Introduction to R

(tutorial can be found at: https://kateto.net/networks-r-igraph)

Assignment

You can assign a value to an object using `assign()`, `<-`, or `=`.

```
x <- 3         # Assignment
x              # Evaluate the expression and print result
y <- 4         # Assignment
y + 5          # Evaluation, y remains 4
z <- x + 17*y  # Assignment
z              # Evaluation
rm(z)          # Remove z: deletes the object.
z              # Error!
```

Value comparison

We can use the standard operators `<`, `>`, `<=`, `>=`, `==`(equality) and `!=` (inequality). Comparisons return Boolean values: `TRUE` or `FALSE` (often abbreviated to just `T` and `F`).

```
2==2  # Equality
2!=2  # Inequality
x <= y # less than or equal: "<", ">", and ">=" also work
```

Special constants

Special constants include:

- **NA** for missing or undefined data
- **NULL** for empty object (e.g. null/empty lists)
- **Inf** and **-Inf** for positive and negative infinity
- **NaN** for results that cannot be reasonably defined

```
# NA - missing or undefined data

5 + NA      # When used in an expression, the result is generally NA
is.na(5+NA) # Check if missing

# NULL - an empty object, e.g. a null/empty list
10 + NULL     # use returns an empty object (length zero)
is.null(NULL) # check if NULL
```

Inf and -Inf represent positive and negative infinity. They can be returned by mathematical operations like division of a number by zero:

```
5/0
is.finite(5/0) # Check if a number is finite (it is not).
```

NaN (Not a Number) - the result of an operation that cannot be reasonably defined, such as dividing zero by zero.

```
0/0
is.nan(0/0)
```

Vectors

Vectors can be constructed by combining their elements with the important R function `c()`.

```
v1 <- c(1, 5, 11, 33)       # Numeric vector, length 4
v2 <- c("hello","world")    # Character vector, length 2 (a vector of
strings)
v3 <- c(TRUE, TRUE, FALSE)  # Logical vector, same as c(T, T, F)
```

Combining different types of elements in one vector will coerce the elements to the least restrictive type:

```
v4 <- c(v1,v2,v3,"boo")     # All elements turn into strings
```

Other ways to create vectors include:

```
v <- 1:7         # same as c(1,2,3,4,5,6,7)
v <- rep(0, 77)  # repeat zero 77 times: v is a vector of 77 zeroes
v <- rep(1:3, times=2) # Repeat 1,2,3 twice
v <- rep(1:10, each=2) # Repeat each element twice
v <- seq(10,20,2) # sequence: numbers between 10 and 20, in jumps of 2
v1 <- 1:5         # 1,2,3,4,5
v2 <- rep(1,5)    # 1,1,1,1,1
```

Check the length of a vector:

```
length(v1)
length(v2)
```

Element-wise operations:

```
v1 + v2      # Element-wise addition
v1 + 1       # Add 1 to each element
v1 * 2       # Multiply each element by 2
v1 + c(1,7)  # This doesn't work: (1,7) is a vector of different length
```

Mathematical operations:

```
sum(v1)      # The sum of all elements
mean(v1)     # The average of all elements
sd(v1)       # The standard deviation
cor(v1,v1*5) # Correlation between v1 and v1*5
```

Logical operations:

```
v1 > 2       # Each element is compared to 2, returns logical vector
v1==v2       # Are corresponding elements equivalent, returns logical
vector.
v1!=v2       # Are corresponding elements *not* equivalent? Same as
!(v1==v2)
(v1>2) | (v2>0)   # | is the boolean OR, returns a vector.
(v1>2) & (v2>0)   # & is the boolean AND, returns a vector.
```

```
(v1>2) || (v2>0)  # || is the boolean OR, returns a single value
(v1>2) && (v2>0)  # && is the boolean AND, ditto
```

Vector elements:

```
v1[3]             # third element of v1
v1[2:4]           # elements 2, 3, 4 of v1
v1[c(1,3)]        # elements 1 and 3 - note that your indexes are a vector
v1[c(T,T,F,F,F)]  # elements 1 and 2 - only the ones that are TRUE
v1[v1>3]          # v1>3 is a logical vector TRUE for elements >3
```

Note that the indexing in R starts from `1`, a fact known to confuse and upset people used to languages that index from `0`.

