**Lesson 1: Variables and Data Structures**

**Segment 1: Fundamentals**

After opening RStudio, it is essential to set up your working environment. Mainly, we would like to set up the working directory and to remove all objects in the current R session, so we can start fresh. The working directory is a default location on your computer that R is pointing at. If you want to save or load a file, you need to know what the current working directory is.

You can either look right above the console in RStudio, or type in:

getwd()

To organize different R projects and to have a special directory for this R lesson, I created a directory called Rlesson on my Desktop. Setting the working directory could be done via graphical user interface (in RStudio) or command line interface:

setwd("~/Desktop/Rlesson")

If you do not change your working directory, later on you may ended up with a cluttered desktop, lost files, or worse.

We would like know if there are any object in the current R session. In RStudio, we can click on the Environment tab to see this list, or simply type:

ls()

## [1] "base.url" "fig.path" "test1" "test2" "test3"

We see that there are three objects, namely test1, test2, and test3. Note that you may see a different list (or an empty list) if there is no object in your current R session.

You may want to delete an object to clear up the working environment. Let's first delete test1:

rm(test1)

If you want to delete every object currently available, you can use ls():

rm(list=ls())

Let's make sure all objects have been removed:

ls()

## character(0)

The character(0) means you do not have any objects in the current R session.

**Segment 2: Variables**

The most basic and crucial element of R would be a variable, which could be assigned a single number, a vector, a matrix, a data frame, and others. Technically speaking, variables can be thought of as containers which refer to any type of objects, such as data structures.

Let's first assign numbers or characters to variables. We can simply assign a single number, such as 42, to a variable:

my.number = 42

This gives the variable my.number a value of 42. You can show the value of my.number with a print command:

print(my.number)

## [1] 42

Note that variable names consist of letters, digits, periods and underscores (\_), and cannot start with a digit. Do not use other special characters or space in a variable name.

Not all values you could want to store in R are numeric. You could store a string of characters using either single or double quotation marks.

chv1 = "hello"

chv2 = "world"

We can print one of these variables:

print(chv1)

## [1] "hello"

Primitively, R could be used as a scientific calculator. For example, we can add two numbers:

6+4

## [1] 10

and we could assign the result of the calculation to a variable:

x = 6+4

print(x)

## [1] 10

A math operation could be done using two numeric variables. For instance, we can create another variable y:

y = 4

x / y

## [1] 2.5

R has many built-in mathematical and statistical functions that are intuitively named and easy to use. You can use exponentiation:

x^2

## [1] 100

or calculate a natural logarithmic value by using a function log():

log(x)

## [1] 2.303

**Segment 3: Vectors**

As R is built for analyzing large data, we must learn how to handle a sequence of numbers or a matrix of numbers. Let’s first look at a sequence of numbers stored in a vector. Instead of storing a single numeric value, we can create a vector consisting of multiple numeric values by using a function c().

v1 = c(1, 5.5, 1e2)

v2 = c(0.14, 0, -2)

This function can also be used to combine two vectors, such as v1 and v2, into a variable v3:

v3 = c(v1, v2)

v3

## [1] 1.00 5.50 100.00 0.14 0.00 -2.00

When we have a vector with more than one value, we can subset the vector using square brackets. Enter an index within square brackets following a variable to retrieve a single value corresponding to this index. Here we look at v3, and we only want to get the second element.

v3[2]

## [1] 5.5

Similarly, you may use a sequence of indices to retrieve a sequence of values corresponding to those indices. Here we look at the second and third elements of vector v3.

v3[c(2,3)]

## [1] 5.5 100.0

Lastly, you can store the output of subsetting into another variable:

v3\_sub = v3[c(2,3)]

A lot of statistical programming in R relies on mathematical operations applied to a vector or a matrix. Basic calculator-like functions may apply to all elements in a given vector. We could add the numeric value 2 to a vector v1:

v1 + 2

## [1] 3.0 7.5 102.0

We could also take the trigonometric function sin and apply it to the vector v1:

sin(v1)

## [1] 0.8415 -0.7055 -0.5064

Now, we have a vector which consists of multiple numeric values and math operations that work element-wise. If needed, we may apply an operation on a subset of a vector:

sin(v1[2])

## [1] -0.7055

Note that an equivalent result can be obtained by firstly applying a desired math operation to a vector and retrieving a subset of the result:

sin(v1)[2]

## [1] -0.7055

You can also perform operations between two vectors. If two vectors are of the same length, corresponding elements will be used. We could multiply v1 and v2:

v1 \* v2

## [1] 0.14 0.00 -200.00

See that the first element of v1

v1

## [1] 1.0 5.5 100.0

and the first element of v2

v2

## [1] 0.14 0.00 -2.00

were multiplied, resulting in the first element of the output. The second element of v1 and the second element of v2 were multiplied, resulting in the second element of the output, and so on.

