Flow control structures I

Yufeng Huang

Associate Professor of Marketing, Simon Business School

August 9, 2022

The past two weeks

- Week 1
  - R/RStudio interface
  - variable assignments
  - vector operation and sub-setting
- Week 2
  - data structure
  - arrays and dimensions
  - lists and data frame subsetting
  - merge, aggregate and reshape
  - data table

This week

- Flow constrol: “if” statement and “for” or “while” loop
- Strings and regular expressions
- Assignment: RFM targeting
  - can we target different consumers by their past behavior?
  - recency, frequency and monetary value are three measures that are quite useful
  - we learn to construct these measures using flow control structures

Basics: expressions[1]

[1] We pretty much know what these are but we haven't dealt with braces much

What's the difference between parenthesis, brackets and braces?

Symbol	Use
[] brackets	Objects
() parenthesis	Functions
{} braces	Expressions

What are the differences between parenthesis, brackets and braces?

```r
# brackets for objects
my_vector[1:10]

# parenthesis for function call
some_function(my_vector)

# brackets for expressions
{
        1 + 1
        mean(1:5)
        my_dataframe <- read.csv("some_file.csv")
}
```

Expressions

- R code is composed of a series of expressions
  - assignment
  - arithmetic operations
  - function calls[2]
  - *conditional statements
  - etc.

[2] We've used lots of built-in functions!

Simple expressions

```r
# assignment expression
a <- 2 + 1

# assignment with function call
b <- log(4)

# arithmetic expression
a^2 + b

## [1] 10.38629
```

Use curly braces to group statements

```
# group statements into expressions
{
        a <- 2 + 1
        b <- log(4)
        a^2 + b
}

## [1] 10.38629
```

Value of an expression is the value of the last statement

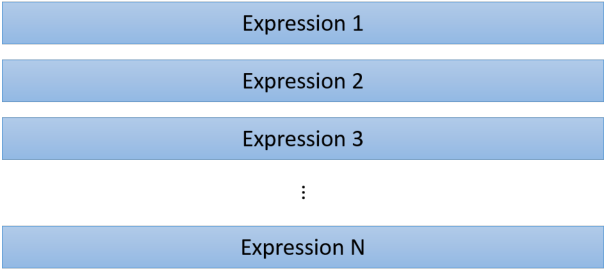
```
# simple expression
1 + 1

## [1] 2

# compound expression with braces

{
        1 + 1
        2 + 1
        3 + 1
}

## [1] 4
```

But assignment statements in a compound expression can be used later

```
# assignment in a compound expression
z <- {x <- 4; y <- x^2; x + y}

x

## [1] 4

y

## [1] 16

z

## [1] 20
```

The flow of R code

In R, code is executed line by line



Expression 1

Expression 2

Expression 3

⋮

Expression N

Like your dinner plan every day

```r
# series of assignments
action <- "order Steak" # day 1
action

## [1] "order Steak"

action <- "order Salad" # day 2
action

## [1] "order Salad"

action <- "order Steak" # day 3
action

## [1] "order Steak"

action <- "order Salad" # day 4
action

## [1] "order Salad"
```

Another way to express this

```
possible_actions <- c(   # define a set of possible actions
        "order Steak",
        "order Salad"
)
d <- 1; # day 1
possible_actions[1]
```

```
## [1] "order Steak"
```

```
d <- 2; # day 2
possible_actions[2]
```

```
## [1] "order Salad"
```

```
d <- 3; # day 3
possible_actions[1]
```

```
## [1] "order Steak"
```

```
d <- 4; # day 4
possible_actions[2]
```

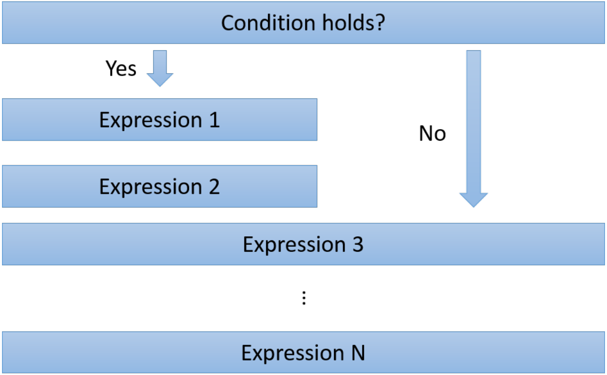
```
## [1] "order Salad"
```

But what's my decision on day 100?

