An introduction to R – Part 2

Harlley Lima & Manel Slokom

November 21, 2018

Overview

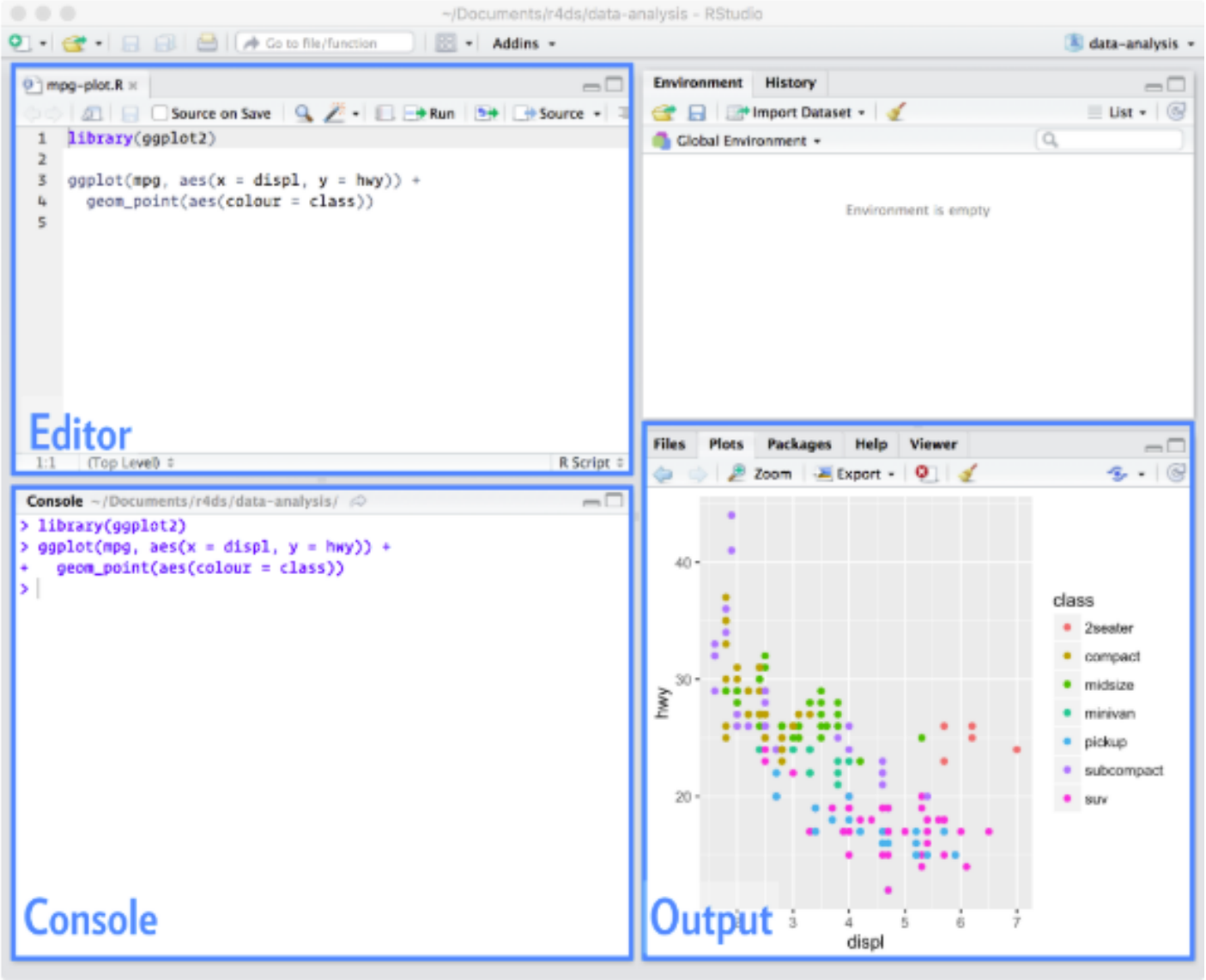
1. Programming with R
   - R Script
   - Control structures
   - User-defined Functions
   - Loop functions

