# Data Structures

301113 Programming for Data Science

WESTERN SYDNEY
UNIVERSITY

School of Computer, Data and Mathematical Sciences

Week 4

# Outline

# Organising Data



We have seen that data can be stored in numeric and string variables, and they can be compounded into vectors.

When dealing with many variables, the data should be organised into structures that represent the relationships within the data.

- Matrices are two dimensional data structures.
- Tables have columns as variables and rows as observations.
- Hierarchical data should be organised in a hierarchical fashion (e.g. family tree).

We will examine the data structures available in R, how to create them, insert values into them and manipulate them.

## Note about computer memory

When we create a variable, the computer allocates a place in its memory (RAM) for that variable.

- When a value is assigned to the variable, the value is placed in the assigned placed in the computer memory.
- When we use the variable, the computer retrieves the value from the computer's memory.

# Outline

# Arrays

One of the fundamental variable types in R is an `array`. An array is a collection of data arranged in to rows and columns.

- A 1d array is a vector.
- A 2d array is a matrix.

Higher dimensional arrays can also be created.

# Vectors



We have already created 1d arrays when using the combine function c.

```
v <- c(4,7,6,2)
```

Each element of a vector can be extracted by providing the index using square brackets. The first element of the vector has index 1, the second element has index 2.

```
print(v[1])
```

```
## [1] 4
```

```
print(v[3])
```

```
## [1] 6
```

```
v[2] * v[4]
```

```
## [1] 14
```

# Vector Length



If we try to access an element that is not within the vector, an `NA` is returned.

```
print(v[5])
```

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

We can however provide negative indices.

```
print(v[-1])
```

```
## [1] 7 6 2
```

Providing a negative index provides all elements except for the provided index.

The function `length` returns the number of elements in the provided vector.

```
length(v)
```

```
## [1] 4
```

# Looping over the Index



There are often times where we want to loop over a piece of code, but we want the loop variable to be the index of a vector, rather than the elements.

```
weight <- c(92.5, 67.8, 72.1, 79.1) # measured in kg
height <- c(1.64, 1.72, 1.77, 1.81) # measured in m
bmi <- rep(0, times = length(weight)) # new function!

for (a in 1:length(weight)) {
    bmi[a] <- weight[a]/height[a]^2
}
```

# Looping over the Index



There are often times where we want to loop over a piece of code, but we want the loop variable to be the index of a vector, rather than the elements.

```
weight <- c(92.5, 67.8, 72.1, 79.1) # measured in kg
height <- c(1.64, 1.72, 1.77, 1.81) # measured in m
bmi <- rep(0, times = length(weight)) # new function!

for (a in 1:length(weight)) {
    bmi[a] <- weight[a]/height[a]^2
}
```

Note that the loop sequence ends at `length(weight)`. We could have used the number 4, but it is good programming practice to not include literals in your code, but instead make make your code dependent on the variables that are provided (see Millennium bug).

# Looping over the Index



There are often times where we want to loop over a piece of code, but we want the loop variable to be the index of a vector, rather than the elements.

```
weight <- c(92.5, 67.8, 72.1, 79.1) # measured in kg
height <- c(1.64, 1.72, 1.77, 1.81) # measured in m
bmi <- rep(0, times = length(weight)) # new function!

for (a in 1:length(weight)) {
    bmi[a] <- weight[a]/height[a]^2
}
```

Note that the loop sequence ends at `length(weight)`. We could have used the number `4`, but it is good programming practice to not include literals in your code, but instead make make your code dependent on the variables that are provided (see Millennium bug).

We just introduced the function `rep`, which has the commonly used arguments `times` and `each`.

# Subsetting Vectors



Rather than extract one element of a vector, we can extract a subset of elements by providing a vector of indices.

```
v[c(1,3)]
```

```
## [1] 4 6
```

# Subsetting Vectors



Rather than extract one element of a vector, we can extract a subset of elements by providing a vector of indices.

```
v[c(1,3)]
```

```
## [1] 4 6
```

A negative vector can also be provided to provide all elements except for those that are provided.

```
v[-c(1,3)]
```

```
## [1] 7 2
```

The vector of indices can be a variable.

# Assigning Values



We can extract values from a vector by providing its index. Note that providing an index also exposes the location of the value, allowing us to update it.

Individual values can be updated using assignment.

```
v[2] <- 10
```

We can also update subsets.

```
v[c(2,3)] <- c(11, 11)
```

## Assigning subsets

Make sure that the vector you are assigning to a subset is the same length as the subset being replaced. If they are not the same, a warning message will be produced and R will perform the replacement as best as it can.

