GC&DS - 03: Variables and Data Frames

Code

1 **Variables/Vectors are Uni-dimensional**

Objects can take on different characteristics. For example, some objects may represent a single value (e.g., a numeric values or a string), some may represent multiple values like a variable or levels of a variable, some may be lists, etc. Consider creating variables.

In `R`, the simplest type of data structure is known as a vector. A vector is a sequence of data elements of the same basic type (e.g., numeric, character, lists, etc.). Members of a vector are called components or elements. You can think of a vector as a variable object. For example, you might have variables representing ages of all your participants, reaction times to stimuli, salaries of people in your company, or IQs of individuals. In many cases, you may have variables in a file and other times you might need to create them. Let's create some vectors to understand their structure. Use `rnorm()` to create a numeric vector object.

Hide

```
iq <- rnorm(n = 1000, mean = 100, sd = 15)  # create a random normal distribution with a leng
th of 1000 values which have a mean of 100 and standard deviation of 15 (e.g., iq distrubtio
n)


iq <- rnorm(1000, 100, 15)      # remember, as long as the order is correct,
                                # you don't need to specify the parameter names

head(iq)                        # look at object head
```

```
## [1] 104.01295  76.04144 102.01924 104.46973  99.75040 123.30198
```

2 Creating Vectors that are Repetitions or Sequences

Let's say you new the first 500 people were female and other 500 were male. Create a vector and put it all into a data frame. Two functions are useful here: `rep()` for creating replications and `seq()` for creating sequences.

Hide

```
c("F", "F", "F", "F", "F")   # combining 5 "F"s is tedious to type
```

```
## [1] "F" "F" "F" "F" "F"
```

Hide

```
rep("F",  times = 5)         # replicate object 5 times
```

```
## [1] "F" "F" "F" "F" "F"
```

Hide

```
rep( c("F", "M"), 5)           # replicate c() vector 5 times
```

```
##  [1] "F" "M" "F" "M" "F" "M" "F" "M" "F" "M"
```

Hide

```
sex <- c(rep("F", 500), rep("M", 500))   # combine with c() and replicate each 500, assign to
sex


head(sex)
```

```
## [1] "F" "F" "F" "F" "F" "F"
```

Hide

```
# or alternatively, create 2 vector objects if that's less confusing
f <- rep("F", 500)
m <- rep("M", 500)



sex <- c(f, m)               # assign the character vectors to the sex vector



seq(from = 1, to = 10)       # use seq() to create a sequence FROM 1 TO  10
```

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

Hide

```
seq(2, 10, by = 2)           # FROM 2 TO 10 BY 2s, dropping from and to arguments
```

```
## [1]  2  4  6  8 10
```

Hide

```
id <- seq(1, 1000)            # assign a sequence of 1 to 1000 to a vector named id



head(id)
```

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

If the number of elements in a vector variable changes, hard coding can be troublesome. Get the `length()` of the sex vector and pass that as the sequence value. This approach is useful for objects that get modified. This approach would be more flexible.

Hide

```
length(sex)                # length will return the length of the vector, including NAs
```

```
## [1] 1000
```

Hide

```
id <- seq(from = 1, to = length(sex) )  # assign a sequence of 1 to 1000 to a vector named id


head(id)
```

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

3 Vectors Can be Different Types

Common vectors types are numeric (integer and double/float) and character. You can check the type using `typeof()`. If you were interested in knowing whether a vector is of a specific type, you can use a set of functions starting with `is.` to ask whether the vector is a specific type (e.g., `is.numeric()` or `is.double()`, `is.integer()`, `is.character()`, etc.). These functions will return a logical `TRUE` or `FALSE`.

Hide

```
is.double(iq)        # is it a double precision/floating point?
```

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

Hide

```
is.numeric(iq)       # is it a set of numeric values?
```

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

Hide

```
is.character(iq)     # is is a set of characters?
```

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

Hide

```
is.integer(iq)       # is it a set of integer values?
```

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

4 Changing Vector Types

You can also change the form of vectors using a set of functions starting with `as.` (e.g., `as.integer()`, `as.character()`, `as.numeric()`). Because IQ scores are integers and not floats containing decimals, let's just change the values created from numeric to an integer using `as.integer()`.

Converting numeric or characters containing numbers to integer…

Hide

```
iq <- as.integer(iq) # make is a character vector


head(iq)             # now integers
```

```
## [1] 104  76 102 104  99 123
```

Converting numeric vectors to character…

Hide

```
iq <- as.character(iq) # make is a character vector


head(iq)             # now the numbers are in quotes representing a string
```

```
## [1] "104" "76"  "102" "104" "99"  "123"
```

Converting character vectors to numeric…

If the characters are numbers…

Hide

```
iq <- as.integer(iq)  # pass an existing vector into as.integer() to convert


head(iq)             # now integers, not floats
```

```
## [1] 104  76 102 104  99 123
```

You can also wrap a function in another function when the initial object is created. Here, `as.integer()` converts the vector returned by `rnorm()` into an integer.

Hide

```
iq <- as.integer(rnorm(1000, 100, 15))  # create initially by wrapping as.integer() around rn
orm()
```

Alternatively, if nesting functions and reading them from the inside out confuses use, you can pipe the object from one function to another using `magrittr`'s pipe operator `%>%`. Just make sure to load the library first, though you should add the function to the top of your file.

