Session-2

Typical Programing Errors

Typical Programing Errors

- Syntactical (spelling mistake)
  - Will get caught very easily! Just run the program.
    - `a = c("hi"; "there")`

Typical Programing Errors

- Syntactical (spelling mistake)
  - Will get caught very easily! Just run the program.
    - `a = c("hi"; "there")`

- Semantic Errors (meaningless operations)
    - For e.g. `"iimb" + 32`
  - Exceptions: like divide by 0.
  - May get caught. A warning will be thrown nonetheless.

Typical Programing Errors

- Syntactical (spelling mistake)
  - Will get caught very easily! Just run the program.
    - `a = c("hi"; "there")`
- Semantic Errors (meaningless operations)
    - For e.g. `"iimb" + 32`
  - Exceptions: like divide by 0.
  - May get caught. A warning will be thrown nonetheless.
- Logical Errors (Unintentional)
  - Program will crash, run forever or give a wrong answer!
    - Like using `i` in place of `j`, adding in place of multiplying, …
    - These will happen when you first learn programming.
    - You'll get better with skill and experience.

Getting Help in R

Getting Help in R

- From Console
  - Just type: ? followed by function name without parenthesis
  - E.g. `?mean; ?sum; ?length;`
  - Clarify:
    - **?mean** - help for the function “mean”
    - **??mean** - will perform the search over the internet (CRAN database)
      - Look for base::mean!
    - **mean()** - call the function mean
    - **mean** - print the definition of the function “mean”

Getting Help in R

- From Console
  - Just type: ? followed by function name without parenthesis
  - E.g. `?mean; ?sum; ?length;`
  - Clarify:
    - **?mean** - help for the function “mean”
    - **??mean** - will perform the search over the internet (CRAN database)
      - Look for base::mean!
    - **mean()** - call the function mean
    - **mean** - print the definition of the function “mean”

- From Web sources
  - Most reliable and easy to incorporate is www.stackoverflow.com.
  - www.r-bloggers.com is also quite helpful.
  - You can use https://cran.r-project.org for any resource on R
    - Read the package vignette and manuals
    - e.g. data.table CRAN; data.table vignette; data.table manual
  - Even typing your question in google will get you good results!
    - 99% of your questions are already answered! You just need to find them!

R Input and Output

R Input and Output

- Simple assignment
  - `X = 1;` (*or* `X <- 1;`)
  - Assignment is always right to left
    - Read 1 goes into X
    - We aren't comparing X with 1 here
    - Learn this by heart! (for first-time programmers)
  - The semi-colon isn't necessary in R, but it's a good practice to use it
    - semi-colon is an instruction demarcation. Meaning you can write `a = 1; b = 2` in one line.
    - `X = ;` is incomplete
  - `#` (prefix) is used as a comment. Use it for helpful comments.
    - Use Ctrl-Shift-C for multi-line comments

R Input and Output

- Simple assignment
  - `X = 1;` (*or* `X <- 1;`)
  - Assignment is always right to left
    - Read 1 goes into X
    - We aren't comparing X with 1 here
    - Learn this by heart! (for first-time programmers)
  - The semi-colon isn't necessary in R, but it's a good practice to use it
    - semi-colon is an instruction demarcation. Meaning you can write `a = 1; b = 2` in one line.
    - `X = ;` is incomplete
  - `#` (prefix) is used as a comment. Use it for helpful comments.
    - Use Ctrl-Shift-C for multi-line comments

- Value of X can be seen by writing X and hitting enter

Vectors

Vectors

- A sequence of numbers. Many ways to input!
  - `Y = c(1,7,-3,41); # concatenate arbitrary numbers`
  - `Y = 1:10; # natural numbers`
  - `Y = seq(1,100,9); # skip by 9`
  - `Y = rep(2, 3); # repeat 2 3-times`
  - `Y = rep(1:2, 3); # repeat the vector c(1,2) 3-times`
  - `Y = rep(1:2, each = 3); # repeat each element of c(1,2) 3-times`
  - `Y = c(); # empty vector`
  - Execute this: `Y = c(1:3, rep(c(5,7), each = 2), rep(9, 4), 7);`

Vectors

- A sequence of numbers. Many ways to input!
  - `Y = c(1,7,-3,41); # concatenate arbitrary numbers`
  - `Y = 1:10; # natural numbers`
  - `Y = seq(1,100,9); # skip by 9`
  - `Y = rep(2, 3); # repeat 2 3-times`
  - `Y = rep(1:2, 3); # repeat the vector c(1,2) 3-times`
  - `Y = rep(1:2, each = 3); # repeat each element of c(1,2) 3-times`
  - `Y = c(); # empty vector`
  - Execute this: `Y = c(1:3, rep(c(5,7), each = 2), rep(9, 4), 7);`