# Logical Subsetting



Rather than providing the index number of the elements we want, we can provide a vector of the same size of our target vector, containing logical values (`TRUE` and `FALSE`). The elements that occur at `TRUE` positions are returned.

This allows us to select elements based on a logical statement.

```
print(v[v > 10])
```

```
## [1] 11 11
```

And also replace elements that match the condition.

```
v[v > 10] <- 1
print(v)
```

```
## [1] 4 1 1 2
```

# Vector element positions



When provided with a logical vector, the function `which` returns the position of the `TRUE` values. This is useful to identify the position of elements that pass a logical test.

```
print(v)
```

```
## [1] 4 1 1 2
```

```
pos <- which(v == 1) # which elements of v equal 1.
print(pos)
```

```
## [1] 2 3
```

The found indicies can be used to subset the vector.

```
print(v[pos]) # show the value of the elements at index positions pos.
```

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

# High BMI



## Problem

Body Mass Index (BMI) is the weight in kilograms of a person divided by the square of their height in meters. A BMI greater than 25 is a warning sign for a person to get a health check. The height and weight of four people were recorded as:

```
weight <- c(92.5, 67.8, 72.1, 79.1) # measured in kg
height <- c(1.64, 1.72, 1.77, 1.81) # measured in m
```

Compute the BMI for each person, then provide in the index value for any person with BMI greater than 25.

# Lookup Vectors



If the positions of a vector have a meaning, we can assign names to the positions. Using names rather than numbers will lead to few errors in the code (due to using wrong numbers).

Names can be assigned when the variable is created.

```
ages = c(Garry = 19, Sarah = 20, Kevin = 18, Diana = 15)
print(ages)
```

```
## Garry Sarah Kevin Diana
##    19    20    18    15
```

# Lookup Vectors



If the positions of a vector have a meaning, we can assign names to the positions. Using names rather than numbers will lead to few errors in the code (due to using wrong numbers).

Names can be assigned when the variable is created.

```
ages = c(Garry = 19, Sarah = 20, Kevin = 18, Diana = 15)
print(ages)
```

```
## Garry Sarah Kevin Diana
##    19    20    18    15
```

Or than can be assigned to an existing vector using the `names` function.

```
names(v) <- c("Carrot", "Tomato", "Spinach", "Celery") # assigning to function output?
print(v)
```

```
##  Carrot  Tomato Spinach  Celery
##       4       1       1       2
```

# Accessing Lookup Vectors (Character Subsetting)



Lookup vectors are simply vectors with named elements, so we can extract elements just as we can from a vector.

We can also extract elements by name.

```
ages["Garry"]
```

```
## Garry
##    19
```

and subset.

```
ages[c("Garry", "Diana")]
```

```
## Garry Diana
##    19    15
```

The `names` function can also be used to return the vectors of names from the vector.

# Matrices



A matrix is a 2d array that can be used for matrix arithmetic. Matrices are created using the function `matrix`.

```
A <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

The elements of a matrix can be extracted by providing a row and column index.

```
print(A[2,3])
```

```
## [1] 6
```

# Column and Row Vectors



The columns and rows of a matrix can be extracted by providing the index of the row or column only.

```
print(A[2,]) # extract the second row
```

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

```
print(A[,2]) # extract the second column
```

```
## [1] 3 4
```

# Subsetting Matrices



Portions of matrices can be extracted by subsetting (just as we did with vectors). The element extracted will be those that satisfy both row and columns subsets.

```
print(A[1, c(1,3)])
```

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

If the rows or columns are not provided, then all indices are returned.

```
print(A[,c(2,3)])
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

# Subsetting Matrices



Portions of matrices can be extracted by subsetting (just as we did with vectors). The element extracted will be those that satisfy both row and columns subsets.

```
print(A[1, c(1,3)])
```

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

If the rows or columns are not provided, then all indices are returned.

```
print(A[,c(2,3)])
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

## Dropping Dimensions

The square brackets operator (to extract elements) has the parameter `drop` with default `TRUE`, meaning that the result is of a lower dimension, then it is coerced to that lower dimension type. So the result of subsetting a matrix could be a vector. If this is not wanted, set `drop` to `FALSE`.

# Assigning Values to Matrices



**Problem**

Write one line of R code to replace the 2 and 6 in matrix `A` with the values 7 and 8.
Note that we have not shown how to do this, but we can infer how to do it using the knowledge we have.

# Vector element-wise arithmetic



Vectors of the same length can use the set of arithmetic, relational and logical operations. The operations are applied to the elements from each vector with the same index.

