Algoritma Academy: Programming for Data Science

Samuel Chan

02 April, 2024

Before you go ahead and run the code in this coursebook, it's often a good idea to go through some initial setup. Under the *Libraries and Setup* tab you'll see some code to initialize our workspace, and the libraries we'll be using for the projects. You may want to make sure that the libraries are installed beforehand by referring back to the packages listed here. Under the *Training Focus* tab we'll outline the syllabus, identify the key objectives and set up expectations for each module.

Background

Algoritma

The following coursebook is produced by the team at Algoritma for its Data Science Academy workshops. The coursebook is intended for a restricted audience only, i.e. the individuals and organizations having received this coursebook directly from the training organization. It may not be reproduced, distributed, translated or adapted in any form outside these individuals and organizations without permission.

Algoritma is a data science education center with bootcamp programs offered in:

- Bahasa Indonesia (Jakarta campus)
- English (Singapore campus)

Lifelong Learning Benefits

If you're an active student or an alumni member, you also qualify for all our future workshops, 100% free of charge as part of your **lifelong learning benefits**. It is a new initiative to help you gain mastery and advance your knowledge in the field of data visualization, machine learning, computer vision, natural language processing (NLP) and other sub-fields of data science. All workshops conducted by us (from 1-day to 5-day series) are available to you free-of-charge, and the benefits **never expire**.

Second Edition

This coursebook is initially written in 2017.

This is the second edition, written in late August 2020. Some of the code has been refactored to work with the latest major version of R, version 4.0. I would like to thank the incredible instructor team at Algoritma for their thorough input and assistance in the authoring and reviewing process.

Libraries and Setup

We'll set-up caching for this notebook given how computationally expensive some of the code we will write can get.

```
# chunk set up
knitr::opts_chunk$set(cache=TRUE,
                      fig.align = "center",
                      comment = "#>"
                      )

# set up scientific notation
options(scipen = 9999)

# clear Global Environment
rm(list=ls())
```

You will need to use `install.packages()` to install any packages that are not already downloaded onto your machine. You then load the package into your workspace using the `library()` function:

```
library(dplyr)
library(skimr)
```

Training Objectives

The primary objective of this course is to provide a comprehensive introduction to the science of statistical programming and the toolsets required to succeed with data science work. The syllabus covers:

- **Basic Programming in R**
- Objects and Environment
- Data Classes in R
- Data Structures in R

- Data Science Workflow
- R Scripts and R Markdown
- **Data Manipulation**

- Read & Extracting Data

- Practical Data Cleansing

- Data Transformation
- **Statistical Computing**

- Organizing your Project

- Modern Tools for Data Analysis

- Reproducible Data Science

By the end of the workshop, Academy students can choose to complete either of the Learn-By-Building modules as their graded assignment:

R Script to clean & transform the data
A programming script that perform various data cleansing tasks and output the result in an appropriate format for further data science work.

Reproducible Data Science
Create an R Markdown file that combines data transformation code with explanatory text. Add formatting styles and hierarchical structure using Markdown.

R Programming

Since you'll spend a great deal of your time working with data in R and RStudio, I think it's important to get yourself very familiar with this IDE (Integrated Development Environment). RStudio is the most popular integrated development for R and is a core tool for data science teams in Airbnb[1], Uber[2] etc., and is a tool we'll be using throughout the Academy workshops.

If you're a seasoned programmer, the **Option + Shift + K** (Alt + Shift + K on Windows) combination will bring up a shortcut reference guide that helps you use RStudio more effectively.

Why learn R at all?

1. **Built by statisticians, for statisticians.**
   R is a statistical programming language created by Ross Ihaka and Robert Gentleman at the Department of Statistics, at the University of Auckland (New Zealand). R is created for the purpose of data analysis and as such, is different in nature from traditional programming languages. R is not just a statistical programming language, it is a complete environment for data scientist and the most widely used data analysis software today[3].

2. **Libraries.**
   R's libraries extend R's graphical abilities, and adds out-of-the-box functionalities for linear and non-linear modeling, statistical tests (confidence tests, P-value, t-test etc), time-series analysis, and various machine learning tools such as regression algorithms, classification algorithms, and clustering algorithms. The R community is noted for its active contributions in terms of packages and boasts nearly 20,000 packages to date.

3. **Open Source.** Part of the reason for its active and rapidly growing community is the open-source nature of R. Users can contribute packages – many of which packaged some of the most advanced statistical tools that are not found in other commercial, proprietary statistical computing softwares.

4. **Used by the biggest software companies in the world.**
   R is used by Google to calculate ROI on advertising campaigns and estimate causal effect (say, estimate the impact of an app feature on app downloads or number of additional sales from an AdWords

[1]How R Helps AirBnB make the most of its data
[2]Uber Engineering's Tech Stack: The Foundation
[3]Microsoft R Open: The Enhanced R Distribution

campaign); In fact, it even released its own R packages to allow other R users to do similar analysis using the same tool[4]. Data Science employees at Google participate in User Groups to discuss how R is used in Google (answer: it's used very widely in a production environment at Google and Google integrates R with many of their own technologies), publishing its own R client for the Google Prediction API, Google's R style guide, and its developers have released a number of R packages over the years. Microsoft first uses R for Azure capacity planning, Xbox's TrueSkill Matchmaking System, player churn analysis, in-game purchase optimization, fraud detection, and other internal services across Microsoft's line of products[5], and then went on to acquire Revolution Analytics, whom products were then rebranded and renewed by Microsoft and now known as Microsoft R Server, Microsoft R Open, Microsoft Data Science Virtual Machine etc.

5. **Ready for big data**
   RHadoop, ParallelR, Revolution R Enterprise and a handful of other toolkits adds powerful big data support, allowing data engineers to create custom parallel and distributed algorithms to handle parallel / map-reduce programming in R. This makes R a popular choice for big data analytics and high performance, enterprise-level analytics platform.

6. **Employability!**
   R is a required skill for data science roles across all top Indonesian's startups: GoJek, Traveloka, Uber, Shopee, Twitter, HappyFresh etc. Do a quick search on job portals (Tech In Asia's Jobs, JobStreet etc) and you'll see R is a highly sought-after language skill.

