Computational Social Science

Programming in R

Dr. Thomas Davidson

Rutgers University

January 26, 2022

Plan

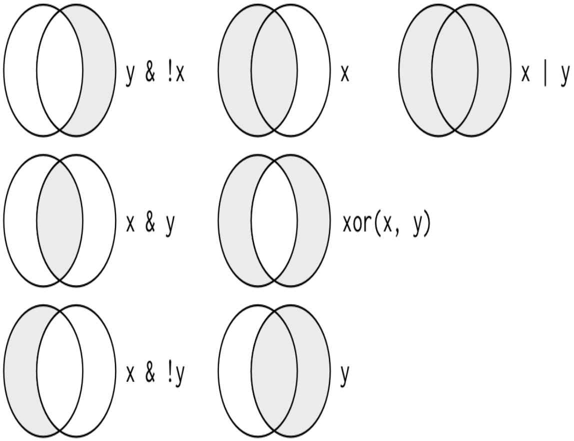
- RStudio overview
- Programming fundamentals
  - Boolean logic
  - If-else statements
  - Loops
  - Functions
  - Pipes

Recap

Data structures

- Basic data types
- Vectors
- Lists
- Matrices
- Data frames and tibbles

Boolean logic in R



Boolean logic

```
TRUE == TRUE  # equals
```

```
## [1] TRUE
```

```
TRUE != FALSE  # not equals
```

```
## [1] TRUE
```

```
TRUE == !FALSE
```

```
## [1] TRUE
```

```
!TRUE != FALSE
```

```
## [1] FALSE
```

Boolean logic

```
TRUE | FALSE  # or
```

```
## [1] TRUE
```

```
TRUE & FALSE  # and
```

```
## [1] FALSE
```

```
TRUE & FALSE == FALSE
```

```
## [1] TRUE
```

Boolean logic

```
TRUE | FALSE & FALSE
```

```
## [1] TRUE
```

```
FALSE | TRUE & FALSE
```

```
## [1] FALSE
```

See the documentation for more on logic in R.

If-else statements

- We often encounter situations where we want to make a choice contingent upon the value of some information received.
- If-else statements allow us to chain together one or more conditional actions.
    - e.g. If time is between 2:00-3:20pm AND day is Monday or Wednesday, attend Computational Data Science lecture. Else, do something else.

If-else statements

The basic syntax. The `if` is followed by a conditional statement in parentheses. If the condition is met, then the code in the braces is executed.

```
x <- TRUE

if (x == TRUE) {
    print("x is true")
}
```

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

If-else statements

In this case we have a vector containing five fruits. We use `sample` to randomly pick one. We can use an if-statement to determine whether we have selected an apple.

```r
fruits <- c("apple", "apple", "orange", "orange", "apple")

f <- sample(fruits, 1)

f <- "orange"

if (f == "apple") {
    print("We selected an apple")
}
```

If-else statements

In the previous example we only have an if-statement. If the condition is not met then nothing happens. Here we add an else statement.

```
fruits <- c("apple", "apple", "orange", "orange", "apple")

f <- sample(fruits, 1)

if (f == "apple") {
    print("We selected an apple")
} else {
    print("We selected an orange")
}
```

```
## [1] "We selected an orange"
```

Note that R can be quite fussy about the syntax. If `else` is on the line below then the function throws an error.

If-else statements

What about this case where we have another fruit? If we only care about apples we could modify the output of our else condition.

```
fruits <- c("apple", "apple", "orange", "orange", "apple", "pineapple")

f <- sample(fruits, 1)

if (f == "apple") {
    print("We selected an apple")
} else {
    print("We selected another fruit.")
}
```

```
## [1] "We selected an apple"
```

If-else statements

We could also use else-if statements to have a separate consideration of all three.

```r
f <- sample(fruits, 1)

if (f == "apple") {
    print("We selected an apple")
} else if (f == "orange") {
    print("We selected an orange")
} else {
    print("We selected a pineapple")
}
```

```
## [1] "We selected an orange"
```

