4. Base R - Matrices and Data Frames

CT5102 - J. Duggan (University of Galway)

Functions

Sometimes data require more complex storage than simple vectors, and thankfully R provides a host of data structures. The most common are the data.frame, matrix and list, followed by the array. — *Jared P. Lander*

Overview

- We have used atomic vectors and lists to store information, which do not provide support for processing *rectangular data*,
- Rectangular data, as the name suggests, is the typical format found in spreadsheets or databases, which comprises rows of data for one or more variable.
- Typically, in a rectangular dataset, every column represents a variable (where each variable can have a different type), and each row contains an observation, for example, a set of values that are related (medical data on a single patient, or weather data at a specific point in time).
- More generally, we can have two types of rectangular (two dimensional) data:
    - Data of a same type, typically numeric, that is stored in a *matrix*
    - data of different types that is stored in a *data frame*.

Learning Outcomes

- How to create a matrix in R, using a one-dimensional vector as input.
- How to subset a matrix, and extend it by adding rows and columns.
- How to create a data frame, and how to subset the data frame using matrix notation.
- How to create a tibble, and understand how a tibble differs from a data frame.
- How to use base R functions `subset()` and `transform()`, which can be used to conveniently process data frames
- How to use the function `apply()` to process matrices and data frame, on either a row or a column basis.
- Key R functions that allow you to manipulate matrices and data frames.
- How to solve all three test exercises.

Matrices

- In R, a matrix is a two-dimensional structure, with rows and columns, that contains the same atomic type.
- A simple way to understand a matrix is that it is essentially an atomic vector in two dimensions, and is created using the `matrix()` function, with the following arguments:
  - `data`, which are the initial values, contained in an atomic vector, supplied to the matrix
  - `nrow`, the desired number of rows
  - `ncol`, the desired number of columns
  - `byrow`, a logical value (default is FALSE), that specifies what way to fill the matrix with data, either filled by row or by column
  - `dimnames`, a list of length 2 giving row and column names respectively.

Matrix Example

```
set.seed(100)
(data <- sample(1:9))
#> [1] 7 6 3 1 2 5 9 4 8
(m1 <- matrix(data, nrow = 3, ncol=3,
              dimnames = list(c("R1","R2","R3"),
                              c("C1","C2","C3"))))
#>    C1 C2 C3
#> R1  7  1  9
#> R2  6  2  4
#> R3  3  5  8
nrow(m1)
#> [1] 3
ncol(m1)
#> [1] 3
dim(m1)
#> [1] 3 3
```

Additional Observations

A number of points that are worth noting:

- The matrix is populated by column as the default. If `by_row` was set to `TRUE`, then the matrix would be populated by row order.
- The row names are column names are set using the `dimnames` argument. This is not required, and row names and column names can always be set on a matrix using the functions `rownames()` and `colnames`.
- The functions `nrow()` and `ncol()` can be used to return the matrix dimensions.
- The function `dim()` provides information on the matrix dimensions, and can be used to resize a matrix, for example, converting a 3x3 to a 1x9. We will explore this resizing option further in the following chapter, when we explore the S3 object system.

Adding a row

- An important property of a matrix is that it can be extended, by adding either rows, using the function `rbind()` or columns, with the function `cbind()`.
- Here is an example of how we can extend the original 3x3 array by adding a row, and then naming that row to be "R4".

```
m1_r <- rbind(m1,c(1,2,3))
rownames(m1_r)[4] <- "R4"
m1_r
#>    C1 C2 C3
#> R1  7  1  9
#> R2  6  2  4
#> R3  3  5  8
#> R4  1  2  3
```

Adding a column

- In a similar way, we can add a new column to the matrix using cbind(), and also name this new column using the function colnames().

```
m1_c <- cbind(m1_r,c(10,20,30,40))
colnames(m1_c)[4] <- "C4"
m1_c
#>    C1 C2 C3 C4
#> R1  7  1  9 10
#> R2  6  2  4 20
#> R3  3  5  8 30
#> R4  1  2  3 40
```

