R Fundamentals

301113 Programming for Data Science

WESTERN SYDNEY
UNIVERSITY

School of Computer, Data and Mathematical Sciences

Week 2

Outline

Motivation

Our team has gathered data and needs to compute a set of statistics from it. We have been asked to write an R script to perform the analysis.

To do this, we need to:

- Read data.
- Perform the processing.
- Print the results.

Outline

Using Variables

When we write a script, we are creating an algorithm (a recipe). A set of things (usually numbers) are provided to the script, the algorithm runs and a result is provided.

Variables are used in programs to make the programs more flexible. If we used variables in cooking recipes, then we could provide the number of people we are cooking for and the recipe would change to suit.

There are many types of variables that we can use. The most commonly used are:

- Numeric
- String
- Data Frames
- Lists

To create a variable, we simply write the new variable name and assign a value using <–.

```
frank <- 8
```

Examining Variable Contents

The contents of a variable can be examined while using the console by typing its name.

```
frank
```

```
[1] 8
```

To print out a variables contents while running a script, we need to tell the script to print out the variable.

```
print(frank)
```

```
[1] 8
```

Note that we can also use `print` in the console.

Numeric and String Variables

Numeric variables contain numbers.

```
volume <- 20.3
```

String variables contain a string of characters.

```
food <- "pizza"
```

Note that strings are encased in quotes. If we encase a number in quotes, it becomes a string.

```
volumeString <- "20.3"
```

We can use numeric operations on numeric variables, but not on strings.

```
volume * 2
```

```
[1] 40.6
```

```
volumeString * 2
```

```
Error in volumeString * 2: non-numeric argument to binary operator
```

Computer Memory

Recall that computers have Main Memory (RAM) and Secondary Storage (such as a hard drive). Information can be recorded in both. We can easily explore our hard drive (and examine the files and folders), but it is more difficult to explore the main memory.

The reason for using two types of storage is:

- Reading and writing to RAM is much faster than reading and writing to a hard drive, so for programs to be fast they record their information into the RAM.
- RAM is volatile (if the computer loses power, the information the RAM is lost), but hard drives retain their information without power.

When a program runs, it holds all of its information (variables and program state) in the main memory to allow the program to run faster. It moves information from the main memory to secondary storage only when it needs to preserve information (e.g. when a user saves a file).

Numeric variables Memory

All variables are stored in the main memory, unless we write code to save the variables to the hard drive (into a file).

The type of the variable defines how it is represented in the memory. Computer memory can only store sequences of 0s and 1s (each called a bit), so codes are needed by the computer to convert the 0s and 1s into the numeric and string values.

Each numeric value is stored using 64 bits

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 = 0

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001 = 1

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010 = 2

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000011 = 3

Strings in Memory

We just stated that computer memory can only record 0s and 1s. So to record letters, a code must be used.

ASCII is a common encoding, but is only useful for the Latin alphabet.

The point of identifying how variables are represented in memory is to make clear that the numeric value 1 and the string "1" are different values and types.

Type of variable

To identify the type of a variable we can use the function `class`

```
class(frank)

[1] "numeric"
```

Scientific Notation

To provide a more readable output, R prints large and small numbers in scientific notation.

```
print(134*10^10)
```

```
[1] 1.34e+12
```

We can also assign values to variables using this notation.

```
planckConstant <- 6.626068e-34
```

Preventing Scientific Notation

If you prefer R not to print using scientific notation, add `options(scipen=999)` to the top of your script.

Error messages

When the R interpreter receives a command that it cannot run, it will report an error.

```
a <- "2"
b <- "1"
a + b
```

```
Error in a + b: non-numeric argument to binary operator
```

It is important to read the error messages. They contain information that will help you fix the problem.

The more error messages you see, the easier it will be to correct your scripts.

Conversion between Numeric and String

R provides the functions:

- `as.numeric` to convert a string to a numeric value.
- `as.character` to convert a numeric value to a string.

We can also test is a variable is numeric or a string type:

- `is.numeric` returns TRUE is the variable is numeric.
- `is.character` returns TRUE is the variable is a string.

Variable Names

Variable names must begin with a letter (or a period ".") and be followed by any combination of letters numbers and underscores and periods.

Writing good programs requires that variables have self-explanatory names. For example the variable name `nameOfLecturer` is likely to contain the name of the lecturer, but it is not clear what the variable `x` contains. Having good variable names makes your programs easier to read and maintain.

Variable names cannot contain spaces, so it is common to use one of the following styles:

- Replace spaces with underscores `name_of_lecturer`
- Use camelcase: `nameOfLecturer`

Variable Names

Variable names must begin with a letter (or a period ".") and be followed by any combination of letters numbers and underscores and periods.

