Data Manipulation Techniques with R

Contents

Preface

Structure of the book

The book is broken down into small sections that aim to demonstrate a single concept at a time. This book is a work in progress. Submit any issues here. Please check back for frequent updates.

Acknowledgements

We would like to thank Aaron Baker for providing the foundation for this work, and also Nic Larsen for all his feedback and suggestions.

1 Introduction to R

1.1 Getting Started

The simplest way to use R is to use it as if it were a calculator. For example, if we want to know what one plus one is, you may type:

```
1 + 1
```

We can use any arithmetic operator, like addition, subtraction, multiplication, divison, exponentiation, and modulus operations:

```
# addition
1 + 1

# subtraction
6 - 4

# multiplication
2 * 2

# division
10 / 5
10.2 / 5

# integer divison
15.2 %/% 5
15.7 %/% 5

# modulus
15.2 %% 5
15.8 %% 5

# exponential
2^3
```

R also provides numerous built-in functions to use in calculations, such as natural logs, exponentiation, square root, absolute value:

```
# natural log
log(10)

# exponentiation
exp(2)
```

```r
# square root
sqrt(4)

# absolute value
abs(-4)
```

R includes extensive facilities for accessing documentation and searching for help. This is useful to get more information about a specific function. The `help()` function and `?` help operator in R provide access to the documentation pages for R functions. For example, to get help with the round() function, we submit the following code:

```r
#help() function
help(round)

#? operator
?round()
```

1.2 Variable Assignment

A basic concept in programming is called a variable. A variable allows you to store a value (e.g. 10) or an object (e.g. a function description) in R. We can then further use the variable's name to access the value or the object that is stored within this variable.

For example, you can assign a value 10 to a variable `my_variable`:

```r
my_variable <- 10
```

To print out the value of the variable, you simply type the name of your variable:

```r
my_variable
```

A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number:

```r
# A variable can be operated on
my_variable + 1

# And passed to a function
sqrt(my_variable + 1)

# To reassign a variable, just reassign in
my_variable <- 3000

# You can also operate on and reassign a variable to itself
my_variable <- my_variable + 1
```

You can broaden assignments beyond numbers:

```r
result <- sqrt(9)

fruit_1 <- "apple"
fruit_2 <- "banana"
fruit_3 <- "cantaloupe"
```

1.3 Finding Variables

To know all the variables currently available in the workspace we use the `ls()` function:

```
print(ls())
```

The `ls()` function can also use patterns to match the variable names.

```
# List the variables starting with the pattern "var".
print(ls(pattern = "var"))
```

1.4 Deleting Variables

Variables can be deleted by using the `rm()` function. Below we delete the variable my_variable:

```
my_variable <- 5
rm(my_variable)
```

All the variables can be deleted by using the `rm()` and `ls()` function together:

```
rm(list = ls())
```

Another common way to remove all variables in the R environment is to click on the little broom icon next to the button `Import Dataset` under the Environment tab. One can also go up to `Session` and do `Restart R` or `New Session` (the `Restart R` and `New session` options might come in handy for when programs crash).

2 Data Structures

2.1 Vectors

A vector is the simplest type of data structure in R. Vectors are particularly important as most of the functions you will write will work with vectors. Simply put, a vector is a sequence of data elements of the same basic type.

We can construct a vector using the combine `(c)` function:

```
# Here is a vector containing three numeric values 3, 5 and 7:
numbers <- c(3, 5, 7)

#Here is a vector of logical values:
logical <- c(TRUE, FALSE, TRUE, FALSE, FALSE)

#Here is a vector of character values:
fruits <- c("apple", "oranges", "banana")
```

We can also change the value in a vector:

```
fruits[2] <- "strawberries"
```

Every vector has two key properties: its type and its length:

```
typeof(fruits)
length(fruits)
```

You can combine vectors using the combine function as well:

```
results <- c(1, 2, 3)
other_results <- c(4, 5, 6)
combined_results <- c(results, other_results)
```

Vectors can be added together in an element-wise operation:

```
total_add <- results + other_results
total_div <- results / other_results
total_mod <- results %% other_results
```