Subsetting

To subset a matrix supply an index for each dimension (row number, column number), and ranges can be applied, similar to subsetting an atomic vector.

```
m1
#>    C1 C2 C3
#> R1  7  1  9
#> R2  6  2  4
#> R3  3  5  8
# Extract the value in row 2, column 2
m1 [2,2]
#> [1] 2
```

```
m1[1:2,1:2]
#>    C1 C2
#> R1  7  1
#> R2  6  2
```

Blank Subsetting

Blank subsetting for one of the dimensions is useful, as it lets you retain all of the rows, or all of the columns. If only one column is subsetted, R will return this as a vector by default. To keep the matrix structure, the additional argument `drop=FALSE` is added.

```
# Extract first row and all columns
m1[1,]
#> C1 C2 C3
#>  7  1  9
# Extract first column, returned as a vector
m1[,1]
#> R1 R2 R3
#>  7  6  3
# Extract all rows and the first column, returned as matrix
m1[,1,drop=FALSE]
#>    C1
#> R1  7
```

Subset by character vector

Overall, the subsetting rules we applied to an atomic vector also apply to a matrix, for example, subsetting by row and column name.

```
# Extract first row and first two columns
m1["R1",c("C1","C2")]
#> C1 C2
#>  7  1
# Extract all rows and first two columns
m1[,c("C1","C2")]
#>    C1 C2
#> R1  7  1
#> R2  6  2
#> R3  3  5
```

Subset by logical vector

Logical vectors can also be used to subset matrices, for example, to extract every second row from the matrix, the following code can be used (as with atomic vectors, the logical vector is recycled).

```
m1
#>    C1 C2 C3
#> R1  7  1  9
#> R2  6  2  4
#> R3  3  5  8
m1[c(T,F),]
#>    C1 C2 C3
#> R1  7  1  9
#> R3  3  5  8
```

The function `is.matrix()`

The function `is.matrix()` is used to test whether the object is a matrix or not. This can be useful as a pre-test, in order to ensure that the data is in the expected format.

```
A <- matrix(1:4,nrow=2)
B <- matrix(1:4,nrow=2,byrow = T)
C <- list(c1=1:2, c2=3:4)
is.matrix(A)
#> [1] TRUE
is.matrix(B)
#> [1] TRUE
is.matrix(C)
#> [1] FALSE
```

Element wise operations

By default, matrix operations such as multiplication, division, exponentiation, addition and subtraction are performed on an element-wise basis. For example, let's explore some simple matrix examples, including addition and multiplication.

```
C <- matrix(1:4,nrow=2)
D <- matrix(1:4,nrow=2,byrow = T)
C
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
D
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
```

Some operations

```
# Multiplication of C and D
C*D
#>      [,1] [,2]
#> [1,]    1    6
#> [2,]    6   16
# Addition of C and D
C+D
#>      [,1] [,2]
#> [1,]    2    5
#> [2,]    5    8
# Multiplying C by a constant
10*C
#>      [,1] [,2]
#> [1,]   10   30
#> [2,]   20   40
```

Matrix multiplication using %*%

```
# Use matrix algebra to multiply two matrices
C
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
D
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
C%*%D
#>      [,1] [,2]
#> [1,]   10   14
#> [2,]   14   20
```

Transpose of a matrix with `t()`

```
t(C)
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
```

`dim()` for matrix dimensions

```r
dim(C)
#> [1] 2 2
```

- The function `dimnames()` returns the row and column names in a list.
- The functions `rownames()` and `colnames()` return the row names and column names as atomic vectors.