Hide

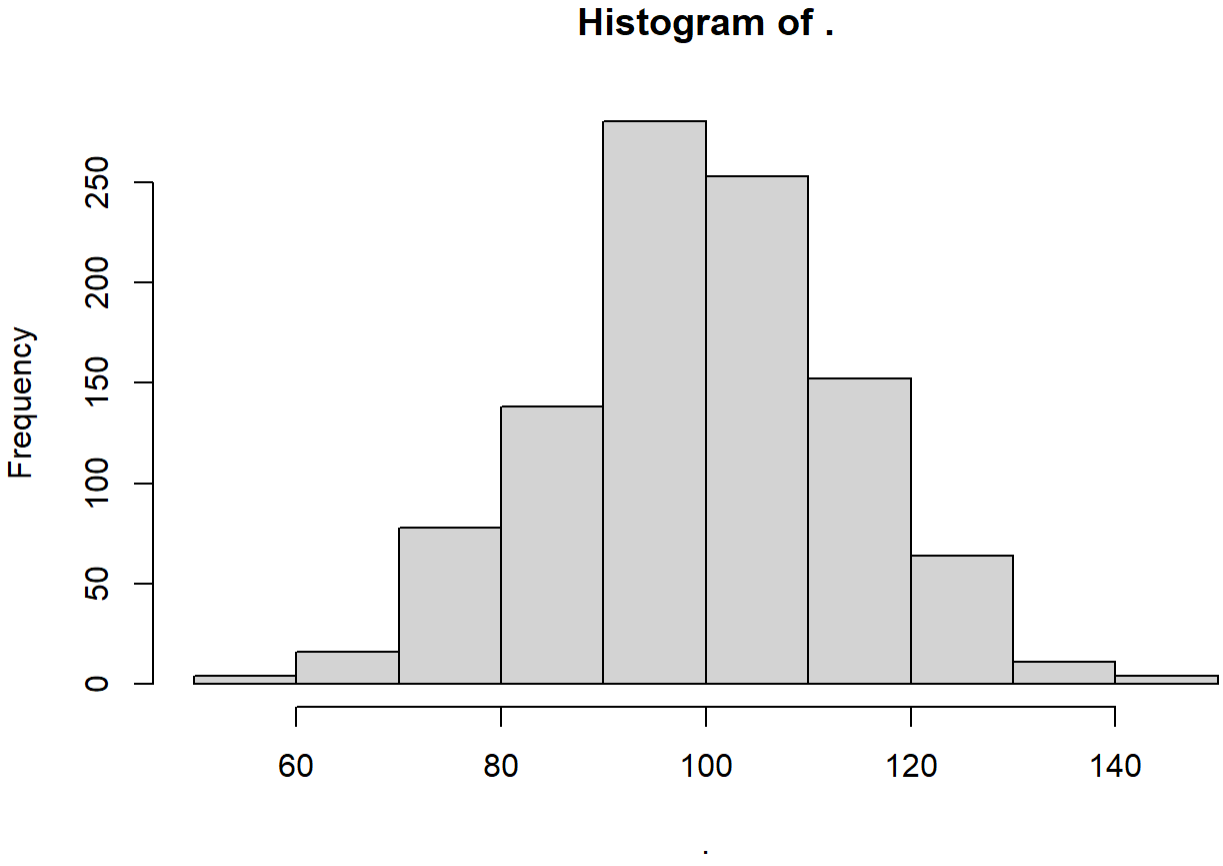
```r
library(magrittr)

iq <- rnorm(1000, 100, 15) %>%    # create the random normal dist
    as.integer()                  # pipe to make integer

head(iq)
```

```
## [1]  84  86  98 116  98  93
```

And to see a plot, use `hist(iq)` or pass the object using the pipe `%>%` :

