MSc Data Science and Advanced Computing
CSMDM21 - Data Analytics and Mining

Practical 6: R basics

Setup: you may refer to **Installing-R-RStudio.pdf** for downloading and installing R and RStudio on your own computer.

In this practical, we will use a case study data set to walk through some basics of data operations and visualizations.

0. Loading the data set

The following install the **dslabs** package, load the **dslabs** library and load the US **murders** dataset:

```
install.packages("dslabs")
library(dslabs)
data(murders)
```

1. Data Types

Variables in R can be of different types. For example, we need to distinguish numbers from character strings and tables from simple lists of numbers. The function **class** helps us determine what type of object we have.

1.1. Data frames

The most common way of storing a dataset in R is in a data frame. Conceptually, we can think of a data frame as a table with rows representing observations and the different variables reported for each observation defining the columns. Data frames are particularly useful for datasets because we can combine different data types into one object.

To see that the murders data is in fact a **data frame**, we type:

```
class(murders)
#> [1] "data.frame"
```

Examining an object

The function **str** is useful for finding out more about the structure of an object:

```
str(murders)
#> 'data.frame':    51 obs. of  5 variables:
#> $ state : chr "Alabama" "Alaska" "Arizona" "Arkansas" ...
#> $ abb : chr "AL" "AK" "AZ" "AR" ...
#> $ region : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2
#>    2 ...
#> $ population: num 4779736 710231 6392017 2915918 37253956 ...
#> $ total : num 135 19 232 93 1257 ...
```

This tells us much more about the object. We see that the table has 51 rows of data and five variables.

We can show the first six lines using the function **head**:

```
head(murders)
#>          state abb region population total
#> 1    Alabama  AL  South    4779736   135
#> 2     Alaska  AK   West     710231    19
#> 3    Arizona  AZ   West    6392017   232
#> 4   Arkansas  AR  South    2915918    93
#> 5 California  CA   West   37253956  1257
#> 6   Colorado  CO   West    5029196    65
```

The accessor: $

For our analysis, we will need to access the different variables represented by columns included in this data frame. To do this, we use the accessor operator **$** in the following way:

```
murders$population
#>  [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097
#>  [8]   897934   601723 19687653  9920000  1360301  1567582 12830632
#> [15]  6483802  3046355  2853118  4339367  4533372  1328361  5773552
#> [22]  6547629  9883640  5303925  2967297  5988927   989415  1826341
#> [29]  2700551  1316470  8791894  2059179 19378102  9535483   672591
#> [36] 11536504  3751351  3831074 12702379  1052567  4625364   814180
#> [43]  6346105 25145561  2763885   625741  8001024  6724540  1852994
#> [50]  5686986   563626
```

1.2. Vectors: numerics, characters, and logical

The object **murders$population** is not one number but several. We call these types of objects **vectors**. A single number is technically a vector of length 1, but in general we use the term vectors to refer to objects with several entries. The function **length** tells you how many entries are in the vector:

```
pop <- murders$population

length(pop)

#> [1] 51
```

This particular vector is **numeric** since population sizes are numbers:

```
class(pop)

#> [1] "numeric"
```

In a numeric vector, every entry must be a number.

To store character strings, vectors can also be of class **character**. For example, the state names are characters:

```
class(murders$state)

#> [1] "character"
```

As with numeric vectors, all entries in a character vector need to be a character.

Another important type of vectors are **logical** vectors. These must be either **TRUE** or **FALSE**.

```
z <- 3 == 2

z

#> [1] FALSE

class(z)

#> [1] "logical"
```

Here the == is a relational operator asking if 3 is equal to 2. In R, if you just use one =, you actually assign a variable, but if you use two == you test for equality.

1.3. Factors

In the murders dataset, we might expect the region to also be a character vector. However, it is not:

```
class(murders$region)

#> [1] "factor"
```

It is a **factor**. Factors are useful for storing categorical data. We can see that there are only 4 regions by using the **levels** function:

```
levels(murders$region)

#> [1] "Northeast"     "South"         "North Central" "West"
```

Note that the levels have an order that is different from the order of appearance in the factor object. The default in R is for the levels to follow alphabetical order. However, often we want the levels to follow a different order. You can specify an order through the **levels** argument when creating the factor with the factor function. For example, in the murders dataset regions are ordered from east to west. The function **reorder** lets us change the order of the levels of a factor variable based on a summary computed on a numeric vector.