Loops

- Often when we program we need to complete the same operation many times. One of the common approaches is to use a loop.
- There are two kinds of loops you will encounter
    - For-loops
        - Iterative over an entire sequence
    - While-loops
        - Iterate over a sequence while a condition is met

For-loops

Here is a simple example where we use a loop to calculate the sum of a sequence of values.

```r
s <- 0  # value to store our sum

for (i in 1:100) {
    # for i from 1 to 100
    s <- s + i  # add i to sum
}

print(s)
```

```
## [1] 5050
```

For-loops

```
s = 0

for i in range(1,101):
  s += i

print(s)
```

```
## 5050
```

The syntax varies slightly across programming languages but the basic structure is very similar, as this Python example shows. Note that for-loops and other functions in R tend to use braces around the operations. We will see this again when we look at functions.

For-loops

```r
is_orange <- logical(length(fruits))  # a vector of logical objects

i = 1  # In this case we need to maintain a counter
for (f in fruits) {
    if (f == "orange") {
        is_orange[i] <- TRUE
    }
    i <- i + 1  # increment counter by 1
}

print(fruits)
```

```
## [1] "apple"     "apple"     "orange"    "orange"    "apple"     "pin
```

```r
print(is_orange)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE FALSE
```

For-loops

Loops can also easily be *nested*. Here we start a second loop within the first one and use it to populate a matrix.

```
M <- matrix(nrow = 5, ncol = 5)

for (i in 1:5) {
    for (j in 1:5) {
        M[i, j] <- i * j
    }
}

print(M)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
## [4,]    4    8   12   16   20
## [5,]    5   10   15   20   25
```

While-loops

A while loop runs when a condition is true and ends when it becomes false. *Make sure the condition will eventually be false to avoid an infinite loop.*

```r
i <- 1  # iterator
while (i < 5) {
    print(i)
    i <- i + 1
}
```

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

While-loops

In this example we iterate over fruits until we get a pineapple.

```r
i <- 1  # iterator
f <- fruits[i]  # define initial value
while (f != "pineapple") {
    print(f)
    i <- i + 1  # increment index
    f <- fruits[i]  # get next f
}
```

```
## [1] "apple"
## [1] "apple"
## [1] "orange"
## [1] "orange"
## [1] "apple"
```

Functions

- A function is a customized sequence of operations
- We use functions to make our code modular and extendable
- There are thousands of functions built into R and available for packages, but sometimes it is useful to create our own
  - *R4DS* contains the following heuristic:
    - "You should consider writing a function whenever you've copied and pasted a block of code more than twice"

Functions

Here is an example of a simple function that returns the mean of a vector of values x.

```r
avg <- function(x) {
    return(sum(x)/length(x))
}

avg(c(5, 6, 6, 4, 3))
```

```
## [1] 4.8
```

We define the function by using the `function` command and assigning it to the name `avg`. The content in the parentheses is called the `argument` of the function. The `return` statement tells the function what output to produce.

Functions

Here is the same function in Python.

```
def avg(x):
  return(sum(x)/len(x))

avg([5,6,6,4,3])
```

```
## 4.8
```

Again you can see that the syntax is slightly different, for example the `def` command is used to define a function on the left hand side, followed by the name.

Functions

Testing

- It is important to test functions to ensure they work as expected
  - Ensure the function will only process valid inputs
  - It is good practice to handle incorrect inputs
    - There are many ways a function could behave that would not raise errors in R but could still be problematic
  - Write unit tests to ensure the function works as expected
    - Make sure to handle edge cases, inputs that require special handling

Functions

Testing

```
avg(c("a", "b", "c"))
```

```
## Error in sum(x): invalid 'type' (character) of argument
```

Functions

Testing

```
avg(c())
```

```
## [1] NaN
```

Functions

Testing

The function can be modified to return a message if input is incorrect. Note the use of two `return` statements within the function.

```r
avg <- function(values) {
    if (!is.numeric(values)) {
        return("Input must be numeric.")
    } else {
        return(sum(values)/length(values))
    }
}
```