The Google's R Style Guide is the one we'll adhere to - if this is the first time you're writing R code, I recommend you adopt these "best practices" as a certain level of strictness can make you a more disciplined and methodical programmer in the long run.

R Programming Basics

It pays to get yourself familiar with R and RStudio, the IDE (interactive development environment). In our workshop, we'll discuss in more details the various functionalities of RStudio's interface, and if this is the first time you're working in a code environment, spend some time to get yourself familiar with this IDE along with the RMarkdown format as you'll be working with it a lot!

Before moving forward, make sure you have a solid understanding of the following:

- R as a statistical programming language

- RStudio as a code editor and integrated development environment
- The RMarkdown format

To get started, let's write our first R code by typing `getwd()` into the Console (bottom of the screen), or by running in from within a Chunk (look for the green "run" button):

```
# This is a comment
getwd()
```

```
#> [1] "G:/My Drive/RnD/Python for Finance/1_programming_for_data_science"
```

```
# setwd(...)
```

[4]CausalImpact: A new open-source package for estimating causal effects in time series
[5]R at Microsoft

Notice the "#" character in the first and third line of the code chunk, indicating to R that it's a comment and should be ignored. `setwd()` was ignored because it's on the same line and to the right of the "#" character. As you may have expected, `setwd()` is used to change our working directory by setting a new one.

R is **case-sensitive** so "Algoritma" and "algoritma" are different symbols and will point to different variables.

```
activity <- "Programming"
activity == "programming"
```

```
#> [1] FALSE
```

```
print(paste(activity, "in data science."))
```

```
#> [1] "Programming in data science."
```

```
# Un-comment the following line; Observe that object 'Activity' don't exist!
# print(Activity) will not work
```

Vectors

Speaking of objects, some of the most common data types that you'll come across are: - character
- numeric
- integer
- logical

The most basic form of an R object is a vector. As a rule, a vector can only contain objects of the same class:

```
vector1 <- c("learning", "data", "science", 2018)
class(vector1)
```

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

```
vector2 <- c(1, FALSE, FALSE, 0)
class(vector2)
```

```
#> [1] "numeric"
```

Also observe how we use the `c()` function to concatenate objects together to form a vector.

`vector1` is now an object in our global environment, but if you're paying attention, you'll notice that it is a **character vector**. While 2018 itself is a numeric, because of the "same-class" rule we learn above, 2018 was coerced into a character so that the resulting vector is valid. 2018 (the numeric) is "2018" (character) as a result:

```
vector1
```

```
#> [1] "learning" "data"     "science"  "2018"
```

Similarly, in `vector2`, `1` is a numeric, and `FALSE` is a logical, and therefore the `FALSE` values are coerced into a numeric. Go ahead and print out `vector2` as a confirmation:

```r
# your code here:
```

R objects may have attributes like `names`, `class`, `length`, `colnames`, `dim` etc:

```r
names(vector2) <- c("User ID", "Active", "Cart Items", "Payment")
length(vector2)
```

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

```r
vector2
```

```
#>    User ID     Active Cart Items    Payment 
#>          1          0          0          0
```

Recall how implicit coercion (R's default) takes place earlier when we create our `vector1` and `vector2`. We could explicitly coerce one class to another:

```r
vector2 <- c(1,FALSE,FALSE,0)
vector2.b <- as.logical(vector2)
vector2.b
```

```
#> [1]  TRUE FALSE FALSE FALSE
```

```r
class(vector2.b)
```

```
#> [1] "logical"
```

Dive Deeper:

Create a vector and name it `customers`. Store 4 names in the vector and make sure it is a `character` vector. Create another vector and name it `age`, store 4 numeric in the vector and make sure it is a `numeric` vector.

```r
# Your code here:
```

2. Use `class()` and `length()` in the code chunk below to verify that you have done the exercise above correctly:

```r
# Your code here:
```

3. Create another vector and name it `suppliers`. Store 3 names in it:

```r
# Your code here
```

4. Join the `customers` and `suppliers` vector into one vector using the concatenate technique you've learned, which is `c()`.

```r
# Your code here:
```

If you've managed to execute the above exercises in the dive deeper section: congratulations! Throughout the course you'll do a number of these exercises, and they are useful revision tools that you should take advantage of to test your knowledge and make sure you have a full grasp of the topics being assessed.

You've see how numeric and character classes and even made a few vectors of your own above! But R has other object types and we'll take a look at them:

```
# character
tempo <- c("Algoritma", "Indonesia", "e-Commerce", "Jakarta")
# numeric
tempo <- c(-1, 1, 2, 3/4, 0.5)
# integer
tempo <- c(1L, 2L)
# integer
tempo <- 5:8
# logical
tempo <- c(TRUE, TRUE, FALSE)
```

A quick note on integers: they cannot take decimal or fractional values, while numerics can. Numerics act more like the "float" or "double" types supported by many other programming languages.

Matrix

When we create a vector and give it a dimension attribute, we end up with a matrix:

```
matri <- matrix(11:16, nrow=3, ncol=2)
dim(matri)
```

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

```
matri
```

```
#>      [,1] [,2]
#> [1,]   11   14
#> [2,]   12   15
#> [3,]   13   16
```

Notice how the values fill up by column from the [1,1] position, which is the most upper-left position.

Once created, we can refer to any row or column using R's subsetting operator:

```
matri[1,]
```

```
#> [1] 11 14
```

```
matri[,2]
```

```
#> [1] 14 15 16
```

We could also have constructed a matrix by giving an existing vector the `dim` attribute:

```
numbers <- 11:16
dim(numbers) <- c(2,3)
numbers
```

```
#>      [,1] [,2] [,3]
#> [1,]   11   13   15
#> [2,]   12   14   16
```

Notice `c(2,3)` means "2 rows, 3 columns". Contrast this to our `matri` object above and the way we constructed matrices using two different approach.

Another interesting way to construct a matrix:

```r
accounts <- c("AlphaMall", "BetaMall", "OmegaMall")
sales <- c(400,320,380)
returns <- c(0,0,480)
netsales <- sales - returns
# cbind = bind as columns
# rbind = bind as rows
# rbind(accounts, sales, returns)

sales_records <- cbind(accounts, sales, netsales)
```

`sales_record` is now a matrix. Go ahead and print it, then observe how 400 (numeric) has been coerced into "400" (strings) so the resulting matrix is a valid R object.