diag() for the diagonal

- The function diag() can be used in two ways. First, it can set all the diagonal elements of a matrix. Second, it can be used to create the identity matrix for a given dimension.

```
diag(C) <- -1
C
#>      [,1] [,2]
#> [1,]   -1    3
#> [2,]    2   -1
I <- diag(3)
I
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
```

eigen() for eigenvalues

The function eigen() can be used for calculating the eigenvalues and eigenvectors of a matrix, which has important applications in many computing problems. Here is an example of a 2x2 matrix, where both eigenvalues and eigenvectors are shown.

```
M <- matrix(c(1,-1,-2,3),nrow=2)
eig <- eigen(M)
eig$values
#> [1] 3.7320508 0.2679492
eig$vectors
#>            [,1]       [,2]
#> [1,]  0.5906905 -0.9390708
#> [2,] -0.8068982 -0.3437238
```

The determinant - `det()`.

```
det(M)
#> [1] 1
```

Summary functions

The functions `colSums()`, `colMeans()`, `rowSums()` and `rowMeans()` can be used to conveniently generate summary statistics for a matrix.

```
C
#>      [,1] [,2]
#> [1,]   -1    3
#> [2,]    2   -1
rowSums(C)
#> [1] 2 1
rowMeans(C)
#> [1] 1.0 0.5
colSums(C)
#> [1] 1 2
colMeans(C)
#> [1] 0.5 1.0
```

Data Frames

- A data frame is similar to a matrix, with a two-dimensional row and column structure
- At a technical level, a data frame is a *list*, with the elements of that list containing equal length vectors
- Crucially, the elements (columns) of a data frame can be of different.
- The data frame, with its row and column structure, will be familiar to anyone who has used a spreadsheet, where each column is a variable, and every row is an observation
- The data frame, and its successor, the *tibble*, will be used extensively in part two of the course
- In most cases, a data frame is created from reading in some rectangular data structure from a file or a database, but a data frame can also be created using the function `data.frame()`
- We could setup a simple data frame with five rows and three columns.
- Note that we set the argument `stringsAsFactors` to be `FALSE`, so that the second column are characters, rather than factors.

Data Frames Example

```r
d <- data.frame(Number=1:5,
                Letter=LETTERS[1:5],
                Flag=c(T,F,T,F,NA),
                stringsAsFactors = F)
d
#>   Number Letter  Flag
#> 1      1      A  TRUE
#> 2      2      B FALSE
#> 3      3      C  TRUE
#> 4      4      D FALSE
#> 5      5      E    NA
```

Summarising data frames

When exploring data frames, the `summary()` function is useful, as it provides a useful summary for each variable (or column) in the data frame.

```
summary(d)
#>      Number      Letter              Flag
#>  Min.   :1   Length:5           Mode :logical
#>  1st Qu.:2   Class :character   FALSE:2
#>  Median :3   Mode  :character   TRUE :2
#>  Mean   :3                      NA's :1
#>  3rd Qu.:4
#>  Max.   :5
```

Subsetting data frame

- An important activity that is required once a data frame is available, is to be able to:
    - subset rows
    - subset columns
    - add new columns.
- Because a data frame is a list and also shares properties of a matrix, we can combine subsetting mechanisms from both of these data structures to subset a data frame, for example, to select the first two rows from the data frame.

```
d[1:2,]
#>   Number Letter  Flag
#> 1      1      A  TRUE
#> 2      2      B FALSE
```

Additional subsetting examples

- To select all rows that have `Flag` set to true

```
d[d$Flag == T,]
#>    Number Letter Flag
#> 1       1      A TRUE
#> 3       3      C TRUE
#> NA     NA   <NA>   NA
```

- To select the first two rows and the last two columns

```
d[1:2,c("Letter","Flag")]
#>   Letter  Flag
#> 1      A  TRUE
#> 2      B FALSE
```

Adding a column using $

- To add a new column, we can simply add a new element as we would with a list.

```
d1 <- d
d1$letter <- letters[1:5]
d1
#>   Number Letter  Flag letter
#> 1      1      A  TRUE      a
#> 2      2      B FALSE      b
#> 3      3      C  TRUE      c
#> 4      4      D FALSE      d
#> 5      5      E    NA      e
```