Note that below, the flow of code is still line by line, but **not every line is executed**

```
# if my choices are so regular,
#    it can be wrapped in a conditional statement

d <- 100
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

## [1] "order Salad"

# reads:
#    on odd number of days I order Steak,
#    otherwise (i.e. on even days) I order Salad
```

Graphically



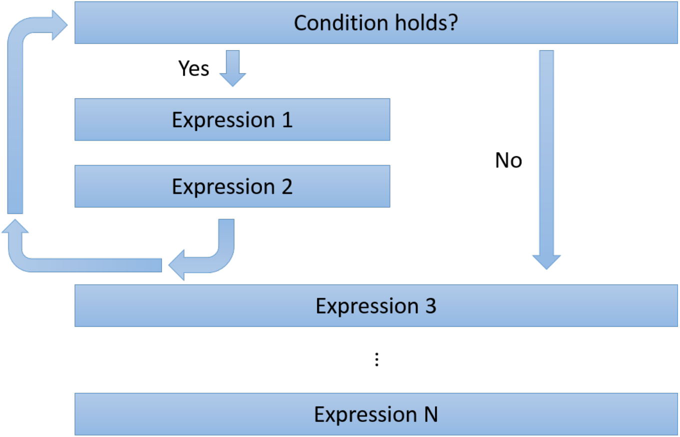
But what are my decisions during days 51-58?

```
# copy and paste everything?

d <- 51
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

## [1] "order Steak"

d <- 52
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

## [1] "order Salad"

d <- 53
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

## [1] "order Steak"

# and so on...
```

If the decision is so regular (i.e. rule is fixed), can write a loop on this

```
# loop version

d <- 51
while (d <= 58) {
        if (d %% 2 == 1) {
                print(possible_actions[1])
        } else {
                print(possible_actions[2])
        }
        d <- d + 1
}

## [1] "order Steak"
## [1] "order Salad"
## [1] "order Steak"
## [1] "order Salad"
## [1] "order Steak"
## [1] "order Salad"
## [1] "order Steak"
## [1] "order Salad"

# reads: starting with day 51
#    while day does not exceed 58
#    execute the same decision rule
#    and add 1 to day
```

Graphically



Flow control statements are used when I have reasons to interrupt the “natural” flow of code

Flow control

- There are times when you don't want to execute statements one after another
- You want to execute a section of code when a condition is fulfilled
- This (and the next) lecture note
  - if-else
  - switch cases
  - for loop
  - while loop
  - repeat loop
- "Theory" is easy but applications can be tricky

If-else

If-else

- If-else statements make it possible to choose between two expressions depending on the value of a (logical) condition
- If condition is satisfied, expression 1 is executed; otherwise expression 2 is executed

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

# equivalent
if (condition) {
        expression1
} else {
        expression2
}
```

If-else

- Can take compound expressions

```
# compound version
if (condition) {
        expression1
        expression2
        expression3
} else {
        expression4
        expression5
}
```

Example

```
# example

if (5 > 2) {
        5 * 2
} else {
        5 / 2
}

## [1] 10

# or one line, but please do this only when code is short

if (5 > 2) 5 * 2 else 5 / 2

## [1] 10
```

Example

```
# I recommend using the braces form because it's more readable

x <- -4

if (x > 0) {
        sqrt(x)
} else {
        sqrt(-x)
}
```

Example: second expression can be empty

```r
x <- -4

if (x > 0) {    # empty second statement
        sqrt(x)
} else NULL

if (x > 0) {    # equivalent to just if()
        sqrt(x)
}
```

If-else

- if() takes a logical expression
- Condition must be of length 1
- Executes first statement if condition is TRUE (**length 1**!)
- Executes second statement if condition is FALSE
- If there is no second statement and the condition is FALSE, just stops

Your turn: what are the output?

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

if (FALSE) {
    print("It's false")
)

if (!TRUE) {
    print("It's not true")
}

if (!FALSE) {
    print("It's not false")
}
```

Nesting if-else

```
# one can nest if-else

if (x >= 2) {
        statement1
} else {
        if (x >= 1) {
                statement2
        } else {
                statement3
        }
}

# now you see the importance of indentation right?
```

Your turn: a different statement

- Which statement is executed if:
    - $x = 0.5$?
    - $x = 1.5$?
    - $x = 2.5$?

```
# different statement
if (x >= 1) {
        if (x >= 2) {
                statement1
        } else {
                statement2
        }
} else {
        statement3
}
```

ifelse()

Example: piece-wise demand curve

- Demand is piece-wise linear
  - i.e. price sensitivity is -2 when prices are above $1, otherwise it is -3
  - more precisely,

$$sales = \begin{cases} 4 - 2 \cdot price & \text{if } price > 1 \\ 5 - 3 \cdot price & \text{if } price \leq 1 \end{cases}$$

  and one can verify that demand is connected at price=1

- Can you hand-draw this?
  - by the way, what behavior generates this figure?