R Script

R Script

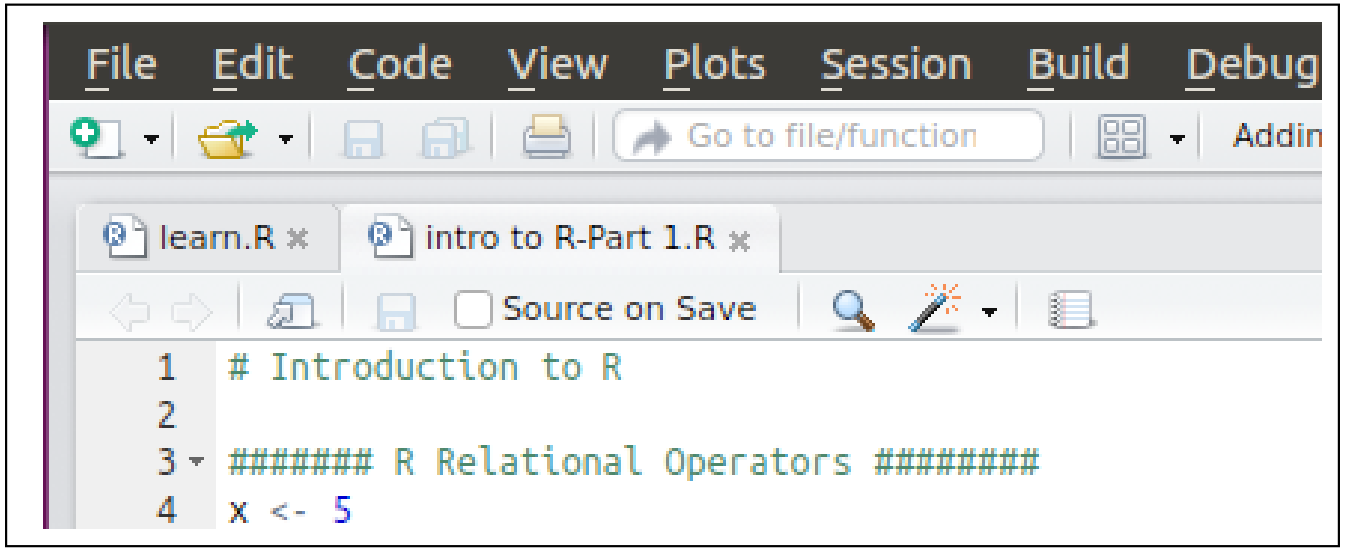
- Everything we have shown so far can be done using R console
- A script is a good way to keep track of what you're doing
- If you have a long analysis, and you want to be able to recreate it later, a good idea is to type it into a script

R Script


~/Documents/r4ds/data-analysis - RStudio
Addins
data-analysis
mpg-plot.R
Source on Save
Run
Source
library(ggplot2)
ggplot(mpg, aes(x = displ, y = hwy)) +
geom_point(aes(colour = class))
Editor
1:1
(Top Level)
R Script
Console ~/Documents/r4ds/data-analysis/
> library(ggplot2)
> ggplot(mpg, aes(x = displ, y = hwy)) +
+ geom_point(aes(colour = class))
>
Console
Environment
History
Import Dataset
List
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
Zoom
Export
hwy
displ
class
2seater
compact
midsize
minivan
pickup
subcompact
suv
Output