Writing good programs requires that variables have self-explanatory names. For example the variable name `nameOfLecturer` is likely to contain the name of the lecturer, but it is not clear what the variable `x` contains. Having good variable names makes your programs easier to read and maintain.

Variable names cannot contain spaces, so it is common to use one of the following styles:

- Replace spaces with underscores `name_of_lecturer`
- Use camelcase: `nameOfLecturer`

Case sensitive names

Note that R is case sensitive, so the variables `Sensitive` and `sensitive` are different variables.

Variable Names

Variable names must begin with a letter (or a period ".") and be followed by any combination of letters numbers and underscores and periods.

Writing good programs requires that variables have self-explanatory names. For example the variable name `nameOfLecturer` is likely to contain the name of the lecturer, but it is not clear what the variable `x` contains. Having good variable names makes your programs easier to read and maintain.

Variable names cannot contain spaces, so it is common to use one of the following styles:

- Replace spaces with underscores `name_of_lecturer`
- Use camelcase: `nameOfLecturer`

Case sensitive names

Note that R is case sensitive, so the variables `Sensitive` and `sensitive` are different variables.

Unique Names

If you create a variable with the same name as another variable or function you will not be able to access the older variable or function.

Predefined Constants

R has a few predefined constants that can be used.

- `LETTERS`
- `letters`
- `pi`
- `month.name`
- `month.abb`

Note that these can be reassigned new values, but when cleared (using `rm`) the constant values remain.

Outline

Comments

It is not always obvious what a block of code will do. Most programming languages provide a way to annotate code, so that we can remind ourselves and others of what the code does and why it is written that way.

In R we write comments (notes) using the hash character # Any text after the hash character will be ignored by the interpreter for that line.

```
## Area of a circle
## Written by Tony Stark

r <- 4 # set the radius of the circle in cm
A <- pi * r^2 # the result has units of cm^2
print(A)
```

Functions

Functions are mechanisms for processing variables. A function has:

- input: the data to be processed
- output: the result of the processing

The input is provided within parentheses () and the output must assigned to a variable.

```
angle <- 4 # angle in degrees
rise <- sin(angle) # call the function sin to compute the sine of angle
print(rise)
```

```
[1] -0.7568025
```

Functions

Functions are mechanisms for processing variables. A function has:

- input: the data to be processed
- output: the result of the processing

The input is provided within parentheses () and the output must assigned to a variable.

```
angle <- 4 # angle in degrees
rise <- sin(angle) # call the function sin to compute the sine of angle
print(rise)
```

```
[1] -0.7568025
```

Problem

The sine of 4 degrees should not be negative! What is wrong with our code?

Function Arguments

The function input is also known as the function arguments. Functions can have multiple arguments, where the first is usually the input to be processed.

Some functions have additional input variables that allow us to change the way the function processes the data. If we don't provide values for these arguments, the default values are used (shown in the help file).

```
log(10, base = 3) # log base 3
```

```
[1] 2.095903
```

```
log(10) # log with the default base.
```

```
[1] 2.302585
```

Function Arguments

The function input is also known as the function arguments. Functions can have multiple arguments, where the first is usually the input to be processed.

Some functions have additional input variables that allow us to change the way the function processes the data. If we don't provide values for these arguments, the default values are used (shown in the help file).

```
log(10, base = 3) # log base 3
```

```
[1] 2.095903
```

```
log(10) # log with the default base.
```

```
[1] 2.302585
```

Problem

What is the default value for the argument `base` in the function `log`?

Mathematical Functions

We have seen the `sin` and `log` functions. Here are a few more commonly used mathematical functions.

For the variable `x`:

- Trigonometric functions: `cos(x)`, `sin(x)`, `tan(x)`,
- Inverse trigonometric functions: `acos(x)`, `asin(x)`, `atan(x)`, `atan2(y, x)`
- Logarithmic functions: `log(x)`, `log10(x)`, `log2(x)`, `exp(x)`
- Absolute value and square root: `abs(x)`, `sqrt(x)`
- Rounding: `ceiling(x)`, `floor(x)`, `trunc(x)`, `round(x, digits = 0)`, `signif(x, digits = 6)`
- More mathematical operations: integer division `x %/% y`, remainder `x %% y`

Using Parentheses

We might want to include multiple operations on one line to make the code more compact.

```
z = 100 * y^3 + 24
```

Note that even though the code is written on one line, R will perform each of the mathematical operations one at a time, so it is important that we know the order.

