DATA SOCIETY®

Intro to R programming - day 2

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Module completion checklist

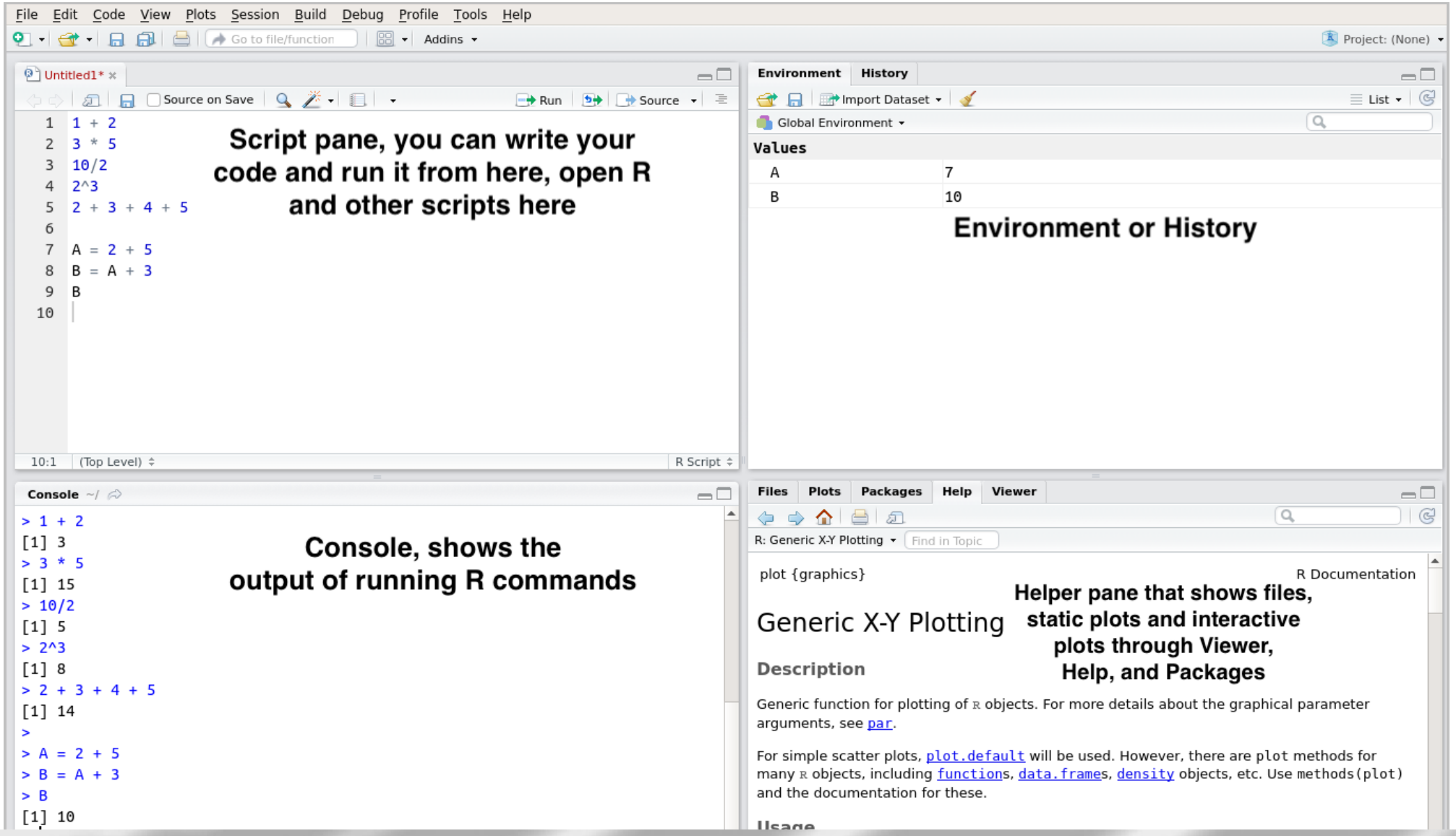
Objective	Complete
Identify the various ways to read, write and view data	
Save the environment and environment variables	
Clear environment	
Load environment and environment variable	
Detect and address missing values in data	
Manipulate data types and structures using flow control structures (for loops, conditionals, etc)	
Save and load custom functions into R environment as a module	

RStudio overview layout recap

A default RStudio layout includes 4 panes:

- **Top left** pane is used as a `Script` pane, you can write your code and run it from here, open R and other scripts here
- **Bottom left** pane has a `Console`, which shows the output of running R commands
- **Top right** is a helper pane that shows your `Environment` or `History`
- **Bottom right** is another helper pane that shows `Files`, static `Plots` and interactive plots through `Viewer`, `Help`, and `Packages`

