Lab 2

Irlanda Ayon-Moreno

3/9/2018

Lab 2: Vectors and other data structures

Gaston Sanchez

Learning Objectives

- Work with vectors of different data types
- Understand the concept of *atomic* structures
- Learn how to subset and slice R vectors
- Understand the concept of *vectorization*
- Understand *recycling* rules in R

Getting the Data File

In this lab, you are going to work with a handful of variables about NBA players from the regular season 2016-2017:

- `player`: name of player.
- `team`: team name abbreviation.
- `position`: player position.
- `age`: age of player.
- `experience`: years of experience in NBA.
- `salary`: salary (in dollars).
- `points`: total scored points.
- `points1`: number of free throws, worth 1 point each.
- `points2`: number of 2-point field goals, worth 2 points each.
- `points3`: number of 3-point field goals, worth 3 points each.

The data is in the file `nba2017-salary-points.RData`, located in the `data/` folder of this github repository. The original source of the data is the website www.basketball-reference.com

Open a new session in Rstudio, and make sure you have a clean workspace by typing this command on the console:

```
# remove existing objects
rm(list = ls())
```

You can download the `.RData` file to your working directory, and then `load()` it with the code below. Do NOT include these commands in your source Rmd file; simply type them directly on the console:

```r
# download RData file into your working directory
rdata <- "https://github.com/ucb-stat133/stat133-fall-2017/raw/master/data/nba2017-salary-poin
download.file(url = rdata, destfile = 'nba2017-salary-points.RData')
```

What's happening in the code above?

The function `download.file()` allows you to download any type of file from the Web. In this case you are downloading the file called `nba2017-salary-points.RData` which is located in the github repository of the course. This file is a binary file. To be more precise, the file extension `.RData` is the default extension used by R for its binary native format.

Where does the file get downloaded? Bby default, the file `nba2017-salary-points.RData` gets downloaded to your **working directory**. If you are curious about what is the current directory to which R is paying attention to, simply type the function `getwd()`—which stands for *get the working directory*.

If you want to specify a specific location for the downloaded file, then modify the `destfile` parameter. For instance, if you are using Mac, and you want the file to be downloaded to your desktop, you can use:

```r
# download RData file to your Desktop (assuming you use Mac)
rdata <- "https://github.com/ucb-stat133/stat133-fall-2017/raw/master/data/nba2017-salary-poin
download.file(url = rdata, destfile = '~/Desktop/nba2017-salary-points.RData')
```

Loading the data file

To load or import the contents of the binary file into your R session you use `load()`. This function allows you to import R binary files. This time, include the code below in your `Rmd` file:

```r
# load data in your R session
load('nba2017-salary-points.RData')
```

Note: the code above will only work as long as your `Rmd` file lives in the same directory of the `.RData` file. So far I'm assuming that you have both files in the working directory. If you run into problems, ask the GSI or any of the lab assistants.

Once you imported (or loaded) the data, use the function `ls()` which allows you to **list** all the available R objects:

```r
# list the available objects with ls()
ls()
```

```
## [1] "player"   "points"   "points1"  "points2"  "points3"  "position"
## [7] "salary"   "team"
```

Check that the following objects are available in your session:

- `player`
- `team`
- `position`

- `salary`
- `points`
- etc

About `.RData` files

Most of the data sets you are going to be working with in real life are going to be stored in some sort of text file. So you probably won't be handling many R binary files (i.e. `.RData` files). However, R binary files are an interesting option for saving intermediate results, and/or for saving R objects (as R objects).

Inspecting the data objects

Once you have some data objects to work with, the first step is to inspect some of their characteristics. R has various functions that allow you to examine objects:

- `typeof()` type of storage of any object
- `class()` gives you the class of the object
- `str()` displays the structure of an object in a compact way
- `mode()` gives the data type (as used in R)
- `object.size()` gives an estimate of the memory space used by an object
- `length()` gives the length (i.e. number of elements)
- `head()` take a peek at the first elements
- `tail()` take a peek at the last elements
- `summary()` shows a summary of a given object

Your turn:

- Use `length()`, `head()`, `tail()`, and `summary()` to start exploring the content of the loaded objects.

```r
length(player)
length(points)
length(points1)
length(points2)
length(points3)
length(position)
length(salary)
length(team)
```

```r
head(player)
head(points)
head(position)
head(salary)
head(team)
tail(points)
summary(points)
```

- Do all the objects have the same length?
  They do all have the same length

- Are there missing values, i.e. NA, in any of the objects?
  There seem to be no missing values

- Find out what is the class of each of the objects `player`, `team`, etc.

```
class(player)
class(points)
class(points1)
class(points2)
class(points3)
class(position)
class(salary)
class(team)
```

- How do you know if any of the loaded objects is a vector?
  Use the `is.vector()` function