- Length of vector: `length(Y);`

Vectors

- A sequence of numbers. Many ways to input!
  - `Y = c(1,7,-3,41); # concatenate arbitrary numbers`
  - `Y = 1:10; # natural numbers`
  - `Y = seq(1,100,9); # skip by 9`
  - `Y = rep(2, 3); # repeat 2 3-times`
  - `Y = rep(1:2, 3); # repeat the vector c(1,2) 3-times`
  - `Y = rep(1:2, each = 3); # repeat each element of c(1,2) 3-times`
  - `Y = c(); # empty vector`
  - Execute this: `Y = c(1:3, rep(c(5,7), each = 2), rep(9, 4), 7);`

- Length of vector: `length(Y);`

- Accessing i^{th} element of vector: `Y[i];`
  - square (not curly or parenthesis) brackets
  - `i` should be between `1` and `length(Y)`
  - Printing the entire vector is as before: `Y`

Objects in R

Objects in R

- 5 basic (atomic) types of objects
  - character – strings
  - numeric – real numbers. Also called double.
  - integer – natural numbers. Default data type for numeric vectors.
    - `typeof(1:10)`
    - Execute: `as.integer(2^31 - 1)` and then `as.integer(2^31)`
    - There is raw data type as well. It represents hexadecimal numbers. Try: `as.raw(255)`
  - complex – complex numbers. We won't use them now!
    - `2 + 3i`
  - logical – True/False (binary)
    - `T, F, TRUE, FALSE`

Objects in R

- 5 basic (atomic) types of objects
  - character – strings
  - numeric – real numbers. Also called double.
  - integer – natural numbers. Default data type for numeric vectors.
    - `typeof(1:10)`
    - Execute: `as.integer(2^31 - 1)` and then `as.integer(2^31)`
    - There is raw data type as well. It represents hexadecimal numbers. Try: `as.raw(255)`
  - complex – complex numbers. We won't use them now!
    - `2 + 3i`
  - logical – True/False (binary)
    - `T, F, TRUE, FALSE`

- Most basic collection of objects is a vector (also called an array)
  - Can only contain objects of same class (i.e. character or integer; not both)
  - "list" is a general object type and can contain heterogeneous objects as its members
    - Any Combination of vector, matrix, atomic types etc.
    - It can even contain another list as an object. E.g. linked-lists!
    - Due to its generality its very slow and hence rarely used with large datasets unless situation demands it

Numbers

Numbers

- Default type of any number is numeric (i.e. real). `typeof(1)`

Numbers

- Default type of any number is numeric (i.e. real). `typeof(1)`
- R can differentiate between corner cases:
  - `1/0` is `Inf` -- `is.infinite();`
  - `0/0` is `NaN` -- `is.nan();`
  - Missing data is `NA` -- `is.na();`
  - Check what's `Inf-Inf` ?

Numbers

- Default type of any number is numeric (i.e. real). `typeof(1)`
- R can differentiate between corner cases:
  - `1/0` is `Inf` -- `is.infinite();`
  - `0/0` is `NaN` -- `is.nan();`
  - Missing data is `NA` -- `is.na();`
  - Check what's `Inf-Inf` ?

- Arithmetic Operations
  - `*` multiplies
  - `/` divides
  - `^` takes exponent
  - `%%` is the modulo (remainder) operator. Try: `7 %% 2;`

Coercion

Coercion

- Mixing Objects
  - Automatically coerced to the same class.
  - Try: `c(1:7, "a"); c(T, 2); c("a", FALSE);`
  - Implicit coercion!
  - Never use unless you know what you're doing!
    - Even then its better to explicitly coerce objects

Coercion

- Mixing Objects
  - Automatically coerced to the same class.
  - Try: `c(1:7, "a"); c(T, 2); c("a", FALSE);`
  - Implicit coercion!
  - Never use unless you know what you're doing!
    - Even then its better to explicitly coerce objects

- Explicit Coercion
  - `as.character(1:5);`
  - `as.numeric("iimb"); # warning!`
  - `as.logical(seq(-2,2,1));`

List

List

- Can carry different types of data together
  - `L = list(1, FALSE, 3.14, "iimb", "c", 4-3i, list("2nd-list"));`
  - Print list: `L;`
  - Check: `typeof(L); typeof(L[4]); typeof(L[[4]]); typeof(L[[7]]);`
  - Single square brackets `[i]` access the i^{th} list embedded in the list `L`
    - It's a pointer to the element (don't bother if you don't know what a pointer is!)
  - Double square brackets `[[i]]` access the i^{th} element
  - Can append elements in list: `L = append(L, "8-th");`
  - `unlist(L);` will coerce all elements into a single type and return a vector
  - Delete an element from a list:
    - I don't know how to do that!
    - Let's google: "delete element from list in R"
    - Open the answer on www.stackoverflow.com

Matrices

Matrices

- Generalization of vectors
  - 2 dimensions instead of one!