1. The contents of parentheses are evaluated first
2. Exponentials are evaluated.
3. Multiplcations and divisions.
4. Additions and subtractions.

Note that if there are multiple of any type, they are evaluated from left to right.

If in doubt, use parentheses!

Using Parentheses

We might want to include multiple operations on one line to make the code more compact.

```
z = 100 * y^3 + 24
```

Note that even though the code is written on one line, R will perform each of the mathematical operations one at a time, so it is important that we know the order.

1. The contents of parentheses are evaluated first
2. Exponentials are evaluated.
3. Multiplcations and divisions.
4. Additions and subtractions.

Note that if there are multiple of any type, they are evaluated from left to right.

If in doubt, use parentheses!

Problem

How is the statement `z = 100 * (y^3 + 24)` evaluated?

Outline

Combining Values

We have seen that a variable can hold a numeric or string value. A data scientist usually works with a sample from some population. It would be inconvenient to have to create a variable for each item in a sample.

R provides a function to combine a set of values into a single variable, allowing us to keep the values together, and process them together. The function is used frequently, so the creators of the function have given it a short name `c`.

```
ages <- c(18, 24, 26, 20, 19, 18, 20, 35)
print(ages)
```

```
[1] 18 24 26 20 19 18 20 35
```

The variable `ages` contains a set of numerical values and so is called a vector or one dimensional array.

Combining Combinations

If we combine two vectors, or a vector and a numeric, we obtain a vector.

```
ages <- c(ages, 41) # append 41 to the set of ages
print(ages)
```

```
[1] 18 24 26 20 19 18 20 35 41
```

Therefore, vectors can be created when all of the data is available, or can be gradually updated as the code progresses.

Generating Sequences

R provides the operator `:` to create sequences with separation of size 1.

```
years <- 2010:2020
print(years)
```

```
 [1] 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020
```

There is also the function `seq` for finer control.

```
years <- seq(from = 1980, to = 2020, by = 5)
print(years)
```

```
[1] 1980 1985 1990 1995 2000 2005 2010 2015 2020
```

Computing Statistics

A statistic is a numerical summary of a sample from some population.

The following functions can be used to compute statistics when provided with a numeric vector.

- Measures of location: `mean`, `median`
- Measures of spread: `sd`, `var`, `range`
- Quantiles: `quantiles`, Interquartile range: `IQR` (third quartile - first quartile)
- Multiple statistics: `summary`

Problem

Compute the median of the ages from the previous slide.

Missing Values

There are often times when we obtain a sample that contains missing values (either the questions were not answered in a survey, or the data was corrupted).

R uses the special value `NA` (not available) to represent missing values.

If mathematical operations are performed on missing values, no error is reported; the result is simply reported as missing.

```
NA + 1 # a missing value + 1 is still missing.
```

```
[1] NA
```

```
NA/10 # a missing value divided by 10 is still missing.
```

```
[1] NA
```

Missing Values and Functions

If we have a vector that contains a missing value and attempt to compute a statistic from it:

```
runningTimes <- c(45.2, 54.7, 34.8, NA, 45.5, 35.9)
mean(runningTimes)
```

```
[1] NA
```

For this case, the mean is the sum of the values (which will result in `NA`), divided by the sample size (which is also `NA`).

Many functions have arguments to change the method of dealing with missing values.

Problem

How can we get compute the mean by ignoring the missing values?

Reading User Supplied Values

Data can be read from the console while the program is running using the function `scan`. The help file of `scan` shows that it has many arguments. To read in a vector of numbers, we only need to set the argument `n`.

```
userAges <- scan(n = 5) # get five numbers from the user.
```

Outline

Formatting Output

During the processing of our script or at the end of the script, we might want to print results to the console. We have seen that numbers can be printed, but it would be more informative if a description of the number was also printed.

Two commonly used formatting functions are:

- `paste`: return output as a string
- `cat`: print the output to the screen

Formatting Output

During the processing of our script or at the end of the script, we might want to print results to the console. We have seen that numbers can be printed, but it would be more informative if a description of the number was also printed.

Two commonly used formatting functions are:

- `paste`: return output as a string
- `cat`: print the output to the screen

```
s <- sd(runningTimes, na.rm=TRUE)
cat("The standard deviation of the running times is", s) # print to screen
```

```
The standard deviation of the running times is 8.145367
```

Formatting Output

During the processing of our script or at the end of the script, we might want to print results to the console. We have seen that numbers can be printed, but it would be more informative if a description of the number was also printed.