Adding a column using `cbind()`

- To add a new column, we can also use the `cbind()` function

```
d2 <- cbind(d,letter2=letters[6:10])
d2
#>   Number Letter  Flag letter2
#> 1      1      A  TRUE       f
#> 2      2      B FALSE       g
#> 3      3      C  TRUE       h
#> 4      4      D FALSE       i
#> 5      5      E    NA       j
```

The `subset()` function

- While subsetting and adding new elements to data frames can be done using the methods just summarised, there are special purpose functions that make the process easier and more intuitive.
- The function `subset(x, subset,select)` returns subsets of vectors, matrices or data frames which meet specified conditions. The main arguments provided to this function when using with data frames are:
  - `x`, the object to be subsetted
  - `subset`, a logical expression indicating which rows should be kept
  - `select`, which indicates the columns to be selected from the data frame. If this is not present, all columns are returned.

```
subset(mtcars,mpg>32,select=c("mpg","disp"))
#>                 mpg disp
#> Fiat 128       32.4 78.7
#> Toyota Corolla 33.9 71.1
```

Other methods

- Of course, we could achieve the same outcome without using `subset()`, and utilise the matrix-type subsetting for data frames, or by using the $ operator.
- However, even though the same result is achieved, the the `subset()` function is easier to read, and therefore would probably be more useful in terms of code maintenance, and making it easier for others to understand the code.

```
mtcars[mtcars[,"mpg"]>32,c("mpg","disp")]
#>                 mpg disp
#> Fiat 128       32.4 78.7
#> Toyota Corolla 33.9 71.1
mtcars[mtcars$mpg>32,c("mpg","disp")]
#>                 mpg disp
#> Fiat 128       32.4 78.7
#> Toyota Corolla 33.9 71.1
```

Adding variables using `transform()`

A second function that can be used to manipulate data frames is `transform(data, ...)`, which takes in the following arguments.

- `data`, which is the data frame
- `...`, which are additional arguments that capture the details of how the new column is created.

```
df1 <- subset(mtcars,mpg>32,select=c("mpg","disp"))
df1 <- transform(df1,kpg=mpg*1.6)
df1
#>                 mpg disp   kpg
#> Fiat 128       32.4 78.7 51.84
#> Toyota Corolla 33.9 71.1 54.24
```

Adding variables using $

An alternative to using `transform()` is to just use the $ operator as follows.

```
df1 <- subset(mtcars,mpg>32,select=c("mpg","disp"))
df1$kpg <- df1$mpg*1.6
df1
#>                 mpg disp   kpg
#> Fiat 128       32.4 78.7 51.84
#> Toyota Corolla 33.9 71.1 54.24
```

Overall, while the functions `subset()` and `transform()` are useful, and are part of base R, and so always available to use, we will migrate to using new ways to subset and manipulate data frames and tibbles in part two, which are part of the `dplyr` package.

Tibbles

- As we move through the course, we will rely more on tibbles than data frames.
- Tibbles are data frames, however they alter some data frame behaviours to make working with packages in the `tidyverse` a little easier, and have two main difference when compared to the `data.frame` (Wickham 2019)
  - Printing, where tibbles by default only show the first 10 rows, and limit the visible columns to those than fit on the screen. Each column also displays its type, which is a useful feature that provides more information on each variable.
  - Subsetting, where a tibble is always returned, and also partial matching is not supported.
- In order to use a tibble, the tibble package must be loaded `library(tibble)`

Creating a tibble

```
#> Warning: package 'tibble' was built under R version 4.1.2
```

- Similar to a data frame, a function can be used to create a tibble, and this takes a set of atomic vectors.
- Notice that by default string values are not by default converted to factors by this function, and this shows another difference with `data.frame`.

```
d1 <- tibble(Number=1:5,
             Letter=LETTERS[1:5],
             Flag=c(T,F,T,F,NA))
d1
#> # A tibble: 5 x 3
#>   Number Letter Flag
#>    <int> <chr>  <lgl>
#> 1      1 A      TRUE
#> 2      2 B      FALSE
```