Matrices

- Generalization of vectors
  - 2 dimensions instead of one!
  - N x K matrix means a matrix having N rows and K columns. Total of NK elements.
  - `M = matrix(nrow = 2, ncol = 3);`
  - Dimensions: `dim(M);`
  - Can think of M as
    - 3 columns vectors each of length 2, *or*
    - 2 row vectors each of length 3
  - Populate matrix:
    - `M = rbind(1:3, 4:6);`
    - `M = matrix(1:6, nrow = 2, byrow = T); # default is by column`
    - `M = cbind(1:2, 3:4, 5:6);`

Matrices (cont.)

- Indexing a matrix
  - `M[i,j]` gives the element at i^{th} row and j^{th} column
  - `M[i,]` gives the entire i^{th} row (a vector)
  - `M[,j]` gives the entire j^{th} column (a vector)

Matrices (cont.)

- Indexing a matrix
  - `M[i,j]` gives the element at i^{th} row and j^{th} column
  - `M[i,]` gives the entire i^{th} row (a vector)
  - `M[,j]` gives the entire j^{th} column (a vector)

- Matrix multiplication
  - `%*%` performs the usual matrix multiplication. Try: `M %*% M`
    - Dimensions must match
    - Try `t(M) %*% M;`
    - `t(M)` takes transpose of a matrix!
  - `M*N` perform element-wise multiplication

Matrices (cont.)

- Identity matrix: `diag(3)`
- Diagonal Matrix: `diag(c(1,5,7)); diag(1:7);`
- Diagonal of a matrix: `diag(M)`
- Trace of a matrix: `sum(diag(M))`
- Inverse of a matrix:
  - Must be a square matrix: `M = matrix(1:9, nrow = 3, ncol = 3);`
    - Another way to create a matrix. Data is entered column-wise.
  - Determinant must be non-zero: `det(M); M[3,3] = 19; det(M);`
  - Inverse: `solve(M);`

Higher (> 2) dimension Arrays

Higher (> 2) dimension Arrays

- `A = array(1:24, dim = c(2,3,4),`
  `dimnames = list(paste0("ROW-", 1:2),`
  `paste0("COL-", 1:3),`
  `paste0("MATRIX-", 1:4)));`
- `A[,,i]` will be a 2x3 matrix, `A[,i,j]` will be a 2x1 vector, `A[i,j,k]` is a scaler.
  - What the dimension of `A[i,,], A[,j,], A[i,,k]` and `A[i,j,]` ?

Higher (> 2) dimension Arrays

- `A = array(1:24, dim = c(2,3,4),`
  `dimnames = list(paste0("ROW-", 1:2),`
  `paste0("COL-", 1:3),`
  `paste0("MATRIX-", 1:4)));`
- `A[,,i]` will be a 2x3 matrix, `A[,i,j]` will be a 2x1 vector, `A[i,j,k]` is a scaler.
  - What the dimension of `A[i,,], A[,j,], A[i,,k]` and `A[i,j,]`?
- They have a limited use
  - For instance, if you need to compute 10,000 matrices (of dimension NxN) and then add them, then it's better to define an array of dimension `c(N,N,10000)` and after computation do `apply(A, c(1,2), sum)`
  - data.frames/data.tables are almost always easier to build, interpret and summarize!

Factors

Factors

- For categorical data
  - Male, female
  - Cities in a dataset
  - Typically useful when the dataset is large but the no. of categories is small
  - Using factors is more descriptive than integer values
    - Rather than using 1 for PGP, 2 for FPM and 3 for Others; its more intuitive to use factors.

Factors

- For categorical data
  - Male, female
  - Cities in a dataset
  - Typically useful when the dataset is large but the no. of categories is small
  - Using factors is more descriptive than integer values
    - Rather than using 1 for PGP, 2 for FPM and 3 for Others; its more intuitive to use factors.
- Example:
  - `sex = rep(c("male", "female"), 5);`
  - `sex_f = as.factor(sex);`
  - `Check: typeof(sex_f); as.integer(sex_f);`

Factors

- For categorical data
  - Male, female
  - Cities in a dataset
  - Typically useful when the dataset is large but the no. of categories is small
  - Using factors is more descriptive than integer values
    - Rather than using 1 for PGP, 2 for FPM and 3 for Others; its more intuitive to use factors.
- Example:
    - `sex = rep(c("male", "female"), 5);`
    - `sex_f = as.factor(sex);`
    - `Check: typeof(sex_f); as.integer(sex_f);`
- Useful in the regression framework using `lm();`
    - Automatically creates dummy for all but one categories.
    - Conversion between integers (like year) and factor can corrupt your data!
    - Try: `as.integer(as.factor(2000:2020))`