A dot product, or an inner product, which is a sum of the products of corresponding elements in two vectors, can be computed by

v1 %\*% v2

## [,1]

## [1,] -199.9

Dot products requires that two vectors must have a same length. Otherwise, you will get an error:

v1 %\*% v3

## Error: non-conformable arguments

When in doubt, check the length of a vector. We can look at the length of v1, or of v3:

length(v1)

## [1] 3

length(v3)

## [1] 6

You may wonder what happens when you apply a math function to a character variable we made previously. Conveniently, R will prohibit you from using math operations on a character vector, since it simply does not make sense. For example, we created the character vector chv2 earlier, and we get an error if we try adding a number to it.

chv2 + 10

## Error: non-numeric argument to binary operator

However, it may not be obvious whether a variable is numeric or not. You can verify the class of a variable using the function class.

class(v2)

## [1] "numeric"

class(chv2)

## [1] "character"

Some numeric values may be stored in a character vector. For example, patient numbers could be simply stored for an identification purpose and years may be used as ordinal categories. Whether to assign a number 42 versus a character “42” depends entirely on the context. For example, we could create a variable v4, and assign "10" and "42" as its contents:

v4 = c("10", "42")

Note that double quotation marks make it a character vector. And since R thinks v4 is a character vector, an attempt to apply a math operation will give an error.

v4 / 10

## Error: non-numeric argument to binary operator

If you would like to use numeric values in a character vector, we can tell R to treat v4 as numbers.

v4 = as.numeric(v4)

Now we can check the class of v4, which has changed to numeric, and we can perform numeric operations on it.

class(v4)

## [1] "numeric"

v4 / 10

## [1] 1.0 4.2

Of course, this forcefully changes the class and therefore you should be careful in using this function.

The function summary provides an easy way to get the feel of data. For a numeric vector, we get six descriptive statistics. For instance, summary of v3

summary(v3)

## Min. 1st Qu. Median Mean 3rd Qu. Max.

## -2.00 0.04 0.57 17.40 4.38 100.00

calculates the mean, median, minimum, max, and other summary statistics.

Depending on the class, summary provides different outputs.

summary(chv2)

## Length Class Mode

## 1 character character

R includes a number of built-in functions to compute various statistics. Many of these are only applicable to numeric values. For instance, we could get the mean or variance:

mean(v3)

## [1] 17.44

var(v3)

## [1] 1642

The quantile function provides the 0, 25th, 50th, 75th, and 100th quantile of the vector:

quantile(v3)

## 0% 25% 50% 75% 100%

## -2.000 0.035 0.570 4.375 100.000

Other widely used functions include sum, median, sd, max, min, and others.

sum(v3)

## [1] 104.6

median(v3)

## [1] 0.57

sd(v3) # standard deviation

## [1] 40.52

max(v3)

## [1] 100

min(v3)

## [1] -2

Elements in a vector have names, which you can access using the function names:

names(v3)

## NULL

NULL here implies that the elements in v3 currently do not have names. We can assign names using =:

names(v2) = c("Cat", "Dog", "Rat")

Now we can look at v2 and see that the names are part of the output:

v2

## Cat Dog Rat

## 0.14 0.00 -2.00

or we can simply extract the names by themselves:

names(v2)

## [1] "Cat" "Dog" "Rat"

**Segment 4: Matrix**

Matrices are like two-dimensional vectors, organizing values into rows and columns. For example, each row may represent a patient, whereas each column contains biomedical characteristics. If you pick one row, you would get all information about the particular patient. If you examine one column, you would get one of many biomedical characteristics about all patients in the matrix.

Whereas vectors in previous sections had only one row, indicated by [1], a matrix may contain multiple rows.