To add more elements to a vector, simply assign them values.

```
v1[6:10] <- 6:10
```

We can also directly assign the vector a length:

```
length(v1) <- 15 # the last 5 elements are added as missing data: NA
```

Factors

Factors are used to store categorical data.

```
eye.col.v <- c("brown", "green", "brown", "blue", "blue", "blue")
#vector
eye.col.f <- factor(c("brown", "green", "brown", "blue", "blue", "blue"))
#factor
eye.col.v
## [1] "brown" "green" "brown" "blue"  "blue"  "blue"
eye.col.f
## [1] brown green brown blue  blue  blue
## Levels: blue brown green
```

R will identify the different levels of the factor - e.g. all distinct values. The data is stored internally as integers - each number corresponding to a factor level.

```
levels(eye.col.f)  # The levels (distinct values) of the factor
(categorical var)
## [1] "blue"  "brown" "green"
as.numeric(eye.col.f)  # As numeric values: 1 is  blue, 2 is brown, 3 is
green
## [1] 2 3 2 1 1 1
as.numeric(eye.col.v)  # The character vector can not be coerced to numeric
## Warning: NAs introduced by coercion
## [1] NA NA NA NA NA NA
as.character(eye.col.f)
## [1] "brown" "green" "brown" "blue"  "blue"  "blue"
as.character(eye.col.v)
## [1] "brown" "green" "brown" "blue"  "blue"  "blue"
```

Matrices and arrays

A matrix is a vector with dimensions:

```
m <- rep(1, 20)   # A vector of 20 elements, all 1
dim(m) <- c(5,4)  # Dimensions set to 5 & 4, so m is now a 5x4 matrix
```

Creating a matrix using `matrix()`:

```
m <- matrix(data=1, nrow=5, ncol=4)  # same matrix as above, 5x4, full of
1s
m <- matrix(1,5,4)                       # same matrix as above
dim(m)                                # What are the dimensions of m?
## [1] 5 4
```

Creating a matrix by combining vectors:

```
m <- cbind(1:5, 5:1, 5:9)  # Bind 3 vectors as columns, 5x3 matrix
m <- rbind(1:5, 5:1, 5:9)  # Bind 3 vectors as rows, 3x5 matrix
```

Selecting matrix elements:

```
m <- matrix(1:10,10,10)
m[2,3]  # Matrix m, row 2, column 3 - a single cell
m[2,]   # The whole second row of m as a vector
m[,2]   # The whole second column of m as a vector
m[1:2,4:6] # submatrix: rows 1 and 2, columns 4, 5 and 6
m[-1,]     # all rows *except* the first one
```

Other operations with matrices:

```
# Are elements in row 1 equivalent to corresponding elements from column 1:
m[1,]==m[,1]
# A logical matrix: TRUE for m elements >3, FALSE otherwise:
m>3
# Selects only TRUE elements - that is ones greater than 3:
m[m>3]
t(m)          # Transpose m
m <- t(m)     # Assign m the transposed m
m %*% t(m)    # %*% does matrix multiplication
m * m         # * does element-wise multiplication
```

Arrays are used when we have more than 2 dimensions. We can create them using the `array()` function:

```
a <- array(data=1:18,dim=c(3,3,2)) # 3d with dimensions 3x3x2
a <- array(1:18,c(3,3,2))          # the same array
```

Lists

Lists are collections of objects. A single list can contain all kinds of elements - character strings, numeric vectors, matrices, other lists, and so on. The elements of lists are often named for easier access.

```
l1 <- list(boo=v1,foo=v2,moo=v3,zoo="Animals!")  # A list with four
components
l2 <- list(v1,v2,v3,"Animals!")
```

Create an empty list:

```
l3 <- list()
l4 <- NULL
```

Accessing list elements:

```
l1["boo"]   # Access boo with single brackets: this returns a list.
l1[["boo"]] # Access boo with double brackets: this returns the numeric
vector

l1[[1]]     # Returns the first component of the list, equivalent to above.

l1$boo      # Named elements can be accessed with the $ operator, as with
[[]]
```

Adding more elements to a list:

```
l3[[1]] <- 11 # add an element to the empty list l3
l4[[3]] <- c(22, 23) # add a vector as element 3 in the empty list l4.
```

Since we added element 3 to the list `l4`above, elements 1 and 2 will be generated and empty (NULL).

```
l1[[5]] <- "More elements!" # The list l1 had 4 elements, we're adding a
5th here.
l1[[8]] <- 1:11
```

We added an 8th element, but not 6th and 7th to the list`l1` above. Elements number 6 and 7 will be created empty (NULL).

```
l1$Something <- "A thing"  # Adds a ninth element - "A thing", named
"Something"
```

Data frames

The data frame is a special kind of list used for storing dataset tables. Think of rows as cases, columns as variables. Each column is a vector or factor.

