

CS 133 - Introduction to Computational and Data Science

Instructor: Renzhi Cao
Computer Science
Department
Pacific Lutheran University
Spring 2017



Homework

- *Read book to page 25.*
- *Final project. Check Sakai, read papers! Due on May 18 and 24!*
- *Project 2 is due today!*

Simple practices

1. Create a vector `v`, and add two elements: “hello”, 133
2. Print the second element of `v`
3. Convert the second element of `v` to numeric number
4. Setup your working directory to a new 'work' folder in your desktop
5. Create a vector numbers from 1 to 6 and find out its class type
6. Create a vector containing following mixed elements {1, 'a', 2, 'b'} and find out its class. Then create a list with the same elements.
7. Get the first two elements from above vector
8. Get the first and third elements from above vector

Matrices

Matrices are vectors with a dimension attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
```

```
> m
```

```
[,1] [,2] [,3] [1,] NA NA NA [2,] NA NA NA
```

```
> dim(m)
```

```
[1] 2 3
```

```
> attributes(m)
```

```
$dim
```

```
[1] 2 3
```

Matrices

Matrices are constructed column-wise, so entries can be thought of starting in the “upper left” corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
```

```
>m
```

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

Matrices

Matrices can also be created directly from vectors by adding a dimension attribute.

R Nuts and Bolts 17

```
> m <- 1:10
```

```
> m
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
> dim(m) <- c(2, 5)
```

```
> m
```

```
> m[1,2]
```

Matrices

Matrices can be created by column-binding or row-binding with the `cbind()` and `rbind()` functions.

```
> x <- 1:3  
> y <- 10:12  
> cbind(x, y)
```

```
> rbind(x, y)
```

Simple practices

1. Create the following matrices and print it out:

```
1  3  5
7  9  11
13 15 17
```

2. Create the following matrices and print it out:

```
1  41  455  474
2  239 121  357
61 65  178  533
```


Factors

Factors are used to represent categorical data (unordered or ordered), like integer vector where each integer has a label.

- Self-describing. “Male” and “Female” is better value compared to 1 and 2.
- Use `factor()` function to create a factor.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
```

```
>x
```

```
>table(x)
```

```
>## See the underlying representation of factor
```

```
> unclass(x)
```

Factors

The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
```

```
> x ## Levels are put in alphabetical order
```

```
[1] yes yes no yes no
```

```
Levels: no yes
```

```
> x <- factor(c("yes", "yes", "no", "yes", "no"), levels <- c("yes", "no"))
```

```
> x
```

```
[1] yes yes no yes no
```

```
Levels: yes no
```

Missing values

Missing values are denoted by NA or NaN for undefined mathematical operations. (NaN means not a number, like 0/0. NA means missing values)

- `is.na()` is used to test objects if they are NA
- `is.nan()` is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN is also NA but the converse is not true

Missing values

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
> ## Return a logical vector indicating which elements are NA

> is.na(x)

> is.nan(x)

> ## Now create a vector with both NA and NaN values

> x <- c(1, 2, NaN, NA, 4)
> is.na(x)

> is.nan(x)
```

Simple practices

1. Create a vector with the values of 1, 3, NA, 5, NaN
2. Test NA
3. Test NaN

Data Frames

Data frames are used to store tabular data in R.

Data frames are represented as a special type of list where every element of the list has to have the same length.

Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

What is this looks like and what is the difference?

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

Data frames have a special attribute called `row.names` which indicate information about each row of the data frame.

Data Frames

Data frames are usually created by reading in a dataset using the `read.table()` or `read.csv()`. Also, be created explicitly with the `data.frame()` function

Data frames can be converted to a matrix by calling `data.matrix()`.

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
```

```
>nrow(x)
```

```
>ncol(x)
```

Simple practices

1. Create a data frame with the following values:

ID	Score	CS133
1	89	TRUE
2	30	FALSE
3	0	FALSE
4	99	TRUE

2. Convert the data frame a matrix `m`, and print the score of ID 3.

Names

R objects can have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3  
> names(x)
```

```
> names(x) <- c("New York", "Seattle", "Los Angeles")
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
```

```
> x
```

Names

Matrices can have both column and row names.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
```

```
> m
```

Column names and row names can be set separately using the `colnames()` and `rownames()` functions.

```
> colnames(m) <- c("h", "f")
```

```
> rownames(m) <- c("x", "z")
```

Summary

There are a variety of different builtin-data types in R. In this chapter we have reviewed the following

- atomic classes: numeric, logical, character, integer, complex
- vectors, lists
- factors
- missing values
- data frames and matrices

Exercises

1. Create a matrix `m` with 2 rows and 2 columns
2. Assign 1 to element at row 1, column 1
3. Assign 30 to element at row 2, column 2
4. Assign `Inf` to element at row 2, column 1
5. print `m`
6. Convert `m` to a character vector `n`
7. Guess what will be `n[!is.na(n)]`?
8. Print the names of vector `n`
9. Set names of vector `n`

Learn more operations on R object.