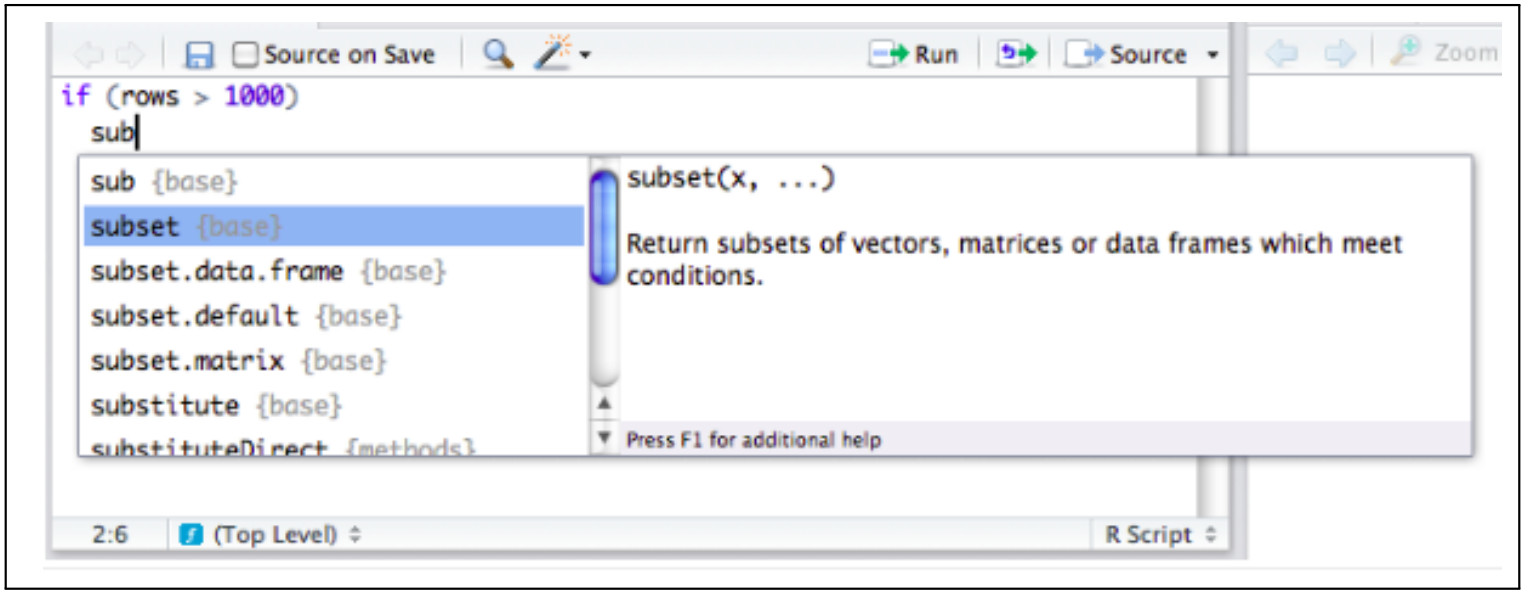
R Script file

- To create a new file you use the: **File -> New File -> R Script (Ctrl + Shift + N)**
- To open an existing file you use the **File -> Open File**
- If you open several files within RStudio they are all available as tabs



Code Completion

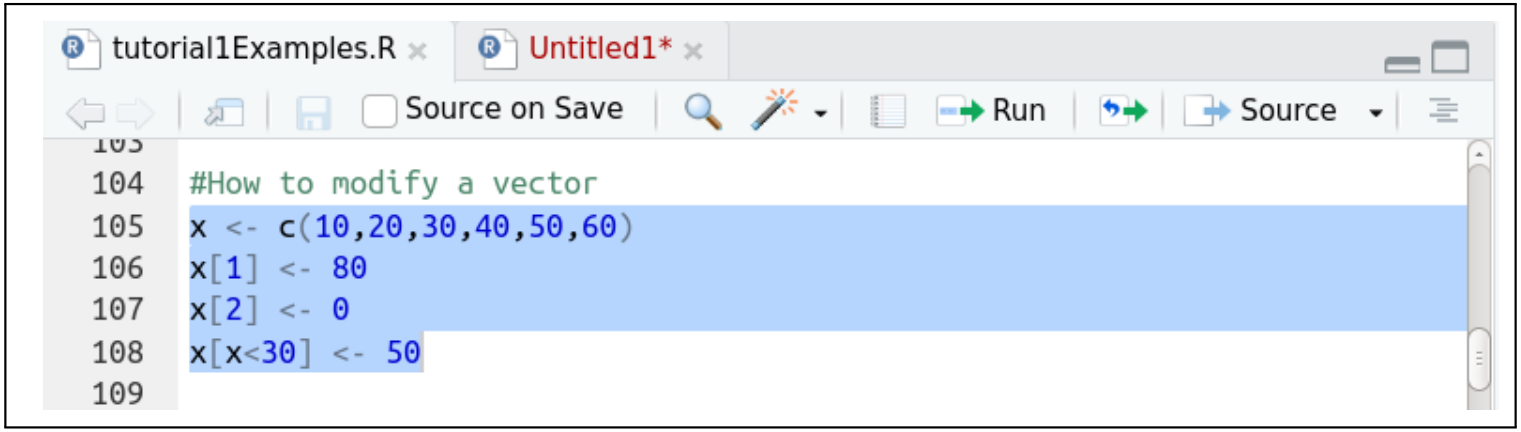
- RStudio supports the automatic completion of code using the Tab key
- If you have an function named `subset`, you can type `sub` and then Tab and RStudio will automatically complete the full name