Before we create a matrix, let’s quickly look at how a sequence of numbers are generated using a colon. We can create a sequence 1 through 6:

1:6

## [1] 1 2 3 4 5 6

To generate a more complicated series of numbers, we could use a function called seq, standing for "sequence". For example, to generate the sequence of numbers from 1 to 12, incremented by 4, we would do:

seq(from=1, to=12, by=4)

## [1] 1 5 9

We can make a sequence of numbers into a matrix, by using a function matrix. For instance, we can create a matrix with three rows and two columns:

ma = matrix(1:6, nrow=3, ncol=2)

ma

## [,1] [,2]

## [1,] 1 4

## [2,] 2 5

## [3,] 3 6

Let's create another matrix mb:

mb = matrix(7:9, nrow=3, ncol=1)

mb

## [,1]

## [1,] 7

## [2,] 8

## [3,] 9

Note that a matrix cannot contain multiple data types. In our case, ma and mb exclusively contain numeric values.

Sometimes we'd like to combine different matrices and vectors. cbind and rbind functions stand for column binding and row binding. It could be used to combine any combination of vectors and matrices, as long as their lengths and dimensions are comparable. Here, we can bind rows of ma with a new vector:

rbind(ma, c(100, 200, 300))

## Warning: number of columns of result is not a multiple of vector length

## (arg 2)

## [,1] [,2]

## [1,] 1 4

## [2,] 2 5

## [3,] 3 6

## [4,] 100 200

Or we can combine ma and mb into a new matrix:

m = cbind(ma, mb)

m

## [,1] [,2] [,3]

## [1,] 1 4 7

## [2,] 2 5 8

## [3,] 3 6 9

See that the matrix m has columns of ma followed by columns of mb.

Try to row-bind ma and mb. Because ma is a 3-by-2 matrix and mb is a 3-by-1 matrix, R returns an error stating that two matrices do not have the same number of columns.

rbind(ma, mb)

## Error: number of columns of matrices must match (see arg 2)

To extract one value, or a set of values, from a matrix, use square brackets with both row and column indices such as [index of row, index of column]. If we would like to know the element in the first row and the third column:

m[1, 3]

## [1] 7

We can also use a sequence of numbers generated with a colon operator within square brackets. Here I would like to know the values in the first row:

m[1, 1:3]

## [1] 1 4 7

Leaving the "row" spot or the "column" spot empty will extract, respectively, an entire column or an entire row.

m[1, ]

## [1] 1 4 7

We could also get multiple rows or columns. Here I can retrieve the first and second columns

m[, 1:2]

## [,1] [,2]

## [1,] 1 4

## [2,] 2 5

## [3,] 3 6

Importantly, you will get an error if you enter an index of row or column that is out of bounds.

m[5, ]

## Error: subscript out of bounds

The matrix m does not have a fifth row, which leads to an error.

Of course, if you have a large matrix or have recently loaded a matrix, you may want to ask R the number of rows or the number of columns for your matrix. nrow is a function that computes the number of rows of a matrix, ncol computes the number of columns.

nrow(m)

## [1] 3

ncol(m)

## [1] 3

You can also use the function dim, short for dimensions, to return both the number of rows and the number of columns.

dim(m)

## [1] 3 3

Matrices being two-dimensional, we could flip the columns and the rows. Such operation is called transpose and is used often in statistics. Simply use the t function to transpose a matrix:

t(m)

## [,1] [,2] [,3]

## [1,] 1 2 3

## [2,] 4 5 6

## [3,] 7 8 9

Compare it with the original matrix m:

m

## [,1] [,2] [,3]

## [1,] 1 4 7

## [2,] 2 5 8

## [3,] 3 6 9

We see that the first row has become the first column in the transposed matrix.

Sometimes the diagonal elements, which are located at [1,1], [2,2], and so on, may contain significant information about the data. Therefore, R provides a quick way to extract those values, using the function diag.

diag(m)

## [1] 1 5 9

The diag function behaves differently based on an input. As we just saw, with a matrix, diag will return a vector of diagonal elements.