```
x <- c(1,2,3)
y <- c(4,5,6)
print(x + y)
```

```
## [1] 5 7 9
```

```
print(x * y)
```

```
## [1]  4 10 18
```

```
print(x < y)
```

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

# Vector inner and outer products



Vectors of the same length can be used with vector inner and outer products.

```
x %*% y # inner product
```

```
##      [,1]
## [1,]   32
```

```
x %o% y # outer product
```

```
##      [,1] [,2] [,3]
## [1,]    4    5    6
## [2,]    8   10   12
## [3,]   12   15   18
```

# Angle between vectors



## Problem

Recall the cosine of the angle θ between two vectors is given by:

$$\cos(\theta) = \frac{x \cdot y}{||x||\,||y||}$$

where:

- $x \cdot y$ is the inner product between $x$ and $y$ and
- $||x||$ is the norm of $x$ (the square root of the inner product of $x$ with itself).

Compute the cosine of the angle between the vectors $x = [3\ 4\ 6]$ and $y = [2\ -1\ 0]$.

# Matrix element-wise arithmetic



Matrices of the same shape (rows and columns) can use the set of arithmetic, relational and logical operations. The operations are applied to the elements from each matrix with the same index.

```
A <- matrix(c(1,2,3,4,5,6), 2, 3)
B <- matrix(c(4,5,6,7,8,9), 2, 3)
print(A * B)
```

```
##      [,1] [,2] [,3]
## [1,]    4   18   40
## [2,]   10   28   54
```

```
print(A < B)
```

```
##      [,1] [,2] [,3]
## [1,] TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE
```

# Matrix products



Matrix multiplication between two matrices can be performed as long as the number columns of the first matrix is equal to the number of rows of the second matrix.

```
A %*% B # matrix product, dimensions don't match
```

```
## Error in A %*% B: non-conformable arguments
```

The function `t` tranposes a matrix, allow providing suitable matrices in this case.

```
A %*% t(B) # matrix product with transposed B
```

```
##      [,1] [,2]
## [1,]   62   71
## [2,]   80   92
```

# Matrix function



Other commonly used matrix functions.

- `diag`: extract the diagonal of a matrix.
- `det`: determinant of a square matrix.
- `solve`: compute the inverse of a square matrix.
- `eigen`: compute the eigenvalues of a square matrix.
- `svd`: compute the singular value decomposition of a matrix.

# BMI again



## Problem

Recall that Body Mass Index (BMI) is the weight in kilograms of a person divided by the square of their height in meters.
We previously computed the BMI of four people using a for loop.
Compute the BMI of the four people using element-wise arithmetic.

```
weight <- c(92.5, 67.8, 72.1, 79.1) # measured in kg
height <- c(1.64, 1.72, 1.77, 1.81) # measured in m
```

# Arrays



We covered 1d arrays (vectors) and 2d arrays (matrices). Higher dimensional arrays can be created using the function `array`. The function takes the value of each element and the size of each dimension as its arguments. The following is a data cube.

```
Z <- array(c(1,2,3,4,5,6,7,8,9,10,11,12), dim = c(2, 3, 2))
print(Z)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

# Array Elements



Elements of an array can be accessed by providing the correct number of dimensions.

```
print(Z[1,2,1])
```

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

```
print(Z[,,2])
```

```
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Subsetting and insertion is the same as when using vectors of matrices.

# Outline

# Creating Lists



A list is a generic data structure that can contain multiply data types. The elements of a list can be named (but do not need to be).

The following list contains a string, a numeric and a logical variable.

```
student <- list(name = "Gavin", age = 24, needsExam = TRUE)
print(student)
```

```
## $name
## [1] "Gavin"
##
## $age
## [1] 24
##
## $needsExam
## [1] TRUE
```

# Lists of lists



List elements can contain any variable type, and so can contain lists. Lists are very flexible; they can store JSON and YAML structured data.

```
student <- list(name = "Gavin", age = 24, needsExam = TRUE,
    examResults = list(name = "First Exam", mark = 87))
print(student)
```

```
## $name
## [1] "Gavin"
##
## $age
## [1] 24
##
## $needsExam
## [1] TRUE
##
## $examResults
## $examResults$name
## [1] "First Exam"
```

# Accessing Lists



Elements of a list can be accessed using either the index position number, or the name of the variable.

```
print(student[1])
```

```
## $name
## [1] "Gavin"
```

```
print(student["name"])
```

```
## $name
## [1] "Gavin"
```

We can also use the $ notation to access variables.