Comparison with `data.frame`

We can now explore some of the differences between the `data.frame` and `tibble` by comparing the two variables `d` and `d1` that we have just created. First, we can observe their structure, using `str()`

```
# Show the data frame
str(d)
#> 'data.frame':    5 obs. of  3 variables:
#>  $ Number: int  1 2 3 4 5
#>  $ Letter: chr  "A" "B" "C" "D" ...
#>  $ Flag  : logi  TRUE FALSE TRUE FALSE NA
# Show the tibble
str(d1)
#> tibble [5 x 3] (S3: tbl_df/tbl/data.frame)
#>  $ Number: int [1:5] 1 2 3 4 5
#>  $ Letter: chr [1:5] "A" "B" "C" "D" ...
#>  $ Flag  : logi [1:5] TRUE FALSE TRUE FALSE NA
```

Subsetting examples

Second, we can the see the difference in subsetting one column from each structure. Notice how the data frame output is changed to an atomic vector, which the tibble structure is retained.

```
# Subset the data frame
d[1:2,"Letter"]
#> [1] "A" "B"
# Subset the tibble
d1[1:2,"Letter"]
#> # A tibble: 2 x 1
#>   Letter
#>   <chr>
#> 1 A
#> 2 B
```

Converting between both structures

If required, it is straighforward to convert between both structures, using the function `tibble::as_tibble()` to convert from a `data.frame` to a `tibble`, and the function `as.data.frame()` to convert from a `tibble` to a `data.frame`.

```
str(as_tibble(d))
#> tibble [5 x 3] (S3: tbl_df/tbl/data.frame)
#>  $ Number: int [1:5] 1 2 3 4 5
#>  $ Letter: chr [1:5] "A" "B" "C" "D" ...
#>  $ Flag  : logi [1:5] TRUE FALSE TRUE FALSE NA
str(as.data.frame(d1))
#> 'data.frame':    5 obs. of  3 variables:
#>  $ Number: int  1 2 3 4 5
#>  $ Letter: chr  "A" "B" "C" "D" ...
#>  $ Flag  : logi  TRUE FALSE TRUE FALSE NA
```

Using apply() (functional) on Matrices and Data Frame

The apply(x,margin,f) is a useful functional to iterate over matrices and data frames. It accepts the following arguments x, which can be a matrix or a data frame margin, a number that indicates whether the iteration is by row (margin=1), column (margin=2), and f, which is the function to be applied during each iteration.

```
grades <- sample(30:90,12,replace = T)
results <- matrix(grades,nrow=4)
rownames(results) <- paste0("St-",1:4)
colnames(results) <- paste0("Sub-",1:3)
results
#>       Sub-1 Sub-2 Sub-3
#> St-1     35    63    72
#> St-2     33    36    85
#> St-3     84    36    47
#> St-4     35    84    90
```

Task 1 - maximum grade for each subject

```
results
#>      Sub-1 Sub-2 Sub-3
#> St-1    35    63    72
#> St-2    33    36    85
#> St-3    84    36    47
#> St-4    35    84    90
max_gr_subject <- apply(results,              # the matrix
                        2,                    # 2 for columns
                        max)   # the function to apply
max_gr_subject
#> Sub-1 Sub-2 Sub-3
#>    84    84    90
```

Task 2 - maximum grade for each student

```
results
#>       Sub-1 Sub-2 Sub-3
#> St-1     35    63    72
#> St-2     33    36    85
#> St-3     84    36    47
#> St-4     35    84    90
max_gr_student <- apply(results,                # the matrix
                        1,                      # 1 for rows
                        max)    # the function to apply
max_gr_student
#> St-1 St-2 St-3 St-4
#>   72   85   84   90
```

Using apply() on Data Frames

- The apply() function can also be used on data frames, again, to iterate and apply a function over either each row, of each column. * For example, if we take a subset of the data frame mtcars, and insert some random NA values, we can then count the missing values by either row or column.

```r
set.seed(100)
my_mtcars <- mtcars[sample(1:6),c("mpg","cyl","disp")]
rows <- sample(1:nrow(my_mtcars),5)
my_mtcars[rows[1],1] <- NA
my_mtcars[rows[2],2] <- NA
my_mtcars[rows[3],3] <- NA
my_mtcars[rows[4],1] <- NA
my_mtcars[rows[5],2] <- NA
```