For a single numeric value, it will create an identity matrix, which is a square matrix with 1s in the diagonal positions.

diag(3)

## [,1] [,2] [,3]

## [1,] 1 0 0

## [2,] 0 1 0

## [3,] 0 0 1

For a vector, it will create a diagonal matrix whose diagonal elements are derived from an input vector. The square matrix then would have both the number of rows and columns matching the length of an input vector.

diag(c(1,2,3))

## [,1] [,2] [,3]

## [1,] 1 0 0

## [2,] 0 2 0

## [3,] 0 0 3

Basic math functions from the beginning of this course can be readily applied to matrices. You can add, subtract, multiply, or divide each element in a matrix by a single numeric value.

m + 3

## [,1] [,2] [,3]

## [1,] 4 7 10

## [2,] 5 8 11

## [3,] 6 9 12

m \* 2

## [,1] [,2] [,3]

## [1,] 2 8 14

## [2,] 4 10 16

## [3,] 6 12 18

You can also perform matrix multiplication. Let's create a new matrix, m2, with 3 rows:

m2 = matrix(21:32, nrow=3)

m2

## [,1] [,2] [,3] [,4]

## [1,] 21 24 27 30

## [2,] 22 25 28 31

## [3,] 23 26 29 32

Now we can multiply m with m2:

m3 = m %\*% m2

m3

## [,1] [,2] [,3] [,4]

## [1,] 270 306 342 378

## [2,] 336 381 426 471

## [3,] 402 456 510 564

Note that each element in m3 is a dot product between a row in m and a column in m2.

**Segment 5: Lists and Data Frames**

While matrices are extremely useful for processing and storing a large dataset, matrices have several limitations that may not suit our needs. For example, what do we do if we would like to put together columns of numeric values and of characters in the same structure? Matrices will force numeric values into characters because only one data type is allowed. Both lists and data frames are more flexible data structures and will allow different data types to be assigned to a single variable.

In R, a list is a vector containing other objects which may be of different data types and of different lengths. Let’s combine multiple variables we have created into a single list. To do this we use a function called list, and give it items we have created before:

list(v1,chv2,ma)

## [[1]]

## [1] 1.0 5.5 100.0

##

## [[2]]

## [1] "world"

##

## [[3]]

## [,1] [,2]

## [1,] 1 4

## [2,] 2 5

## [3,] 3 6

We could also assign names to objects within a list:

my.list = list(numeric=v1, character=chv2, matrix=ma)

We can slice a list by its index:

my.list[1]

## $numeric

## [1] 1.0 5.5 100.0

This output is still a list containing the first member:

class(my.list[1])

## [1] "list"

If you would like to extract the content, you need to use *two* sets of square brackets

my.list[[1]]

## [1] 1.0 5.5 100.0

class(my.list[[1]])

## [1] "numeric"

Alternatively, the content of a member in a list can be accessed by its name. We can learn the names of a list with the names function:

names(my.list)

## [1] "numeric" "character" "matrix"

And can use a dollar sign ($) to extract one member of the list:

my.list$matrix

## [,1] [,2]

## [1,] 1 4

## [2,] 2 5

## [3,] 3 6

class(my.list$matrix)

## [1] "matrix"

Data frames are lists with a set of restrictions. Most precisely, a data frame is a list of vectors which are conveniently arranged as columns. All vectors or columns in a data frame must have the same length. With statistical programming in mind, data frames mimic matrices when needed and appropriate. Most functions, such as colnames, cbind, and dim, used for a matrix are also applicable to data frames.

R comes with built-in datasets that can be retrieved by name, using data function. In this class, we are going to utilize mtcars, a dataset built into R.

data(mtcars)

class(mtcars)

## [1] "data.frame"

mtcars contains statistics about 32 cars in 1974, including miles per gallon, weight, number of cylinders, and others. Each row is one car, and each column one of characteristics.

You can see a help file about mtcars with:

?mtcars or help(mtcars)

RStudio provides a functionality to display the data in a spreadsheet, using the View(function).

View(mtcars)

However, you might not want to look at all the data points in a data frame. One of the most useful functions is head, which shows the first 6 rows of a data frame. Since the data may be very large, it is a good way to get an idea of its contents:

head(mtcars)

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1

## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

Just how we obtained summary statistics of a vector, we can apply summary function to a data frame:

summary(mtcars)