Executing Code

Executing a single line

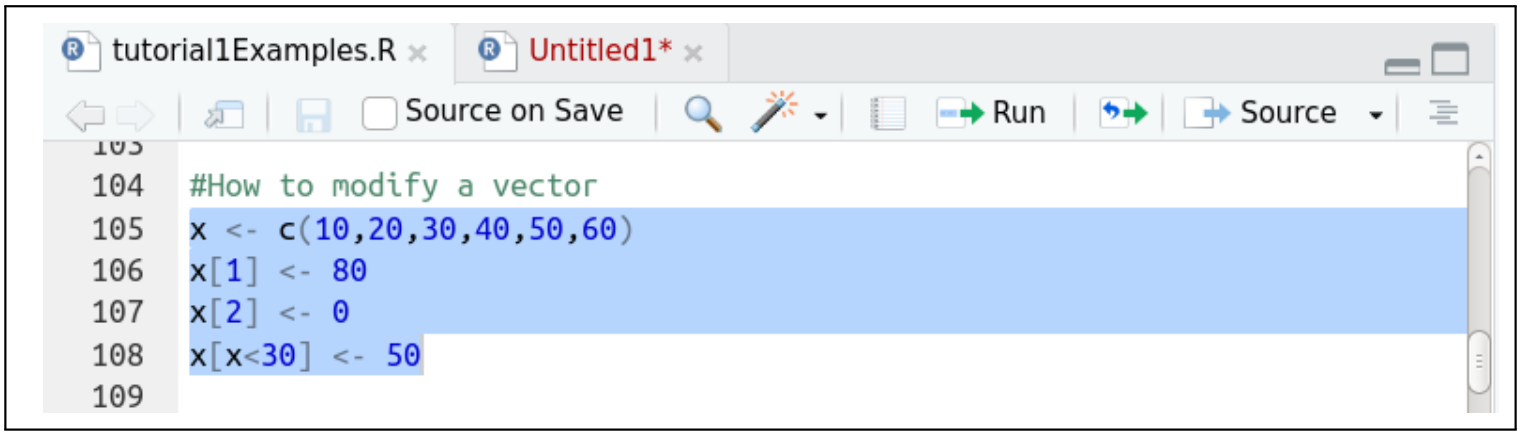
- To execute the line of source code where the cursor currently resides you press the **Ctrl + Enter** key (or use the Run toolbar button)
- After executing the line of code, RStudio automatically advances the cursor to the next line



Executing Code

Executing multiple lines

- Select the lines and press the **Ctrl + Enter** key (or use the Run toolbar button)
- To run the entire document press the **Ctrl + Shift + Enter** key



Keyboard Shortcuts

- Ctrl + Shift + N: New document
- Ctrl + O: Open document
- Ctrl + S: Save active document
- **Ctrl + 1**: Move focus to the Source Editor
- **Ctrl + 2**: Move focus to the Console

Control structures

Control structures

- Allow to control the flow of execution of a series of statements
- There are different types of control flow statements:
  - executing a set of statements only if some condition is met
  - executing a set of statements a given number of times
  - executing a set of statements until some condition is met
  - stop executing one thing or quit entirely
- Control structures allow to put some "logic" into the code