RStudio overview panes recap

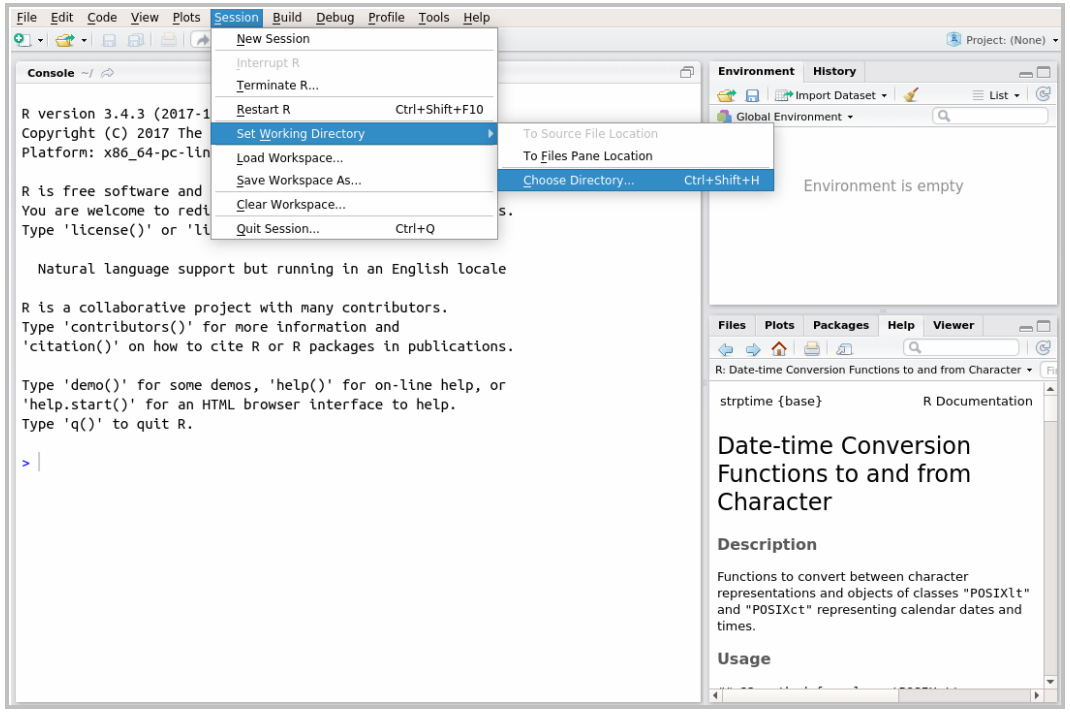


R's working directory

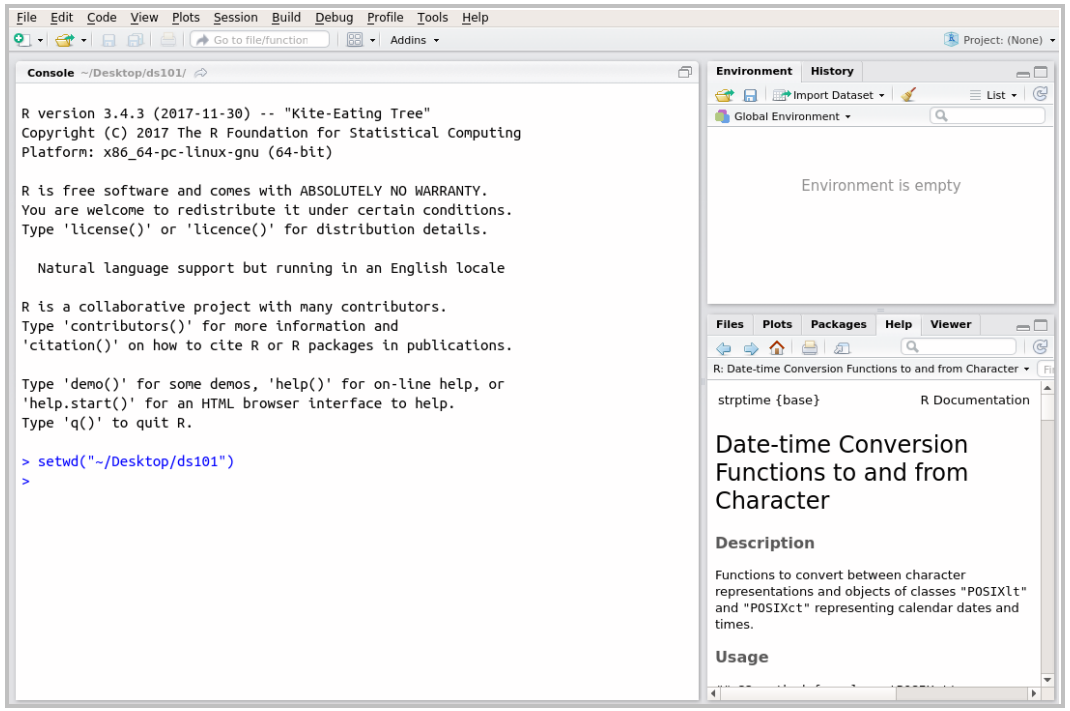
- A folder on your machine, which R treats as your "sandbox" and saves your files and loads your data from is called a **working directory**
- R has a **default** working directory, which can be found and set through RStudio's `Global Options`
- We can `set` the working directory
- We can `get` the working directory
- We can `encode` directory paths into `variables` and change them without having to manually type the paths every time

Setting R's working directory

- You can set your working directory via RStudio's GUI



- Once the directory is set, you will see the command executed in the `Console`



Setting your working directory via command line

- You can set your working directory via command line (on Mac/Linux)

```
# To set working directory, call `setwd` with the path to the folder.
setwd("~/Desktop/af-werx")

# To check the current working directory, use `getwd`.
getwd()
```

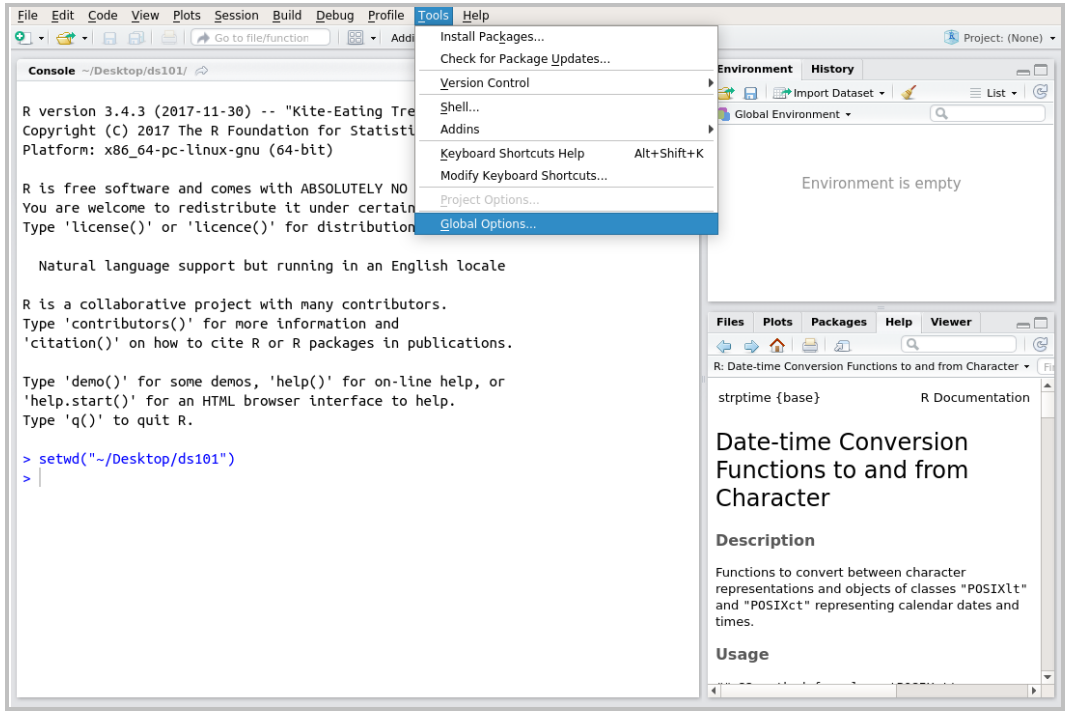
```
[1] "/home/[your-user-name]/Desktop/af-werx"
```

- You can set your working directory via command line (on Windows)