- How do you know that a given vector is of a certain data type?
  Use the `typeof()` function

Manipulating Vectors: Subsetting

Create a vector `four` by selecting the first four elements in `player`:

```
four <- head(player, n = 4)
```

Single brackets `[ ]` are used to subset (i.e. subscript, split) vectors. Find out what happens if you specify:

- number one: `four[1]`

```
four[1]
```

```
## [1] "Al Horford"
```

- an index of zero: `four[0]`?

```
four[0]
```

```
## character(0)
```

- a negative index: `four[-1]`?

```
four[-1]
```

```
## [1] "Amir Johnson"      "Avery Bradley"     "Demetrius Jackson"
```

- various negative indices: `four[-c(1,2,3)]`?

```
four[-c(1,2,3)]
```

```
## [1] "Demetrius Jackson"
```

- an index greater than the length of the vector: `four[5]`?

```
four[5]
```

```
## [1] NA
```

- repeated indices: `four[c(1,2,2,3,3,3)]`?

```
four[c(1,2,2,3,3,3)]
```

```
## [1] "Al Horford"    "Amir Johnson"  "Amir Johnson"  "Avery Bradley"
## [5] "Avery Bradley" "Avery Bradley"
```

Often, you will need to generate vectors of numeric sequences, like the first five elements `1:5`, or from the first till the last element `1:length(player)`. R provides the colon operator `:`, and the functions `seq()`, and `rep()` to create various types of sequences.

Figure out how to use `seq()`, `rep()` to extract:

- all the even elements in `player`

```
player[seq(from = 2, to = length(player), by =2)]
```

- all the odd elements in `salary`

```
salary[seq(from = 1, to = length(salary), by = 2)]
```

- all multiples of 5 (e.g. 5, 10, 15, etc) of `team`

```
team[seq(from = 5, to = length(team), by = 5)]
```

- elements in positions 10, 20, 30, 40, etc of `points`

```
points[seq(from = 10, to = length(points), by = 10)]
```

- all the even elements in `team` but this time in reverse order

```
rev(team[seq(from = 2, to = length(team), by = 2)])
```

Another kind of subsetting/subscripting is the so-called **logical subsetting**. This type of subsetting implies using a logical vector inside the brackets:

```
four[c(TRUE, TRUE, TRUE, TRUE)]
four[c(TRUE, TRUE, FALSE, FALSE)]
four[c(FALSE, FALSE, TRUE, TRUE)]
four[c(TRUE, FALSE, TRUE, FALSE)]
four[c(FALSE, FALSE, FALSE, FALSE)]

# recycling
four[TRUE]
four[c(TRUE, FALSE)]
```

When subsetting a vector logically, most of the times you won't really be providing an explicit vector of `TRUE`'s and `FALSE`s. Just imagine having a vector of 100 or 1000 or 1000000 elements, and trying to do logical subsetting by manually creating a logical vector of the same length. That would be very boring. Instead, you will be providing a logical condition or a comparison operation that returns a logical vector.

A **comparison operation** occurs when you use comparison operators such as:

- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal
- `==` equal
- `!=` different

```
a <- c(2, 4, 6, 7, 8, 10)

# elements greater than 6
a > 6
```

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

```
# elements different from 6
a != 6
```

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

Notice that a comparison operation always returns a logical vector. Here's how you actually extract the values of a vector based on logical indexing:

```
scored_four <- scored[1:4]

# elements greater than 100
scored_four[scored_four > 100]

# elements less than 100
scored_four[scored_four < 100]

# elements less than or equal to 10
scored_four[scored_four <= 10]

# elements different from 10
scored_four[scored_four != 10]
```

In addition to using comparison operators, you can also use **logical operators** to produce a logical vector. The most common type of logical operators are:

- `&` AND
- `|` OR
- `!` negation

Run the following commands to see what R does:

```
# AND
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

```
FALSE & FALSE
```

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

```
# OR
TRUE | TRUE
```

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

```
TRUE | FALSE
```

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

```
FALSE | FALSE
```

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

```
# NOT
!TRUE
```

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

```
!FALSE
```

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

Logical operators allow you to combine several comparisons:

```
# players of Golden State (GSW)
player[team == 'GSW']

# name of players with salaries greater than 20 million dollars
player[salary > 20000000]

# name of players with scored points between 1000 and 1200 (exclusive)
player[points > 1000 & points < 1200]
```

Your turn

Write commands, using bracket notation, to answer the following questions (you may need to use `min()`, `max()`, `which()`, `which.min()`, `which.max()`):

- players in position Center, of Warriors (GSW)

```
player[position == "C" & team == 'GSW']
```

- players of both GSW (warriors) and LAL (lakers)

```
player[team == "GSW"  | team == "LAL"]
```

- players in positions Shooting Guard and Point Guards, of Lakers (LAL)