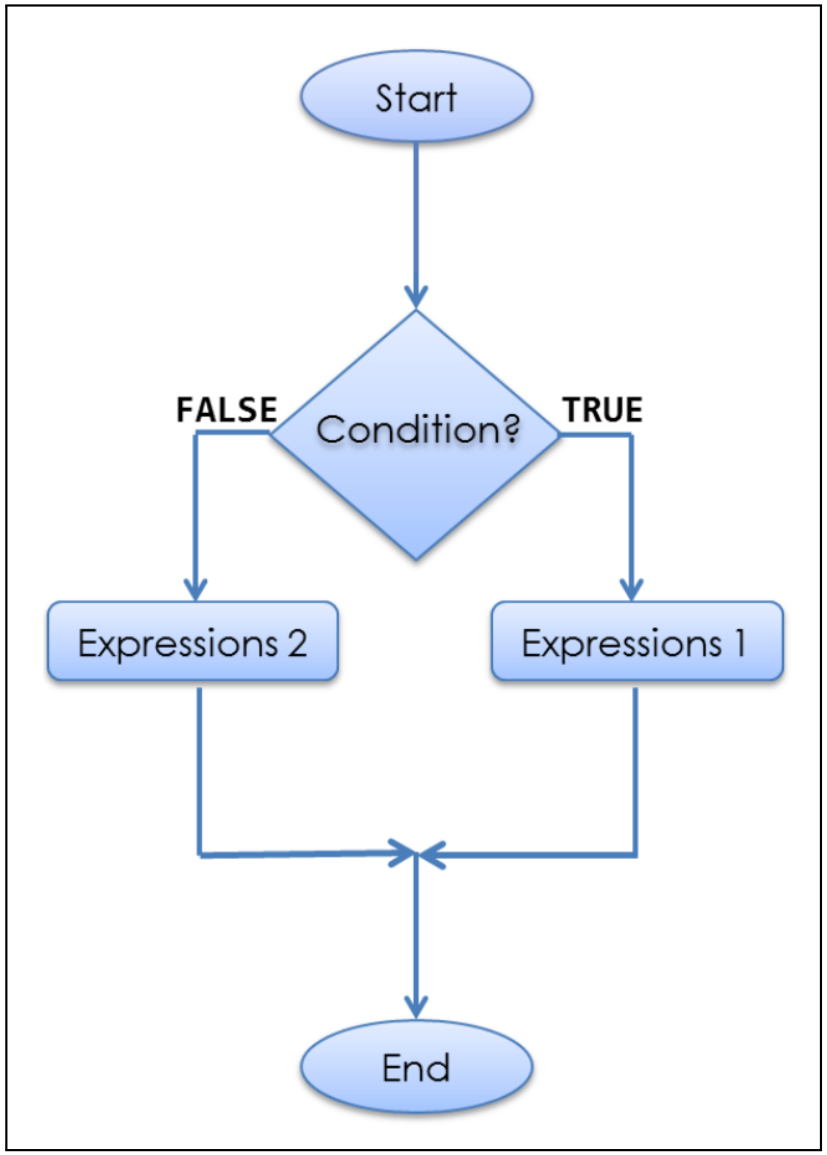
Conditional execution

If statements

- `if` statements allow to do different things based on the value of some condition.
- Conditional expressions in R.

Simple condition

if (*condition*) {
# condition is TRUE
} **else** {
# condition is FALSE
}



If statements

```
> x <- 10
> if ( x > 0 )
  {
   print(``This is Positive number'')
  }
[1] ``This is Positive number''
```

If-else statements

if-else

if (*this*) {
  # Do that
} **else if** (that) {
  # Do something else
} **else** {
  #
}

Example

```
> x <- -10
> if(x >= 0)
  {
    print("Non-negative number")
  } else {
    print("Negative number")
  }
```

If-else statements

```
> x <- c(2, 1, 3)
> if(sqrt(9) > 2)
  {
    mean(x)
  } else {
    sum(x)
  }
[1] 2
```

```
> if(sqrt(9) > 4) { mean(x) } else { sum(x) }
[1] 6
```

ifelse

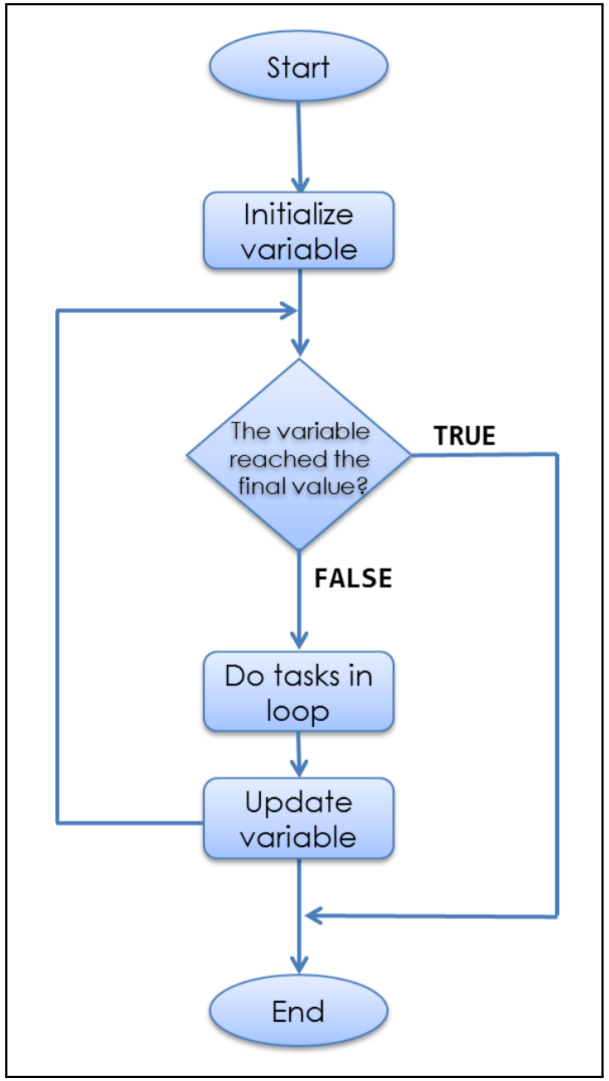
ifelse (test, TrueValue, FalseValue)

```
> x <- c(2, 1, 3, 6, 8, 1)
> y <- ifelse(x > 4, x, sum(x))
> y
[1] 21 21 21 6 8 21
```