Two commonly used formatting functions are:

- `paste`: return output as a string
- `cat`: print the output to the screen

```
s <- sd(runningTimes, na.rm=TRUE)
cat("The standard deviation of the running times is", s) # print to screen
```

```
The standard deviation of the running times is 8.145367
```

```
text <- paste("The standard deviation of the running times is", s) # store in variable
print(text)
```

```
[1] "The standard deviation of the running times is 8.14536678118303"
```

Plotting

R provides many functions for plotting data. The most commonly used methods are:

- `plot`: Scatter plot
- `barlpot`: Bar plot
- `hist`: Histogram
- `boxplot`: Box plot

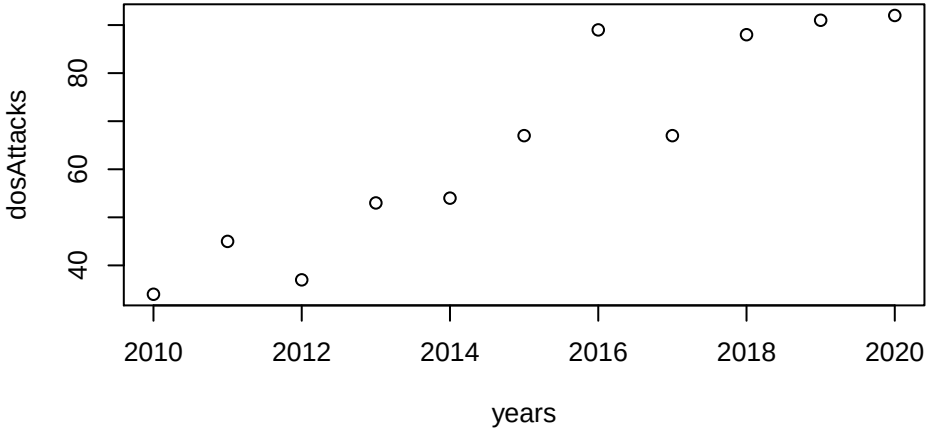
R plots have been used in many textbooks and publications.

The Web site R Graph Gallery is dedicated to showing the many types of plots and code required to obtain the plots.

Scatter Plot

A scatter plot plots the position of a set of points in a Cartesian plane.

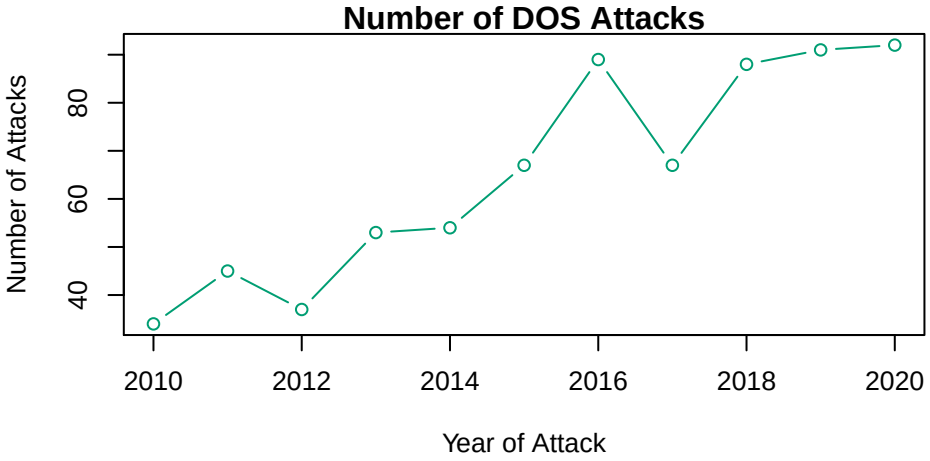
```
years <- 2010:2020
dosAttacks <- c(34, 45, 37, 53, 54, 67, 89, 67, 88, 91, 92)
plot(years, dosAttacks)
```



Scatter Plot with Additions

A scatter plot plots the position of a set of points in a Cartesian plane.

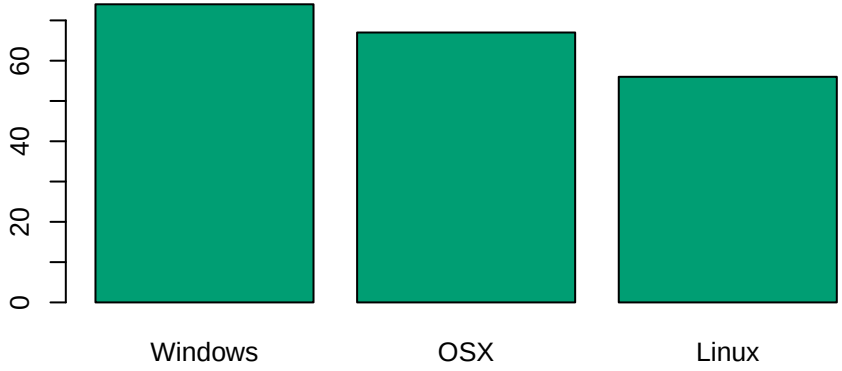
```
plot(years, dosAttacks, xlab = "Year of Attack", ylab = "Number of Attacks",
    main = "Number of DOS Attacks", type = "b", col = 2)
```



Bar Plot

A bar plot plots the frequency of a set of categories.