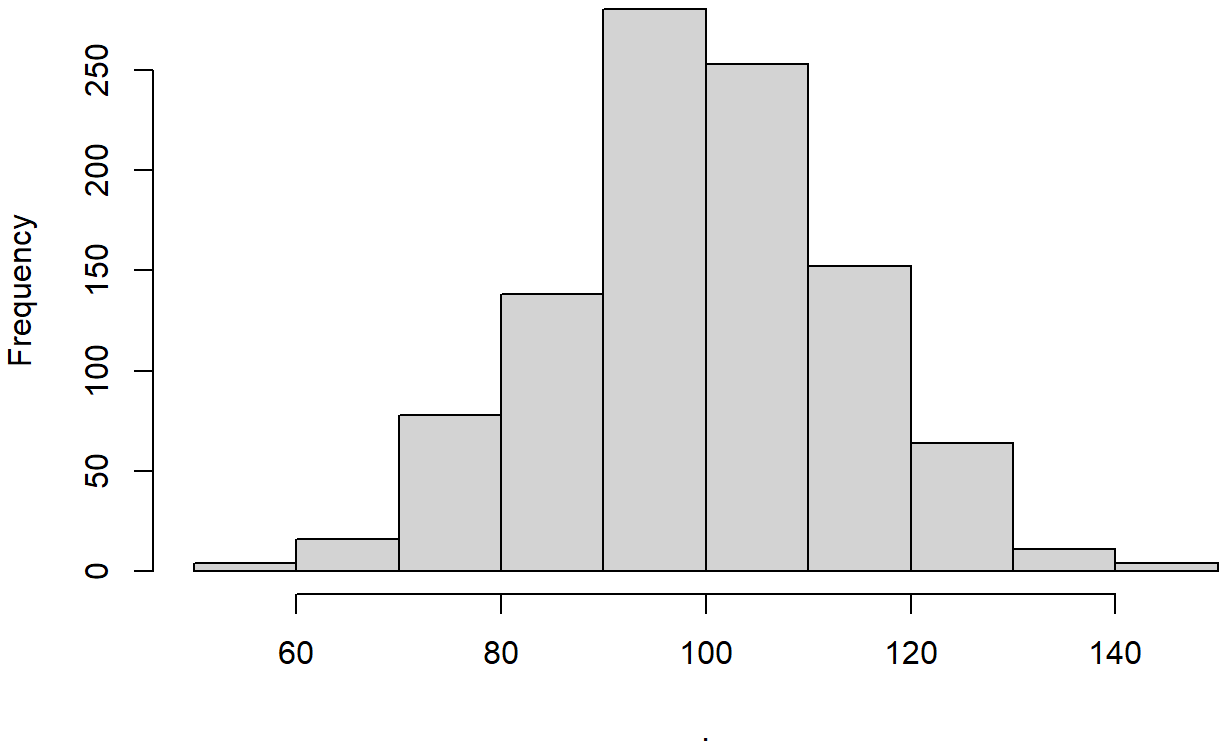
Hide

```r
iq %>% hist(.)    # and pipe the vector to the histogram function in base R
```



Hide

```r
iq %>% hist(., main = "IQ Distribution")   # add a title by passing a string argument to main
```



.

If the characters are not numbers as strings, you'll need a different approach. For example, you can use `ifelse()` or `dplyr::case_when()` to test whether the element of a vector matches some condition and if yes (do one thing), otherwise no (do something else).

- `ifelse()`
- `case_when()`

Hide

```
# convert M and F to 0 and 1. If == M, make 0, else 1. Fine for two groups
ifelse(sex == "M", 0, 1) %>%
  head()
```

```
## [1] 1 1 1 1 1 1
```

When there are more than two groups, *re-coding* can be confusing with `ifelse()` because you'll need to nest an `ifelse()` inside an `ifelse()`.

`dplyr`'s `case_when()` is similar to `ifesle()`,

We need to convert these sloppy data:

Hide

```
temp_sex <- c("M", "m", "F", "f", NA)

dplyr::case_when(
  temp_sex == "M" ~ 0,
  temp_sex == "F" ~ 1
  )
```

```
## [1]  0 NA  1 NA NA
```

But note that if case is inconsistent, `NA`'s will replace strings that you might want recoded. An easy fix for casing is to convert the vector (without assignment) by passing it to `tolower()` or `toupper()` and then perform the logical conversion on the case change elements.

Hide

```
dplyr::case_when(
  toupper(temp_sex) == "M" ~ 0,
  toupper(temp_sex) == "F" ~ 1
  )
```

```
## [1]  0  0  1  1 NA
```

5 Vectors Contain Ordered Components/Elements

Because vectors are uni-dimensional and composed of components or elements, those components have an order or position within the vector. Some value has to be first, some value has to be last, and if there are more than 2, all other elements assume some ordered position between the two. You can examine their elements using `[]`. Behind the scene, this is what R is essentially doing when you call the `iq` object but you can also specify an element's position using a number or set of numbers.

Hide

```
iq[] %>% head()          # inspect all elements w
```

```
## [1]  84  86  98 116  98  93
```

Hide

```
iq[1000] %>% head()      # the element in the 1000th position
```

```
## [1] 93
```

Hide

```
iq[1001] %>% head()      # Nothing is beyond the length of 1000
```

```
## [1] NA
```

Hide

```
iq[1:5]       # the first 5 positions
```

```
## [1]  84  86  98 116  98
```

Hide

```
iq[100:105]   # use : to find the 100th through 105th
```

```
## [1] 110 122 104 111  97  80
```

Hide

```
#iq[100,105]   # not understood by the interpreter



iq[c(100,105)] # use c() to combine positions, like the 100th and the 105th only
```

```
## [1] 110  80
```

Hide

```
iq[c(1:5,100,105)] # use c() to 'combine' different positions,
```

```
## [1]  84  86  98 116  98 110  80
```

Hide

```
                   # like the 1st through 5th, 100th, 105th
```

6 Orders of Elements Can be Reordered

You might want or need to change the order of elements in vectors. One such change involves sorting with `sort()`.

Hide

```
sort(iq) %>% head()                 # sort from lowest to highest
```

```
## [1] 56 58 59 60 61 61
```

Hide

```
sort(iq, decreasing = T) %>% head()  # sort from highest to lowest
```

```
## [1] 149 146 145 141 138 136
```

You can also modify values of elements by referencing them by their index/position and assign that index a new value.

If for example, you found errors for certain positions, say `c(1,52, 99, 108)`, you could set them to `NA`.

