Programming with data in R

Susan Holmes (c)

Logical statements

Suppose I use the data on births from fivethirtyeight we looked at in the last session. I would like to extract all the children born on Friday the 13th.

Download the "births.csv" data from our folder:
https://stanford.box.com/s/jm9pe4c4ve1kuw2kenlxohogo56ogsve

You can place the data directly in your working directory to make it easier to use. If you are not sure where your current working directory is, you can always check your directory using command getwd().

```
load("births.RData")
head(births)
```

```
##   year month date_of_month day_of_week births
## 1 2000     1             1           6   9083
## 2 2000     1             2           7   8006
## 3 2000     1             3           1  11363
## 4 2000     1             4           2  13032
## 5 2000     1             5           3  12558
## 6 2000     1             6           4  12466
```

```
summary(births)
```

```
##       year          month         date_of_month    day_of_week
##  Min.   :2000   Min.   : 1.000   Min.   : 1.00   Min.   :1
##  1st Qu.:2003   1st Qu.: 4.000   1st Qu.: 8.00   1st Qu.:2
##  Median :2007   Median : 7.000   Median :16.00   Median :4
##  Mean   :2007   Mean   : 6.523   Mean   :15.73   Mean   :4
##  3rd Qu.:2011   3rd Qu.:10.000   3rd Qu.:23.00   3rd Qu.:6
##  Max.   :2014   Max.   :12.000   Max.   :31.00   Max.   :7
##      births
##  Min.   : 5728
##  1st Qu.: 8740
##  Median :12343
##  Mean   :11350
##  3rd Qu.:13082
##  Max.   :16081
```

I can see that the data on day of the week is not a factor but a numeric encoding, in fact Friday is encoded as 5.

```
Fridays=births[which(births[,4]==5),]
```

Which of the Friday births occurred on the 13th?

```
Fridays13=Fridays[which(Fridays[,3]==13),]
head(Fridays13)
```

```
##      year month date_of_month day_of_week births
## 287  2000    10            13           5  11723
## 469  2001     4            13           5  10881
## 560  2001     7            13           5  12187
## 987  2002     9            13           5  13028
## 1078 2002    12            13           5  11600
## 1260 2003     6            13           5  12013
```

```
dim(Fridays13)
```

```
## [1] 25  5
```

```
Weekendbirths=births[which(births[,4]%in%c(6,7)),]
Weekdaybirths=births[which(births[,4]<6),]
```

Valid ways of generating TRUE or FALSE

Sign	Meaning	Example
==	Equals	day_of_week == 5
!=	Does not equal	year != 0

Sign	Meaning	Example
>	Greater than	day_of_week > 5
>=	Greater than or equal	date_of the week >= 6
<	Less than	day_of_week <2
<=	Less than or equal to	day_of_week <=1
%in%	Included in	births[,4]%in%c(6,7)
is.na()	Is a missing value	is.na(births[,4])

Conditions and directions

```
if (condition){
Do something
} else {
Do something different
}
```

```
if (mean(Weekendbirths[,5])> mean(Weekdaybirths[,5]))
{ cat("More weekend babies on average") }else {
  cat("There are less weekend babies on average")
}
```

```
## There are less weekend babies on average
```

Loops and repeats

One may want to repeat a computation for each different element of a vector, sometimes we need to do this with loops and sometimes we avoid this with what is called `vectorization`.

A typical loop:

```
set.seed(431)
mat43=replicate(4,sample(3,3))
mat43l=rep(0,4)
for (j in 1:4)
{mat43l[j]=max(mat43[,j])}
mat43l
```

```
## [1] 3 3 3 3
```

A vectorized version:

```
apply(mat43,2,max)
```

```
## [1] 3 3 3 3
```

Why is vectorization faster, since the number of operations seems always the same?

```
apropos("apply")
```

```
##  [1] ".mapply"    "apply"      "dendrapply" "eapply"     "kernapply"
##  [6] "lapply"     "mapply"     "rapply"     "sapply"     "tapply"
```

```
## [11] "vapply"
```

```
?apply
```

Because R is interpreted it deals with assigning types and memory to variables on the fly. The inner representation of every variable is a vector, it expects to act on vectors even if only one number is involved.