Functions can be applied to each element of the vector. However, not all functions are vectorized (we will cover this later):

```
total_rou <- round(total_div, 1)

names <- c("Simmons", "Race", "Healey", "LaBarr", "Villanes")
names <- sort(names, decreasing = TRUE)
```

There are also functions for constructing useful types of vectors:

```
# Replicate - replicates a value X number of times
identity_vector <- rep(1,10)
identity_vector

# Sequence operator ":" From:To
sequence_vector <- 1:10

# Sequence function - allows sequencing by a value
sequence_by_two_vector <- seq(0,10, 2)
sequence_by_two_vector

# Sample function - take a random sample from a vector
random_vector <- sample(sequence_by_two_vector,3)

#A common trick to randomly permute all elements in a vector is sample(vector_name)
permute <- sample(numbers)
```

Finally, a vector doesn't have to be made with consistent elements, but it will force them to one type:

```
values <- c("IAA", 1, "2021", 5)
typeof(values)
```

As we mentioned earlier, the elements of a vector can only be of one type. If we want elements of different types… we will need a list.

2.2 Lists

Lists are objects which contain elements of different types like – numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. A list is created using list() function.

```
# Here are 3 vectors of different types
v1 <- c("apple", "banana")
v2 <- c("dog", "cat", "bunny", "pig", "cow", "horse")
v3 <- seq(0,10,by=2)

v1;v2;v3
```

We can put all of these vectors into a list:

```
# Here are 3 vectors of different types
l1 <- list(v1, v2, v3)

class(l1)
typeof(l1)
```

List elements can be accessed via index. Each of the components of a list is accessed via the double bracket [[x]] syntax:

```
# First element in the list
l1[[1]]

# First element in the first element (vector) of the list
l1[[1]][1]
```

List elements can be named:

```
names(l1) <- c("Fruit", "Animals", "Even_Numbers")
```

Or they can be named upon creating the list:

```
l1 <- list(Fruit=v1, Animals=v2, Even_Numbers=v3)
```

The elements in a list can be retrieved by name:

```
l1$Fruit
l1$Fruit[1]

l1$Even_Numbers
max(l1$Even_Numbers)
```

A list can even contain lists:

```
l2 <- list(Odd_Numbers=seq(1,10,by=2), list1=l1)

# First Vector
l2$Odd_Numbers
l2[[1]]

# l1 within L2
l2$list1
l2[[2]]

l2$list1$Fruit
l2[[2]][1]

l2$list1$Fruit[1]
l2[[2]][[1]][1]

l2[[2]][[1]][3] <- "blueberry"
l2$list1$Fruit
l2
```

To combine lists:

```
l4 <- list(More_Fruit=c("melon", "orange"))
l1 <- c(l1,l4)
```

Finally, if we want to update part of a list, we can just operate on and update it accordingly:

```
l1$Fruit <- c(l1$Fruit, l4$More_Fruit)
l1$Fruit
```

2.3 Arrays

Arrays are similar to vectors, except that they are multi-dimensional. An array is created using the array() function, and the dim parameter to specify the dimensions:

```
# 1 dimension - 1 row (vector)
sequence_array_1_dim <- array(1:10, dim=10)
sequence_array_1_dim

# 2 dimensions - 2 rows, 5 columns
sequence_array_2_dim <- array(1:10, dim= c(2,5))
sequence_array_2_dim

# 3 dimensions - 2 rows, 5 columns, 2 tables
sequence_array_3_dim <- array(1:20, dim = c(2,5,2))
sequence_array_3_dim

# 4 dimensions - 2 rows, 5 columns, 2 tables, 2 sets
sequence_array_4_dim <- array(1:40, dim = c(2,5,2,2))
sequence_array_4_dim
```

Similar to vectors, functions can be applied to arrays:

```
dim(sequence_array_2_dim)
length(sequence_array_2_dim)
sum(sequence_array_2_dim)
max(sequence_array_2_dim)
```

2.4 Matrices

Matrices are objects in which the elements are arranged in a two-dimensional rectangular layout. A matrix is created using the matrix() function:

```
# A matrix is a 2-demensional (row x column) array
A <- matrix(1:25, 5, 5)

B <- matrix(1:25, 5, 5, byrow=TRUE) #if you want to fill down the row instead of across the columns

C <- matrix(1:5, 1, 5)

D <- matrix(1:5, 5, 1)
```