Hide

```
iq_backup <- iq   # create a back for illustration


iq[c(1,52, 99, 108)]    # the recorded values
```

```
## [1]  84 129 104  85
```

Hide

```
iq[c(1,52, 99, 108)] <- NA



iq[c(1,52, 99, 108)]   # after reassignment
```

```
## [1] NA NA NA NA
```

Hide

```
iq <- iq_backup        # restore iq using the backup
```

7 Removing an Object from Memory

You can remove any object from memory by passing it to `rm()`. Let's remove the backup object because it's no longer needed.

Hide

```
rm(iq_backup)
```

8 Data Frames are Two-Dimensional

A data frame is a two-dimensional structure in which each column contains values of one variable and each row contains one set of values from each column. Data frames are row and column objects. If you think of a data frame is a bunch of vectors, you can create a data frame from the vectors just created. Use `data.frame()`.

Hide

```
DF <- data.frame(id, sex, iq)      # create data frame with existing objects;
                                   # must all be same length!


head(DF)
```

```
##   id sex  iq
## 1  1   F  84
## 2  2   F  86
## 3  3   F  98
## 4  4   F 116
## 5  5   F  98
## 6  6   F  93
```

Notice that column names of the data frame are inherited by the vector names. If you wanted to change the column names at creation of the data frame, just use `name = vector` .

Hide

```
DF <- data.frame(Id = id, Sex = sex, Iq = iq)  # or if you wanted to assign names to columns
that were not the name of the vectors
```

But you could always just pass a character vector of the same length as `names()` and assign them to the existing names but this requires an extra step.

Hide

```
names(DF) <- c("Id", "Sex", "Iq")    # note that the names are the same so nothing actually c
hanges here
```

9 Two-Dimensional Objects are Not Always Data Frames

As a word of caution, sometimes two-dimensional objects are not data frames. Use `is.data.frame()` to check.

Hide

```
is.data.frame(DF)
```

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

Hide

```
matrix_aint_no_dataframe <- matrix(1:12, nrow = 4, ncol = 3)   # a matrix with 2 dimensions



is.data.frame(matrix_aint_no_dataframe)       # but is not a data frame
```

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

Hide

```
as.data.frame(matrix_aint_no_dataframe) # can be made into a data frame using as.data.frame()
```

```
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

Hide

```
matrix_aint_no_dataframe                # but won't be changed unless you use assignment <-
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

Hide

```
DF_was_a_matrix <- as.data.frame(matrix_aint_no_dataframe)  # assign it to an object



is.data.frame(DF_was_a_matrix)
```

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

10 **Examining the Data Frame**

There are a set of functions you can used to inspect a data frame. For example, you can look its row x column dimensions using `dim()` or examine the structure of the vectors using `str()` or even look at the top or bottom rows using `head()` and `tail()`.

Hide

```
dim(DF)   # returns the numeric vector of Row and Column counts
```

```
## [1] 1000    3
```

Hide

```
dim(DF)[1] # because there are two values, the row count is the first element
```

```
## [1] 1000
```

Hide

```
dim(DF)[2] # because there are two values, the column count the second element
```

```
## [1] 3
```

Hide

```
str(DF)   # returns more information about each vector
```

```
## 'data.frame':    1000 obs. of  3 variables:
##  $ Id : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Sex: chr  "F" "F" "F" "F" ...
##  $ Iq : int  84 86 98 116 98 93 96 75 110 99 ...
```

Hide

```
head(DF)  # returns the first 6 rows
```

```
##   Id Sex  Iq
## 1  1   F  84
## 2  2   F  86
## 3  3   F  98
## 4  4   F 116
## 5  5   F  98
## 6  6   F  93
```

Hide

```
tail(DF)  # returns the last 6 rows
```

```
##        Id Sex  Iq
## 995   995   M  92
## 996   996   M 105
## 997   997   M  82
## 998   998   M  92
## 999   999   M 108
## 1000 1000   M  93
```

11 Data Frame Column/Variable Names

You can look at the names of the columns in your data frame using `names()`

Hide

```
names(DF)
```

```
## [1] "Id"  "Sex" "Iq"
```

Hide

```
# or
colnames(DF)
```

```
## [1] "Id"  "Sex" "Iq"
```

12 Changing Order of Variables in a Data Frame

If you want to rearrange the order of columns, one easy way to do this is using `dplyr::relocate()` . But rather than calling `library::function()` , just load the library.

`dplyr::relocate()` : returns a new order

Hide

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Hide

```
DF %>%
  relocate(., "Sex", .after = "Id") %>%           # take "Age" and put after "Id"
  head(.)
```

```
##   Id Sex  Iq
## 1  1   F  84
## 2  2   F  86
## 3  3   F  98
## 4  4   F 116
## 5  5   F  98
## 6  6   F  93
```

Hide

```
DF %>%
  relocate(., "Sex", .after = last_col()) %>%     # take "Age" and put after last column
  head(.)
```

```
##   Id  Iq Sex
## 1  1  84   F
## 2  2  86   F
## 3  3  98   F
## 4  4 116   F
## 5  5  98   F
## 6  6  93   F
```

Hide

```
DF %>%
  dplyr::relocate(., "Sex", .before = "Id") %>%          # take "Age" and put before "Id"
  head(.)
```

```
##   Sex Id  Iq
## 1   F  1  84
## 2   F  2  86
## 3   F  3  98
## 4   F  4 116
## 5   F  5  98
## 6   F  6  93
```

Hide

```
DF %>%
  relocate(where(is.numeric)) %>%                 # move numerics left
  head(.)
```

```
##   Id  Iq Sex
## 1  1  84   F
## 2  2  86   F
## 3  3  98   F
## 4  4 116   F
## 5  5  98   F
## 6  6  93   F
```

Hide

```
DF %>%
  relocate(where(is.numeric), .after = where(is.character)) %>% # move numerics to a location
  head(.)
```

```
##   Sex Id  Iq
## 1   F  1  84
## 2   F  2  86
## 3   F  3  98
## 4   F  4 116
## 5   F  5  98
## 6   F  6  93
```

Note that if you want to change the order of columns in the dataframe, you will need to use assignment to replace the old object with the new one with a new order.