```
print(student$name)
```

```
## [1] "Gavin"
```

# Double Brackets



There are many functions that manipulate lists, so by default, when we access an element or subset of a list, the results is returned as the element stored in a list.

```
class(student["name"])
```

```
## [1] "list"
```

To obtain the element (rather than the element stored in a list), we must use the double bracket notation.

```
class(student[["name"]])
```

```
## [1] "character"
```

Note that the $ notation is equivalent to the double bracket notation.

# Accessing Nested Lists



Accessing the contents of a nested list requires double brackets (instead of single brackets). Printing the variable provides clues on how to access.

```
student[["examResults"]][["name"]]
```

```
## [1] "First Exam"
```

Or we can use the $ notation.

```
student$examResults$name
```

```
## [1] "First Exam"
```

# Modifying lists



The names and length of the top level of a list is provided using the functions `names` and `length`.

```
names(student)
```

```
## [1] "name"        "age"         "needsExam"   "examResults"
```

The elements and names can be modified just as done with vectors. New elements can be added by creating a new name, or using an unused index.

```
student$cricket = "batsman"
```

# Hierarchical Data



## Problem

Write the code to store the following data structure in a list.

1. Server name: frodo
   - OS: Linux
   - Hack attempts: 2, 5, 10, 15
2. Server name: bilbo
   - OS: Windows
   - Hack attempts: 13, 3, 14, 10

# Outline

# Tabular data



Data often comes in the form of a table, where the rows of the table represent an object and the columns represent a variable that we are measuring from the objects. For example:

- Students are the rows, student grades are the columns.
- Food Products are the rows, nutrition information are the columns.
- Machine learning methods are the rows, results from testing are the columns.
- Houses are the rows, qualities such as size, location, price are the columns.
- Cricket players are the rows, batting and bowling averages are the columns.

We find this data in database tables, spreadsheets and text files (e.g. CSV)

# Creating Data Frames



Data Frames are Rs representation for tabular data. Each of the variables can be a different type (non-compound), but they must be the same for each observation.

Data frames can be created using the function `data.frame`. The columns are provided as vectors. Columns require names, so if we don't provide names, R provides a name.

```
cricket <- data.frame(name = c("Steve Smith", "Pat Cummins"),
    battingAverage = c(61.80, 16.46),
    bowlingAverage = c(56.47, 21.59))
print(cricket)
```

```
##          name battingAverage bowlingAverage
## 1 Steve Smith          61.80          56.47
## 2 Pat Cummins          16.46          21.59
```

# Accessing Data Frames



Elements of a data frame can be accessed:

- just as matrices are accessed (providing the row and column number),
- or as lists are accessed (using single and double bracket notation or $ notation).

They can also be subsetted in the same way.

## Provided Data

There are many famous tabular data sets provided by R. Running the function `data()` provides the list of data. Let's look at the `mtcars` data. The data also has a help file.

# Simple Statistics



Recall that we can compute the mean and standard deviation using the functions `mean` and `sd`. We can compute these statistics of the `mpg` column for the `mtcars` data.

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

```
sd(mtcars$mpg)
```

```
## [1] 6.026948
```

We can examine the count of the automatic and manual transmission cars using `table`.

```
table(mtcars$am)
```

```
##
##  0  1
## 19 13
```

# Aggregating Data



R provides a method of describing relationships called formula notation, that can be used with data frames. An example of this can be shown with the `aggregate` function.

Find the mean miles per gallon with respect to the number of cylinders.

```
aggregate(mpg ~ cyl, mean, data = mtcars)
```

```
##   cyl      mpg
## 1   4 26.66364
## 2   6 19.74286
## 3   8 15.10000
```

The above can be read as “aggregate mpg with respect to cyl, where aggregation is performed using the mean, and the data is taken from mtcars.”

# Comparing Variables



A data frame can be plotted, providing a plot for each variable against each other. The formula notation can be used to plot specific variables.

```
plot(hp ~ wt,  data = mtcars) # numeric vs numeric
boxplot(mpg ~ cyl, data = mtcars) # numeric vs categorical
```

To compare two categorical variables, we can examine the two way table.

```
with(mtcars, table(cyl, gear)) # new function!
```

# Comparing Variables



A data frame can be plotted, providing a plot for each variable against each other. The formula notation can be used to plot specific variables.

```
plot(hp ~ wt,  data = mtcars) # numeric vs numeric
boxplot(mpg ~ cyl, data = mtcars) # numeric vs categorical
```

To compare two categorical variables, we can examine the two way table.