## mpg cyl disp hp

## Min. :10.4 Min. :4.00 Min. : 71.1 Min. : 52.0

## 1st Qu.:15.4 1st Qu.:4.00 1st Qu.:120.8 1st Qu.: 96.5

## Median :19.2 Median :6.00 Median :196.3 Median :123.0

## Mean :20.1 Mean :6.19 Mean :230.7 Mean :146.7

## 3rd Qu.:22.8 3rd Qu.:8.00 3rd Qu.:326.0 3rd Qu.:180.0

## Max. :33.9 Max. :8.00 Max. :472.0 Max. :335.0

## drat wt qsec vs

## Min. :2.76 Min. :1.51 Min. :14.5 Min. :0.000

## 1st Qu.:3.08 1st Qu.:2.58 1st Qu.:16.9 1st Qu.:0.000

## Median :3.69 Median :3.33 Median :17.7 Median :0.000

## Mean :3.60 Mean :3.22 Mean :17.8 Mean :0.438

## 3rd Qu.:3.92 3rd Qu.:3.61 3rd Qu.:18.9 3rd Qu.:1.000

## Max. :4.93 Max. :5.42 Max. :22.9 Max. :1.000

## am gear carb

## Min. :0.000 Min. :3.00 Min. :1.00

## 1st Qu.:0.000 1st Qu.:3.00 1st Qu.:2.00

## Median :0.000 Median :4.00 Median :2.00

## Mean :0.406 Mean :3.69 Mean :2.81

## 3rd Qu.:1.000 3rd Qu.:4.00 3rd Qu.:4.00

## Max. :1.000 Max. :5.00 Max. :8.00

See that each column is summarized independently, and for each column, we get the six summary statistics, such as min, max, median, and mean.

As a data frame can be thought of as a list of vectors that have the same length, we can also access each column by their names.

names(mtcars)

## [1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"

## [11] "carb"

Note that this is equivalent to looking at the column names:

colnames(mtcars)

## [1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"

## [11] "carb"

You can retrieve a specific column by name. For instance, we could look just at miles per gallon (mpg):

mtcars$mpg

## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2

## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4

## [29] 15.8 19.7 15.0 21.4

Rather than using the dollar sign ($), we could also access the column like

mtcars[, "mpg"]

## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2

## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4

## [29] 15.8 19.7 15.0 21.4

Alternatively, we can use the column index within square brackets to subset a column from a data frame. This is identical to how we subset a matrix. To get just the first column:

mtcars[, 1]

## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2

## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4

## [29] 15.8 19.7 15.0 21.4

We can obtain multiple rows at once as well:

mtcars[1:3, ]

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

**Segment 6: Logical Vectors and Operators**

Another type of variable is a logical value: TRUE or FALSE. We can create a logical vector or matrix, as well as using mathematical operations, such as inequalities, on numbers to dynamically generate logical variables.

Using the function c, we can make a vector of logical values. Note that TRUE and FALSE are not wrapped by quotation marks.

y = c(TRUE, FALSE, TRUE)

y

## [1] TRUE FALSE TRUE

Note that the class of y is "logical":

class(y)

## [1] "logical"

Note that TRUE and FALSE, being capitalized, are reserved and treated specially by R. Therefore, you can not and should not name your variable TRUE or FALSE.

You can compare numeric values in a vector with any value you choose:

v2 > 0

## Cat Dog Rat

## TRUE FALSE FALSE

If you apply a logical operator to a matrix, it will work on each element. Here we ask which elements of m are greater than or equal to 5.

m >= 5

## [,1] [,2] [,3]

## [1,] FALSE FALSE TRUE

## [2,] FALSE TRUE TRUE

## [3,] FALSE TRUE TRUE

However, what if you want "all automatic cars" from the mtcars dataset, or "all cars with mpg > 20"? We can first ask R which elements of mtcars$mpg is greater than 20:

mtcars$mpg > 20

## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE

## [12] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE

## [23] FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE TRUE

This logical vector can be used to subset rows of the data frame. TRUE means "keep the row", FALSE means drop it. Place this before the comma in the square brackets:

v = mtcars$mpg > 20

efficient.cars = mtcars[v, ]

efficient.cars

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

## Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

## Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2

## Fiat 128 32.4 4 78.7 66 4.08 2.200 19.47 1 1 4 1

## Honda Civic 30.4 4 75.7 52 4.93 1.615 18.52 1 1 4 2

## Toyota Corolla 33.9 4 71.1 65 4.22 1.835 19.90 1 1 4 1

## Toyota Corona 21.5 4 120.1 97 3.70 2.465 20.01 1 0 3 1

## Fiat X1-9 27.3 4 79.0 66 4.08 1.935 18.90 1 1 4 1

## Porsche 914-2 26.0 4 120.3 91 4.43 2.140 16.70 0 1 5 2

## Lotus Europa 30.4 4 95.1 113 3.77 1.513 16.90 1 1 5 2

## Volvo 142E 21.4 4 121.0 109 4.11 2.780 18.60 1 1 4 2

Alternatively, and more concisely, we could also put the expression directly in front of the comma:

efficient.cars = mtcars[mtcars$mpg > 20, ]