Suppose we want the levels of the region by the total number of murders rather than alphabetical order. If there are values associated with each level, we can use the **reorder** function and specify a data summary to determine the order. The following code takes the sum of the total murders in each region, and reorders the factor following these sums.

```
region <- murders$region
value <- murders$total
region <- reorder(region, value, FUN = sum)
levels(region)
#> [1] "Northeast"     "North Central" "West"          "South"
```

The new order is in agreement with the fact that the Northeast has the least murders and the South has the most.

1.4. Lists

Data frames are a special case of **lists**. Lists are useful because you can store any combination of different types. You can create a list using the **list** function like this:

```
record <- list(name = "John Doe",
             student_id = 1234,
             grades = c(95, 82, 91, 97, 93),
             final_grade = "A")
```

We can create **vectors** using the function **c**, which stands for concatenate.

This list includes a character, a number, a vector with five numbers, and another character.

```
record
#> $name
#> [1] "John Doe"
#>
#> $student_id
#> [1] 1234
#>
#> $grades
#> [1] 95 82 91 97 93
#>
#> $final_grade
#> [1] "A"
class(record)
#> [1] "list"
```

As with data frames, you can extract the components of a list with the accessor **$**.

```
record$student_id
#> [1] 1234
```

We can also use double square brackets **[[** like this:

```
record[["student_id"]]
#> [1] 1234
```

You should get used to the fact that in R, there are often several ways to do the same thing, such as accessing entries.

You might also encounter lists without variable names.

```
record2 <- list("John Doe", 1234)
record2
#> [[1]]
#> [1] "John Doe"
#>
#> [[2]]
#> [1] 1234
```

If a list does not have names, you cannot extract the elements with **$**, but you can still use the brackets method and instead of providing the variable name, you provide the list index:

```
record2[[1]]
#> [1] "John Doe"
```

Exercise 1

1. Install the **dslabs** package, load the **dslabs** library and load the US **murders** dataset if you have not done so.
2. What are the column names used by the data frame?
3. Extract the state abbreviations and assign them to the object **a**. What is the class of this object?
4. With one line of code, use the function levels and length to determine the number of regions defined by this dataset.
5. The function **table** takes a vector and returns the frequency of each element. You can quickly see how many states are in each region by applying this function. Use this function in one line of code to create a table of states per region.

2. Vectors

In R, the most basic objects available to store data are vectors. As we have seen, complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector. Here we learn more about this important class.

2.1. Creating vectors

We can create vectors using the function **c**, which stands for concatenate.

```
codes <- c(380, 124, 818)
codes
#> [1] 380 124 818
```

We can also create character vectors. We use the single/double quotes to denote that the entries are characters rather than variable names.

```
country <- c("italy", "canada", "egypt")
```

is the same as

```
country <- c('italy', 'canada', 'egypt')
```

2.2. Names

Sometimes it is useful to name the entries of a vector. For example, when defining a vector of country codes, we can use the names to connect the two:

```
codes <- c(italy = 380, canada = 124, egypt = 818)
codes
#>  italy canada  egypt
#>    380    124    818
```

The object codes continues to be a numeric vector:

```
class(codes)
#> [1] "numeric"
```

but with names, we can extract the country names:

```
names(codes)
#> [1] "italy"  "canada" "egypt"
```

We can also assign names using the **names** functions:

```
codes <- c(380, 124, 818)
country <- c("italy","canada","egypt")
names(codes) <- country
codes
#>  italy canada  egypt
#>    380    124    818
```

2.3. Sequences

Another useful function for creating vectors generates sequences:

```
seq(1, 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

The first argument defines the start, and the second defines the end which is included. The default is to go up in increments of 1, but a third argument lets us tell it how much to jump:

```
seq(1, 10, 2)
#> [1] 1 3 5 7 9
```

If we want consecutive integers, we can use the following shorthand:

```
1:10
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

When we use these functions, R produces integers, not numerics, because they are typically used to index something:

```
class(1:10)
#> [1] "integer"
```

However, if we create a sequence including non-integers, the class changes:

```
class(seq(1, 10, 0.5))
#> [1] "numeric"
```

2.4. Subsetting

We use square brackets to access specific elements of a vector. For the vector codes we defined above, we can access the second element using:

```
codes[2]
#> canada
#>    124
```

You can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
#> italy egypt
#>   380   818
```

The sequences defined above are particularly useful if we want to access, say, the first two elements:

```
codes[1:2]
#>  italy canada
#>    380    124
```

If the elements have names, we can also access the entries using these names. Below are two examples.