Dive Deeper:
1. You learned how to bind the three vectors by columns. Now create a matrix named `sales_records` and bind `sales`, `returns` and `netsales` by rows (instead of columns). You can do this with `rbind` (row-bind)

```r
# Your code here
```

2. You can optionally check that you've done the above step correctly by printing out the matrix and / or use `dim()` to verify that is in fact a 3x3 matrix. Now assign `accounts` as column names to your matrix. To assign column names to a matrix, we can use `colnames(mymatrix) <- c("Name1", "Name2", "Name3")`:

```r
# Your code here
```

3. Print our `sales_records`:

```r
# Your code here
```

Dive Deeper

Recall that I've repeatedly stressed that as a rule, a vector can contain objects of the same class? Consider the following code:

- 1. What is the class of the resulting vector `quiz1`?
- 2. What is the dimensions attribute of `quiz1`?
- 3. How many times did implicit coercion happened?

List

There is a type of R object that is exempted from the rule we repeatedly mention above, and it's the **List**:

```r
our.list <- list(TRUE, "TRUE", c(1,6,12), 1+5i)
our.list
```

```
#> [[1]]
#> [1] TRUE
#>
#> [[2]]
#> [1] "TRUE"
#>
#> [[3]]
#> [1]  1  6 12
#>
#> [[4]]
#> [1] 1+5i
```

A list, as we've observed above, can contain elements that are of different classes from other members of the list. You can can subset from a list much like how you've done earlier: however, any subsets using a single square bracket `[]` will return a list. To return the elements itself, use double square-brackets: `[[]]`

Demonstration of subsetting elements from our list:

```
our.list[3]
```

```
#> [[1]]
#> [1]  1  6 12
```

```
our.list[[3]]
```

```
#> [1]  1  6 12
```

```
class(our.list[3])
```

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

```
class(our.list[[3]])
```

```
#> [1] "numeric"
```

Factors

Another important concept in R is factors - many statistical modeling techniques and prediction algorithms treat factors specially either as a target outcome (in machine learning language) or dependent variable (in statistics) while many other modeling techniques treat factors specially when they're used as independent variables. Factors is useful in representing categorical variables whether or not they are unordered (cash, credit, transfer) or ordered (high volume, normal volume, low volume):

```
categories <- factor(c("OfficeSupplies", "Computers", "Packaging", "Machinery", "Building"))
categories # levels are sorted alphabetically unless through the levels argument
```

```
#> [1] OfficeSupplies Computers      Packaging      Machinery      Building
#> Levels: Building Computers Machinery OfficeSupplies Packaging
```

Data Frames

Data frames can be thought of as a special case of lists where every element of the list has to have the same length. Each element of the list can be thought of as a column in the data frame.

```
categories_df <- data.frame(categories=c("OfficeSupplies", "Computers", "Packaging", "Machinery", "Buil
categories_df
```

```
#>       categories category_id
#> 1 OfficeSupplies         111
#> 2      Computers         112
#> 3      Packaging         113
#> 4      Machinery         114
#> 5       Building         115
```

And we can perform mathematical operations on our dataframes, the same way we can do it with matrices. If we need to update our system by adding one new category on the top of the list such that all existing IDs are incremented by one, we can do so:

```
categories_df$category_id + 1
```

```
#> [1] 112 113 114 115 116
```

Notice that here we're accessing the `category_id` column using the '$' operator.

Hopefully by now you also observe how R conveniently applies implicit coercion so our data frame and matrix can be multiplied. This is another nice property of R!

```
class(1-TRUE)
```

```
#> [1] "numeric"
```

```
TRUE + TRUE * 34
```

```
#> [1] 35
```

R Programming with Retail

With the foundations laid, let's now take a look at a real life dataset and apply our newly acquired knowledge.

First make sure the data you'll like to work with is also in your current directory, and use the `read.csv()` to read our csv file into your global environment. Having our CSV in the same directory as the one we're working in isn't required, we may have used the full path as well. However, to keep our projects organized I would recommend you keep your scripts, working files, and its dependent data in the same directory whenever it's convenient to do so:

```
retail <- read.csv("data_input/retail.csv")
names(retail)
```

```
#>  [1] "Row.ID"       "Order.ID"     "Order.Date"   "Ship.Date"    "Ship.Mode"
#>  [6] "Customer.ID"  "Segment"      "Product.ID"   "Category"     "Sub.Category"
#> [11] "Product.Name" "Sales"        "Quantity"     "Discount"     "Profit"
```

The two lines of code above does two things:
- Read our csv file into R so we can begin working on it
- Use `names()` to get the names of our dataset variable

If you have tried calling `names(Retail)` you would have gotten an error that says `object 'Retail' not found`. This is because R is case-sensitive, so `Retail` and `retail` are different things. Correct the following code so it prints the dimensions of the dataframe:

```
# Will throw an error: Retail not found
dim(Retail)
```

```
#> Error in eval(expr, envir, enclos): object 'Retail' not found
```

Notice also that R commands are separated either by a semi-colon (';'), or by a newline. So we can write our code like the above, or we could have separate the commands with ';'. For the most part, we will stick to writing code using the new line format as it makes our code more readable and it follows best practice. An example of two commands on the same line:

```
purchases <- 15; purchases * 2;
```

```
#> [1] 30
```

Dive Deeper: Inspect the structure of the data using `str()`
Call `str()` on our `retail` dataset the same way you use `names()`. `str()` returns the structure of an R Object and we'll be using it a lot given how helpful that is.

```
# Your code here
```

Now if you've previously been working with data in a speadsheet-like environment, using `names()` and `str()` to inspect data may taking a bit of getting used to - however, I can assure you the benefits will become apparent (from a programmability perspective but also, very soon, you'll be dealing with data with thousands of variables and a spreadsheet environment just isn't going to make much sense). For a relatively small dataset as this, you can still view the full CSV in its raw format through the `View(retail)` command, or clicking on the "spreadsheet" icon next to the data you'll like to inspect in the Environment pane.