Next time we are going to learn how to get data In and Out of R. Please Read the book.

Subsetting of R objects

There are three operators that can be used to extract subsets of R objects.

- The `[]` operator always returns an object of the same class as the original. It can be used to select multiple elements of an object
- The `[[` operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- The `$` operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of `[[`.

Subsetting a vector

```
> x <- c("a", "b", "c", "c", "d", "a")
```

```
> x[1] ## Extract the first element
```

```
> x[2] ## Extract the second element
```

The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence.

```
> x[1:4]
```

```
> x[c(1, 3, 4)]
```

Subsetting a vector

We can also pass a logical sequence to the `[]` operator to extract elements of a vector that satisfy a given condition.

```
> u <- x > "a"
```

```
> u
```

```
> x[u]
```

```
> x[x > "a"]
```


Subsetting a matrix

Matrices can be subsetting in the usual way with (i,j) type indices. Here, we create simple 2*3 matrix with the matrix function.

```
> x <- matrix(1:6, 2, 3)
```

```
>x
```

We can access the $(1, 2)$ or the $(2, 1)$ element of this matrix using the appropriate indices.

```
> x[1, 2]
```

```
> x[2, 1]
```

```
> x[1, ] ## Extract the first row
```

```
> x[, 2] ## Extract the second column
```

Subsetting a matrix

Dropping matrix dimensions

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1*1 matrix. Often, this is exactly what we want, but this behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
```

```
> x[1, 2]
```

```
> x[1, 2, drop = FALSE]
```

```
> x[1, ]
```

```
> x[1, , drop = FALSE]
```

Subsetting lists

Lists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
> x <- list(foo = 1:4, bar = 0.6)
```

```
>x
```

The `[[` operator can be used to extract single elements from a list. Here we extract the first element of the list.

```
> x[[1]]
```

Subsetting lists

The `[[` operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the `$` operator to extract elements by name.

```
> x[["bar"]]
```

```
> x$bar
```

Subsetting lists

One thing that differentiates the `[[` operator from the `$` is that the `[[` operator can be used with computed indices. The `$` operator can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
```

```
> name <- "foo"
```

```
>
```

```
> ## computed index for "foo"
```

```
> x[[name]]
```

```
>## the element "name" doesn't exist
```

```
> x$name
```

```
> ## element "foo" does exist
```

```
> x$foo
```

Subsetting Nested Elements of a List

The `[]` operator can take an integer sequence if you want to extract a nested element of a list.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
```

```
>
```

```
> ## Get the 3rd element of the 1st element
```

```
> x[[c(1, 3)]]
```

```
> ## Same as above
```

```
> x[[1]][[3]]
```

```
> ## 1st element of the 2nd element
```

```
> x[[c(2, 1)]]
```

Partial matching

Partial matching of names is allowed with `[[` and `$`. This is often very useful during interactive work if the object you're working with has very long element names.

```
> x <- list(aardvark = 1:5)
```

```
> x$a
```

```
> x[["a"]]
```

```
> x[["a", exact = FALSE]]
```

Removing NA values

A common task in data analysis is removing missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
```

```
> bad <- is.na(x)
```

```
> print(bad)
```

```
> x[!bad]
```


Removing NA values

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
> x <- c(1, 2, NA, 4, NA, 5)
```

```
> y <- c("a", "b", NA, "d", NA, "f")
```

```
> good <- complete.cases(x, y)
```

```
> good
```

```
> x[good]
```

```
> y[good]
```

Removing NA values

You can use `complete.cases` on data frames too.

```
> head(airquality)
```

```
> good <- complete.cases(airquality)
```

```
> head(airquality[good, ])
```

Vectorized operations

Many operations in R are vectorized, meaning that operations occur in parallel in certain R objects. This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages.

```
> x <- 1:4
```

```
> y <- 6:9
```

```
> z <- x + y
```

```
> z
```

```
> x >= 2
```

```
> x-y
```

```
> x*y
```

Vectorized operations

Matrix operations are also vectorized, making for nicely compact notation.

```
> x <- matrix(1:4, 2, 2)
```

```
> y <- matrix(rep(10, 4), 2, 2)
```

```
> ## element-wise multiplication
```

```
> x*y
```

```
> ## element-wise division
```

```
> x/y
```

```
> ## true matrix multiplication
```

```
> x %*% y
```

Exercises

1. Create a vector `v` with the following elements: 3, 5 , 7 , 9 , 10 , 133
2. Print second, third, and fifth element
3. Print second, third, and fifth element
4. Get all elements which is larger than 8
5. Create a 2*3 matrix `m` based on the previous vector `v`.
6. Print first row of matrix `m`
7. Print second column of matrix `m`
8. Create a list `l` with the same elements to `v`
9. Print the second element of `l`
10. Create vector `v2` with: 3, NA, 4, 5
11. Removing missing value in `v3`
12. Practice on course website