13 Referencing Vectors In a Data Frame Using $

Because the data frame contains vectors of certain names, the vectors in the data frame do not exist outside of the data frame object. In order to reference them, you can use the `$`.

Hide

```
#Iq     # Error: object 'IQ' not found
```

But the vector does exist inside the data frame. Using the `$` will allow you to specify the vector in a data frame: `dataframename$vector`.

Hide

```
head(DF$Iq)
```

```
## [1]  84  86  98 116  98  93
```

14 Inspecting Dataframe Objects using Bracket Notation `[]`

Data frames are row and column structures, so you can inspect its elements using `[]` as you did to inspect vectors. However, because data frames contain both rows and columns, `[]` operates differently on them than on one-dimension vectors.

Hide

```
iq[1000]      # the 1000th element of the vector
```

```
## [1] 93
```

Hide

```
DF$Iq[1000]    # remember, the vector in the data frame is a vector too
```

```
## [1] 93
```

Hide

```
DF$Iq[1:5]     # first 5
```

```
## [1]  84  86  98 116  98
```

Note: `dplyr::slice()` can do a lot of the same as the above but more about that later.

15 Parsing a Data Frame by It's Rows and Columns

When examining a data frame you specify the rows and columns before and after a comma. You can pass numbers to represent the row number and or the column number or pass no numbers to see all rows and all columns (e.g., : `[,]`).

Hide

```
DF[,] %>% head()           # All rows and all columns -- same as DF or DF[]
```

```
##   Id Sex  Iq
## 1  1   F  84
## 2  2   F  86
## 3  3   F  98
## 4  4   F 116
## 5  5   F  98
## 6  6   F  93
```

Hide

```
DF[1000, 3]      # the 1000th row, 3rd column (see above for str() or dim() )
```

```
## [1] 93
```

Hide

```
DF[, "Iq"] %>% head()        # all rows and the IQ column ONLY
```

```
## [1]  84  86  98 116  98  93
```

Hide

```
DF[1000, "Iq"]     # the 1000th row and the IQ column ONLY
```

```
## [1] 93
```

Hide

```
#DF[ 1000, "Sex" "IQ" ] # Want more than one column? Hmm.
                       # Error: unexpected string constant in "DF[ 1000, "Sex" "IQ""


c("Sex", "Iq")    # Use c() to combine them and then pass the combined names as the argument
```

```
## [1] "Sex" "Iq"
```

Hide

```
DF[1:10, c("Sex", "Iq") ] # 1 through 10th row and BOTH Sex and IQ columns
```

```
##    Sex  Iq
## 1    F  84
## 2    F  86
## 3    F  98
## 4    F 116
## 5    F  98
## 6    F  93
## 7    F  96
## 8    F  75
## 9    F 110
## 10   F  99
```

16 **Modifying a Data Frame Using base R**

There are ways to modify a data frame using base R or by using `dplyr` .

Using base R, you can add or modify variables in a data frame. Be careful not to overwrite an existing variables unless that is your intention.

Assign a New Variable using `$`

Hide

```
DF$New <- "2021"      # Add the year as a string or character vector; assigns same string to all Rows


head(DF$New)
```

```
## [1] "2021" "2021" "2021" "2021" "2021" "2021"
```

Hide

```
DF$New2 <- 2021       # Add the year as a numeric vector; assigns same value to all rows


head(DF$New2)
```

```
## [1] 2021 2021 2021 2021 2021 2021
```

Using the `$` can sometimes be clunky. But you can also assign a new variable using `[]`

Hide

```
DF[, "New"] <- "2021" # Add the year as a string or character vector; assigns same string to all Rows


head(DF$New)
```

```
## [1] "2021" "2021" "2021" "2021" "2021" "2021"
```

Hide

```
DF[, "New2"] <- 2021   # Add the year as a numeric vector; assigns same value to all rows


head(DF$New)
```

```
## [1] "2021" "2021" "2021" "2021" "2021" "2021"
```

17 **Modifying an Existing Variable in a Data Frame Using base R**

You have already done this above. Whenever you assign a vector to an existing column of a data frame, you will overwrite it.

Hide

```
names(DF)    # returns the names of DF. New and New2 are there
```

```
## [1] "Id"   "Sex"  "Iq"   "New"  "New2"
```

Hide

```
DF$New <- 1  # or DF[, "New] <- 1


head(DF$New)
```

```
## [1] 1 1 1 1 1 1
```

18 Rename an Existing Variable in a Data Frame Using base R

There are various ways to rename column variables. You can assign an existing vector to a new column and simply remove the old column, you can change the name manually, and you can use libraries.

Find the column number and assign a new string to it.