```
codes["canada"]
#> canada
#>    124
codes[c("egypt","italy")]
#> egypt italy
#>   818   380
```

Exercise 2

1. Use the function **c** to create a vector with the average high temperatures in January for some countries, which are 35, 88, 42, 84, 81, and 30 degrees Fahrenheit. Call the object **temp**.
2. Now create a vector with the city names (Beijing, Lagos, Paris, Rio de Janeiro, San Juan, and Toronto) and call the object **city**.
3. Use the **names** function and the objects defined in the previous questions to associate the temperature data with its corresponding city.
4. Use the **[** and **:** operators to access the temperature of the first three cities on the list.
5. Use the **[** operator to access the temperature of Paris and San Juan.

3. Sorting

Now that we have mastered some basic R knowledge, let’s try to gain some insights into the safety of different states in the context of gun murders.

3.1. sort

Say we want to rank the states from least to most gun murders. The function **sort** sorts a vector in increasing order. We can therefore see the largest number of gun murders by typing:

```
sort(murders$total)
#>  [1]    2    4    5    5    7    8   11   12   12   16   19   21   22
#> [14]   27   32   36   38   53   63   65   67   84   93   93   97   97
#> [27]   99  111  116  118  120  135  142  207  219  232  246  250  286
#> [40]  293  310  321  351  364  376  413  457  517  669  805 1257
```

However, this does not give us information about which states have which murder totals. For example, we don’t know which state had 1257.

3.2. order

The function **order** is closer to what we want. It takes a vector as input and returns the vector of indexes that sorts the input vector. This means we can order the state names by their total murders. We first obtain the index that orders the vectors according to murder totals and then index the state names vector:

```
ind <- order(murders$total)
murders$abb[ind]
#>  [1] "VT" "ND" "NH" "WY" "HI" "SD" "ME" "ID" "MT" "RI" "AK" "IA" "UT"
#> [14] "WV" "NE" "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI"
#> [27] "DC" "OK" "KY" "MA" "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC"
#> [40] "MD" "OH" "MO" "LA" "IL" "GA" "MI" "PA" "NY" "FL" "TX" "CA"
```

According to the above, CA (California) had the most murders.

3.3. max and which.max

If we are only interested in the entry with the largest value, we can use **max** for the value:

```
max(murders$total)
#> [1] 1257
```

and **which.max** for the index of the largest value:

```
i_max <- which.max(murders$total)
murders$state[i_max]
#> [1] "California"
```

For the minimum, we can use **min** and **which.min** in the same way.

Exercise 3

1. Use the **$** operator to access the population size data and store it as the object **pop**. Then use the **sort** function to redefine pop so that it is sorted. Finally, use the **[** operator to report the smallest population size.
2. Now instead of the smallest population size, find the index of the entry with the smallest population size. Hint: use **order** instead of **sort**.
3. We can actually perform the same operation as in the previous question using the function **which.min**. Write one line of code that does this.
4. Now we know how small the smallest state is and we know which row represents it. Which state is it? Report the name of the state with the smallest population.

4. Indexing

R provides a powerful and convenient way of indexing vectors. We can, for example, subset a vector based on properties of another vector.

4.1. Subsetting with logicals

We can calculate the murder rate using:

```
murder_rate <- murders$total / murders$population * 100000
```

Imagine you are moving from Italy where, according to an ABC news report, the murder rate is only 0.71 per 100,000. You would prefer to move to a state with a similar murder rate. Another powerful feature of R is that we can use logicals to index vectors. If we compare a vector to a single number, it actually performs the test for each entry. The following is an example related to the question above:

```
ind <- murder_rate < 0.71
```

Note that we get back a logical vector with TRUE for each entry smaller than or equal to 0.71. To see which states these are, we can leverage the fact that vectors can be indexed with logicals.

```
murders$state[ind]
#> [1] "Hawaii"        "Iowa"          "New Hampshire" "North Dakota"
#> [5] "Vermont"
```

In order to count how many are TRUE, the function sum returns the sum of the entries of a vector and logical vectors get coerced to numeric with TRUE coded as 1 and FALSE as 0. Thus we can count the states using:

```
sum(ind)
#> [1] 5
```

4.2. Logical operators

Suppose we like the mountains and we want to move to a safe state in the western region of the country. We want the murder rate to be at most 1. In this case, we want two different things to be true. Here we can use the logical operator **and**, which in R is represented with **&**. This operation results in TRUE only when both logicals are TRUE.

For our example, we can form two logicals:

```
west <- murders$region == "West"
safe <- murder_rate <= 1
```

and we can use **&** to get a vector of logicals that tells us which states satisfy both conditions:

```
ind <- safe & west
murders$state[ind]
#> [1] "Hawaii"  "Idaho"   "Oregon"  "Utah"    "Wyoming"
```

4.3. which

Suppose we want to look up California's murder rate. For this type of operation, it is convenient to convert vectors of logicals into indexes instead of keeping long vectors of logicals. The function **which** tells us which entries of a logical vector are TRUE.