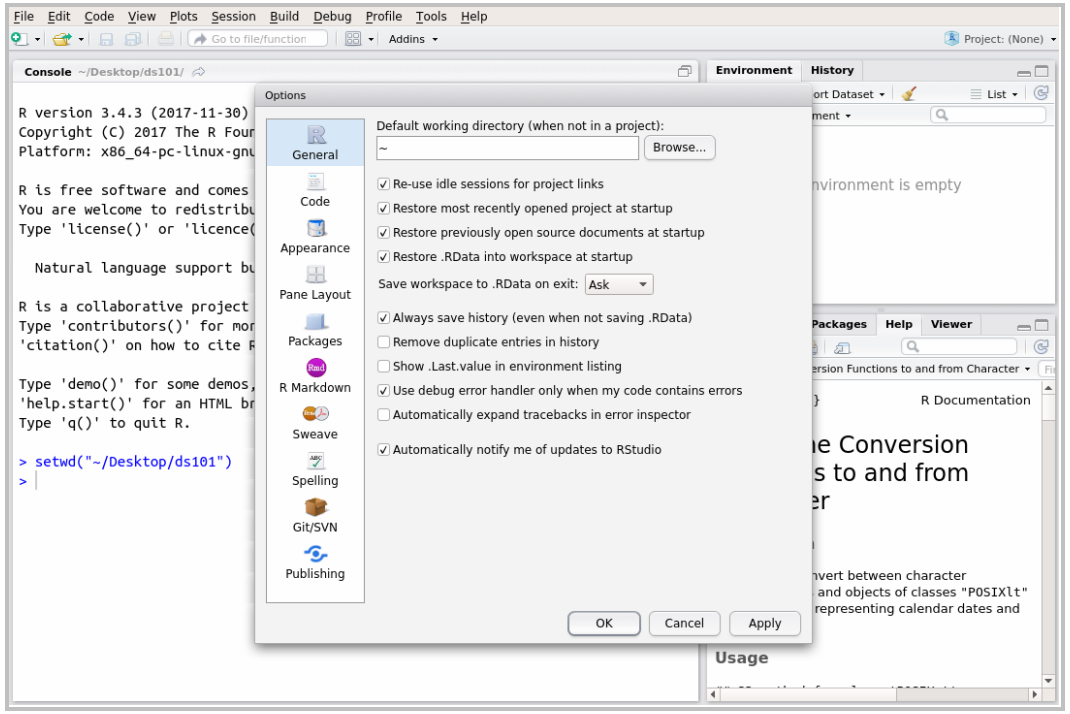
```
# To set working directory, call `setwd` with the path to the folder.
setwd("C:/Users/[your-user-name]/Desktop/af-werx")

# To check the current working directory, use `getwd`.
getwd()
```

```
[1] "C:/Users/[your-user-name]/Desktop/af-werx"
```

R's default working directory



- You can also set a default working directory for whenever R just starts



- To see what it currently is, you can look at the very first option in the `General` section of the `Global Options`
- To change it, just click on `Browse` and select a default working directory

Setting the working directory

- In order to maximize the efficiency of your workflow, you may want to encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac/Linux).
main_dir = "~/Desktop/af-werx"
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Windows).

main_dir = "C:/Users/[username]/Desktop/af-werx"
```

Directory settings

1. We will store all datasets in the `data` directory inside of the `af-werx` folder, so we'll save its path to a `data_dir` variable
2. We will save all of the plots in the `plots` directory inside of the `af-werx` folder, so we'll save its path to a `plot_dir` variable

To append a string to another string, use `paste0` command and pass the strings you would like to paste together.

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = paste0(main_dir, "/data")
data_dir
```

```
[1] "~/Desktop/af-werx/data"
```

```
# Make `plots_dir` from the `main_dir` and
# remainder of the path to plots directory.
plot_dir = paste0(main_dir, "/plots")
plot_dir
```

```
[1] "~/Desktop/af-werx/plots"
```

Directory settings

- Now all you have to do to switch between working directories is to use a variable instead of typing the full path every time

```
# Set working directory to where the data is.
setwd(data_dir)
```

```
# Print working directory (Mac/Linux).
getwd()
```

```
[1] "/home/[your-user-name]/Desktop/af-werx/data"
```

```
# Print working directory (Windows).
getwd()
```

```
[1] "C:/Users/[your-user-name]/Desktop/af-werx/data"
```

Loading CSV files to R

- Most of the time you will be working with data that was generated elsewhere and you need to load it to your R environment
- R works with many different data types, but the most common one is CSV

```
# Set working directory to where the data is.
setwd(data_dir)

# To read a C[omma] S[eparated] V[alues] file into
# R, you can use a simple command `read.csv`.
temp_heart_data = read.csv("temp_heart_rate.csv",    #<- provide file name
                           header = TRUE,            #<- if file has header set to TRUE
                           stringsAsFactors = FALSE) #<- read strings as characters, not as factors
```

Viewing data in R

```
# Inspect the structure of the data.
str(temp_heart_data)
```

```
'data.frame':   130 obs. of  3 variables:
 $ Gender    : chr  "Male" "Male" "Male" "Male" ...
 $ Body.Temp : num  96.3 96.7 96.9 97 97.1 97.1 97.1 97.2 97.3 97.4 ...
 $ Heart.Rate: int  70 71 74 80 73 75 82 64 69 70 ...
```

```
# Inspect the `head` (first 4 rows).
head(temp_heart_data, 4)
```

```
  Gender Body.Temp Heart.Rate
1   Male      96.3         70
2   Male      96.7         71
3   Male      96.9         74
4   Male      97.0         80
```

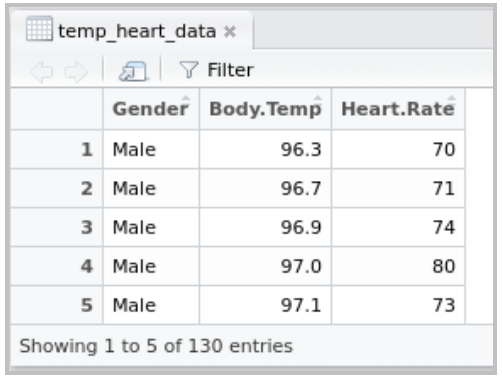
```
# Inspect the `tail` (last 4 rows).
tail(temp_heart_data, 4)
```

```
    Gender Body.Temp Heart.Rate
127 Female      99.4         77
128 Female      99.9         79
129 Female     100.0         78
130 Female     100.8         77
```

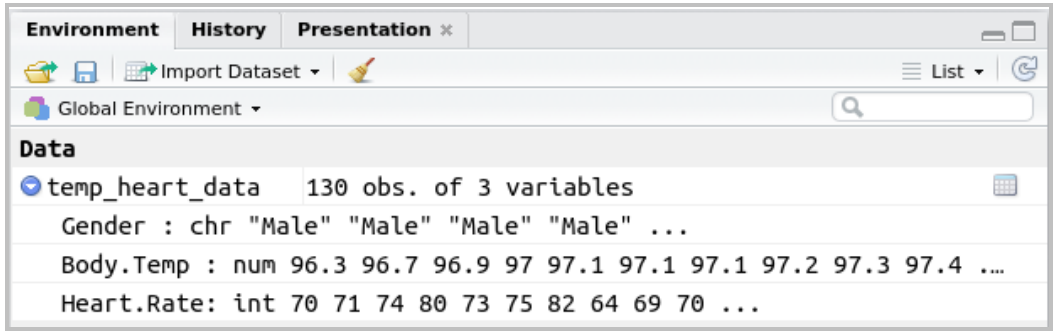
Viewing data in R

- View in the tabular data explorer

```
View(temp_heart_data)
```



- You can also see the loaded data and variables in the `Environment` pane of R Studio