Matrices are convenient because we can do linear algebra with them:

```
# Element-wise operators
A + B
A - B
A * B
A / B

# Matrix Multiplication
C %*% D

# Transpose
C
t(C)
```

We can also name the matrix columns and and rows:

```
colnames(A) <- c("Column_1", "Column_2", "Column_3", "Column_4", "Column_5")
A

colnames(A)[4] <- "Changed_It"
A

rownames(A)  <- c("Row_1", "Row_2", "Row_3", "Row_4", "Row_5")
A
```

We can also use some helpful functions for matrices:

```
colSums(A)
rowSums(A)
sum(A)
dim(A)
length(A)
```

And we can extract information from a matrix using similar index notation (row, column):

```
A[2,1] # Single element

A[,1] # All rows in the first column
A[,c(1,2)] # All rows for columns 1 and 2

A[1,] # All columns in the first row
A[c(1,2),] # All columns for rows 1 and 2

A["Row_1","Column_2"]
```

2.5 Data Frames

Data Frames are data displayed in a format as a table. A data frame is essentially a 2 dimensional matrix (row x column). You can think of it like an excel table or relational database table. Data Frames can have different types of data inside it, and we use the data.frame() function to create a data frame:

```
# Create a data frame called dt1 from a set of vectors:
dt1 <- data.frame (
  names =  c("Simmons", "Race", "Healey", "LaBarr", "Villanes"),
  class = c("Time Series", "Linear Algebra", "Visualization", "Finance", "Programming"),
  female = c(1,1,0,0,1)
)
```

To view the data frame:

```
# View the table
View(dt1)

# Head of the table
head(dt1, 3)
```

To change the column names:

```
colnames(dt1)
colnames(dt1) <- c("Last_Name", "Class_Taught", "Female?")
colnames(dt1)
colnames(dt1)[2] <- "Classes_Taught"
```

```
colnames(dt1)
```

To reference a column by name:

```
dt1$Last_Name
dt1$Classes_Taught
```

Data frame indexes - similar to vectors and arrays. We can perform the following actions:

```
# Retrieve a row
# leaving the row or column blank means "everything"
# the negative operator "-" means "everything but"
dt1[1,]
dt1[2:5,]
dt1[-1,]

# Retrieve a column
dt1[,1]
dt1[,-3]
dt1[,c(1:2)]
dt1[,"Last_Name"]
dt1[,c("Last_Name","Female?")]

# Retrieve an element
dt1[5,2]

# Reassign an element
dt1[5,2] <- "R Programming"
dt1[5,]

#Find elements that meet a condition
dt1$`Female?`==1
dt1$Last_Name=='Race'

# Filter by an value
dt1[dt1$`Female?`==1,]
dt1[dt1$Last_Name=='Race',]

# Add a new row - rbind (row-bind) function. The cbind function is its column version.
dt1 <- rbind(dt1, c("Larsen", "Teaching Assistant", 0))
dt1

# Add a new column
dt1$First_Name <- c("Susan", "Shaina", "Christopher", "Aric", "Andrea", "Nicholas")

# Change the order of columns
dt1 <- dt1[,c(4,1:3)]

# Make columns based off other columns
dt1$First_Name_Length <- nchar(dt1$First_Name)
dt1$Last_Name_Length <- nchar(dt1$Last_Name)
dt1$Total_Name_Length <- dt1$First_Name_Length + dt1$Last_Name_Length + 1
dt1$Full_Name <- paste(dt1$First_Name, dt1$Last_Name, sep=" ")
dt1
```

```
# Remove some intermediate step columns
dt1 <- dt1[,-c(5,6)]
```

2.6 Working with different types

We can examine the class of each object using the class function:

```
total_add
class(total_add)

names
class(names)

l2
class(l2)

sequence_array_1_dim
class(sequence_array_1_dim)

A
class(A)

dt1
class(dt1)

dt1$First_Name
class(dt1$First_Name)
```

Finally, we can also do some conversions between them using the "as" functions:

```
new_array <- as.array(total_add)
class(new_array)

old_vector <- as.vector(new_array)
class(old_vector)

dfA <- as.data.frame(A)
class(dfA)
```