Repetitive execution: for loops, repeat and while

Repetitive execution

- Loop is used to repeatedly carry out some computation.



Loop "For"

```
> for (variable in vector) {
    commands
  }
```

```
> for (i in 1:5) {
    print(i)
  }
 [1] 1
 [1] 2
 [1] 3
 [1] 4
 [1] 5
```

Loop "For"

Note that the variable that is set in a for loop is changed in the calling environment.

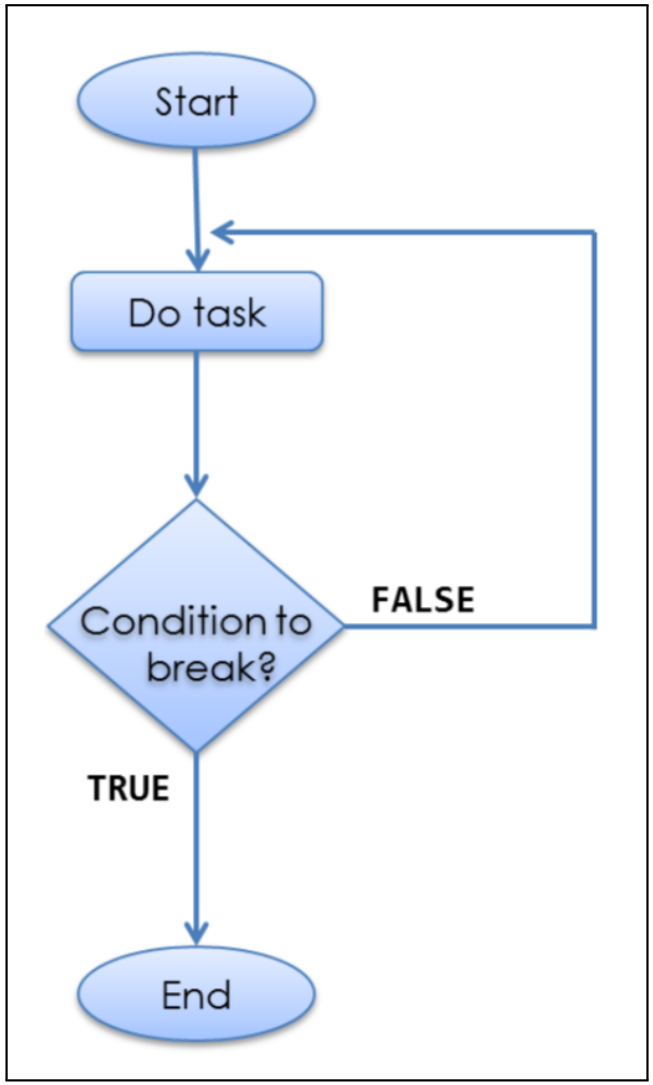
```
> i <- 1
> for (i in seq(5, 10, by = 2)) {
    print(i)
  }
[1] 5
[1] 7
[1] 9

> i
[1] 9
```

Loop "Repeat"

- "Repeat": repeats the same expression.
- The syntax is:

```
> repeat {
    expression
  }
```