Hide

```
names(DF)    # New looks like position 4
```

```
## [1] "Id"   "Sex"  "Iq"   "New"  "New2"
```

Hide

```
names(DF)[4] # Yes
```

```
## [1] "New"
```

Hide

```
names(DF)[4] <- "NewName"


names(DF)    # position 4 now has a new name
```

```
## [1] "Id"      "Sex"     "Iq"      "NewName" "New2"
```

Assign an existing column vector to a new column vector, then reassign the data frame by eliminating the old one.

Hide

```
DF$NewName2 <- DF$NewName  # assign NewName to NewName2



names(DF)     # NewName2 is there
```

```
## [1] "Id"       "Sex"      "Iq"       "NewName"  "New2"     "NewName2"
```

Hide

```
DF[, c("Id", "Sex", "Iq", "NewName2") ] %>% head()    # Use c() to combine the column names y
ou want to keep
```

```
##   Id Sex  Iq NewName2
## 1  1   F  84        1
## 2  2   F  86        1
## 3  3   F  98        1
## 4  4   F 116        1
## 5  5   F  98        1
## 6  6   F  93        1
```

Hide

```
DF <- DF[, c("Id", "Sex", "Iq", "NewName2") ] # Assign this new subsetted data frame to DF



names(DF)     # Now gone
```

```
## [1] "Id"       "Sex"      "Iq"       "NewName2"
```

19 **Examine the Levels of a Variable in a Data Frame**

Levels are for factor variables, so you need to check whether you are dealing with a factor if you wanted the variable to be a factor.

Hide

```
levels(DF$Sex)     # list levels of variable in DF. Retuns NULL if it's not a factor
```

```
## NULL
```

Hide

```
head(DF$Sex)
```

```
## [1] "F" "F" "F" "F" "F" "F"
```

Is it a factor?

Hide

```
is.factor(DF$Sex)  # nope
```

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

Reassigning the vector as a factor.

Hide

```
DF$Sex <- as.factor(DF$Sex)   # use as.factor to wrap the object and then assign to existing



is.factor(DF$Sex) # Now it is
```

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

Hide

```
head(DF$Sex)      # looks different
```

```
## [1] F F F F F F
## Levels: F M
```

Hide

```
levels(DF$Sex)    # Has 2 levels
```

```
## [1] "F" "M"
```

20 **Modifying/Creating Variables in a Data Frame Using `dplyr`**

The `mutate()` function will allow for creating new variables or changing existing variables. Similar to overwriting variable names with `mutate()`, `rename()` will rename them. Passing the data frame with `%>%` makes the code easy to follow.

Single instance

Hide

```
head(DF)
```

```
##   Id Sex  Iq NewName2
## 1  1   F  84        1
## 2  2   F  86        1
## 3  3   F  98        1
## 4  4   F 116        1
## 5  5   F  98        1
## 6  6   F  93        1
```

Hide

```
str(DF)
```

```
## 'data.frame':    1000 obs. of  4 variables:
##  $ Id      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Sex     : Factor w/ 2 levels "F","M": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Iq      : int  84 86 98 116 98 93 96 75 110 99 ...
##  $ NewName2: num  1 1 1 1 1 1 1 1 1 1 ...
```

Hide

```
DF %>%
  dplyr::mutate(.,
                Sex_f = as.factor(Sex),          # convert the character to a factor
                New_var1 = 0,                    # set to some constant
                New_var2 = Iq/2,                 # using math
                New_var3 = dplyr::case_when(     # conditional...
                  Iq <= 80 ~ 1,
                  Iq > 80 & Iq <= 115 ~ 2,
                  Iq > 115 ~ 3,
                ),
                New_var4 = "Exp 1",              # a constant character
                New_var5 = dplyr::ntile(Iq, 5)   # quintiles based on specific variable
  ) %>%
  dplyr::mutate(
    New_var4 = paste(New_var4, "*", sep = "")    # mutate() to change a variable too
  ) %>%
  dplyr::rename(., Pid = Id) %>%                 # then rename Id to Pid
  head(.)
```

```
##   Pid Sex  Iq NewName2 Sex_f New_var1 New_var2 New_var3 New_var4 New_var5
## 1   1   F  84        1     F        0     42.0        2   Exp 1*        1
## 2   2   F  86        1     F        0     43.0        2   Exp 1*        1
## 3   3   F  98        1     F        0     49.0        2   Exp 1*        3
## 4   4   F 116        1     F        0     58.0        3   Exp 1*        5
## 5   5   F  98        1     F        0     49.0        2   Exp 1*        3
## 6   6   F  93        1     F        0     46.5        2   Exp 1*        2
```

Multiple instances

When you wish to perform the same type of function or operation across a set of variables, mutating each individually is unnecessary unless you like working harder and not smarter. For such cases, `dplyr::across()` will serve you well. However, given each new variable needs it's own name (if it's not obvious, column variables cannot be redundant), you'll need to pass some special code as the argument for `.names` so that your variable names are unique and are meaningful to your goal.

Note that the argument for `.names` operates in way a similar to how `paste()` concatenates character strings. The new names will be the result of gluing together the column names by passing `{.col}` along with other characters.

A couple key arguments will need to be passed for `.cols` , `.fns` , and `.names` .