3 Programming Fundamentals

3.1 Logicals

A logical is a binary representation of True and False:

```
a <- TRUE

b <- FALSE

typeof(b)
```

Logicals are used for evaluating comparisons, such as:

```
#equality
# note the double equals == operator is a logical comparison, opposed to a single equal = being an as
```

```r
2==2
typeof(2==2)

'cat' == 'dog'

'cat' == 'cat'

#Not equal operator
2!=2

# greater than/less than
2>2

2>=2

2<1

2<=1

# Null Values
V1 <- 1
V1[10] <- 10

is.na(V1)
!is.na(V1)

# Contained within set
V1

1 %in% V1
10 %in% V1
2 %in% V1


c(1,2) %in% V1

# If we want not in, then use the not ! operator around the entire statement
!(1 %in% V1)

# Note, logicals can be used with other data types as well
a <- c(1,2,3,4,5)
a <= 1

b <- c(1,2,7,9,5)
a == b
a != b

A <- matrix(1:10,2,5)
B <- matrix(seq(1,20,2),2,5)

A == B
A != B
```

```
identical(a,b)
identical(A,B)
```

3.2 Conditionals

Conditionals are expressions that perform different computations or actions depending on whether a predefined boolean condition is TRUE or FALSE. Conditionals allow us to control the flow of execution. We use logicals to evaluate conditional statements. Conditionals include if, else, ifelse.

The syntax of if statement is:

```
if (test_expression) {
statement
}
```

If the test_expression is `TRUE`, the statement gets executed. But if it is `FALSE`, nothing happens. For example:

```
x <- 5
if(x > 0){
print("Positive number")
}
```

If you will be executing a single statement after the conditional, you can withold the { }

```
x <- 5
if(x > 0) print("Positive number")
```

If we want code to execute when the test_expression is `FALSE`, then we add an else statement:

```
if (test_expression) {
statement
} else {
statement2
}
```

For example:

```
x <- -5
if(x > 0){
print("Non-negative number")
} else {
print("Negative number")
}
```

Note that you can use multiple logicals in the test_expression, such as:

- Or operator |
- And operator &

For example:

```
mean_values <- 3.1
std_dev_values <- 1.95

if (mean_values>=0 & std_dev_values>=1) {
    print(paste("The mean", mean_values, "is above 0, and the standard deviation", std_dev_values, "is
}
```

Additionally, you can use the ifelse function to quickly assign a value based on a condition. The syntax for the ifelse function is `ifelse(test_expression, yes, no)`. For example:

```r
std_dev_vector <- c(1.2,0.8,0.3,2.4)
ifelse(std_dev_vector>=1, "above one", "below one")
```

3.3 Loops

Occassionally, we need perform a task in an iterative cycle, also known as a loop. According to the R base manual, among the control flow commands, the loop constructs are `for`, `while` and `repeat`, with the additional clauses `break` and `next`.

A for loop is used for iterating over a sequence (they iterate a defined number of times):

```r
sequence <- seq(0,50,5)

#Note this will iterate 11 times, the number of elements in "sequence"
for(i in 1:length(sequence)){
  print(paste("Now at iteration", i, ", value of sequence is", sequence[i]))
  Sys.sleep(0.50)
}
```

By default, R will iterate over all the elements in a vector without needing to designate a number

```r
for(value in sequence){
  print(paste("Value of sequence is", value))
  Sys.sleep(0.50)
}
```

With the break statement, we can stop the loop before it has looped through all the items:

```r
for(i in 1:length(sequence)){
  if(i>5) break
  print(paste("Now at iteration", i, ", value of sequence is", sequence[i]))
  Sys.sleep(0.50)
}
```

With the next statement, we can skip an iteration without terminating the loop:

```r
for(i in 1:length(sequence)){
  if((i %% 2)!=0) next
  print(paste("Now at iteration", i, ", value of sequence is", sequence[i]))
  Sys.sleep(0.50)
}
```

The while loops are used to loop until a specific condition is met:

```r
while (test_expression)
{
statement
}
```