Other file types and commands in R

Command	File type
`read.csv("filename.csv")`	File with comma separated values
`read.table("filename")`	Tabulated data in a text file
`read.spss("filename.spss")`	File produced in SPSS
`read.dta("filename.dta")`	File produced in STATA
`read.ssd("filename.ssd")`	File produced in SAS
`readJPEG("filename.jpg")`	Read JPEG image files

Writing CSV files

- The most natural way to share tabular data is through saving your data to a CSV file

```
# Let's save the first 10 rows of our data to a variable.
temp_heart_subset = temp_heart_data[1:10, ]
temp_heart_subset
```

```
   Gender Body.Temp Heart.Rate
1    Male      96.3         70
2    Male      96.7         71
3    Male      96.9         74
4    Male      97.0         80
5    Male      97.1         73
6    Male      97.1         75
7    Male      97.1         82
8    Male      97.2         64
9    Male      97.3         69
10   Male      97.4         70
```

```
# Set working directory to where the data is.
setwd(data_dir)

# Write data to a CSV file providing 3 arguments:
write.csv(temp_heart_subset,              #<- name of variable to save
          "temp_heart_rate_subset.csv", #<- name of file where to save
          row.names = FALSE)            #<- logical value for row names
```

Module completion checklist

Objective	Complete
Identify the various ways to read, write and view data	✔
Save the environment and environment variables	
Clear environment	
Load environment and environment variable	
Detect and address missing values in data	
Manipulate data types and structures using flow control structures (for loops, conditionals, etc)	
Save and load custom functions into R environment as a module	

Listing objects in your environment

```
# List all objects that are in your environment.
ls()
```

```
 [1] "data_dir"          "directory"         "head"
 [4] "highlight_js"      "main_dir"          "platform"
 [7] "plot_dir"          "session_info"      "temp_heart_data"
[10] "temp_heart_subset"
```

Capturing your environment

- Sometimes you would like to quickly load data you've pre-saved into R, but
  - it may not be in tabular format
  - you want to capture the state of your environment at some point of your work
- You can do that by using an image (i.e. snapshot) of your environment and saving it to an `*.RData` file

```
# Set working directory to where the data is saved.
setwd(data_dir)

# Save image of the environment to RData file.
save.image(file = "my_working_environment.RData")
```

Clearing objects from environment

```
# Remove individual variable(s).
rm(X, x, this_is_a_valid_name, This.Is.Also.A.Valid.Name, unnamed_list)

# List all objects again to check.
ls()
```

```
 [1] "data_dir"           "directory"          "head"
 [4] "highlight_js"       "main_dir"           "platform"
 [7] "plot_dir"           "session_info"       "temp_heart_data"
[10] "temp_heart_subset"
```

Notice that the variables we have removed are gone!

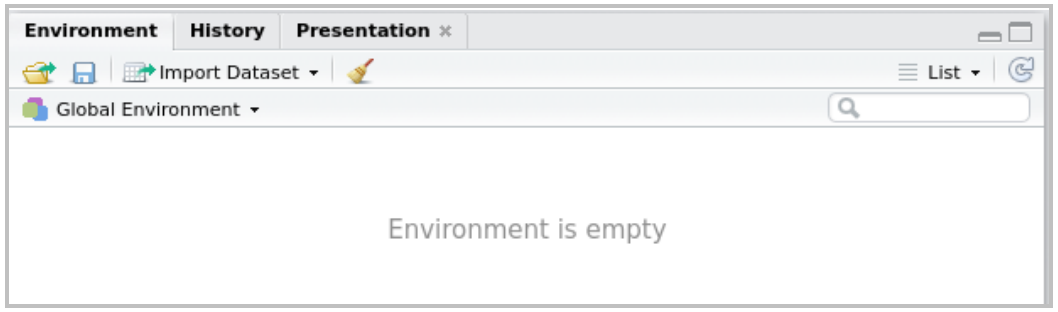
Clearing the entire environment

- The clear environment will always return an empty vector of type `character`

```
# Remove all variables listed in the
environment.
rm(list = ls())

# List all objects again to check again.
ls()
```

```
character(0)
```

- The clear environment will always show like this in the `Environment` pane



You can also clear the environment by clicking on the broom icon at the top of the environment pane

Loading the entire environment

```
# Set working directory to where the data is saved.
setwd("~/Desktop/af-werx/data")

# Load the environment.
load("my_working_environment.RData")
```

Keep in mind, since we have cleared the environment, the `data_dir` and all other directory variables are gone, so we will have to type the full path name to the `data` folder!

Check if variables were loaded

```
# List all objects that are in your environment.
ls()
```

```
 [1] "data_dir"           "directory"          "head"
 [4] "highlight_js"       "main_dir"           "platform"
 [7] "plot_dir"           "session_info"       "temp_heart_data"
[10] "temp_heart_subset"
```

Saving individual variables to `*.RData` file

- Instead of saving the entire environment to `*.RData` file, you can save individual variable(s)

```
# Set working directory to where the data is saved.
setwd(data_dir)

# Save image of individual variable(s) to `RData`.
save(main_dir, data_dir, plot_dir,
     file = "directory_variables.RData")

# Clear the enviroment from all of the variables.
rm(list = ls())

# Reload just the directory variables.
load("directory_variables.RData")

# List the items in the environment.
ls()
```

```
[1] "data_dir" "main_dir" "plot_dir"
```

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Identify the various ways to read, write and view data	✔
Save the environment and environment variables	✔
Clear environment	✔
Load environment and environment variable	✔
Detect and address missing values in data	
Manipulate data types and structures using flow control structures (for loops, conditionals, etc)	
Save and load custom functions into R environment as a module	

Directory settings

- In order to maximize the efficiency of your workflow, you may want to encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac/Linux).
main_dir = "~/Desktop/af-werx"

# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/af-werx"

# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = paste0(main_dir, "/data")