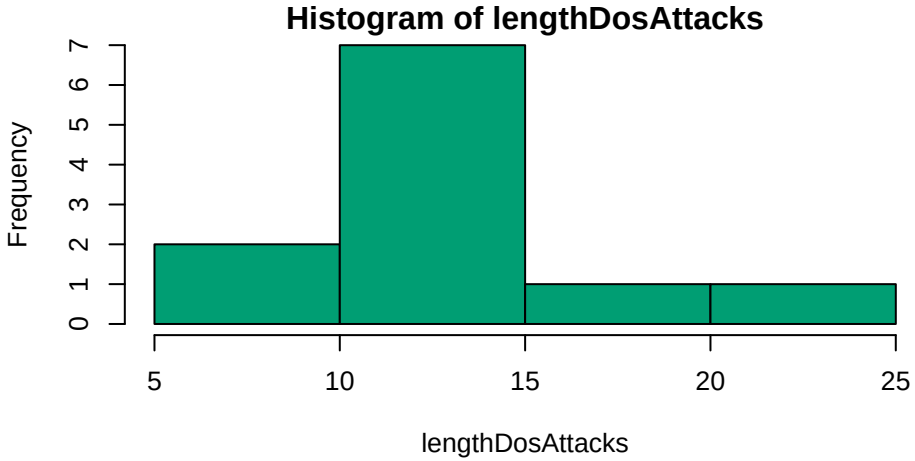
```
os <- c("Windows", "OSX", "Linux")
dosAttacks <- c(74, 67, 56)
barplot(dosAttacks, names.arg = os, col = 2)
```



Histogram

A histogram shows the frequency of a set of numeric values.

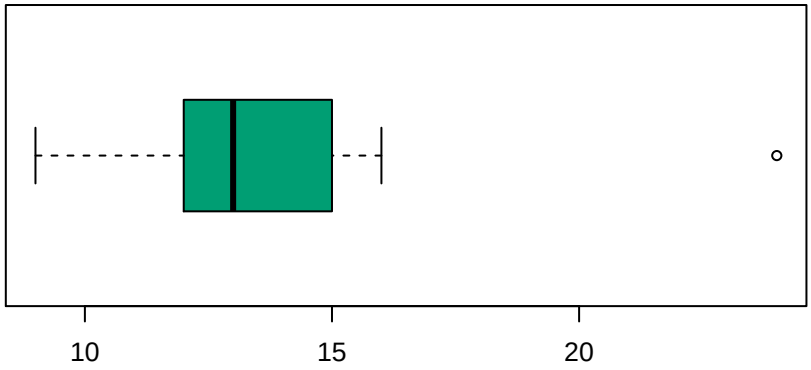
```
lengthDosAttacks <- c(24, 12, 13, 15, 10, 9, 12, 15, 13, 14, 16)
hist(lengthDosAttacks, col = 2)
```



Box plot

A box plot shows the distribution of a set of numeric values.

```
lengthDosAttacks <- c(24, 12, 13, 15, 10, 9, 12, 15, 13, 14, 16)
boxplot(lengthDosAttacks, col = 2, horizontal = TRUE)
```



A Typical Script

When writing code, we need to save it in a script, and we must make sure that the script is organised. Here is a typical script.

```
## Script Name
## Written by: Author <author@address.com>
## Description of script

# First part of the script usually reads and cleans data

# Second part of script usually processes the data

# Third part of the script usually provides output such as statistics or plots.
```

There should also be comments throughout the code to describe what each block of code is doing.

Bringing it all together

Example

Using what we have learnt, let's write a script to:

1. Read 5 values supplied by a user.
2. Compute the mean and standard deviation.
3. Print out the statistics.
4. Plot a histogram of the sample.

Summary

- Data is stored in variables (in the computer RAM) and can have different types (we examined the Numeric and String types).
- Functions take a set of inputs (some have default values) and provide one output.
- There are many predefined functions in R.
- Variables can be combined to form Vectors.
- We can plot the values in vectors.
- Comments are not essential to run the code, but are needed to describe the program to whoever is reading it.