I don't recommend you use the `View()` command, because in real life you don't always know beforehand the size of data, and taking a peek at the first few rows or last few rows of data would have given you a good idea into the underlying structure of the data.

To see the first 6 observations, we could have just done `head(retail)`. We can pass in an extra argument, n, so the function would return the first n number of rows instead of the default 6. The following code returns the first 5 rows of our data:

```
head(retail, 5)
```

```
#>   Row.ID       Order.ID Order.Date Ship.Date      Ship.Mode Customer.ID
#> 1      1 CA-2016-152156    11/8/16  11/11/16   Second Class    CG-12520
#> 2      2 CA-2016-152156    11/8/16  11/11/16   Second Class    CG-12520
#> 3      3 CA-2016-138688    6/12/16   6/16/16   Second Class    DV-13045
#> 4      4 US-2015-108966   10/11/15  10/18/15 Standard Class    SO-20335
#> 5      5 US-2015-108966   10/11/15  10/18/15 Standard Class    SO-20335
#>     Segment      Product.ID        Category Sub.Category
#> 1  Consumer FUR-BO-10001798       Furniture    Bookcases
#> 2  Consumer FUR-CH-10000454       Furniture       Chairs
#> 3 Corporate OFF-LA-10000240 Office Supplies       Labels
#> 4  Consumer FUR-TA-10000577       Furniture       Tables
#> 5  Consumer OFF-ST-10000760 Office Supplies      Storage
#>                                                  Product.Name    Sales Quantity
#> 1                           Bush Somerset Collection Bookcase 261.9600        2
#> 2 Hon Deluxe Fabric Upholstered Stacking Chairs, Rounded Back 731.9400        3
#> 3   Self-Adhesive Address Labels for Typewriters by Universal  14.6200        2
#> 4               Bretford CR4500 Series Slim Rectangular Table 957.5775        5
#> 5                              Eldon Fold 'N Roll Cart System  22.3680        2
#>   Discount    Profit
#> 1     0.00   41.9136
#> 2     0.00  219.5820
#> 3     0.00    6.8714
#> 4     0.45 -383.0310
#> 5     0.20    2.5164
```

I'd now like to drop the first two variables: `Row.ID` and `Order.ID` since we won't be using them. Recall that in R, we can achieve that with `retail[,-c(1:2)]` or `retail[,3:15]`. The first option explicitly eliminates the first two variables while the latter retain only the third variable to the last.

Data Structures in R

Another thing I'd like to do is to change the type of our `Order.Date` and `Ship.Date` variables. They are currently stored as a Factor (''), which means R will treat them as categorical data. Since they are dates are not categorical, let's perform the conversion to Date using `as.Date()`. Because our dates are in the **mm/dd/yy** format, we would specify an additional argument to `as.Date()` indicating the format:

```
# loadn data
retail <- read.csv("data_input/retail.csv")

# subset columns/variables
retail <- retail[,-c(1:2)]

# date transformation
retail$Order.Date <- as.Date(retail$Order.Date, "%m/%d/%y")
retail$Ship.Date <- as.Date(retail$Ship.Date, "%m/%d/%y")

# quick check
head(retail)
```

```
#>   Order.Date  Ship.Date      Ship.Mode Customer.ID   Segment      Product.ID
#> 1 2016-11-08 2016-11-11   Second Class    CG-12520  Consumer FUR-BO-10001798
#> 2 2016-11-08 2016-11-11   Second Class    CG-12520  Consumer FUR-CH-10000454
#> 3 2016-06-12 2016-06-16   Second Class    DV-13045 Corporate OFF-LA-10000240
#> 4 2015-10-11 2015-10-18 Standard Class    SO-20335  Consumer FUR-TA-10000577
#> 5 2015-10-11 2015-10-18 Standard Class    SO-20335  Consumer OFF-ST-10000760
#> 6 2014-06-09 2014-06-14 Standard Class    BH-11710  Consumer FUR-FU-10001487
#>          Category Sub.Category
#> 1       Furniture    Bookcases
#> 2       Furniture       Chairs
#> 3 Office Supplies       Labels
#> 4       Furniture       Tables
#> 5 Office Supplies      Storage
#> 6       Furniture  Furnishings
#>                                                       Product.Name    Sales
#> 1                                Bush Somerset Collection Bookcase 261.9600
#> 2      Hon Deluxe Fabric Upholstered Stacking Chairs, Rounded Back 731.9400
#> 3        Self-Adhesive Address Labels for Typewriters by Universal  14.6200
#> 4                    Bretford CR4500 Series Slim Rectangular Table 957.5775
#> 5                                   Eldon Fold 'N Roll Cart System  22.3680
#> 6 Eldon Expressions Wood and Plastic Desk Accessories, Cherry Wood  48.8600
#>   Quantity Discount    Profit
#> 1        2     0.00   41.9136
#> 2        3     0.00  219.5820
#> 3        2     0.00    6.8714
#> 4        5     0.45 -383.0310
#> 5        2     0.20    2.5164
#> 6        7     0.00   14.1694
```

We will also remove the `Product.ID` and `Discount` variables as they won't be used in this workshop. We'll take this opportunity to learn another one of R's built-in function: `subset()`.

