More Advanced Programming in R: Lecture Notes

PS239T

Spring 2018

We've already learned about types in R. Now let's discuss some more advanced programming in R - namely:

1. String operations
2. Flow control
3. Functions
4. Loops
5. Vectorized Functions

Setup environment

```r
# remove all objects
rm(list=ls())

# set working directory
setwd(dir="~/Dropbox/berkeley/Git-Repos/PS239T/04_r-data-analysis/")
```

1. Strings

1.1 Intro to Operations on Strings

```r
firstName <- "Schnitzel"
lastName <- "von Krumm"
```

Unlike in Python, R does not have a reserved operator for string concatenation such as `+`. Furthermore, using the usual concatenation operator `c()` on two or more character strings will not create a single character string, but rather a **vector** of character strings.

```r
fullName <- c(firstName, lastName)
print(fullName)
```

```
## [1] "Schnitzel" "von Krumm"
```

```r
length(fullName)
```

```
## [1] 2
```

In order to combine two or more character strings into one larger character string, we use the `paste()` function. This function takes character strings or vectors and collapses their values into a single character string, with each value separated by a character string selected by the user.

```r
fullName<-paste(firstName, lastName)
print(fullName)
```

```
## [1] "Schnitzel von Krumm"
```

```r
fullName<-paste(firstName, lastName, sep = "+")
print(fullName)
```

```
## [1] "Schnitzel+von Krumm"
```

```
fullName<-paste(firstName, lastName, sep = "___")
print(fullName)
```

```
## [1] "Schnitzel___von Krumm"
```

As with Python, R can also extract substrings based on the index position of its characters. There are, however, two critical differences. First, **index positions in R start at 1**. This is in contrast to Python, where indexation starts at 0.

Second, **object subsets using index positions in R contain all the elements in the specified range**. If some object called `data` contains five elements, `data[2:4]` will return the elements at the second, third, and fourth positions. By contrast, the same subset in Python would return the objects at the third and fourth positions (or second and third positions, depending upon whether your index starts at 0 or 1).

Third, **R does not allow indexing of character strings***. Instead, you must use the `substr()` function. Note that this function must receive both the `start` and `stop` arguments. So if you want to get all the characters between some index and the end of the string, you must make use of the `nchar()` function, which will tell you the length of a character string.

```
fullName<-paste(firstName, lastName)

# this won't work like in Python
fullName[1] # R sees the string as a unitary object - it can't be indexed this way
```

```
## [1] "Schnitzel von Krumm"
```

```
fullName[1:4]
```

```
## [1] "Schnitzel von Krumm" NA                    NA
## [4] NA
```

```
# So use this instead
substr(x = fullName, start = 1, stop = 1)
```

```
## [1] "S"
```

```
substr(x = fullName, start = 5, stop = 5)
```

```
## [1] "i"
```

```
substr(x = fullName, start = 1, stop = 10)
```

```
## [1] "Schnitzel "
```

```
substr(x = fullName, start = 11, stop = nchar(fullName))
```

```
## [1] "von Krumm"
```

1.2 Character String Methods

Like Python, R has a number of string methods, though these exist as individual rather than "mix-and-match" functions. For example:

```
toupper(x = fullName)
```

```
## [1] "SCHNITZEL VON KRUMM"
```

```
tolower(x = fullName)
```

```
## [1] "schnitzel von krumm"
```

```
strsplit(x = fullName, split = " ")
```

```
## [[1]]
## [1] "Schnitzel" "von"       "Krumm"
```

```
strsplit(x = fullName, split = "n")
```

```
## [[1]]
## [1] "Sch"      "itzel vo" " Krumm"
```

```
gsub(pattern = "K", replacement = "C", x = fullName)
```

```
## [1] "Schnitzel von Crumm"
```

```
gsub(pattern = "Schnitzel", replacement = "Mike", x = fullName)
```

```
## [1] "Mike von Krumm"
```

```
gsub(pattern = "von krumm", replacement = "Duderino", x = fullName) # Note the importance of cases! Thi
```

```
## [1] "Schnitzel von Krumm"
```

```
gsub(pattern = " ", replacement = "", x = fullName) # The same function is used for replacements and st
```

```
## [1] "SchnitzelvonKrumm"
```