# Make `plots_dir` from the `main_dir` and
# remainder of the path to plots directory.
plot_dir = paste0(main_dir, "/plots")
```

Subsidies for friendly pharma

- The **CDC** has **warned that Disease X poses a national threat**, so the government has offered subsidies to pharmaceutical companies that lower the cost of medications in high-risk, low-income communities
- Pharmaceutical company **BigFriendPharma** thinks that if they have more control over their chemical processing, then they **can figure out which combination of biological characteristics and processing manufacturing measurements produce the highest yield of their product**
- Once they optimize their process, they can lower the cost of their product and qualify for those sweet subsidies

Introducing CMP dataset

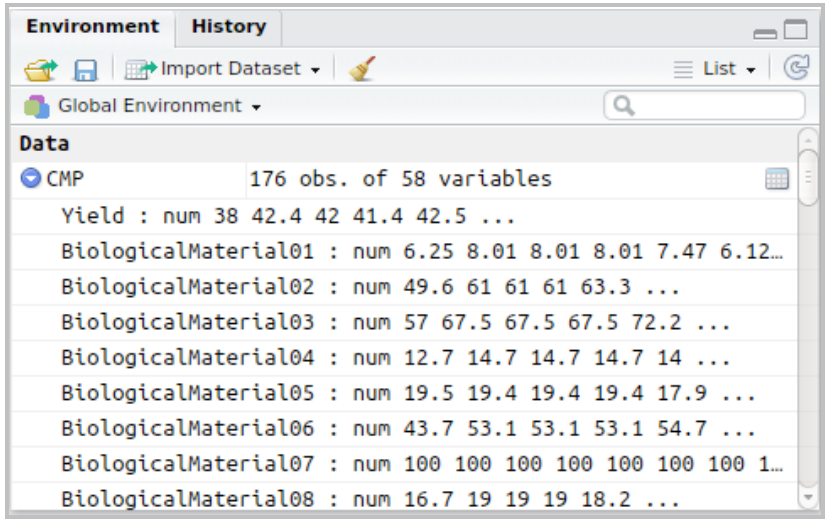
We are going to explore a new dataset called `ChemicalManufacturingProcess` from `AppliedPredictiveModeling` package in R. This dataset includes information about:

- A chemical manufacturing **process**, in which the goal is to understand the relationship between the process and the resulting final product **yield**
- Raw material in this process is put through a sequence of 27 steps to generate the final pharmaceutical product
- Of the 57 characteristics, there are
  - **12 measurements of** the biological starting **material**, and
  - **45 measurements of** the manufacturing **process**
- The starting **material** is generated from a biological unit and has a range of quality and characteristics
- The **process** variables include measurements such as temperature, drying time, washing time, and concentrations of by-products at various steps

Loading dataset

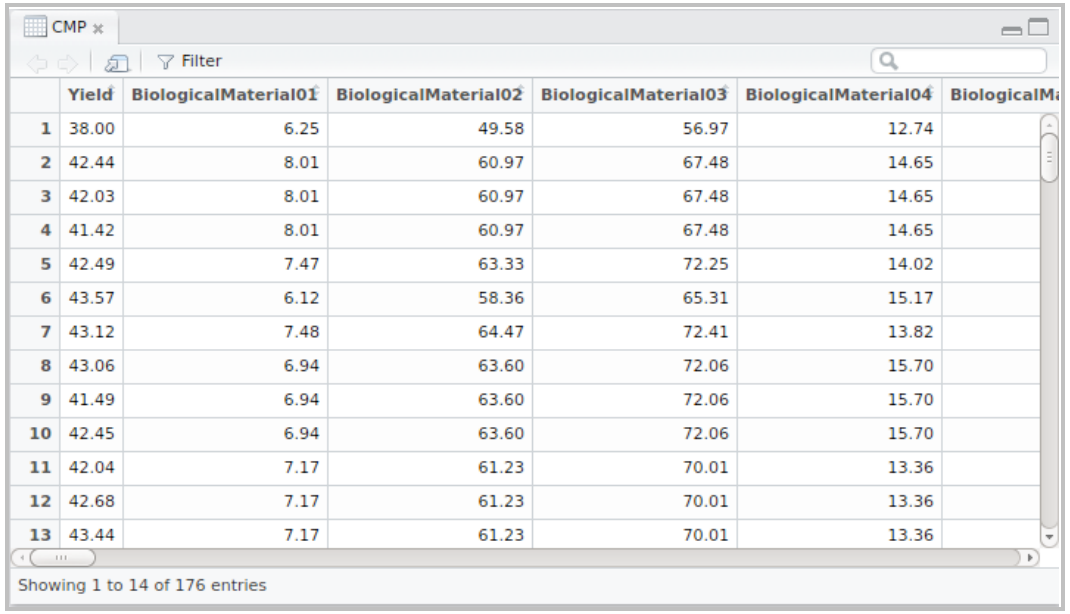
- Let's load the dataset from our `data_dir` into R's environment

```
# Set working directory to where we store data.
setwd(data_dir)

# Read CSV file called
# "ChemicalManufacturingProcess.csv"
CMP =
  read.csv("ChemicalManufacturingProcess.csv",
           header = TRUE,
           stringsAsFactors = FALSE)
```



- The dataset consists of `176` observations and `58` variables

```
# View CMP dataset in the
# tabular data explorer.
View(CMP)
```



	Yield	BiologicalMaterial01	BiologicalMaterial02	BiologicalMaterial03	BiologicalMaterial04	BiologicalMa
1	38.00	6.25	49.58	56.97	12.74	
2	42.44	8.01	60.97	67.48	14.65	
3	42.03	8.01	60.97	67.48	14.65	
4	41.42	8.01	60.97	67.48	14.65	
5	42.49	7.47	63.33	72.25	14.02	
6	43.57	6.12	58.36	65.31	15.17	
7	43.12	7.48	64.47	72.41	13.82	
8	43.06	6.94	63.60	72.06	15.70	
9	41.49	6.94	63.60	72.06	15.70	
10	42.45	6.94	63.60	72.06	15.70	
11	42.04	7.17	61.23	70.01	13.36	
12	42.68	7.17	61.23	70.01	13.36	
13	43.44	7.17	61.23	70.01	13.36	

Understanding data

- In this module, we will explore a subset of this dataset, which includes the following variables
  - **yield**
  - **3 material** variables, and
  - **3 process** variables

CMP

Filter

	Yield	BiologicalMaterial01	BiologicalMaterial02	BiologicalMaterial03
1	38.00	6.25	49.58	56.97
2	42.44	8.01	60.97	67.48
3	42.03	8.01	60.97	67.48
4	41.42	8.01	60.97	67.48
5	42.49	7.47	63.33	72.25
6	43.57	6.12	58.36	65.31
7	43.12	7.48	64.47	72.41
8	43.06	6.94	63.60	72.06
9	41.49	6.94	63.60	72.06

Showing 1 to 10 of 176 entries

CMP

Filter

ManufacturingProcess01	ManufacturingProcess02	ManufacturingProcess03
NA	NA	NA
0.0	0.0	NA
0.0	0.0	NA
0.0	0.0	NA
10.7	0.0	NA
12.0	0.0	NA
11.5	0.0	1.56
12.0	0.0	1.55
12.0	0.0	1.56

Showing 1 to 10 of 176 entries

Subsetting data

```
# Let's make a vector of column indices we would like to save.
column_ids = c(1:4,   #<- concatenate a range of IDs
               14:16) #<- with another a range of IDs
column_ids
```