Creating a dataframe:

```
dfr1 <- data.frame( ID=1:4,
                    FirstName=c("John","Jim","Jane","Jill"),
                    Female=c(F,F,T,T),
                    Age=c(22,33,44,55) )
dfr1$FirstName   # Access the second column of dfr1.
## [1] John Jim  Jane Jill
## Levels: Jane Jill Jim John
```

Notice that R thinks that `dfr1$FirstName` is a categorical variable and so it's treating it like a factor, not a character vector. Let's get rid of the factor by telling R to treat 'FirstName' as a vector:

```
dfr1$FirstName <- as.vector(dfr1$FirstName)
```

Alternatively, you can tell R you don't like factors from the start using `stringsAsFactors=FALSE`

```
dfr2 <- data.frame(FirstName=c("John","Jim","Jane","Jill"),
stringsAsFactors=F)

dfr2$FirstName   # Success: not a factor.
## [1] "John" "Jim"  "Jane" "Jill"
```

Access elements of the data frame:

```
dfr1[1,]   # First row, all columns
dfr1[,1]   # First column, all rows
dfr1$Age   # Age column, all rows
dfr1[1:2,3:4] # Rows 1 and 2, columns 3 and 4 - the gender and age of John
& Jim
dfr1[c(1,3),] # Rows 1 and 3, all columns
```

Find the names of everyone over the age of 30 in the data:

```
dfr1[dfr1$Age>30,2]
## [1] "Jim"  "Jane" "Jill"
```

Find the average age of all females in the data:

```
mean ( dfr1[dfr1$Female==TRUE,4] )
## [1] 49.5
```

Flow control and loops

The controls and loops in R are fairly straightforward (see below). They determine if a block of code will be executed, and how many times. Blocks of code in R are enclosed in curly brackets `{}`.

```
# if (condition) expr1 else expr2
x <- 5; y <- 10
if (x==0) y <- 0 else y <- y/x #

y
## [1] 2
# for (variable in sequence) expr

ASum <- 0; AProd <- 1



for (i in 1:x)
{
  ASum <- ASum + i
  AProd <- AProd * i
}

ASum  # equivalent to sum(1:x)
## [1] 15
AProd # equivalemt to prod(1:x)
## [1] 120
```

```
# while (condintion) expr

while (x > 0) {print(x); x <- x-1;}

# repeat expr, use break to exit the loop

repeat { print(x); x <- x+1; if (x>10) break}
```

Functions in R

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows −

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

Function Components

The different parts of a function are −

- **Function Name** − This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body** − The function body contains a collection of statements that defines what the function does.
- **Return Value** − The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

Built-in Function

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs.

```
# Create a sequence of numbers from 32 to 44.
```

```
print(seq(32,44))

# Find mean of numbers from 25 to 82.
print(mean(25:82))

# Find sum of numbers frm 41 to 68.
print(sum(41:68))
```

When we execute the above code, it produces the following result −

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
   for(i in 1:a) {
      b <- i^2
      print(b)
   }
}
```

Calling a Function

```
# Call the function new.function supplying 6 as an argument.
new.function(6)
```

When we execute the above code, it produces the following result −

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
# Create a function without an argument.
new.function <- function() {
   for(i in 1:5) {
      print(i^2)
   }
}

# Call the function without supplying an argument.
new.function()
```

When we execute the above code, it produces the following result −

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.
new.function <- function(a,b,c) {
   result <- a * b + c
   print(result)
}

# Call the function by position of arguments.
new.function(5,3,11)

# Call the function by names of the arguments.
new.function(a = 11, b = 5, c = 3)
```

When we execute the above code, it produces the following result −

```
[1] 26
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.
new.function <- function(a = 3, b = 6) {
   result <- a * b
   print(result)
}

# Call the function without giving any argument.
new.function()

# Call the function with giving new values of the argument.
new.function(9,5)
```

When we execute the above code, it produces the following result −

```
[1] 18
[1] 45
```

Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
new.function <- function(a, b) {
   print(a^2)
   print(a)
   print(b)
}

# Evaluate the function without supplying one of the arguments.
new.function(6)
```

When we execute the above code, it produces the following result −

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```

Homework

1. Write a function which takes a matrix A as an argument and determines the sum of the matrix columns.

2. Write a function which takes a matrix A as an argument and determines if it is symmetric.

3. Write a function which determines the matrix product A^k for a given positive number k and a square matrix A. Provide a solution with complexity in the order of $O(logk)$ with respect to the number of matrix multiplications.