You can combine multiple conditions using & (and) or | (or), such as looking for automatic gearshift cars with mpg > 20. Here we provide two conditions: that the mpg is greater than 20, and that the gearshift is automatic:

efficient.auto = mtcars[mtcars$mpg > 20 & mtcars$am == 0, ]

Then we can look at the first few rows of efficient.auto:

head(efficient.auto, 3)

## mpg cyl disp hp drat wt qsec vs am gear carb

## Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

## Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

## Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2

We can confirm they have mpg greater than 20 and am equal to 0.

**Assignments:**

To install Swirl, go to any interactive R terminal and type

install.packages("swirl")

library("swirl")

Then install this class's quizzes with the line:

install\_course\_github("dgrtwo", "RData", branch="quizzes", multi=TRUE)

### Taking a Quiz

Once you've finished one of the videos and you wish to take the associated quiz, you simply start swirl:

library("swirl")

swirl()

You'll be prompted for your name and informed of the basics of how Swirl works. Soon you'll reach a page that looks like:

| Please choose a course, or type 0 to exit swirl.

1: Lesson 1. Variables and Data Structures

2: Lesson 2. Visualizing Data Using ggplot2

3: Lesson 3. Statistical Testing and Prediction

4: Lesson 4. Exploratory Data Analysis with data.table

5: Take me to the swirl course repository!

Selection:

Choose whichever lesson you are currently taking, and you'll get to choose which quiz to take:

| Please choose a lesson, or type 0 to return to course menu.

1: Segment 2.1 Introduction

2: Segment 2.2 Scatter Plots

3: Segment 2.3 Faceting and Additional Options

4: Segment 2.4 Histograms and Density Plots

5: Segment 2.5 Boxplots and Violin Plots

6: Segment 2.6 Input- Getting Data into the Right Format

7: Segment 2.7 Output- Saving Your Plots

Selection:

Simply select the quiz you wish to take and it will start.

# Qualitative Data

# A data sample is called qualitative, also known as categorical, if its values belong to a collection of known defined non-overlapping classes. Common examples include student letter grade (A, B, C, D or F), commercial bond rating (AAA, AAB, ...) and consumer clothing shoe sizes (1, 2, 3, ...).

The tutorials in this section are based on an R built-in data frame named painters. It is a compilation of technical information of a few eighteenth century classical painters. The data set belongs to the MASS package, and has to be pre-loaded into the R workspace prior to its use.

> library(MASS)      # load the MASS package   
> painters   
              Composition Drawing Colour Expression School   
Da Udine               10       8     16          3      A   
Da Vinci               15      16      4         14      A   
Del Piombo              8      13     16          7      A   
Del Sarto              12      16      9          8      A   
Fr. Penni               0      15      8          0      A   
Guilio Romano          15      16      4         14      A   
                    .................

The last School column contains the information of school classification of the painters. The schools are named as A, B, ..., etc, and the School variable is qualitative.

> painters$School   
 [1] A A A A A A A A A A B B B B B B C C C C C C D D D D   
[27] D D D D D D E E E E E E E F F F F G G G G G G G H H   
[53] H H   
Levels: A B C D E F G H

For further details of the painters data set, please consult the R documentation.

> help(painters)

# Frequency Distribution of Qualitative Data

The frequency distribution of a data variable is a summary of the data occurrence in a collection of non-overlapping categories.

#### Example

In the data set [painters](http://www.r-tutor.com/node/19), the frequency distribution of the School variable is a summary of the number of painters in each school.

#### Problem

Find the frequency distribution of the painter schools in the data set painters.

#### Solution

We apply the table function to compute the frequency distribution of the School variable.