`subset()` returns subsets of vectors, matrices or data frames based on a specified condition:

```
retail <- subset(retail, select=-c(Product.ID, Discount))
str(retail)
```

```
#> 'data.frame':    9994 obs. of  11 variables:
#>  $ Order.Date  : Date, format: "2016-11-08" "2016-11-08" ...
#>  $ Ship.Date   : Date, format: "2016-11-11" "2016-11-11" ...
#>  $ Ship.Mode   : chr  "Second Class" "Second Class" "Second Class" "Standard Class" ...
#>  $ Customer.ID : chr  "CG-12520" "CG-12520" "DV-13045" "SO-20335" ...
#>  $ Segment     : chr  "Consumer" "Consumer" "Corporate" "Consumer" ...
#>  $ Category    : chr  "Furniture" "Furniture" "Office Supplies" "Furniture" ...
#>  $ Sub.Category: chr  "Bookcases" "Chairs" "Labels" "Tables" ...
#>  $ Product.Name: chr  "Bush Somerset Collection Bookcase" "Hon Deluxe Fabric Upholstered Stacking Cha
#>  $ Sales       : num  262 731.9 14.6 957.6 22.4 ...
#>  $ Quantity    : int  2 3 2 5 2 7 4 6 3 5 ...
#>  $ Profit      : num  41.91 219.58 6.87 -383.03 2.52 ...
```

Notice now that `Customer.ID` and `Product.Name` are not categorical variables and hence should not be have the Factor type. Just like how we used `as.Date()` to convert a variable to a date type object, we can use `as.Character()` to convert these two variables to a character type.

```
retail$Customer.ID <- as.character(retail$Customer.ID)
retail$Product.Name <- as.character(retail$Product.Name)
names(retail)
```

```
#>  [1] "Order.Date"   "Ship.Date"    "Ship.Mode"    "Customer.ID"  "Segment"
#>  [6] "Category"     "Sub.Category" "Product.Name" "Sales"        "Quantity"
#> [11] "Profit"
```

Our variables in our dataframe are now stored in the right type. We have variables with the following type in our `retail` dataset:
- Factor (`Factor`)
- Date (`Date`)
- Numeric (`num`)
- Integer (`int`)

Dive Deeper: Inspect the structure of the data using `str()`
Integers are different from numerics in that integers cannot take decimal or fractional values (but instead have to be whole numbers) while numerics can.

Can you write three lines of code so the resulting dataframe has `prices` as a numeric variable, `discount` and `shipping` as a logical variable:

```
set.seed(100)
prices <- sample(400:600, 8)
discount <- c("FALSE", "FALSE", "TRUE", "FALSE", "FALSE", "TRUE", "FALSE", "TRUE")
shipping <- rbinom(8, 1, 0.4)

dat <- data.frame(prices, discount, shipping)
# ==== Your Solution ====



# ==== Your Solution ====

str(dat)
```

```
#> 'data.frame':    8 obs. of  3 variables:
#>  $ prices  : int  501 511 550 597 403 454 469 497
#>  $ discount: chr  "FALSE" "FALSE" "TRUE" "FALSE" ...
#>  $ shipping: int  0 0 1 0 1 0 1 0
```

R has a built-in function, `summary()` that returns quick summary statistics on each of the variable in our dataset. The following commands are valid:
- `summary(retail)`
- `summary(retail[,1:4])`
- `summary(retail$Sales)`

When `summary()` is called on factor (categorical) variables, it gives us a count on each of the categorical level (more formally called **factor level**), and on numeric variables it will print the 5 number summary of that variable instead. The five number summary is a set of descriptive statistics that provide information about our data and consists of the minimum, maximum, median, first and third quantile:

```
summary(retail)
```

```
#>    Order.Date           Ship.Date            Ship.Mode
#>  Min.   :2014-01-03   Min.   :2014-01-07   Length:9994
#>  1st Qu.:2015-05-23   1st Qu.:2015-05-27   Class :character
#>  Median :2016-06-26   Median :2016-06-29   Mode  :character
#>  Mean   :2016-04-30   Mean   :2016-05-03
#>  3rd Qu.:2017-05-14   3rd Qu.:2017-05-18
#>  Max.   :2017-12-30   Max.   :2018-01-05
#>  Customer.ID          Segment             Category          Sub.Category
#>  Length:9994        Length:9994        Length:9994        Length:9994      
#>  Class :character   Class :character   Class :character   Class :character 
#>  Mode  :character   Mode  :character   Mode  :character   Mode  :character 
#>                                                                            
#>                                                                            
#>                                                                            
#>  Product.Name           Sales              Quantity        Profit        
#>  Length:9994        Min.   :    0.444   Min.   : 1.00   Min.   :-6599.978  
#>  Class :character   1st Qu.:   17.280   1st Qu.: 2.00   1st Qu.:    1.729  
#>  Mode  :character   Median :   54.490   Median : 3.00   Median :    8.666  
#>                     Mean   :  229.858   Mean   : 3.79   Mean   :   28.657  
#>                     3rd Qu.:  209.940   3rd Qu.: 5.00   3rd Qu.:   29.364  
#>                     Max.   :22638.480   Max.   :14.00   Max.   : 8399.976  
```

Take a minute to go through the result. Realize how useful this function could be - it packs in a ton of information on the distribution of our data, giving u compact yet useful summary of your data.

Subsetting

Earlier we used `retail[,-c(1:2)]`, which drops the first two columns based on what we specify as the selector. This square bracket notation (using `data[row, column]`) allows us to select the desired row, column, or both convenient.

R however has more indexing features for accessing object elements and taking subsets of observations from a dataset. In the following chunks, you will see practical examples of conditional subsetting using two approaches: - `data[subset_conditions, ]`
- `subset(data, subset_conditions)`

They are syntactically different, but yields the same result.

We could, for example, select observations of transactions that has a `Profit` greater or equal to $5,000 using either approach, as demonstrated below:

```r
retail[retail$Profit >= 5000, ]
```

```
#>      Order.Date  Ship.Date      Ship.Mode Customer.ID   Segment   Category
#> 4191 2017-11-17 2017-11-22 Standard Class    HL-15040  Consumer Technology
#> 6827 2016-10-02 2016-10-09 Standard Class    TC-20980 Corporate Technology
#> 8154 2017-03-23 2017-03-25    First Class    RB-19360  Consumer Technology
#>      Sub.Category                          Product.Name    Sales Quantity
#> 4191      Copiers Canon imageCLASS 2200 Advanced Copier 10499.97        3
#> 6827      Copiers Canon imageCLASS 2200 Advanced Copier 17499.95        5
#> 8154      Copiers Canon imageCLASS 2200 Advanced Copier 13999.96        4
#>        Profit
#> 4191 5039.986
#> 6827 8399.976
#> 8154 6719.981
```

```r
# equivalent:
subset(retail, Profit >= 5000)
```

```
#>      Order.Date  Ship.Date      Ship.Mode Customer.ID   Segment   Category
#> 4191 2017-11-17 2017-11-22 Standard Class    HL-15040  Consumer Technology
#> 6827 2016-10-02 2016-10-09 Standard Class    TC-20980 Corporate Technology
#> 8154 2017-03-23 2017-03-25    First Class    RB-19360  Consumer Technology
#>      Sub.Category                        Product.Name    Sales Quantity
#> 4191      Copiers Canon imageCLASS 2200 Advanced Copier 10499.97        3
#> 6827      Copiers Canon imageCLASS 2200 Advanced Copier 17499.95        5
#> 8154      Copiers Canon imageCLASS 2200 Advanced Copier 13999.96        4
#>        Profit
#> 4191 5039.986
#> 6827 8399.976
#> 8154 6719.981
```