```
with(mtcars, table(cyl, gear)) # new function!
```

**Opening a data environment**

The above function `with` provides the function `table` with access to the variables within its data frame, without having to add `mtcars` to each access.

# Subset function



We stated that data frames can be subsetted using the square bracket and dollar notations. R also provides a function to subset data.

```
subset(mtcars, subset = (mpg > 30))
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

There is also an argument to select which variables to include in the result.

```
subset(mtcars, subset = (mpg > 30), select = c(mpg, cyl, wt))
```

```
##                 mpg cyl    wt
## Fiat 128       32.4   4 2.200
## Honda Civic    30.4   4 1.615
## Toyota Corolla 33.9   4 1.835
## Lotus Europa   30.4   4 1.513
```

# Equivalent Subset



## Problem

Using the square bracket notation, provide the subset that would be returned by `subset(mtcars, subset = (mpg > 30), select = c(mpg, cyl, wt))`

# Named rows and columns



We previously saw that elements in a vector or list can be named. We also stated that columns in a data frame must be named. The names are accessed using the function `names`.

The `mtcars` data also contain row names (the names of the cars). The column and row names can be accessed using the functions `colnames()` and `rownames()`.

Just like the `names` function, we can use these to change the names.

# Sorting and Ordering



Objects that have an order can be sorted or their order can be provided.

```
sort(mtcars$mpg)
```

```
##  [1] 10.4 10.4 13.3 14.3 14.7 15.0 15.2 15.2 15.5 15.8 16.4 17.3 17.8 18.1 18.7
## [16] 19.2 19.2 19.7 21.0 21.0 21.4 21.4 21.5 22.8 22.8 24.4 26.0 27.3 30.4 30.4
## [31] 32.4 33.9
```

```
order(mtcars$mpg)
```

```
##  [1] 15 16 24  7 17 31 14 23 22 29 12 13 11  6  5 10 25 30  1  2  4 32 21  3  9
## [26]  8 27 26 19 28 18 20
```

The order or a variable can used to sort the table by that variable.

```
o = order(mtcars$mpg)
mtcars[o,] ## use o as the indices wanted
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
```

# Factors



A factor is a categorical variable. The categories of the variable are called levels. Encoding variables as factors allows R to handle them appropriately (especially if they have number codes). For example:

```
quality <- c(1,2,2,3,2,2) # numeric
quality <- c("Low","Medium","Medium","High","Medium","Medium") # string
quality <- factor(quality, levels = c("Low", "Medium", "High")) # factor
```

Many functions behave differently depending on the type of data that they receive. E.g. `summary` and `plot`.

# Ordered Factors



Another benefit of using the factors is that they can be ordered.

```
quality <- factor(quality, levels = c("Low", "Medium", "High"), ordered = TRUE)
print(quality)
```

```
## [1] Low    Medium Medium High   Medium Medium
## Levels: Low < Medium < High
```

This allows sorting and ordering of the data.

```
order(quality)
```

```
## [1] 1 2 3 5 6 4
```

# Loading and Saving variables



Recall that when the R session ends, the memory is released back to the OS, and so all variables are lost. R provides the functions `save` and `load` to save and load variables for use in another session.

```
save(quality, cricket, student, file = "cricketAnalysis.Rdata")
```

As many variables as needed can be listed. The variables are stored in the binary file with name provided.

The variables can then be loaded using:

```
load("cricketAnalysis.Rdata")
```

Use `ls()` to check that the variables have loaded.

# Loading and Saving variables



Recall that when the R session ends, the memory is released back to the OS, and so all variables are lost. R provides the functions `save` and `load` to save and load variables for use in another session.

```
save(quality, cricket, student, file = "cricketAnalysis.Rdata")
```

As many variables as needed can be listed. The variables are stored in the binary file with name provided.

The variables can then be loaded using:

```
load("cricketAnalysis.Rdata")
```

Use `ls()` to check that the variables have loaded.

## Working Directory

The data file will be saved in the current working directory. Remember to view or change the working directory using `getwd()` and `setwd()`.

# Summary



- Data can be stored in the vector, matrix, array, list and data frame data structures.
- Each of these data structures can be assigned to a variable, updated and accessed.
- Vector, matrix and array data structures must contain the same variable types for all elements.
- List are very flexible; they can be hierarchical and can contain multiple types of elements.
- Data frames are tabular objects that are most commonly used for data analysis. Each column of a data frame can have a different type.
- Some functions behave differently based on the variable type provided as input.