2. Flow Control

Almost all the conditional operators used in Python also work in R. The basic loop set up is also very similar, with some small syntax adjustments. Note that `if()` is a function whose arguments must be specified inside parentheses. `else`, however, is a reserved operator that takes no arguments. Note that there is no `elif` option — one simply writes `else if()`. Whereas operations to be executed after conditional evaluations in Python come after a `:`, R operations must only be enclosed in curly brackets: `{}`. Furthermore, there is no requirement for indentation. The only thing to keep in mind is that **each new operation must be on a separate line**.

```
x <- 5
if(x<0){
  print("x is negative")
}

x <- -5
if(x<0){
  print("x is negative")
}
```

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

```
x <- 5
if(x<0){
  print("x is negative")
} else{
  print("x is positive")
}
```

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

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

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

R also does some class coercion that makes boolean evaluations harder to break than in Python. But be careful — R has a set of special coercions used for fast logical evaluation and subsetting. Specifically, `TRUE` is considered equal to `1`, while `FALSE` is equal to `0`. The boolean logicals can also be specified as a full word in all caps, or simply as `T` or `F`.

```r
1<2
```

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

```r
"1"<2
```

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

```r
"a"<2
```

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

```r
TRUE<2
```

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

```r
TRUE=="TRUE"
```

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

```r
T=="TRUE"
```

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

```r
TRUE=="T"
```

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

```r
TRUE=="FALSE"
```

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

```r
TRUE==0
```

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

```r
TRUE==1
```

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

```r
FALSE==0
```

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

```r
FALSE<=1
```

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

3. Functions

While functions are defined in Python using the `def` reserved operator, R sees functions as just another type of named object. Thus, they require explicit assignment to an object. This is done using the function `function()`, which creates a function taking the arguments specified in parentheses.

```
simple.function<-function(x){
  print(x+1)
}
simple.function(x = 2)
```

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

```
less.simple.function<-function(x,y){
  print(x-y+1)
}
less.simple.function(x = 2, y = 10)
```

```
## [1] -7
```

With respect to returning function output, most of the same rules apply as with Python. Be sure to remember that `return()` will only process a single object, so multiple items must usually be returned as a list. Note that your ordering of the functions matters, too.

```
dumbfun<-function(x){
  return(x)
  print("This will never print :(")
}
dumbfun(x = "something")
```

```
## [1] "something"
```

```
dumbfun<-function(x){
  print("Why did I print?")
  return(x)
}
dumbfun(x = "something")
```

```
## [1] "Why did I print?"

## [1] "something"
```

```
dumbfun<-function(x,y){
  thing1<-x
  thing2<-y
  return(list(thing1, thing2))
}
dumbfun(x = "some text", y = "some data")
```

```
## [[1]]
## [1] "some text"
##
## [[2]]
## [1] "some data"
```

```
dumbfun(x = c(5,10,15), y = "some data")
```

```
## [[1]]
## [1]  5 10 15
```

```
##
## [[2]]
## [1] "some data"
```

R functions also allow you to set default argument values:

```
less.simple.function<-function(x,y=0){
  print(x-y+1)
}
less.simple.function(x = 2)
```

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

```
less.simple.function(x = 2, y = 10)
```

```
## [1] -7
```