```
ind <- which(murders$state == "California")
murder_rate[ind]
#> [1] 3.37
```

4.4. match

If instead of just one state, we want to find out the murder rates for several states, say New York, Florida, and Texas, we can use the function **match**. This function tells us which indexes of a second vector match each of the entries of a first vector:

```
ind <- match(c("New York", "Florida", "Texas"), murders$state)
ind
#> [1] 33 10 44
```

Now we can look at the murder rates:

```
murder_rate[ind]
#> [1] 2.67 3.40 3.20
```

4.5. %in%

If rather than an index we want a logical that tells us whether or not each element of a first vector is in a second, we can use the function **%in%**. Let's imagine you are not sure if Boston, Dakota, and Washington are states. You can find out like this:

```
c("Boston", "Dakota", "Washington") %in% murders$state
#> [1] FALSE FALSE  TRUE
```

Exercise 4

1. In the example, we have computed the per 100,000 murder rate for each state and stored it in an object called **murder_rate**. Use logical operators to create a logical vector named **low** that tells us which entries of murder_rate are lower than 1.
2. Now use the results from the previous question and the function **which** to determine the indices of murder_rate associated with values lower than 1.
3. Use the results from the previous question to report the names of the states with murder rates lower than 1.
4. Now extend the code from questions 2 and 3 to report the states in the Northeast with murder rates lower than 1. Hint: use the previously defined logical vector **low** and the logical operator **&**.
5. How many states are below the average murder rate?

5. Basic plots

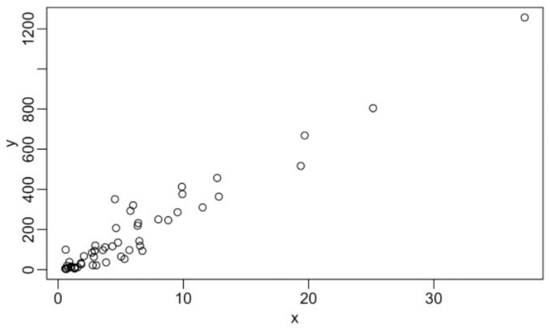
5.1. plot

The plot function can be used to make scatterplots. Here is a plot of total murders versus population.

```
x <- murders$population / 10^6

y <- murders$total

plot(x, y)
```



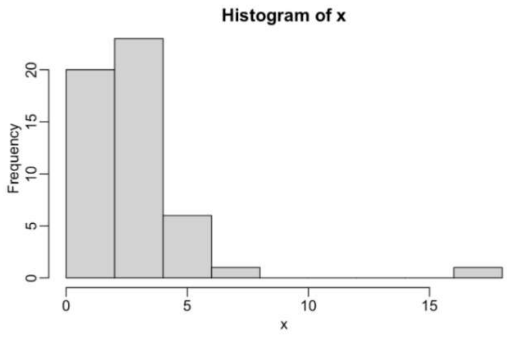
For a quick plot that avoids accessing variables twice, we can use the with function:

```
with(murders, plot(population, total))
```

5.2. hist

Histograms are a powerful graphical summary of a list of numbers that gives you a general overview of the types of values you have. We can make a histogram of our murder rates by:

```
x <- with(murders, total / population * 100000)
hist(x)
```



We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme case with a murder rate of more than 15:

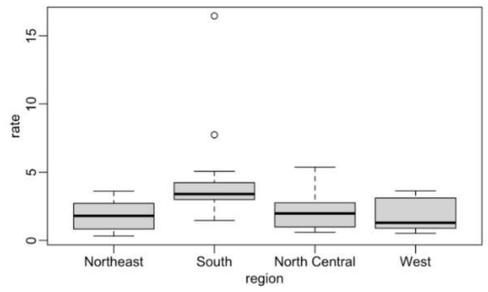
```
murders$state[which.max(x)]

#> [1] "District of Columbia"
```

5.3. boxplot

Boxplots provide a more terse summary than histograms, but they are easier to stack with other boxplots. For example, here we can use them to compare the different regions:

```
murders$rate <- with(murders, total / population * 100000)
boxplot(rate~region, data = murders)
```



Exercise 5

1. Create a histogram of the state populations.
2. Generate boxplots of the state populations by region.