switch() function

Switch statements can also accept integer arguments, which will act as indices to choose a corresponding element:

```r
# Convert a number into a day of the week
day_num <- 3 # Change this value!

switch(day_num,
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday")
```

```
[1] "Tuesday"
```

Congruent Vectors Strategy

`if-else` limitations

As we saw it, if-statements don't work well with vectors.

For example, suppose we want to transform a vector `x` so that:

- Negative elements are set to 0.
- Non-negative elements are squared.

```r
# Naive attempt
x <- c(-4, 5, 10, -3, 2, 1)

if (x < 0) { x = 0 }

if (x >= 0) { x = x^2 }
```

if-else limitations

Unfortunately, if-statements are NOT vectorized

```r
x <- c(-4, 5, 10, -3, 2, 1)

# Using an if-statement doesn't work for this:
if (x < 0) { x = 0 }
```

```
Error in if (x < 0) {: the condition has length > 1
```

Congruent Vectors

Instead, use the so-called **congruent vectors** strategy which involves:

1. An input vector (or vectors) to use in conditions.
2. An output vector to store the results.

Use the input vector to conditionally assign elements to the output vector.

```r
x <- c(-4, 5, 10, -3, 2, 1)
output <- x
output[x < 0] <- 0
output[x > 0] <- x[x > 0]^2
output
```

```
[1]   0  25 100   0   4   1
```

`ifelse()` function

ifelse() function

- R also has a vectorized `ifelse()` function.
- `ifelse()` can be useful when the “condition” evaluates into a logical vector that does not have just one element.

For example:

```
x <- c(-1, 10, 20, -3)
ifelse(x < 0, 0, x)
```

```
[1]  0 10 20  0
```

Note: The `ifelse()` function is not the *panacea*; it is less efficient than the congruent vectors strategy.

dplyr's case_when()

dplyr's case_when() function

Interestingly, the package "dplyr" provides its general vectorized if-else function called **case_when()**

- This function allows you to vectorize multiple if-else statements.
- Each case is evaluated sequentially.
- The first match for each element determines the corresponding value in the output vector.
- The tilde **~** operator is used to determine the assigned value in each case.

Example `case_when()`

Example in which `x` is a numeric vector, and we want to transform its elements so that: negative elements are set to 0, and Non-negative elements are squared.

```
x <- c(-1, 10, 20, -3)

case_when(
  x >= 0 ~ x^2,
  x < 0 ~ 0,
  .default = NA
)
```

```
[1]   0 100 400   0
```

Note: don't forget to load `library(tidyverse)`

Example `case_when()`

The previous command can also be implemented like this:

```r
x <- c(-1, 10, 20, -3)

case_when(
  x < 0 ~ 0,
  x >= 0 ~ x^2,
  .default = NA
)
```

```
[1]   0 100 400   0
```

Example case_when()

```r
# Convert the day of the week into a number
day <- "Tuesday" # Change this value!

num_day = case_when(
  day == "Sunday" ~ 1,
  day == "Monday" ~ 2,
  day == "Tuesday" ~ 3,
  day == "Wednesday" ~ 4,
  day == "Thursday" ~ 5,
  day == "Friday" ~ 6,
  day == "Saturday" ~ 7,
  .default = NA)

num_day
```

```
[1] 3
```

Example case_when()

```r
# Convert the day of the week into a number
days <- c("Tue", "Fri", "Sun", "Mon", "Tue", "Wed", "Unk")

num_days = case_when(
  days == "Sun" ~ 1,
  days == "Mon" ~ 2,
  days == "Tue" ~ 3,
  days == "Wed" ~ 4,
  days == "Thu" ~ 5,
  days == "Fri" ~ 6,
  days == "Sat" ~ 7,
  .default = NA)

num_days
```

```
[1]  3  6  1  2  3  4 NA
```