Here, test_expression is evaluated and the body of the loop is entered if the result is TRUE. The statements inside the loop are executed and the flow returns to evaluate the test_expression again. This is repeated each time until test_expression evaluates to FALSE, in which case, the loop exits. For example:

```r
#While loops - iterate so long as the condition is true
#With while loops, it's important to set your starting conditions
continue <- TRUE
i <- 0
while(continue==TRUE){
  i = i + 1
```

```
  print(paste("At iteration", i, "continue still set to", continue))
  if(i>=5){
    continue <- FALSE
    print(paste("Iteration", i, "reached. Continue set to", continue))
    print(paste("While loop terminated upon reaching iteration", i))
  }
  Sys.sleep(0.50)
}
```

A repeat loop is used to iterate over a block of code multiple number of times. There is no condition check in repeat loop to exit the loop. We must ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. Failing to do so will result into an infinite loop.

```
repeat {
statement
}
```

For example:

```
x <- 1
repeat {
print(x)
x = x+1
if (x == 6){
break
}
}
```

Finally, we can nest loops of the same type or different types - for example, for every value of i it will do a loop of n:

```
continue <- TRUE
i <- 0

while(continue==TRUE){
  i = i + 1
  print(i)
  if(i>=5) continue <- FALSE

  for(n in 1:3){
    print(paste("The value of i is", i, ",and the value of n is", n))
    Sys.sleep(0.25)
  }

}
```

3.4 Functions

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.

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, arg_n) {
   Function_Body
}
```

An example of a custom function is the following:

```
numbers <- seq(1:10)

calculator <- function(data, type){
  if(type=="mean"){
    value <- mean(data)
  } else if (type=="sum"){
    value <- sum(data)
  } else if (type=="min"){
    value <- min(data)
  } else {
    value <- "function not in calculator"
  }
  return(value)
}

calculator(numbers, "mean")
calculator(numbers, "sum")
calculator(numbers, "min")
calculator(numbers, "max")
```

To provide a default in the parameters, use the equal sign:

```
calculator <- function(data, type='mean'){
  if(type=="mean"){
    value <- mean(data)
  } else if (type=="sum"){
    value <- sum(data)
  } else if (type=="min"){
    value <- min(data)
  } else {
    value <- "function not in calculator"
  }
  return(value)
}

calculator(numbers)
```

If we want to return multiple values, we need store them in a vector or list and return that object:

```
univariate_summary <- function(data){
  ntot <- length(data)
  mean <- mean(data)
  std <- sd(data)
  percentiles  <- quantile(data, probs=c(0, 0.25, 0.50, 0.75, 1))

  l1 <- list(n_obs=ntot, mean=mean, std=std, percentiles=percentiles)
  return(l1)
}

results <- univariate_summary(numbers)
results$percentiles
results$std
```

4 Data Wrangling

4.1 Libraries

Up until now, we haven't used packages - only functions available in base R. However, the power of R truly exists in the libraries, which are sets of functions that any user can create and share.

As of June 2019, there were over 14,000 packages available on the Comprehensive R Archive Network, or CRAN, the public clearing house for R packages. CRAN lists all libraries here: https://cran.r-project.org/web/packages/available_packages_by_name.html

To install a library, you can either:

- Packages --> Install --> Name
- Use the function install.packages()

Once a package is installed, you need to load it using the library or require function.

These are the libraries we'll be using today. Install if you don't have:

```
install.packages(c("data.table","dplyr","stringr"))
```

Now, import the libraries:

```
library(data.table)
library(dplyr)
library(stringr)
```

4.2 Importing & Exporting Data

Before importing data, we generally want to set the working directory. This is a folder where we will be accessing or storing data and outputs:

```
setwd("C:/Users/avillan/Documents/Class 2022/Class_Code")
```

Another useful function is `getwd()`. This function returns an absolute filepath representing the current working directory of the R process, or `NULL` if the working directory is not available:

```
getwd()
```

To import data, we can use the `read.` function

```
df_auto <- read.csv("Auto_MPG.csv", stringsAsFactors = FALSE)
```

We can check the data set:

```
class(df_auto)
View(df_auto)
dim(df_auto)
nrow(df_auto)
summary(df_auto)
```

Note that Horsepower is a character class