Count the missing values by row

First, to count number of missing values by row, the following code can be used.

```
my_mtcars
#>                    mpg cyl disp
#> Mazda RX4 Wag     21.0   6  160
#> Datsun 710          NA   4  108
#> Mazda RX4         21.0   6   NA
#> Valiant           18.1  NA  225
#> Hornet Sportabout 18.7  NA  360
#> Hornet 4 Drive      NA   6  258
(n_rm <- apply(my_mtcars,1,function(x)sum(is.na(x))))
#>     Mazda RX4 Wag        Datsun 710         Mazda RX4
#>                 0                 1                 1
#> Hornet Sportabout    Hornet 4 Drive
#>                 1                 1
sum(n_rm)
```

Count the missing values by column

Second, to count number of missing values by column, the following code can be used.

```
n_cm <- apply(my_mtcars,2,function(x)sum(is.na(x)))
n_cm
#>  mpg  cyl disp
#>    2    2    1
sum(n_cm)
#> [1] 5
```

Using `lapply()` on Data Frames

- Given that a data frame is also a list, and that `lapply()` processes lists, it also means that the `lapply()` functional can be used to process a data frame.
- So when processing data frames with `lapply()`, the most important thing to remember is that the data frame will be processed column by column.

Code Example

```
s1 <- mtcars |>
        subset(select=c("mpg","cyl","disp")) |>
        lapply(function(x)mean(x))
s1
#> $mpg
#> [1] 20.09062
#>
#> $cyl
#> [1] 6.1875
#>
#> $disp
#> [1] 230.7219
```

Mini-Case 1: Modelling Social Networks using Matrices

- The first mini-case is a fascinating application area for matrices is the study of networks, which impact many areas, including science, technology, business and nature Barabasi(2016). Here, we model a social media network.
  - We generate an adjacency matrix A which allows us to keep track of the links between N users.
  - The matrix A has N rows and N columns (i.e. it is a square matrix), where $A_{ij} = 1$ if $person_i$ follows $person_j$, and $A_{ij} = 0$ if $person_i$ does not follow $person_j$.
  - The first step is to create 49 numbers, with the arbitrary probability that 40% of these have the value 1.
  - A square matrix is then created, and the rows and columns named `P-1` through to `P-10`.
  - To complete the logic of the adjacency matrix, all the diagonal elements are set to zero using the R function `diag()`, as a person cannot follow themselves.

Setup the adjacency matrix

```
# Set the number of users
N = 7
# Set the seed to ensure the matrix can be reproduced
set.seed(100)
# Sample, with replacement, 100 numbers, with 20% of a 1
v <- sample(0:1,N^2,replace = T,prob = c(0.6,.4))
# Create the matrix of 10x10
m <- matrix(v,nrow=N)

# Set the row and column names
rownames(m) <- paste0("P-",1:N)
colnames(m) <- rownames(m)

# Set the diagonal to zero, and display
diag(m) <- 0
m
#>     P-1 P-2 P-3 P-4 P-5 P-6 P-7
```

Display the adjacency matrix

```
m
#>     P-1 P-2 P-3 P-4 P-5 P-6 P-7
#> P-1   0   0   1   1   0   1   1
#> P-2   0   0   1   0   0   0   1
#> P-3   0   0   0   1   0   1   1
#> P-4   0   1   0   0   1   1   0
#> P-5   0   1   0   0   0   0   1
#> P-6   0   0   1   1   1   0   1
#> P-7   1   0   0   1   1   1   0
```