We can specify more than one conditions using the respective "or" ("|"), "and" ("&") logical operators:

```
retail[retail$Profit >= 4500 | retail$Profit <= -4500, ]
```

```
#>      Order.Date  Ship.Date      Ship.Mode Customer.ID   Segment        Category
#> 4099 2014-09-23 2014-09-28 Standard Class    SC-20095  Consumer Office Supplies
#> 4191 2017-11-17 2017-11-22 Standard Class    HL-15040  Consumer      Technology
#> 6827 2016-10-02 2016-10-09 Standard Class    TC-20980 Corporate      Technology
#> 7773 2016-11-25 2016-12-02 Standard Class    CS-12505  Consumer      Technology
#> 8154 2017-03-23 2017-03-25    First Class    RB-19360  Consumer      Technology
#> 9040 2016-12-17 2016-12-21 Standard Class    AB-10105  Consumer Office Supplies
#>      Sub.Category                                    Product.Name     Sales
#> 4099      Binders             Ibico EPK-21 Electric Binding System  9449.950
#> 4191      Copiers            Canon imageCLASS 2200 Advanced Copier 10499.970
#> 6827      Copiers            Canon imageCLASS 2200 Advanced Copier 17499.950
#> 7773     Machines        Cubify CubeX 3D Printer Double Head Print  4499.985
#> 8154      Copiers            Canon imageCLASS 2200 Advanced Copier 13999.960
#> 9040      Binders GBC Ibimaster 500 Manual ProClick Binding System  9892.740
#>      Quantity    Profit
#> 4099        5  4630.475
#> 4191        3  5039.986
#> 6827        5  8399.976
#> 7773        5 -6599.978
#> 8154        4  6719.981
#> 9040       13  4946.370
```

```
# equivalent:
# subset(retail, Profit >= 4500 | Profit <= -4500)
```

Notice that in R, a test of equality is performed with `==` and not `=`. `!=` on the other hand is used to indicate the opposite:

```
4 != 3
```

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

We can combine what we've learned above with conditional subsetting, but as extra exercise let's see how we can use that in conjunction with `table()` to get a desired contingency table output:

```
retail.cons <- retail[retail$Segment == "Consumer", ]
table(retail.cons$Ship.Mode, retail.cons$Category)
```

```
#> 
#>                 Furniture Office Supplies Technology
#>   First Class          164             456        149
#>   Same Day              66             193         58
#>   Second Class         236             603        181
#>   Standard Class       647            1875        563
```

Graded Assignment: Which product segment makes up our high-value transactions?

This part of the assignment is graded for Academy students. Please fill up your answers in the provided answer sheet. Every correct answer is worth **(1) Point**.

Can you adapt the above code to produce a two-dimensional matrix (`Segment` against `Category`)? Use the matrix to answer the following questions:

Question 1: Which following segment makes up the most of our ">1000 Sales" transaction? Subset the data for `retail$Sales >= 1000` and then use `table()` with the "Segment" and "Category" variables as its parameters

Question 2: Among the transactions that ship on "First Class", how many of them were office supplies (to two decimal points)?

We saw earlier that we could use `head()` to peek at the first 6 rows of data. An equivalent for the last 6 rows of data is - you guessed it! - `tail()`.

Cross-Tabulations and Aggregates

I'd like to show you how you can create a cross-tabulation table that allows us to obtain a basic picture of the interrelation between two variables. To get a contingency table displaying the frequency of each data point, we will pass in the corresponding formula to the `xtabs` functions

```
xtabs(~ Sub.Category + Category, retail)
```

```
#>              Category
#> Sub.Category  Furniture Office Supplies Technology
#>   Accessories         0               0        775
#>   Appliances          0             466          0
#>   Art                 0             796          0
#>   Binders             0            1523          0
#>   Bookcases         228               0          0
#>   Chairs            617               0          0
#>   Copiers             0               0         68
#>   Envelopes           0             254          0
#>   Fasteners           0             217          0
#>   Furnishings       957               0          0
#>   Labels              0             364          0
#>   Machines            0               0        115
#>   Paper               0            1370          0
#>   Phones              0               0        889
#>   Storage             0             846          0
#>   Supplies            0             190          0
#>   Tables            319               0          0
```

Notice we passed in `Sub.Category` and `Category` to the right hand side of the formula, which is how we'd let the function know which variables to be used in the cross tabulations.