```
[1]  1  2  3  4 14 15 16
```

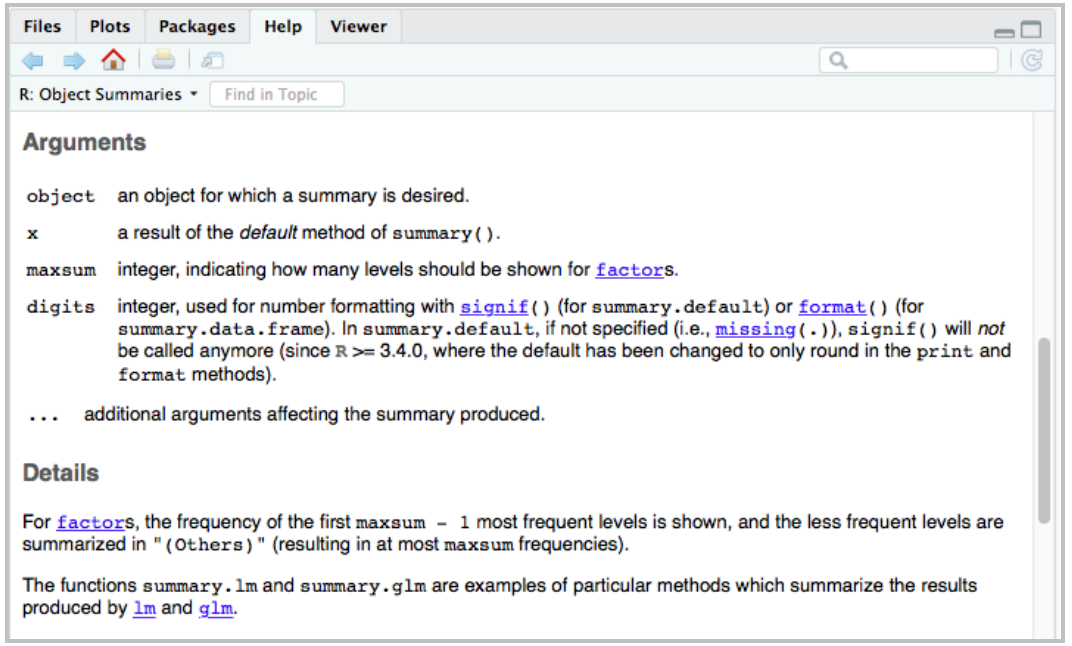
```
# Let's save the subset into a new variable.
CMP_subset = CMP[ ,column_ids]
str(CMP_subset)
```

```
'data.frame':   176 obs. of  7 variables:
 $ Yield                 : num  38 42.4 42 41.4 42.5 ...
 $ BiologicalMaterial01  : num  6.25 8.01 8.01 8.01 7.47 6.12 7.48 6.94 6.94 6.94 ...
 $ BiologicalMaterial02  : num  49.6 61 61 61 63.3 ...
 $ BiologicalMaterial03  : num  57 67.5 67.5 67.5 72.2 ...
 $ ManufacturingProcess01: num  NA 0 0 0 10.7 12 11.5 12 12 12 ...
 $ ManufacturingProcess02: num  NA 0 0 0 0 0 0 0 0 0 ...
 $ ManufacturingProcess03: num  NA NA NA NA NA NA 1.56 1.55 1.56 1.55 ...
```

Summary statistics

- To get quick summary statistics of your dataframe or one single column within the dataframe, use `summary`

```
?summary

summary(data) #<- dataframe or single column
```



Summary statistics of CMP

```
summary(CMP_subset) #<- getting summary statistics of CMP_subset
```

```
     Yield       BiologicalMaterial01 BiologicalMaterial02
 Min.   :35.25   Min.   :4.580        Min.   :46.87
 1st Qu.:38.75   1st Qu.:5.978        1st Qu.:52.68
 Median :39.97   Median :6.305        Median :55.09
 Mean   :40.18   Mean   :6.411        Mean   :55.69
 3rd Qu.:41.48   3rd Qu.:6.870        3rd Qu.:58.74
 Max.   :46.34   Max.   :8.810        Max.   :64.75

 BiologicalMaterial03 ManufacturingProcess01 ManufacturingProcess02
 Min.   :56.97        Min.   : 0.00          Min.   : 0.00
 1st Qu.:64.98        1st Qu.:10.80          1st Qu.:19.30
 Median :67.22        Median :11.40          Median :21.00
 Mean   :67.70        Mean   :11.21          Mean   :16.68
 3rd Qu.:70.43        3rd Qu.:12.15          3rd Qu.:21.50
 Max.   :78.25        Max.   :14.10          Max.   :22.50
                      NA's   :1              NA's   :3
 ManufacturingProcess03
 Min.   :1.47
 1st Qu.:1.53
 Median :1.54
 Mean   :1.54
 3rd Qu.:1.55
 Max.   :1.60
 NA's   :15
```

Working with missing data: max values

```
# Let's try and compute the maximum value of the first manufacturing process.
max_process01 = max(CMP_subset$ManufacturingProcess01)
max_process01
```

```
[1] NA
```

- Notice that we get `NA` in return

```
max_process02 = max(CMP_subset$ManufacturingProcess01, na.rm = TRUE)
max_process02
```

```
[1] 14.1
```

- Notice that we now get an actual number by using `na.rm = TRUE` to ignore NA values

Working with missing data: imputing

- What if the function you are using does not have `na.rm` or removing `NA`s might affect and skew the results?
- Data imputation with one of the following values will help to overcome this:
  - `0`
  - `mean`
  - `median`
  - any other special value appropriate for a given dataset and data type (e.g. handling of categorical variables with missing data should be handled differently from imputing numeric variables!)

Working with missing data

- Function `is.na` will provide a vector of `TRUE` or `FALSE` values for each element of a given vector
- For datasets with an even moderate number of data points, it is hard to track which elements are indeed `NA`

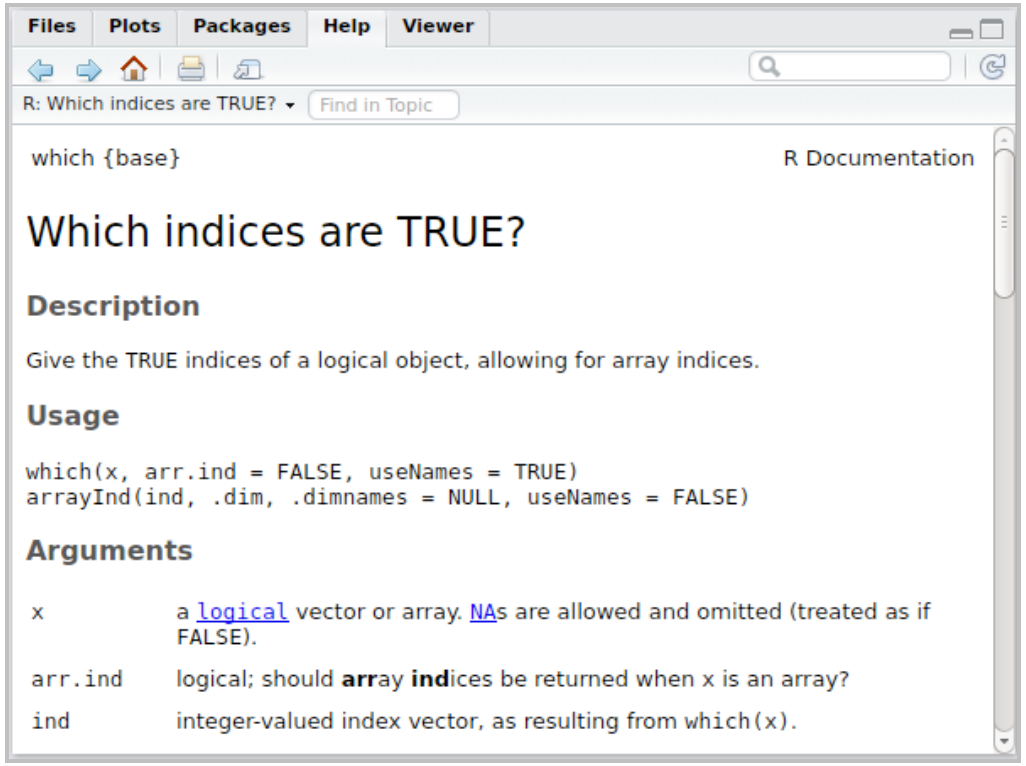
```
# Let's take a look at `ManufacturingProcess01`
# and see if any of the values in it are `NA`.
is.na(CMP_subset$ManufacturingProcess01)
```