Visualise the adjacency matrix

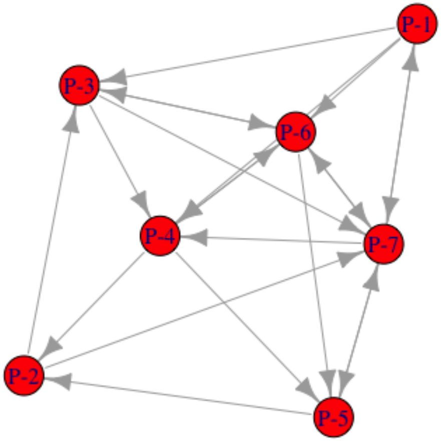
Using the CRAN package `igraph`, we can easily visualise the adjacency matrix as a network graph.

```
# Include the igraph library
suppressPackageStartupMessages(library(igraph))

# Create the igraph data structure from m and display
g1 <- graph_from_adjacency_matrix(m)
g1
#> IGRAPH 7dff4f1 DN-- 7 22 --
#> + attr: name (v/c)
#> + edges from 7dff4f1 (vertex names):
#>  [1] P-1->P-3 P-1->P-4 P-1->P-6 P-1->P-7 P-2->P-3 P-2->P-7
#>  [9] P-3->P-7 P-4->P-2 P-4->P-5 P-4->P-6 P-5->P-2 P-5->P-7
#> [17] P-6->P-5 P-6->P-7 P-7->P-1 P-7->P-4 P-7->P-5 P-7->P-6

# plot(g1,vertex.size=20,vertex.color="red")
```

Output from igraph



Processing the matrix - sum rows

Show the number of people a person follows.

```
m
#>     P-1 P-2 P-3 P-4 P-5 P-6 P-7
#> P-1   0   0   1   1   0   1   1
#> P-2   0   0   1   0   0   0   1
#> P-3   0   0   0   1   0   1   1
#> P-4   0   1   0   0   1   1   0
#> P-5   0   1   0   0   0   0   1
#> P-6   0   0   1   1   1   0   1
#> P-7   1   0   0   1   1   1   0
row_sum <- apply(m,1,sum)
row_sum
#> P-1 P-2 P-3 P-4 P-5 P-6 P-7
#>   4   2   3   3   2   4   4
names(which(row_sum==max(row_sum)))
#> [1] "P-1" "P-6" "P-7"
```

Processing the matrix - sum columns

Show the number of followers for a person

```
m
#>     P-1 P-2 P-3 P-4 P-5 P-6 P-7
#> P-1   0   0   1   1   0   1   1
#> P-2   0   0   1   0   0   0   1
#> P-3   0   0   0   1   0   1   1
#> P-4   0   1   0   0   1   1   0
#> P-5   0   1   0   0   0   0   1
#> P-6   0   0   1   1   1   0   1
#> P-7   1   0   0   1   1   1   0
col_sum <- apply(m,2,sum)
col_sum
#> P-1 P-2 P-3 P-4 P-5 P-6 P-7
#>   1   2   3   4   3   4   5
names(which(col_sum==max(col_sum)))
#> [1] "P-7"
```

Mini-Case 2: A Pipeline for Processing data frames

- In this example, for the data frame `mtcars`, the following processing actions will be taken:
  - Two columns from `mtcars` will be selected, `mpg` and `disp`.
  - A new column `kpg` will be added, which converts `mpg` to kilometers per gallon, using the multiplier 1.6
  - A new column `dm_ratio` will be added, which is the ratio of `disp` and `mpg`.
  - The first six observations will then be shown
- The solution will use the R pipe operator, which chains the sequence of data processing stages together.
- The functions `subset()` and `transform()` will also be used, the first to select a subset of columns, the second to add new columns to the data frame.