Functions

Testing

```r
# Unit tests
avg(c("a", "b", "c"))
```

```
## [1] "Input must be numeric."
```

```r
avg(c())
```

```
## [1] "Input must be numeric."
```

```r
avg(c(2.6, 2.4))
```

```
## [1] 2.5
```

Functions

Testing

While the return statement did avoid errors, we still have to be careful. The error message we printed could cause downstream errors if not handled. In this case we prevent an error by rejecting invalid inputs. See `help("tryCatch")` for more on error handling.

```
output <- avg(c("a", "b", "c"))
output/2
```

```
## Error in output/2: non-numeric argument to binary operator
```

Pipes

- Pipes are a tool designed to allow you to chain together a sequence of operations
- The pipe is designed to improve the readability of complex chains of function
- Implemented in the `magrittr` package but loaded in `tidyverse`

Pipes

Note how pipes allow us to chain operations from left to right, rather than nesting them from inner to outer. In this case we take a sequence from 1 to 10, get the square root of each value, sum the roots, then print the sum.

```
library(magrittr)

print(sum(sqrt(seq(1:10))))  # nested functions
```

```
## [1] 22.46828
```

```
seq(1:10) %>%
    sqrt() %>%
    sum() %>%
    print()  # using pipes
```

```
## [1] 22.46828
```

Pipes

We can also use pipes to do basic arithmetic using pipes. Note again the difference between the nested operations and the pipe operator.

```
((1 + 2) - 10) * 10
```

```
## [1] -70
```

```
1 %>%
    add(2) %>%
    subtract(10) %>%
    multiply_by(10)
```

```
## [1] -70
```

Note how magrittr provides aliases for certain mathematical operations as shown in the second line. This StackOverflow post has some further discussion.

Pipes

Pipes are particularly useful we're working with tabular data. Here's an example without pipes or nesting. Each line produces an object that is then passed as input to the following line.

```
library(tidyverse)
library(nycflights13)



not_delayed <- filter(flights, !is.na(dep_delay), !is.na(arr_delay))
grouped <- group_by(not_delayed, year, month, day)
summary <- summarize(grouped, mean = mean(dep_delay))
print(summary)
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.4
##  2  2013     1     2 13.7
##  3  2013     1     3 10.9
```

Pipes

In this case the expressions have been nested. This is better as we are not unnecessarily storing intermediate objects.

```
summarize(group_by(
  filter(flights, !is.na(dep_delay), !is.na(arr_delay)),
  year, month, day), mean = mean(dep_delay))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.4
##  2  2013     1     2 13.7
##  3  2013     1     3 10.9
##  4  2013     1     4  8.97
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.56
##  9  2013     1     9  2.30
```

Pipes

Here we have the same expression using a pipe. Note how much easier this is to read.

```
q <- flights %>% filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(year, month, day) %>% summarize(mean = mean(dep_delay))
q
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##      year month   day  mean
##     <int> <int> <int> <dbl>
##   1  2013     1     1 11.4
##   2  2013     1     2 13.7
##   3  2013     1     3 10.9
##   4  2013     1     4  8.97
##   5  2013     1     5  5.73
##   6  2013     1     6  7.15
##   7  2013     1     7  5.42
##   8  2013     1     8  2.56
##   9  2013     1     9  2.30
```

Working with tabular data

- Often we will be using intermediate objects like lists, vectors, and matrices to produce tabular data
  - We can then use tables to create visualizations and conduct statistical analysis
- The `tidyverse` contains an entire suite of functions designed for such tasks including
  - `dplyr` contains functions for basic manipulation and merges and joins
  - `tidyr` implements functions for data cleaning and shape transformations
  - `purr` contains methods to easily map functions to lists and data frames.
- RStudio cheatsheets is an excellent resource.

Next week

- Monday
  - Working with tabular data using the `tidyverse`
  - Visualization using `ggplot2`
- Wednesday
  - Data and file management
  - Github and version control
- Homework 1 released