```
  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Working with missing data

```
?which
```

- The `which` function is an invaluable utility function in R `base` package
- It takes either a vector/array of `logical` values (like the one we produced in the previous slide), or
- It takes a vector/array of any values and a comparison statement with one of the comparison operators (==, !=, >, <, >=, <=) and a value to which we are comparing
- It returns the `indices` of all `TRUE` values of the `logical` vector, or the indices of all the values that meet the condition we specified



Identifying NA values

```
# Let's save this vector of logical values to a variable.
is_na = is.na(CMP_subset$ManufacturingProcess01)

# To determine WHICH elements in the vector are `TRUE`,
# we will use `which` function.

# Since we already have a vector of `TRUE` or `FALSE` logical values,
# we only have to give it to `which` and it will return all of the
# indices of values that are `TRUE`.
which(is_na)
```

```
[1] 1
```

```
# This is also a correct way to set it up.
which(is_na == TRUE)
```

```
[1] 1
```

Locating NA values

- Now that we know which entry in the `ManufacturingProcess01` is `NA`, we can select it programmatically, without having to type its index manually

```
# Let's save the index to a variable.
na_id = which(is_na)
na_id
```

```
[1] 1
```

```
# Let's view the value at the `na_id` index.
CMP_subset$ManufacturingProcess01[na_id]
```

```
[1] NA
```

Mean replacement

- We need to compute a value suitable to be a replacement for the given `NA`
- For demonstration purposes, we will use the `mean` of the variable as a replacement

```
# Compute the mean of the `ManufacturingProcess01`.
mean_process01 = mean(CMP_subset$ManufacturingProcess01)
mean_process01
```

```
[1] NA
```

- Don't forget to set `na.rm = TRUE` in order to compute the `mean` of the variable that contains `NA`s!

```
# Compute the mean of the `ManufacturingProcess01` and set `na.rm` to `TRUE`.
mean_process01 = mean(CMP_subset$ManufacturingProcess01, na.rm = TRUE)
mean_process01
```

```
[1] 11.20743
```

Working with missing data

- Now we can finally take that mean and assign it to the value in the vector

```
# Assign the mean to the entry with the `NA`.
CMP_subset$ManufacturingProcess01[na_id] = mean_process01
CMP_subset$ManufacturingProcess01[na_id]
```

```
[1] 11.20743
```

- Instead of the `NA`, we have the `mean` value of this column!
- Let's compute the `max` of of the column without `na.rm` specified to see if it works:

```
max_process01 = max(CMP_subset$ManufacturingProcess01)
max_process01
```

```
[1] 14.1
```

Repeat the process

- Now we repeat the process for the remaining manufacturing variables

```
# Impute missing values of `ManufacturingProcess02` with the mean.
is_na = is.na(CMP_subset$ManufacturingProcess02)
na_id = which(is_na)
mean_process02 = mean(CMP_subset$ManufacturingProcess02, na.rm = TRUE)
CMP_subset$ManufacturingProcess02[na_id] = mean_process02

# Impute missing values of `ManufacturingProcess03` with the mean.
is_na = is.na(CMP_subset$ManufacturingProcess03)
na_id = which(is_na)
mean_process03 = mean(CMP_subset$ManufacturingProcess03, na.rm = TRUE)
CMP_subset$ManufacturingProcess03[na_id] = mean_process03
```

Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Identify the various ways to read, write and view data	✔
Save the environment and environment variables	✔
Clear environment	✔
Load environment and environment variable	✔
Detect and address missing values in data	✔
Manipulate data types and structures using flow control structures (for loops, conditionals, etc)	
Save and load custom functions into R environment as a module	

Laundry algorithm

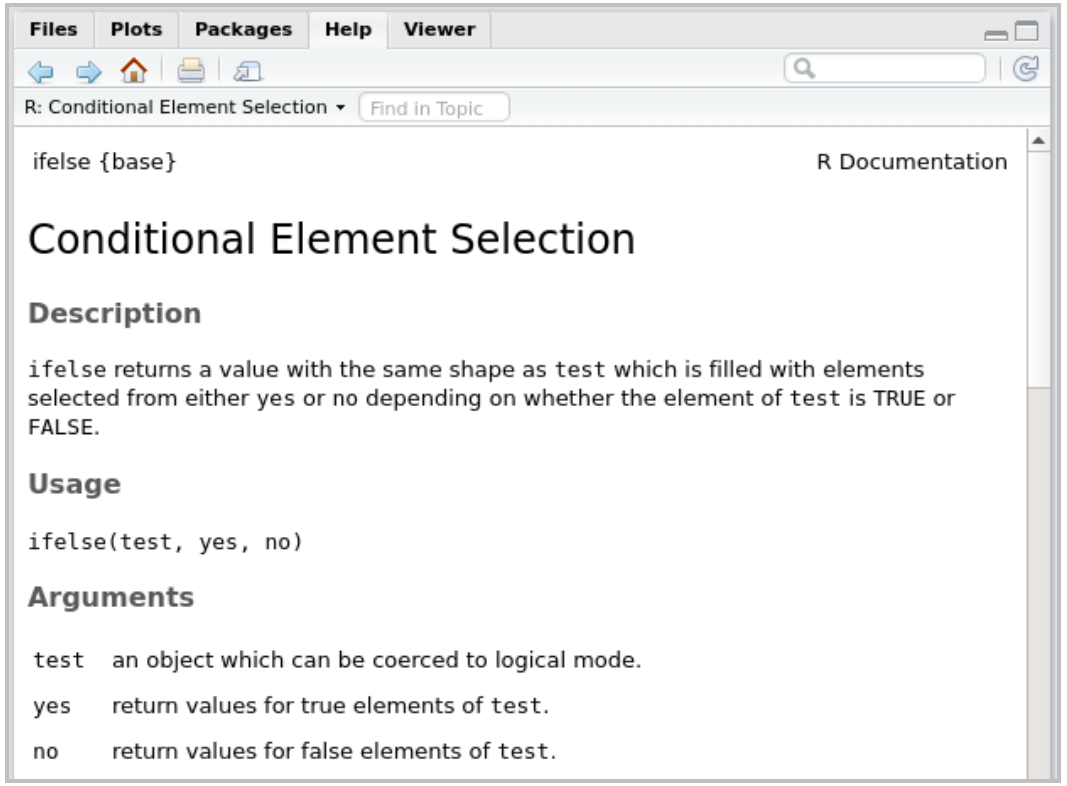
Control structures and functions

- No introduction to any programming language is complete without learning about control structures and functions
- If you understand the data types, basic data structures, control structures, and function definition, you will be able to complete most of the tasks related to problem solving using programming languages
- We will introduce you to
  - Conditional statements using `if`, `if...else`, and `ifelse`
  - Loops using `for`
  - Function definitions using `function`

Conditionals: `ifelse` function

```
?ifelse
```

- The simplest conditional is the `ifelse` function. It has 3 arguments:
  - the condition for which we are testing (i.e. the `test`)
  - the value that is returned in case the condition specified is met
  - the value that is returned in case the condition specified is NOT met



Keep in mind, `ifelse` must return a value! This function is identical to `ifelse` in Excel and similar to single / inline if-else constructs in other programming languages.