Solution

```
mtcars_1 <- mtcars |>
            subset(select=c("mpg","disp")) |>
            transform(kpg=mpg*1.6,
                      dm_ratio=disp/mpg) |>
            head()
mtcars_1
#>                    mpg disp   kpg  dm_ratio
#> Mazda RX4         21.0  160 33.60  7.619048
#> Mazda RX4 Wag     21.0  160 33.60  7.619048
#> Datsun 710        22.8  108 36.48  4.736842
#> Hornet 4 Drive    21.4  258 34.24 12.056075
#> Hornet Sportabout 18.7  360 29.92 19.251337
#> Valiant           18.1  225 28.96 12.430939
```

Useful R Functions (1/3)

R Function	Description
`as.data.frame()`	Can be used to convert a tibble to a data frame
`apply()`	Provides a mechanism to iterate over rectangular data, either by row or by column.
`cbind()`	Adds a new vector as a column to a matrix
`colnames()`	Can be used to set the column names, or to see the current column names of a matrix
`colMeans()`	Calculates the mean of each column in a matrix
`colSums()`	Calculates the sum of each column in a matrix
`data.frame()`	Constructs a data frame
`diag()`	Can be used to set the diagonal of an existing matrix, or to generate a square identity matrix
`dim()`	Returns the matrix dimensions (row followed by column) as an atomic vector. Can also be used to reset the dimensions of a matrix.

Useful R Functions (2/3)

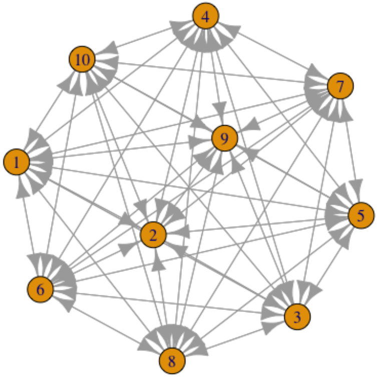
R Function	Description
`dimnames()`	Returns, as a list, the row and column names of a matrix
`eigen()`	Calculates the eigenvalues and eigenvectors of a matrix
`is.matrix()`	Returns `TRUE` if the object is a matrix
`matrix()`	Creaes a matrix from the given set of arguments
`rbind()`	Adds a new vector as a row to a matrix
`rownames()`	Can be used to set the row names, or to see the current row names of a matrix
`t()`	Returns the transpose of a matrix
`rowMeans()`	Calculates the mean of each row in a matrix
`rowSums()`	Calculates the sum of each row in a matrix
`subset()`	Provides a mechanism to iterate over rectangular data, either by row or by column.

Useful R Functions (3/3)

R Function	Description
`tibble::tibble()`	Constructs a tibble
`tibble::as_tibble()`	Converts a data frame to a tibble
`transform()`	Can be used as a way to add columns to a data frame.

Exercise 1 - Create a network

Generate the following network using matrices and `igraph`. All nodes are connected to eachother (i.e. it is a fully connected network).



Exercise 2

Use the subset() function to generate the following tibbles from the tibble ggplot2::mpg. Use the R pipe operator (|>) where necessary.

```
# The car with the maximum displacement,
# with a subset of features
max_displ
#> # A tibble: 1 x 6
#>   manufacturer model      year displ   cty class
#>   <chr>        <chr>     <int> <dbl> <int> <chr>
#> 1 chevrolet    corvette   2008     7    15 2seater
```

Exercise 3

```
# The first six audi cars, with selected columns
# Also, implement the solution without the subset() function
audi_6
#> # A tibble: 6 x 5
#>    year manufacturer model displ   cty
#>   <int> <chr>        <chr> <dbl> <int>
#> 1  1999 audi         a4      1.8    18
#> 2  1999 audi         a4      1.8    21
#> 3  2008 audi         a4      2      20
#> 4  2008 audi         a4      2      21
#> 5  1999 audi         a4      2.8    16
#> 6  1999 audi         a4      2.8    18
```

Lecture Summary

- How to create a matrix in R, using a one-dimensional vector as input.
- How to subset a matrix, and extend it by adding rows and columns.
- How to create a data frame, and how to subset the data frame using matrix notation.
- How to create a tibble, and understand how a tibble differs from a data frame.
- How to use base R functions `subset()` and `transform()`, which can be used to conveniently process data frames
- How to use the function `apply()` to process matrices and data frame, on either a row or a column basis.
- Key R functions that allow you to manipulate matrices and data frames.
- How to solve all three test exercises.