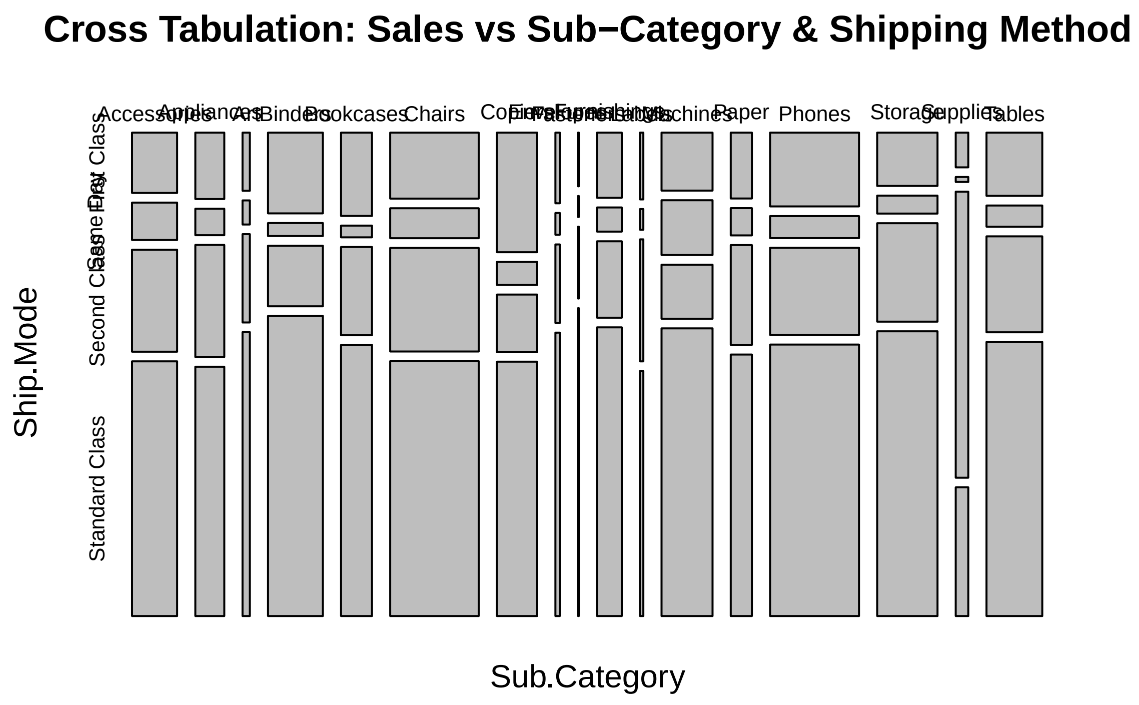
On the left hand side of the formula, we may optionally specify a vector. This allows us to examine the relationship between the explanatory variables (`Sub.Category` and `Category`) and a response variable, say in this case, `Sales`.

```
xtabs(Sales ~ Sub.Category + Category, retail)
```

```
#>              Category
#> Sub.Category  Furniture Office Supplies Technology
#>   Accessories      0.00            0.00  167380.32
#>   Appliances       0.00       107532.16       0.00
#>   Art              0.00        27118.79       0.00
#>   Binders          0.00       203412.73       0.00
#>   Bookcases   114880.00            0.00       0.00
#>   Chairs      328449.10            0.00       0.00
#>   Copiers          0.00            0.00  149528.03
#>   Envelopes        0.00        16476.40       0.00
#>   Fasteners        0.00         3024.28       0.00
#>   Furnishings  91705.16            0.00       0.00
#>   Labels           0.00        12486.31       0.00
#>   Machines         0.00            0.00  189238.63
#>   Paper            0.00        78479.21       0.00
#>   Phones           0.00            0.00  330007.05
#>   Storage          0.00       223843.61       0.00
#>   Supplies         0.00        46673.54       0.00
#>   Tables      206965.53            0.00       0.00
```

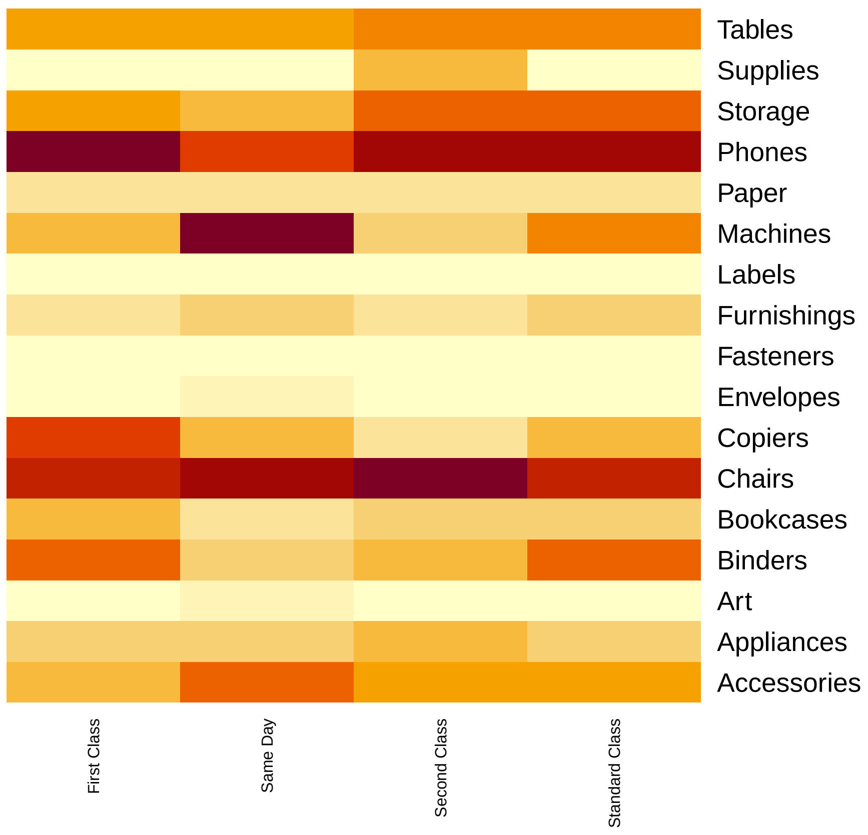
We can wrap the above code in a `plot()` function, and R will plot the cross-tabulation for us. Just to change things up a little, I'm plotting the cross tabulation of sales as explained by `Sub.Category` and `Ship.Mode` instead. I've also added a main title for our plot using the `main` parameter:

```
plot(xtabs(Sales ~ Sub.Category + Ship.Mode, retail), main="Cross Tabulation: Sales vs Sub-Category & Sh
```



Another way to visualize the cross-tabulation above is through the use of heatmap. In R a heatmap is created using the `heatmap` function, so all we need to do is to swap the `plot()` function above with `heatmap()`. I'd also set the heatmap to scale in the column direction - this makes the heatmap output more sensible:

```
heatmap(xtabs(Sales ~ Sub.Category + Ship.Mode, retail), Colv = NA, Rowv = NA, cexCol = 0.6,scale = "co
```