```
player[team == 'LAL' & (position == 'SG' | position == 'PG')]
```

- subset Small Forwards of GSW and LAL

```
player[(team == "GSW" | team == "LAL") & position == "SF"]
```

- name of the player with largest salary

```
player[which.max(salary)]
```

- name of the player with smallest salary

```
player[which.min(salary)]
```

- name of the player with largest number of scored points

```
player[which.max(points)]
```

- salary of the player with largest number of points

```
salary[which.max(points)]
```

- largest salary of all Centers

```
max(salary[position == "c"])
```

```
## Warning in max(salary[position == "c"]): no non-missing arguments to max;
## returning -Inf
```

- team of the player with the largest number of scored points

```
team[which.max(points)]
```

- name of the player with the largest number of 3-pointers

```
player[which.max(points3)]
```

Subsetting with Character Vectors

A third type of subsetting involves passing a character vector inside brackets. When you do this, the characters are supposed to be names of the manipulated vector.

None of the vectors `player`, `position`, `scored`, and `salary` have names. You can confirm that with the `names()` function applied on any of the vectors:

```
names(player)
```

```
## NULL
```

- Create a vector `warriors_player` by selecting the players of `team == 'GSW'`

```
warriors_player <- player[team == "GSW"]
```

- then create a vector `warriors_salary` with their salaries, and another vector `warriors_points` with their `points`.

```
warriors_salary <- salary[team == "GSW"]
warriors_points <- points[team == "GSW"]
```

- Assign `warriors_player` as the names of `warriors_salary`.

```
names(warriors_salary) <- warriors_player
```

You should have a vector `warriors_salary` with named elements. Now you can use character subsetting:

```
warriors_salary["Andre Iguodala"]
```

```
## Andre Iguodala
##       11131368
```

```
warriors_salary[c("Stephen Curry", "Kevin Durant")]
```

```
## Stephen Curry  Kevin Durant
##      12112359      26540100
```

Some plotting

Use the function `plot()` to make a scatterplot of `points` and `salary`

```
plot(points, salary)
text(points, salary, labels = player)
```

Keep in mind that `plot()` is a generic function. This means that the behavior of `plot()` depends on the type of input. When you pass two numeric vectors, `plot()` will attempt to create a scatter plot.

Looking at the generated plot, can you see any issues?

To get a better display of the scatterplot, let's create two vectors `log_points` and `log_salary` by transforming `points` and `salary` with the logarithm function `log()`

```
log_points <- log(points)
log_salary <- log(salary)
```

Make another scatterplot but now use the log-transformed vectors:

```
plot(log_points, log_salary)
```

To add the names of the players in the plot, you can use the low-level graphing function `text()`:

```
plot(log_points, log_salary)
text(log_points, log_salary, labels = player)
```

Now we have another problem. The labels in the plot are very messy. A quick and dirty fix is to use `abbreviate()` to shorten the displayed names:

```r
plot(log_points, log_salary)
text(log_points, log_salary, labels = abbreviate(player))
```

Your Turn: create a scatterplot of points and salary for the Warriors (GSW), displaying the names of the players. Generate two scatterplots, one with raw values (original scale, and another plot with log-transformations).

Vectorization

When you create the vectors `log_points <- log(points)` and `log_salary <- log(salary)`, what you're doing is applying a function to a vector, which in turn acts on all elements of the vector.

This is called **Vectorization** in R parlance. Most functions that operate with vectors in R are **vectorized** functions. This means that an action is applied to all elements of the vector without the need to explicitly type commands to traverse all the elements.

In many other programming languages, you would have to use a set of commands to loop over each element of a vector (or list of numbers) to transform them. But not in R.

Another example of vectorization would be the calculation of the square root of all the elements in `points` and `salary`:

```r
sqrt(points)
sqrt(salary)
```

Why should you care about vectorization?

If you are new to programming, learning about R's vectorization will be very natural (you won't stop to think about it too much). If you have some previous programming experience in other languages (e.g. C, python, perl), you know that vectorization does not tend to be a native thing.

Vectorization is essential in R. It saves you from typing many lines of code, and you will exploit vectorization with other useful functions known as the *apply* family functions (we'll talk about them later in the course).

Recycling

Closely related with the concept of *vectorization* we have the notion of **Recycling**. To explain *recycling* let's see an example.

`salary` is given in dollars, but what if you need to obtain the salaries in millions of dollars?. Create a new vector `salary_millions` with the converted values in millions of dollars.

```r
salary_millions <- round(salary / 1000000, 2)
```

Take the values in `points1`, `points2`, and `points3` and figure out to create a new vector `scored_points` that gives you the same values of `points`.