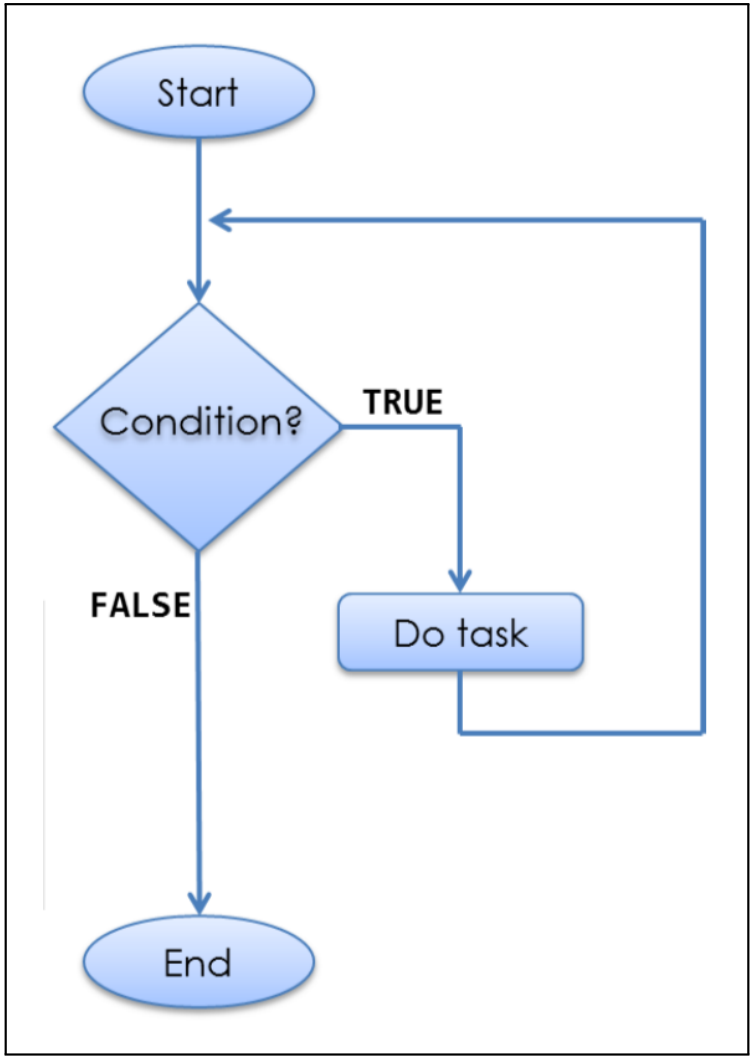
Loop "Repeat"

```
> x <- 7
> repeat {
    print(x);
    x <- x+ 2;
    if(x > 10) { break }
  }
 [1] 7
 [1] 9
```

Loop "While"

- The while structure evaluates a expression as long as a stated condition is TRUE.
- The syntax is:

```
> while (condition) {
      expression
  }
```



Loop "While"

```
> x <- 0
> while (x < 10) {
    print(x);
    x <- x + 5
  }
 [1] 0
 [1] 5
```

User-defined Functions

Summary measures functions

- R offers functions to compute summary measures for quantitative variables
- These functions operate on a vector `x`

R function	Description
`mean(x)`	arithmetic mean of `x`
`median(x)`	median of `x`
`var(x)`	variance of `x`
`sd(x)`	standard deviation of `x`
`range(x)`	minimum and maximum values of `x`

User-defined Functions

- One of the great strengths of R is the user's ability to add functions
- The structure of a function is given below:

```
myfunction <- function(arg1, arg2, ... ){
  statements
  return(object)
}
```

- Function elements:
  - Function name
  - Function arguments
  - Statements
  - Return

User-defined Functions

```
f <- function(num) {
  for(i in seq_len(num)) {
    print("Hello, world!")
  }
}
f(3)
```

- Function elements:
  - Function name: `f`
  - Function arguments: `num`
  - Statements: `function's body`
  - Return: It doesn't return anything
- **Tip: In general, if you find yourself doing a lot of cutting and pasting, thats usually a good sign that you might need to write a function**

User-defined Functions

```
f <- function(num) {
  hello <- "Hello, world!"
  for(i in seq_len(num)) {
    print(hello)
  }
  chars <- nchar(hello) * num
  chars
}
a = f(3)
a
```

- Function elements:
  - Function name: `f`
  - Function arguments: `num`
  - Statements: `function's body`
  - Return: chars. In R, the return value of a function is always the very last expression that is evaluated.