```
typeof(df_auto$Horsepower)
df_auto$Horsepower
summary(df_auto$Horsepower)
```

Upon inspection, it read NAs as question marks (a character), so everything became a character. We can fix this by reassigning as numeric:

```
df_auto$Horsepower <- as.numeric(df_auto$Horsepower)
summary(df_auto)
df_auto$Horsepower
summary(df_auto$Horsepower)
```

Recall if you want to fix a column name you can use the colnames() function:

```
colnames(df_auto)
colnames(df_auto)[8] <- "Origin"
colnames(df_auto)
```

If we want to export the data, we can use write.csv. The row.names = FALSE argument will stop the writer from outputting a row number:

```
write.csv(df_auto, file="Auto_MPG_Clean.csv", row.names = FALSE)
```

In addition to importing the data using code, one can also import data into RStudio with point-and-click. Go to `Import Dataset` in the Environment panel and select the relevant format. From there, you can set the datatype of each column, as well as specify whether to include or to skip a column or row. RStudio automatically generates the corresponding code that you can copy and paste for later reproducibility.

4.3 dplyr and tidyverse

One of the most difficult (yet, interesting!) parts of open-source software is the many ways to achieve the same task. This can make recalling functions and syntax difficult, increasing the time to write or read code. Thus, people have started creating packages that structure coding tasks into a simple but comprehensive set of functions. You'll see this referred to as a "grammar". This framework attempts to simplify the majority of required functions into a simple set of "verbs" to remember.

If you're familiar with SQL - think about how easy it is to read and write:

```
SELECT * FROM table WHERE column = 'some_value'
```

In SQL, `SELECT`, `FROM`, `WHERE` are "verbs" that provide the framework for numerous tasks.

The packages plyr, dplyr, ggplot2, stringr (and others) work within this framework. "tidyverse" is a collection of packages that encompases all of these https://www.tidyverse.org/

4.4 Data manipulation - dplyr basics

Select: select columns by name or helper function (choose which variables you want to look at):

```
select(df_auto, MPG, Cylinders)
select(df_auto, -Origin)
select(df_auto, contains("Model"))
```

The base R equivalents are:

```
df_auto$MPG
df_auto[,c('MPG','Cylinders')]
df_auto[,-c('Origin')]
df_auto[,grepl("Model", names(df_auto))]
```

Slice: Select rows by position (choose which rows of data you want):

```
df_auto$row_number <- 1:nrow(df_auto) #adding a new variable called row_number to help us see the ind

slice(df_auto,1:5)
slice(df_auto, 150:n()) #all the rows between 150 and the end
```

```
slice(df_auto, -1:-5) #removes rows between values, equivalent to slice(df_auto, 6:n())

df_auto <- select(df_auto, -row_number) #remove the row_number column
```

The base R equivalents are:

```
df_auto[1:5,]
df_auto[150:nrow(df_auto),]
df_auto[-c(1:5),]
```

Rename: Rename the columns of a data frame

```
colnames(df_auto)

df_auto <- rename(df_auto, Miles_Per_Gallon=MPG) #rename MPG to Miles_Per_Gallon
colnames(df_auto)

df_auto <- rename(df_auto, MPG=Miles_Per_Gallon) #rename back Miles_Per_Gallon to MPG
colnames(df_auto)
```

The base R equivalents are:

```
colnames(df_auto)[1] <- 'Miles_Per_Gallon'
colnames(df_auto)

colnames(df_auto)[1] <- 'MPG'
colnames(df_auto)
```

Filter: Extract rows that meet logical criteria (filter the data frame by one, or multiple, conditions)

```
filter(df_auto, MPG>20)
filter(df_auto, (MPG>20 & Acceleration>20))
filter(df_auto, MPG==max(MPG))
filter(df_auto, MPG==min(MPG))
```

The base R equivalents are:

```
df_auto[df_auto$MPG>20,]
df_auto[(df_auto$MPG>20 & df_auto$Acceleration>20),]
df_auto[which.max(df_auto$MPG),]
df_auto[which.min(df_auto$MPG),]
```

Sample: Randomly select a sample of n size or a proportion

```
sample_n(df_auto, 5)
sample_frac(df_auto, 0.05)
```

The base R equivalents are:

```
df_auto[sample(1:nrow(df_auto),5),]
df_auto[sample(1:nrow(df_auto),ceiling(0.05*nrow(df_auto))),]
```

Mutate: Compute and append one or more new columns (create a new variable - returns the data frame)

```
mutate(df_auto, HP_Per_Cylinder=Horsepower/Cylinders)
mutate(df_auto, Avg_HP_Per_Cylinder = mean(Horsepower/Cylinders, na.rm=TRUE))
```

The base R equivalents are:

```
df_auto$HP_Per_Cylinder <- df_auto$Horsepower / df_auto$Cylinders
df_auto$Avg_HP_Per_Cylinder <- mean(df_auto$Horsepower / df_auto$Cylinders, na.rm = TRUE)
```

About the na.rm=TRUE argument. Some arithmatic functions sum, avg, count, etc. will NA the entire column if they encounter a single NA within that column/vector. Setting na.rm=TRUE will tell the interpreter to "remove" the NA and carry forth with the calculation.

Summarise: Summarise data into single row of values (aggregates the data into a single result. think avg, mean, min, max, etc.)

```
summarise(df_auto,
          Avg_HP_Per_Cylinder=mean(Horsepower/Cylinders, na.rm = TRUE),
          Min_HP_Per_Cylinder=min(Horsepower/Cylinders, na.rm=TRUE)
          )
```

The base R equivalents are:

```
data.frame(
    Avg_HP_Per_Cylinder = mean(df_auto$Horsepower/df_auto$Cylinders, na.rm = TRUE),
    Min_HP_Per_Cylinder = min(df_auto$Horsepower/df_auto$Cylinders, na.rm=TRUE)
  )
```

4.5 dplyr - pipe operator

Consider the previous operations. If we wanted to manipulate our data set in order, we traditionally have to do a reassignment each time:

```
ddf_auto_hold <- rename(df_auto, Miles_Per_Gallon=MPG) #rename

df_auto_hold <- mutate(df_auto_hold, HP_Per_Cylinder=Horsepower/Cylinders) #then create

df_auto_hold <- filter(df_auto_hold, HP_Per_Cylinder>20) #then filter

df_auto_hold <- select(df_auto_hold, Car_Name, HP_Per_Cylinder) #then select

View(df_auto_hold)
```

This is truely cumbersome the more complicated the data manipulation becomes. To avoid this, we can use the pipe operator %>%. The pipe "passes along" the results of one function to the next:

```
df_auto_hold <- df_auto %>% #note the pipe operator
  rename(Miles_Per_Gallon=MPG) %>% #note the first argument is not required, because the pipe is pass
  mutate(HP_Per_Cylinder=Horsepower/Cylinders) %>%
  filter(HP_Per_Cylinder>20) %>%
  select(Car_Name, HP_Per_Cylinder)

View(df_auto_hold)
```

4.6 stringr package

The package stringr is a similar grammar language for manipulating strings. All the functions start with str_

Consider that I want to extract the Car Make and Model from the Car Name:

```
View(df_auto_hold)
```

We note that the first word in Car_Name is the make, and the second word is the model. The str_split function will split the string into a vector of individual tokens:

```
df_auto_hold <- df_auto %>%
  mutate(Car_Make=str_split(Car_Name," "))

View(df_auto_hold)
```

The str_split creates a list, which we can use to try to get the value:

```
df_auto_hold <- df_auto %>%
  mutate(Car_Make=str_split(Car_Name," ")[[1]][1])
View(df_auto_hold)
```

Note, this is treating the entire column, instead of by row. One way to address this is to rowise() by Car_Name, then create the variable:

```
df_auto_hold <- df_auto %>%
  rowwise() %>%
  mutate(Car_Make=str_split(Car_Name," ")[[1]][1])

View(df_auto_hold)
```

An alternative to this is the group_by() function:

```
df_auto_hold <- df_auto %>%
  group_by(Car_Name) %>%
  mutate(Car_Make=str_split(Car_Name," ")[[1]][1])

View(df_auto_hold)
```

With the structure built, I can just pipe on more functions to get additional variables:

```
df_auto <- df_auto %>%
  rowwise() %>%
  mutate(Car_Make=str_split(Car_Name," ")[[1]][1]) %>%
  mutate(Car_Model=paste(str_split(Car_Name," ")[[1]][-1], collapse=" ")) #everything but the first

View(df_auto)
```

Now that we have Car Make and Model, we might want to summarize some information. We saw summarize the entire column earlier, but we can additionally summarize by a group. We use the group_by() to do so:

```
# Average MPG by Make and Model Year
df_auto %>%
  group_by(Car_Make, Model_Year) %>%
  summarise(Average_MPG=mean(MPG)) %>%
  arrange(Model_Year) %>%
  filter(Model_Year==70) %>%
  ungroup(Car_Make, Model_Year)
```

4.7 Combining Data - dplyr

Typically when working with data, especially relational databases, we have information stored in multiple tables. For instance, we have the Auto data set:

```
View(df_auto)

# But we also have data on when a brand started:
Car_Make <- c("amc", "audi", "bmw", "buick", "chevrolet", "datsun", "dodge", "ford")
First_Year <- c(1954, 1910, 1916, 1903, 1911, 1931, 1900, 1903)
```

```
df_auto_start <- data.frame(Car_Make=Car_Make,
                            First_Year=First_Year,
                            stringsAsFactors = FALSE)

View(df_auto_start)
```

We are interested in the relationship between founding year and average MPG for the max year. For this, we need to get my First Year data combined with the Auto MPG Data. This is accomplished via joins.

Dplyr has different join types. The easiest way to think about this is as a Venn-Diagram.

- inner_join - only returns the data where both frames contain the join key(s), removes all non-matching rows
- left_join - maintains all of the data on base table (left/top), and joins matching data from the joining table. NAs created for joining table
- right_join - opposite of left. Maintains all of the data on the joining table and matches from the base table. NAs created for base table
- full_join - returns all data from both tables. NAs created for both tables.

Inner_join:

```
df_auto_hold <- df_auto %>%
  inner_join(df_auto_start, by="Car_Make")

View(df_auto_hold)

# Note, we've effectively filtered a lot of data because the inner_join only keeps matching records.
dim(df_auto)
dim(df_auto_hold)

# Joins can pipe together with other functions just as we expect
df_auto_hold <- df_auto %>%
  inner_join(df_auto_start, by="Car_Make") %>%
  group_by(Car_Make, First_Year) %>%
  summarise(Average_MPG=mean(MPG))

View(df_auto_hold)

plot(df_auto_hold$First_Year, df_auto_hold$Average_MPG)
```

Left_join:

```
df_auto_hold <- df_auto %>%
  left_join(df_auto_start, by="Car_Make")

# note, we've retained all of the records but it's placed "NA" in the column where it couldn't find a
dim(df_auto)
dim(df_auto_hold)

is.na(df_auto_hold$First_Year)

View(df_auto_hold)

df_auto_hold <- df_auto %>%
  left_join(df_auto_start, by="Car_Make") %>%
  group_by(Car_Make, First_Year) %>%
```

```
    summarise(Average_MPG=mean(MPG))

View(df_auto_hold)
```

4.8 Cheat Sheet

We're not going to cover every function here, but what is great about these packages is the the cheatsheets. The community has made to provide reference. I highly recommend downloading them and keeping them close:

- dplyr: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
- stringr: https://github.com/rstudio/cheatsheets/blob/master/strings.pdf
- a whole lot more... https://www.rstudio.com/resources/cheatsheets/

4.9 Extra - sqldf

If you like SQL, there is a package allows you to manipulate data in a in-memory, or permanent database:

```
install.packages("sqldf")
library(sqldf)

df_auto_hold <- sqldf("select * from df_auto where Car_Make='amc'")
View(df_auto_hold)
```

We will learn a lot more about SQL in the Fall semester.

5 Resources

Here is a list of resources that might help you in your R journey:

5.1 Cheat Sheets

- Base R Cheat Sheet
- Advanced R Cheat Sheet
- Data Wrangling with dplyr and tidyr Cheat Sheet
- Work with strings with stringr Cheat Sheet

5.2 Programming Style

- Google's R Style Guide

5.3 Other resources

- R Graph Gallery

5.4 Books

- R for Data Science
- SAS Programming for R Users