```
scored_points <- points1 + points2 * 2 + points3 * 3
```

What you just did (assuming that you did things correctly) is called **Recycling**. To understand this concept, you need to remember that R does not have a data structure for scalars (single numbers). Scalars are in reality vectors of length 1.

Factors

As mentioned before, vectors are the most essential type of data structure in R. They are *atomic* structures (can contain only one type of data): integers, real numbers, logical values, characters, complex numbers.

Related to vectors, there is another important data structure in R called **factor**. Factors are data structures exclusively designed to handle categorical data.

The object `team` is an R factor. You can confirm this by using `is.factor()` or `class()`

```
is.factor(team)
```

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

Creating Factors

Use `factor()` to create an object `position_fac` by converting `position` into a factor:

```
position_fac <- factor(position)
```

If you have a factor, you can invoke `table()` to get a table with the frequencies (i.e. counts) of the factor categories or *levels*:

```
table(position_fac)
```

```
## position_fac
##  C PF PG SF SG
## 89 89 85 83 95
```

Manipulating Factors

Because factors are internally stored as integers, you can manipulate factors as any other vector:

```
position_fac[1:5]
```

```
## [1] C  PF SG PG SF
## Levels: C PF PG SF SG
```

Practice manipulating `position_fac` to get:

- positions of Warriors

```r
position_fac[team == 'GSW']
```

- positions of players with salaries greater than 15 millions

```r
position_fac[salary > 15000000]
```

- frequencies (counts) of positions with salaries greater than 15 millions

```r
table(position_fac[salary > 15000000])
```

- relative frequencies (proportions) of 'SG' (Shooting Guards) in each team

```r
prop.table(table(team[position_fac == 'SG']))
```

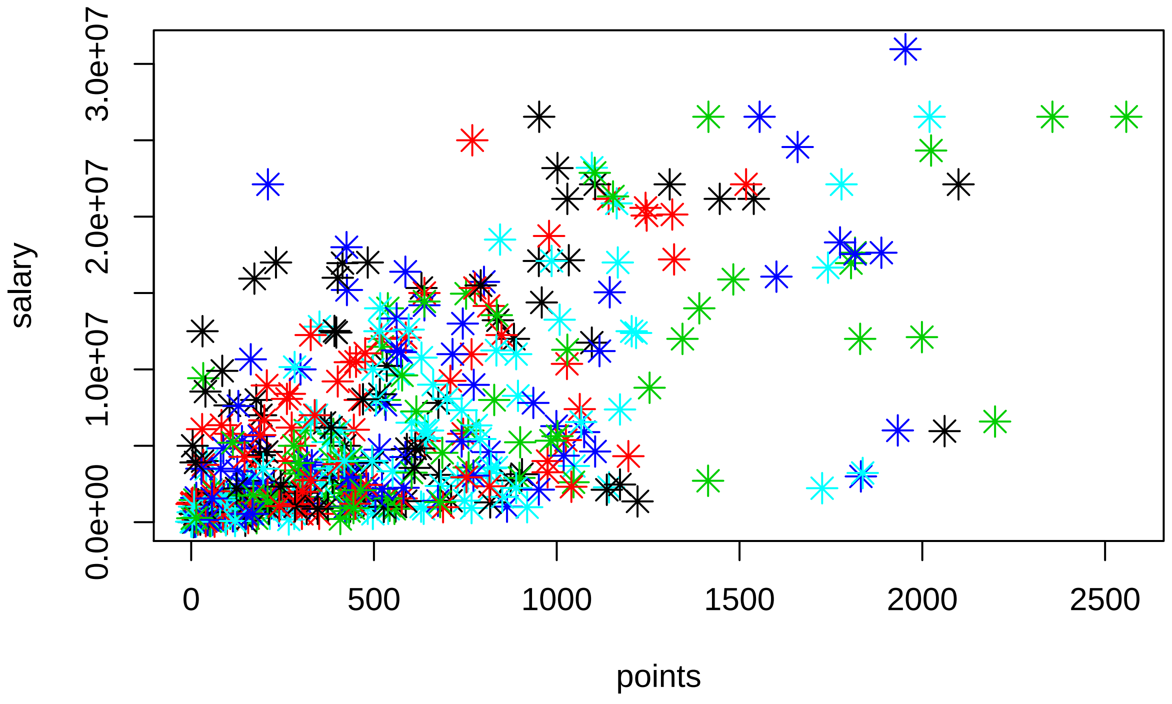
More Plots

Let's go back to the scatterplot of `points` and `salary`

```r
plot(points, salary)
```

- Use your factor `position_fac` to add some color to the dots in the scatterplot.
- Pass the factor to the `col =` parameter inside `plot()`
- Experiment with other `plot()` arguments like the *point character* `pch =`, the *size of dots* with the parameter `cex =`, the axes labels `xlab` and `ylab` and so on.

```r
plot(points, salary, col = position_fac, cex = 1.5, pch = 8)
```


salary
0.0e+00
1.0e+07
2.0e+07
3.0e+07
points