User-defined Functions

```
f <- function(num) {
  hello <- "Hello, world!"
  for(i in seq_len(num)) {
    print(hello)
  }
  chars <- nchar(hello) * num
  chars
}
a = f(3)
a
```

- The user must specify the value of the argument `num`. If it is not specified by the user, R will throw an error:

```
f()
Error in f():  argument "num" is missing, with no
default
```

User-defined Functions

```
f <- function(num = 1) {
  hello <- "Hello, world!"
  for(i in seq_len(num)) {
    print(hello)
  }
  chars <- nchar(hello) * num
  chars
}
a = f()
a
```

- We can modify this behavior by setting a default value for the argument `num`

User-defined Functions

Functions have named arguments which can optionally have default values. Because all function arguments have names, they can be specified using their name.

```
> f(2)
[1] "Hello, world!"
[1] "Hello, world!"
[1] 26
>
```

Argument matching

R functions arguments can be matched *positionally* or by name. Given the following function:

```
g <- function(n, mean = 0, sd = 1) {...}
```

- `mydata <- g(100, 2, 1)`
  - 100 is assigned to `n`
  - 2 is assigned to `mean`
  - 1 is assigned to `sd`

Argument matching

Given a function which sums all elements of the vector:

```
mysum <- function(vec, neg = TRUE) {...}
```

it has two arguments:

- `vec`: the vector of numbers
- `neg`: is a logical indicating whether negative values should be summed or not

Argument matching

Given a function which sums all elements of the vector:

```
mysum <- function(vec, neg = TRUE) {...}
```

- `mysum(v)`
  - Positional match first argument, and TRUE for `neg`
- `mysum(vec = v)`
  - Specify `vec` argument by name, and TRUE for `neg`
- `mysum(vec = v, neg = FALSE)`
  - Specify both arguments by name
- When specifying the function arguments by name, it doesn't matter in what order you specify them: `mysum(neg = FALSE, vec = v)`

The ... Argument

- Functions can have a special argument ...
- The ... argument is necessary when the number of arguments passed to the function cannot be known in advance

```
i01 <- function(...){
  x <- list(...)
  x
}
y <- i01(1,2,3)
x <- i01(1,2,3,4,5)
```

Loop functions

Looping on the command line

- Writing "for" and "while" loops is useful but not particularly easy
- using loops in R is slow
- R has important family of functions which implement looping in a compact form

Functions

- **lapply()**: Loop over a list and evaluate a function on each element
- **sapply()**: same as "lapply" but try to simplify the result
- **apply()**: apply a function over the margins of an array

lapply()

1. lapply() loops over a list, iterating over each element in that list
2. it applies a function to each element of the list (function specified by you)
3. returns a list

- "lapply()" takes the following arguments: (1) a list, (2) a function, (3) other (`?lapply`).
- The output will be coerced to a list using *as.list()*

lapply()

To apply the `mean()` function to all elements of a list.

```
x <- list(a = 1:5, b = 10:15)
lapply(x, mean)
```

Note that here we passed the `mean` function as an argument to the `lapply()` function.

sapply()

- The sapply() function behaves similarly to `lapply()`.
- The only difference is in the return value
- `sapply()` will try to simplify the result of `lapply()` (if possible).

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

sapply()

```
x <- list(a = 1:4, b = 10:15, c = 20:25, d = 30:35)
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 12.5

$c
[1] 22.5

$d
[1] 32.5
```

```
> sapply(x, mean)
a          b          c          d
2.5        12.5       22.5       32.5
```

apply()

The function apply() can be used to apply a function to the rows (second argument equal to 1) or columns (second argument equal to 2) of a matrix, arrays (> 2) dimensions or data.frames.

```
# matrix a
> a
> apply(a, 1, sum) # Apply sum to rows
> apply(a, 2, mean) # Apply mean to columns
```

The result is equivalent to:

```
 rowSums(a) # form row sums
colMeans(a) # Apply mean to columns
```

References

Hadley Wickham & Garrett Grolemund (2017)
R for data science: import, tidy, transform, visualize, and model data
O'Reilly.

Roger D. Peng (2015)
R Programming for Data Science

Venables W.N. & Smith D. M. (2018)
An introduction to R: Notes on R - A programming for Data Analysis and Graphics
Version 3.5.1.

Emmanuel Paradis (2005)
R for beginners

Link:
Advanced R
https://adv-r.hadley.nz/

The End