With respect to specifying arguments, one can either use argument **position** specifications (i.e., the order) or argument **name** specifications. The latter is strongly preferred, as it is very easy to accidentally specify incorrect argument values.

```
send<-function(message, recipient, cc=NULL, bcc=NULL){
  print(paste(message, recipient, sep = ", "))
  print(paste("CC:", cc, sep = " "))
  print(paste("BCC:", bcc, sep = " "))
}
send(message = "Hello", recipient = "World", cc = "Rachel", bcc = "Laura")
```

```
## [1] "Hello, World"
## [1] "CC: Rachel"
## [1] "BCC: Laura"
```

```
send("Hello", "World", "Rachel", "Laura")
```

```
## [1] "Hello, World"
## [1] "CC: Rachel"
## [1] "BCC: Laura"
```

```
send("Hello", "Rachel", "Laura", "World")
```

```
## [1] "Hello, Rachel"
## [1] "CC: Laura"
## [1] "BCC: World"
```

```
send(message = "Hello", cc = "Rachel", bcc = c("Laura", "Rochelle"), recipient = "World")
```

```
## [1] "Hello, World"
## [1] "CC: Rachel"
## [1] "BCC: Laura"    "BCC: Rochelle"
```

4. Loops

Loops in R also work basically the same way as in Python, with just a few adjustments. First, recall that index positions in R start at 1. Second, `while()` and `for()` are functions rather than reserved operators, meaning they must take arguments in parentheses. Third, just like `else`, the `in` operator *is* reserved and takes no arguments in parentheses. Fourth, the conditional execution must appear between curly brackets. Finally, indentation is meaningless, but each new operation must appear on a new line.

```r
fruits<-c("apples", "oranges", "pears", "bananas")

# a while loop
i<-1
while(i<=length(fruits)){
  print(fruits[i])
  i<-i+1
}
```

```
## [1] "apples"
## [1] "oranges"
## [1] "pears"
## [1] "bananas"
```

```r
# a for loop
for(i in 1:length(fruits)){
  print(fruits[i])
}
```

```
## [1] "apples"
## [1] "oranges"
## [1] "pears"
## [1] "bananas"
```

Can you improve the for loop above by making the code more concise?

5. Vectorization

5.1 Vectorized functions

Most of R's functions are vectorised, meaning that the function will operate on all elements of a vector without needing to loop through and act on each element one at a time. This makes writing code more concise, easy to read, and less error prone.

```r
x <- 1:4
x * 2
```

```
## [1] 2 4 6 8
```

The multiplication happened to each element of the vector, simultaneously.

We can also add two vectors together:

```r
x <- 1:4
y <- 6:9
x + y
```

```
## [1]  7  9 11 13
```

Each element of x was added to its corresponding element of y:

```
x:  1  2  3  4
    +  +  +  +
y:  6  7  8  9
---------------
    7  9 11 13
```

Most functions also operate element-wise on vectors:

```r
x <- 1:4
log(x)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944
```

To learn more about using and speeding up for loops, check out the answers to this post: http://stackoverflow.com/questions/2908822/speed-up-the-loop-operation-in-r

5.2 Apply Family

Unlike in Python, while and for loops in R can be very slow. For this reason, R has a number of built-in iteration methods to speed up execution times. In many cases, packages will have "behind-the-scenes" ways to avoid for loops, but what if you need to write your own function?

A common method of getting around for loops is the **apply** family of functions. These take a data structure and a function, and applies a function over all the elements in the object.

```r
fruit <-c("apple", "orange", "pear", "banana")

# make function that takes in only one element
make.plural <- function(x){
   plural <- paste(x, 's', sep = '')
   return(plural)
}

make.plural('apple')
```

```
## [1] "apples"
```

```r
# apply that function to every element
lapply(fruit, make.plural) # returns a list
```

```
## [[1]]
## [1] "apples"
##
## [[2]]
## [1] "oranges"
##
## [[3]]
## [1] "pears"
##
## [[4]]
## [1] "bananas"
```

```r
sapply(fruit, make.plural) # returns a named vector
```

```
##     apple    orange      pear    banana
##  "apples" "oranges"   "pears" "bananas"
```

The multivariate version of `sapply` is `mapply`. Use this if you have a function that takes in 2 or more arguments.