Just like how there's more than one way to create a visual representation of our cross-tabulation data, there are also more than one way to summarize data across multiple variables. We've learned about cross-tabulation using `xtabs` earlier but another equally common statistical tool is the aggregate function, using `aggregate`. The function call is almost the same as `xtabs` except is requires an additional parameter, which is the function we want to use for the aggregation:

```
aggregate(Sales ~ Category + Sub.Category, retail, sum)
```

```
#>           Category Sub.Category     Sales
#> 1       Technology  Accessories 167380.32
#> 2  Office Supplies   Appliances 107532.16
#> 3  Office Supplies          Art  27118.79
#> 4  Office Supplies      Binders 203412.73
#> 5        Furniture    Bookcases 114880.00
#> 6        Furniture       Chairs 328449.10
#> 7       Technology      Copiers 149528.03
#> 8  Office Supplies    Envelopes  16476.40
#> 9  Office Supplies    Fasteners   3024.28
#> 10       Furniture  Furnishings  91705.16
#> 11 Office Supplies       Labels  12486.31
#> 12      Technology     Machines 189238.63
#> 13 Office Supplies        Paper  78479.21
#> 14      Technology       Phones 330007.05
#> 15 Office Supplies      Storage 223843.61
#> 16 Office Supplies     Supplies  46673.54
#> 17       Furniture       Tables 206965.53
```

Compare that to the first few rows of results we obtained from `xtabs()`:

```
head(xtabs(Sales ~ Sub.Category + Category, retail))
```

```
#>              Category
#> Sub.Category  Furniture Office Supplies Technology
#>   Accessories      0.00            0.00  167380.32
#>   Appliances       0.00       107532.16       0.00
#>   Art              0.00        27118.79       0.00
#>   Binders          0.00       203412.73       0.00
#>   Bookcases   114880.00            0.00       0.00
#>   Chairs      328449.10            0.00       0.00
```

Dive Deeper: Analyzing profitability by Category and Shipment Mode

Supposed you were assigned by the company to identify the type of transactions that result in the highest profit on average as well as the ones that result in the biggest losses (or lowest profit) per transaction, how would you go about it?

Use the `aggregate()` function with `Sub.Category` and `Ship.Mode`, but replace the `sum` with `mean` so the function finds the "average" profit instead of total profit from each group instead. If you did this correctly, you should observe that Copiers are great profit makers, and that customers that ship Copiers on First Class bags an average profit in excess of $1,200 per transaction. Sweet!

- What are the top 6 groups measured by average profit? Use the `mean` for this.
- What the bottom (worst) 6 groups measured by average profit? Use the `mean` for this.
- Use the answer provided at the end of this course book as reference.

Supposed we have no concern about the average transaction nor the shipment mode, we could change the formula in our `aggregate` function to take a much simpler form. The following code sums profit across each sub-category:

```
aggregate(Profit ~ Sub.Category, retail, sum)
```

```
#>    Sub.Category      Profit
#> 1   Accessories  41936.6357
#> 2    Appliances  18138.0054
#> 3           Art   6527.7870
#> 4       Binders  30221.7633
#> 5     Bookcases  -3472.5560
#> 6        Chairs  26590.1663
#> 7       Copiers  55617.8249
#> 8     Envelopes   6964.1767
#> 9     Fasteners    949.5182
#> 10  Furnishings  13059.1436
#> 11       Labels   5546.2540
#> 12     Machines   3384.7569
#> 13        Paper  34053.5693
#> 14       Phones  44515.7306
#> 15      Storage  21278.8264
#> 16     Supplies  -1189.0995
#> 17       Tables -17725.4811
```

And we can confirm the above by summing across the row values in our `xtabs` as well, using a handy function called `rowSums`:

```
as.data.frame(rowSums(xtabs(Profit ~ Sub.Category + Ship.Mode, retail)))
```

```
#>             rowSums(xtabs(Profit ~ Sub.Category + Ship.Mode, retail))
#> Accessories                                                41936.6357
#> Appliances                                                 18138.0054
#> Art                                                         6527.7870
#> Binders                                                    30221.7633
#> Bookcases                                                  -3472.5560
#> Chairs                                                     26590.1663
#> Copiers                                                    55617.8249
#> Envelopes                                                   6964.1767
#> Fasteners                                                    949.5182
#> Furnishings                                                13059.1436
#> Labels                                                      5546.2540
#> Machines                                                    3384.7569
#> Paper                                                      34053.5693
#> Phones                                                     44515.7306
#> Storage                                                    21278.8264
#> Supplies                                                   -1189.0995
#> Tables                                                    -17725.4811
```

R Scripts and Reproducible Research

If you are new to writing code but you've scored at least 2 of the 3 quizzes in this coursebook - congratulations! We'll now finish strongly by attempting one of the two learn-by-building modules. As this is a graded task for our Academy students, completion of the task is not optional and count towards your final score. You can choose to complete either of the following task:

R Script to clean & transform the data
Write a R script containing a function (name the function however way you want) that reads `retail.csv` as input, perform the necessary transformation and export a cross-tabulation numeric result OR plot as output. This is the base requirement but more advanced students are free to customize their script to add any extra functionalities.

```
# Sourcing the scipt and running the function should print a cross-tabulation result or plot
source("lbb1.R")
crstab()
```

For graders: Student scores a maximum 2 out of (2) possible points. Check that the R script executes and return a cross tabulation plot (`plot(xtabs())`) with no errors, warnings or missing variables / values.

Reproducible Data Science
Create an R Markdown file that combines your step-by-step data transformation code with some explanatory text. Add formatting styles and hierarchical structure using Markdown.

For graders: Student scores a maximum 2 out of (2) possible points. Check that the RMD file compiles to HTML with at least **two** headings, **two** explanatory paragraph, and the final output is a business recommendation written in English or Bahasa Indonesia on profitable categories.

Writing your code as R scripts make all of these metrics possible for further automation and integration with other tools and services, while writing a R Markdown presents your findings and recommendations in a way that is friendly to non-technical / managerial team members.

Tips on writing R Scripts and functions

As an example, here's how you can write a function, named "weeklyreport":

```r
library(skimr)
library(dplyr)
weeklyreport <- function(){
  retail <- read.csv("data_input/retail.csv") %>%
  group_by(Segment) %>%
  skim(Category, Profit)
}
```

And now you can call the function you created:

```r
weeklyreport()
```

Annotations