Actual example with larger numbers:

```
mat43=replicate(5000,sample(30000,1000))
dim(mat43)
```

```
## [1] 1000 5000
```

```
system.time(apply(mat43,2,max))
```

```
##    user  system elapsed
##   0.061   0.009   0.070
```

```
mat43l=rep(0,5000)
system.time(for (j in 1:5000)
{mat43l[j]=max(mat43[,j])})
```

```
##    user  system elapsed
##   0.041   0.003   0.044
```

Functions in R

R is called a functional programming language because the actions we take are done using functions, even quitting at the end is done using `q()`.

Example

```
library(readxl)
read_excel
```

```
## function (path, sheet = 1, col_names = TRUE, col_types = NULL,
##     na = "", skip = 0)
## {
##     path <- check_file(path)
##     ext <- tolower(tools::file_ext(path))
##     switch(excel_format(path), xls = read_xls(path, sheet, col_names,
##         col_types, na, skip), xlsx = read_xlsx(path, sheet, col_names,
##         col_types, na, skip))
## }
## <environment: namespace:readxl>
```

Writing our own functions

Useful if you are execute the same set of commands on different data or with different parameters.

```
vec=c(1,2,3,5,7,11,13,17,19,23)
(vec^2)+ 1
```

```
##  [1]   2   5  10  26  50 122 170 290 362 530
```

```
(vec^2)+ 3
```

```
##  [1]   4   7  12  28  52 124 172 292 364 532
```

```
(vec^3)
```

```
##  [1]    1    8   27  125  343 1331 2197 4913 6859 12167
```

Suppose we wanted to take another vector and see which of its elements were divisible by 2,3,5, or other numbers.

We write a function

```
ExpAnd <- function(vec,exponent,addto)
{
  vec^exponent+addto
}
```

We will put it and edit it in the top part of our RStudio IDE and save it as a file that we can also `source` later.

```
ExpAnd <- function(vec,exponent,addto)
{
  out=vec^exponent+addto
  return(out)
}
```

```
ExpAnd <- function(vec,exponent,addto)
{##Function that takes argument vec to the power
  ## exp, adds add and then outputs the result
  out=vec^exponent+addto
  return(out)
}
```

Question What happens if you apply your function with vec = 3, exponent=4, and addto=4 ?

Question What happens if you type:

```
ExpAnd()
```

Put some default values in the function:

```
ExpAnd <- function(vec=seq(4,25,3),exponent=2,addto=3)
{
  # Function that takes argument vec to the power
  #  exp, adds add and then outputs the result
  out<-vec^exponent+addto
  return(out)
}
ExpAnd()
```

```
## [1]  19  52 103 172 259 364 487 628
```

Question: Try calling the function ExpAnd with `ExpAnd(vec=seq(4,25,3),exponent=2,addto="3")`

```
ExpAnd(vec=seq(4,25,3),exponent=2,addto="3")
```

```
ExpAnd <- function(vec=seq(4,25,3),exponent=2,addto=3){
  # Function that takes argument vec to the power
  # exp, adds add and then outputs the result
  if (any(!is.numeric(c(vec,exponent,addto))))
     stop("One of the arguments is not numeric.")
  out <- vec^exponent+addto
  return(out)
}
```

What happens if we now type:

```
ExpAnd(addto="4")
```

Functions are quite robust to some changes in the input:

```
ExpAnd(vec=matrix(c(2,3,4,1,1,2,2,7),ncol=2),3,0)
```

```
##      [,1] [,2]
## [1,]    8    1
## [2,]   27    8
## [3,]   64    8
## [4,]    1  343
```

Note: Passing an unspecified number of parameters to a function

We can pass extra, unspecified arguments to a function by using the `...` notation in the argument list.

```
add20 <- function(x, ...) {
    k <- x+20
    return(k)
}
```

Summary of this Session:

- We have introduced the notion of logical variables that test certain facts.
- We saw how to combine the function `which` with a logical statement to take a subset of the data.
- We can execute blocks of commands encapsulated with `{}` using `if` and `else` with logical conditions.
- We saw that functions are an important component of R programming.

  Their basic elements are a name, an argument and an output that is returned using the `return` function.
- We edit functions in an external file and can call them in later session by using the function `source()`.
- We can test the flow of the function and stop it if something goes awry.

Question: Go to the cheatsheet: Base R

Look at all the functions we have not tried yet and try the examples.