Applying if/else on vector logical conditions lead to an error

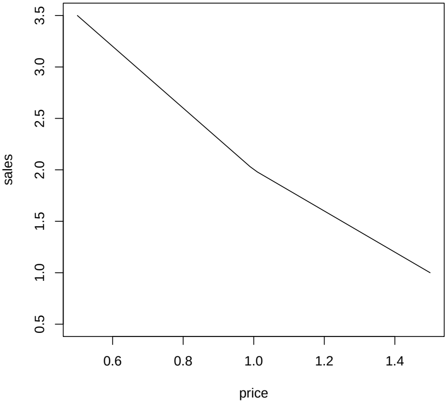
```r
# price is a vector of length 20
price <- seq(0.5, 1.5, length.out = 50)

# but we can't apply the if-else statement to a vector
if (price > 1) {
        sales <- 4 - 2*price
} else {
        sales <- 5 - 3*price
}

       ## Error in if (price > 1) {:  the condition has length > 1
```

ifelse() as a **function**

- One alternative is to use ifelse()
- Note that ifelse() is a function, while if (cond) {expr1} else {expr2} is a flow control structure

```
# ifelse instead of if-then-else
sales <- ifelse(price > 1, 4 - 2*price, 5 - 3*price)
```

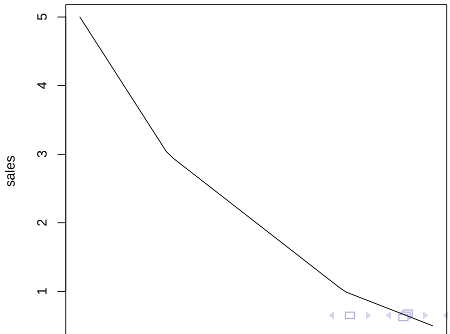
```r
# plot the demand curve (maintain the same axis)
plot(price, sales, ylim = c(0.5, 3.5), type = 'l')
```



Nest ifelse()

But do note that ifelse() is a function

```r
# Nesting ifelse()
sales <- ifelse(price <= 0.75,
        9 - 8*price,
        ifelse(price <= 1.25,
                6 - 4*price,
                3.5 - 2*price)
)
plot(price, sales, type = 'l')
```



Switch statement

Example problem: match first name with last name

first	last
Kristina	Brecko
Hana	Choi
Paul	Ellickson
Ron	Goettler
Avery	Haviv
Yufeng	Huang
Mitch	Lovett
Paul	Nelson
Takeaki	Sunada

Naturally, switch() gives a multiple-choice problem

```r
# say I want Paul
first <- "Paul"

last <- switch(
    first,
    Kristina = "Brecko",
    Hana = "Choi",
    Ron = "Goettler",
    Avery = "Haviv",
    Yufeng = "Huang",
    Mitch = "Lovett",
    Takeaki = "Sunada",
    Paul = "Which Paul do you want"
)

last

## [1] "Which Paul do you want"
```

Write this in if-else

```
# let's define a rule
if (first == "Kristina") {
    last <- "Brecko"
} else {
    if (first == "Hana") {
        last <- "Choi"
    } else {
        if (first == "Ron") {
            last <- "Goettler"
        } else {
            if (first == "Avery") {
                last <- "Haviv"
            } else {
                if (first == "Yufeng") {
                    last <- "Huang"
                } else {
                    if (first == "Mitch") {
                        last <- "Lovett"
                    } else {
                        if (first == "Takeaki") {
                            last <- "Sunada"
                        } else {
                                last <- "Which Paul do you want?"
                        }
                    }
                }
            }
        }
    }
}

last

## [1] "Which Paul do you want?"
```

Switch()

- Switch() function selects among multiple alternatives
- Input is a character string
- Followed by named arguments
  - matches input with name
  - and gives corresponding output
- Switch() is a special case of if (string $==$ name) $\{$output $=$ output arg$\}$
- But easy to deal with multiple cases

Summary

- The flow of code goes top to bottom unless we modify it
- That's why we want flow control
- If-then-else statement re-routes the flow given conditions
  - avoids certain section of code if conditions are not met
- Switch statement is a convenient alternative to multiple if-else