5.3 Speeding Up For Loops

So, what can you do if you really do need a for loop? There are two key steps you can take: first, take advantage of R's vectorization capabilities, which tell R ahead of time that all the data the for loop is working on has the same class, making it more efficient. Second, create and store as much of the information outside the loop (i.e., in the global environment, or in the highest level loop if you have nested loops) as you can. Take a look at the example below, which explains each step in more detail.

```
fruit <- c("apple", "orange", "pear", "banana", "plum",
           "peach", "lime", "lemon", "grape", "cherry")
veggie <-c("celery", "beets", "arugula", "carrots", "radish",
            "turnip", "broccoli", "cauliflower", "pea", "cucumber")
produce <- data.frame(fruit, veggie)
# We're going to make this bigger so that we really have something to work with:
lotsofproduce <- do.call("rbind", replicate(1000, produce, simplify = FALSE))

# let's say we want to want to fill in a new column, "matches," with a "yes" whenever the entries for f

# This is some code to time our for loop:
  # start the clock!
ptm <- proc.time()

lotsofproduce$matches <- "No"
for (i in 1:nrow(lotsofproduce)) {
  if(grepl("a", lotsofproduce$fruit[i])==grepl("a", lotsofproduce$veggie[i])){
    lotsofproduce$matches[i] <- "Yes"
  }
}

# stop the clock
proc.time() - ptm
```

```
##    user  system elapsed
##   0.453   0.001   0.454
```

We got the results we wanted (this time - it's not a very big dataset!). But can we improve on our code?

```
# First we could put in the vectors instead of making it loop through a dataframe:
ptm <- proc.time()

lotsofproduce$matches <- "No"
for (i in 1:nrow(lotsofproduce)) {
  if(grepl("a", fruit[i])==grepl("a", veggie[i])){
    lotsofproduce$matches[i] <- "Yes"
  }
}

proc.time() - ptm
```

```
##    user  system elapsed
##   0.317   0.004   0.322
```

That looks like only a little bit of time saved, but it's actually about a quarter of the function time that we cut, and it all adds up when you're looping through thousands of rows!

```
# Then we could move our conditional logic outside the for loop:
ptm <- proc.time()

lotsofproduce$matches <- "No"
cond1 <- as.logical(grepl("a", lotsofproduce$fruit)) # this creates a logical vector with the results o
cond2 <- as.logical(grepl("a", lotsofproduce$veggie))
```

```r
for (i in 1:nrow(lotsofproduce)) {
  if(cond1[i]==cond2[i]){
    lotsofproduce$matches[i] <- "Yes"
  }
}

proc.time() - ptm
```

```
##    user  system elapsed
##   0.130   0.000   0.131
```

This cut our loop by more than half again! Can we do better than that?

```r
# Then we could vectorize matches and change our subsetting to use brackets (which takes less time for
ptm <- proc.time()

matches <- rep("No", nrow(lotsofproduce))
cond1 <- as.logical(grepl("a", lotsofproduce$fruit)) # this creates a logical vector with the results o
cond2 <- as.logical(grepl("a", lotsofproduce$veggie)) # vector for second conditional test
for (i in 1:nrow(lotsofproduce)) {
  if(cond1[i]==cond2[i]){
    matches[i] <- "Yes"
  }
}
lotsofproduce <- cbind(lotsofproduce, matches)

proc.time() - ptm
```

```
##    user  system elapsed
##   0.013   0.000   0.012
```

A fraction of a second! Can't beat that, right?

```r
# In fact, we could do this even faster by not using a loop at all. Instead, we can use an ifelse state
ptm <- proc.time()

cond1 <- as.logical(grepl("a", lotsofproduce$fruit))
cond2 <- as.logical(grepl("a", lotsofproduce$veggie))
lotsofproduce$matches <- ifelse(cond1==cond2,"Yes", "No")

proc.time() - ptm
```

```
##    user  system elapsed
##   0.006   0.000   0.006
```