> library(MASS)                 # load the MASS package   
> school = painters$School      # the painter schools   
> school.freq = table(school)   # apply the table function

#### Answer

The frequency distribution of the schools is:

> school.freq   
school   
 A  B  C  D  E  F  G  H   
10  6  6 10  7  4  7  4

#### Exercise

1. Find the frequency distribution of the composition scores in painters.
2. Find programmatically the school that has the most painters.
3. Find the relative frequency distribution of the painter schools in the data set painters.

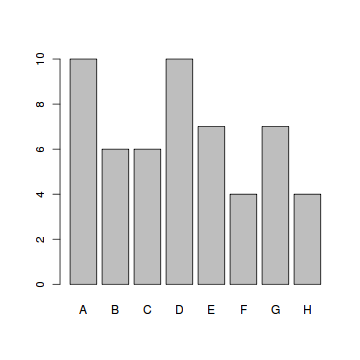
Then we find the sample size of painters with the nrow function, and divide the frequency distribution with it. Therefore the relative frequency distribution is:

> school.relfreq = school.freq / nrow(painters)

We can apply the barplot function to produce its bar graph.

> barplot(school.freq)         # apply the barplot function

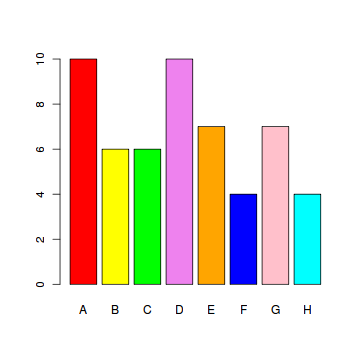
The bar graph of the school variable is:



#### Enhanced Solution

To colorize the bar graph, we select a color palette and set it in the col argument of barplot.

> colors = c("red", "yellow", "green", "violet",   
+   "orange", "blue", "pink", "cyan")   
> barplot(school.freq,         # apply the barplot function   
+   col=colors)                # set the color palette



#### Exercise

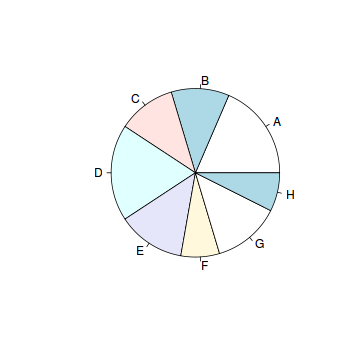
Find the bar graph of the composition scores in painters.

Then we apply the pie function to produce its pie chart.

> pie(school.freq)              # apply the pie function

#### Answer

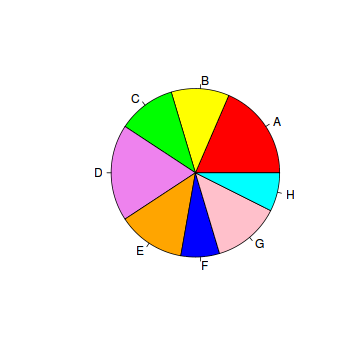
The pie chart of the school variable is:



#### Enhanced Solution

To colorize the pie chart, we select a color palette and set it in the col argument of pie.

> colors = c("red", "yellow", "green", "violet",   
+   "orange", "blue", "pink", "cyan")   
> pie(school.freq,             # apply the pie function   
+   col=colors)                # set the color palette



#### Exercise

Find the pie chart of the composition scores in painters.

#### Problem

Find out the mean composition score of school C in the data set painters.

#### Solution

The solution consists of a few steps:

1. Create a logical index vector for school C.

> library(MASS)                 # load the MASS package   
> school = painters$School      # the painter schools   
> c\_school = school == "C"      # the logical index vector

1. Find the child data set of painters for school C. For explanation, please consult the tutorial of [Data Frame Row Slice](http://www.r-tutor.com/node/17).

> c\_painters = painters[c\_school, ]  # child data set

1. Find the mean composition score of school C.

> mean(c\_painters$Composition)   
[1] 13.167

#### Answer

The mean composition score of school C is 13.167.

#### Alternative Solution

Instead of computing the mean composition score manually for each school, use the tapply function to compute them all at once.

> tapply(painters$Composition, painters$School, mean)   
     A      B      C      D      E      F      G      H   
10.400 12.167 13.167  9.100 13.571  7.250 13.857 14.000

#### Exercise

1. Find programmatically the school with the highest composition scores.
2. Find the percentage of painters whose color score is equal to or above 14.