- `across(.cols = the_columns, .fns = the_funtion_to_apply, .names = the_new_var_names)`

The examples below use `across()` within `mutate()` but you can also pair it with `summarise()` . Here are some examples for creating z-scores for numeric variables by passing variables to `.cols` by their names with a character vector using `c()` , by their starting characters using `starts_with()` , by their contained characters using `contains()` , and by their numeric type using `where()` .

Hide

```
DF %>%
  #select(., c("Id", "Iq")) %>%

  # across variables in character vector
  mutate(., across(.cols = c("Iq"), .fns = scale, .names = "z_{.col}")) %>%

  # across variables with character match
  mutate(., across(starts_with("Iq"), scale, .names = "zIq_{.col}")) %>%

  # across variable with characters contained in
  mutate(., across(contains("i"), scale, .names = "zi_{.col}")) %>%

  # across all numeric types
  mutate(., across(where(is.numeric), scale, .names = "znum_{.col}")) %>%
  head(.)
```

```
##   Id Sex  Iq NewName2       z_Iq     zIq_Iq     zi_Id      zi_Iq    zi_z_Iq
## 1  1   F  84        1 -1.1122449 -1.1122449 -1.729454 -1.1122449 -1.1122449
## 2  2   F  86        1 -0.9731883 -0.9731883 -1.725992 -0.9731883 -0.9731883
## 3  3   F  98        1 -0.1388481 -0.1388481 -1.722530 -0.1388481 -0.1388481
## 4  4   F 116        1  1.1126621  1.1126621 -1.719067  1.1126621  1.1126621
## 5  5   F  98        1 -0.1388481 -0.1388481 -1.715605 -0.1388481 -0.1388481
## 6  6   F  93        1 -0.4864898 -0.4864898 -1.712142 -0.4864898 -0.4864898
##    zi_zIq_Iq   znum_Id    znum_Iq znum_NewName2  znum_z_Iq znum_zIq_Iq
## 1 -1.1122449 -1.729454 -1.1122449           NaN -1.1122449  -1.1122449
## 2 -0.9731883 -1.725992 -0.9731883           NaN -0.9731883  -0.9731883
## 3 -0.1388481 -1.722530 -0.1388481           NaN -0.1388481  -0.1388481
## 4  1.1126621 -1.719067  1.1126621           NaN  1.1126621   1.1126621
## 5 -0.1388481 -1.715605 -0.1388481           NaN -0.1388481  -0.1388481
## 6 -0.4864898 -1.712142 -0.4864898           NaN -0.4864898  -0.4864898
##   znum_zi_Id znum_zi_Iq znum_zi_z_Iq znum_zi_zIq_Iq
## 1  -1.729454 -1.1122449   -1.1122449     -1.1122449
## 2  -1.725992 -0.9731883   -0.9731883     -0.9731883
## 3  -1.722530 -0.1388481   -0.1388481     -0.1388481
## 4  -1.719067  1.1126621    1.1126621      1.1126621
## 5  -1.715605 -0.1388481   -0.1388481     -0.1388481
## 6  -1.712142 -0.4864898   -0.4864898     -0.4864898
```

If your data frame or tibble contains only numeric variables, however, you can use `mutate_all()`. Similarly, you can subset your data frame to include only the numeric variable and use `mutate_all()` but the previous function approaches would likely be better to use if you want the other variables retained in the data frame.

Here are two examples of subsetting and then creating by selecting using either `select_if(., is.numeric)` or `select(where(is.numeric))`

- `select_if(., is.numeric)` : selects columns IF they are numeric
- `select(where(is.numeric))` : selects columns where there are numeric variables

Hide

```
DF %>%
  select_if(., is.numeric) %>%       # if type function
  select(., -c("Id", "NewName2")) %>%            # select out
  mutate_all(., ~scale(.)) %>%
  head(.)
```

```
##           Iq
## 1 -1.1122449
## 2 -0.9731883
## 3 -0.1388481
## 4  1.1126621
## 5 -0.1388481
## 6 -0.4864898
```

Hide

```
DF %>%
  select(., where(is.numeric)) %>%   # where there are numerics
  select(., -c("Id", "NewName2")) %>%            # select out
  mutate_all(., ~scale(.)) %>%
  head(.)
```

```
##           Iq
## 1 -1.1122449
## 2 -0.9731883
## 3 -0.1388481
## 4  1.1126621
## 5 -0.1388481
## 6 -0.4864898
```

21 Arranging/Sorting a Data Frame using `dplyr`

For arranging using `dplyr`, you will use `arrange()` and at very least pass arguments for the data frame (first) followed by the variables (what the documentation refers to as `...`. You can add other arguments as well, however, but R will need to know what to arrange and how to arrange it.

- `arrange(.data, ..., .by_group = FALSE)`

Note, the variables can be represented as separated arguments in the function separated by commas or they can be passed as a single argument as a single vector containing the variable.

Examples:

- `arrange(., var1, var2)` : # the variables by their names
- `arrange_at(., c("var1", "var2"))` # the variables as a combined vector

By default, `arrange()` will sort in a ascending manner.

Hide

```
DF %>%
  arrange(., Iq) %>%
  head(.)
```

```
##    Id Sex Iq NewName2
## 1 878   M 56        1
## 2 837   M 58        1
## 3  15   F 59        1
## 4 428   F 60        1
## 5 297   F 61        1
## 6 879   M 61        1
```

To sort in a descending manner, wrap the variable in `desc()`.

Hide

```
DF %>%
  arrange(., desc(Iq)) %>%
  head(.)
```

```
##    Id Sex  Iq NewName2
## 1  61   F 149        1
## 2 236   F 146        1
## 3 152   F 145        1
## 4 719   M 141        1
## 5 239   F 138        1
## 6 125   F 136        1
```

Hide

```
DF %>%
  arrange(., desc(Sex)) %>%
  head(.)
```

```
##    Id Sex  Iq NewName2
## 1 501   M  91        1
## 2 502   M  89        1
## 3 503   M  96        1
## 4 504   M 102        1
## 5 505   M  88        1
## 6 506   M  79        1
```

Sorting on multiple variables…

Hide

```
DF %>%
  arrange(., Sex, Iq) %>%
  view(., show = 10)
```

Show 10 **entries** **Search:**

	Id	Sex	Iq	NewName2
	All	All	All	All
1	15	F	59	1
2	428	F	60	1
3	297	F	61	1

	Id	Sex	Iq	NewName2
4	54	F	63	1
5	430	F	65	1
6	462	F	68	1
7	35	F	70	1
8	123	F	70	1
9	42	F	71	1
10	90	F	71	1

Showing 1 to 10 of 1,000 entries Previous 1 2 3 4 5 … 100 Next

Hide

```
DF %>%
  arrange(., across(contains("Se"))) %>%    # arranges based on variables containing "se" (e.g., Sex)
  view(., show = 10)
```

Show 10 **entries** **Search:**

	Id	Sex	Iq	NewName2
	All	All	All	All
1	1	F	84	1
2	2	F	86	1
3	3	F	98	1
4	4	F	116	1
5	5	F	98	1
6	6	F	93	1
7	7	F	96	1

	Id	Sex	Iq	NewName2
8	8	F	75	1
9	9	F	110	1
10	10	F	99	1

Showing 1 to 10 of 1,000 entries Previous 1 2 3 4 5 … 100 Next

If you want to pass a vector of names, you'll need to use an `unquote` operator to unqote the vector object so that dplyr understands it. A double bang `!!` will unquote one character vector and a triple bang `!!!` will unquote more.

Hide

```
DF %>%
  arrange(!!! rlang::syms(c("Sex", "Iq"))) %>%
  view(., show = 10)
```

Show 10 **entries** **Search:**

	Id	Sex	Iq	NewName2
	All	All	All	All
1	15	F	59	1
2	428	F	60	1
3	297	F	61	1
4	54	F	63	1
5	430	F	65	1
6	462	F	68	1
7	35	F	70	1
8	123	F	70	1
9	42	F	71	1
10	90	F	71	1

Showing 1 to 10 of 1,000 entries Previous 1 2 3 4 5 … 100 Next

Hide

```
# or
vars <- c("Sex", "Iq")

DF %>%
  arrange(!!! rlang::syms(vars)) %>%
  view(., show = 10)
```

Show 10 **entries** **Search:**

	Id	Sex	Iq	NewName2
	All	All	All	All
1	15	F	59	1
2	428	F	60	1
3	297	F	61	1
4	54	F	63	1
5	430	F	65	1
6	462	F	68	1
7	35	F	70	1
8	123	F	70	1
9	42	F	71	1
10	90	F	71	1

Showing 1 to 10 of 1,000 entries Previous 1 2 3 4 5 … 100 Next

However, this is confusing. There is an old function named `arrange_at()` which makes this easier. As of now, the function is not deprecated and there is no plan to remove it from `dplyr` . All you need to do is pass the vector object and the data frame will be sorted in the indexed order.

Hide

```
DF %>%
  arrange_at(., vars) %>%
  view(., show = 10)
```

Show 10 entries Search:

	Id	Sex	Iq	NewName2
	All	All	All	All
1	15	F	59	1
2	428	F	60	1
3	297	F	61	1
4	54	F	63	1
5	430	F	65	1
6	462	F	68	1
7	35	F	70	1
8	123	F	70	1
9	42	F	71	1
10	90	F	71	1

Showing 1 to 10 of 1,000 entries Previous 1 2 3 4 5 … 100 Next

22 Flagging Complete (or missing) Cases Across a Variable Group

If you want to find out quickly whether a case/row contains completed cases for a certain grouping of variables, you can subset the variables by name and create a new variable and assign a logical `TRUE` or `FALSE`.

Hide

```
DF %>%
  mutate(., complete_all = complete.cases(across(everything()))) %>%    # across all variable
s
  mutate(., complete_i   = complete.cases(across(contains("i")))) %>%   # those containing
  mutate(., complete_num = complete.cases(across(where(is.numeric)))) %>%   # those numeric
  head()
```

```
##   Id Sex  Iq NewName2 complete_all complete_i complete_num
## 1  1   F  84        1         TRUE       TRUE          TRUE
## 2  2   F  86        1         TRUE       TRUE          TRUE
## 3  3   F  98        1         TRUE       TRUE          TRUE
## 4  4   F 116        1         TRUE       TRUE          TRUE
## 5  5   F  98        1         TRUE       TRUE          TRUE
## 6  6   F  93        1         TRUE       TRUE          TRUE
```