Ifelse example

- Let's say we want to take `Yield` from the `CMP` dataset and convert it to either `above average` or `below average`
- Let's demonstrate how we can use `ifelse` here

```
meanCMP_yield = mean(CMP$Yield)

CMP$new_yield = ifelse(CMP$Yield >= meanCMP_yield,    #<- if CMP$Yield is greater than
                                                      #   or equal to the mean of Yield
                       "above_average",               #<- Then new_yield = above average
                       "below_average")               #<- Else new_yield = below average


head(CMP[,c("Yield","new_yield")])
```

```
  Yield     new_yield
1 38.00 below_average
2 42.44 above_average
3 42.03 above_average
4 41.42 above_average
5 42.49 above_average
6 43.57 above_average
```

Loops: `for` loop

A `for` loop is used when we have a **finite** set of distinct repeated actions to be performed:

- It has an explicit `start` and `end`
- Arguments to `for` loop can take several forms, the most common one includes
  - an arbitrary `counter` or `index` variable
  - the `in` word to indicate that the counter is an element of a sequence on its right hand side
  - a `sequence` of indices through which to **loop**, defined in the `start:end` format

```
# Basic for loop.
for(i in 1:num_of_repetitions){
  perform action on element at index i
}
```

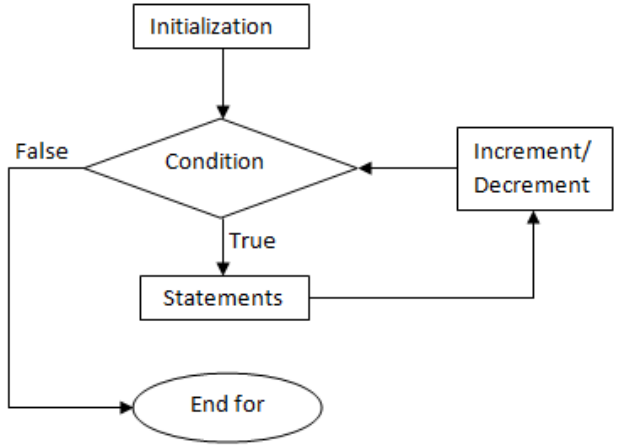


fig: Flowchart for for loop

*Index `i` is an arbitrary letter that we use to let the loop know which element is **current**. We pass that variable (i.e. index) to the data object, so the work is ONLY done on the **current** (i.e. `i-th`) element of the object.*

Defining start and end of loop

- We can give the start and end points of the loop in different ways:
  - give the numbers in the index
  - give variables set equal to index numbers
  - go from 1 to length of list
- In this case, we only want to print the variable names that start at index `3` and end at index `6`
- We only need to adjust the `start` and `end` indices in the `for` loop

```
CMP_subset_variables = colnames(CMP_subset)

# Adjust the start index.
seq_start = 3

# Adjust the end index.
seq_end = 6

# Loop through the seq_start and seq_end
# variables.
for(i in seq_start:seq_end){
  print(CMP_subset_variables[i])
}
```

```
[1] "BiologicalMaterial02"
[1] "BiologicalMaterial03"
[1] "ManufacturingProcess01"
[1] "ManufacturingProcess02"
```

Functions in R

- **Functions** are chunks of code that allow you to:
  - generalize your code, so that you can
  - re-use your code later
- They make your code:
  - more **abstract** so that it can be used with different data and/or parameters
  - more **modular** so that it can be used as a part of another larger chunk of code, script or even a program
  - **clean**, as they isolate actions performed and allow you to trace the flow of your code with ease

```
# Basic function with no arguments.
function(){
  perform action
}

# Basic function with 1 argument.
function(argument){
  perform action given argument
}

# Basic function with 2 (or more) arguments.
function(argument1, argument2){
  perform action given argument1, argument2
}
```

Function without arguments

```
# Make a function that prints "Hello" and
# assign it to `PrintHello` variable.

PrintHello = function(){ #<- declare function
  print("Hello!")        #<- perform action
}

# Invoke function by calling `PrintHello()`.
PrintHello()
```

```
[1] "Hello!"
```

```
# Make function that returns the first few
# digits of `pi` and assign to `GetPi` variable.

GetPi = function(){       #<- declare function
  pi_num = 3.14159265359 #<- compute value
  return(pi_num)          #<- return value
}

# Invoke function by calling `GetPi()`.
GetPi()
```

```
[1] 3.141593
```

The function definition consists of 2 main parts:

1. The chosen function name set equal to the `function` keyword, followed by empty `()`
2. The body of the function that is defined within the `{}`, it can either:
   i. perform an **action**
   ii. return a specific value

Function with arguments

```
# Make a function that prints "Hello, [name]".
PrintHello = function(name){    #<- add `name` argument to function declaration

  # Save message to print to a variable.
  hello_name = paste0("Hello ", #<-  concatenate "Hello "
                      name,     #<-  with the `name` from function argument, and
                      "!")      #<-  with the remainder of the message to print

  print(hello_name)             #<- print message
}

# Invoke function by calling `PrintHello([name])`.
PrintHello("User")
```

```
[1] "Hello User!"
```

Setting default arguments for functions

```
# Make function that rounds to the first `n` digits of `pi`.
GetPi = function(n){              #<- add `n` argument to function declaration

  pi_num = round(3.14159265359, #<- round `pi`
                 n)             #<- to `n` digits
  return(pi_num)
}

# Invoke function by calling `GetPi([n])`.
GetPi(3)
```

```
[1] 3.142
```

Call function without arguments

- What happens if you try and invoke the function that requires arguments without passing the argument to it?
- It either fails or returns unexpected results!
- To overcome potential errors or getting results that we don't expect, we can set default arguments to functions

```
PrintHello()
```

```
Error in paste0("Hello ", name, "!") :
  argument "name" is missing, with no default
```

```
GetPi()
```

```
[1] 3
```

Wrapping it all into function

- To define a function, we need to assign it to a variable (i.e. `ImputeNAsWithMean`) and add an argument to `()`
- We then need to substitute every instance of specific dataset name with our argument (i.e. `dataset`)
- We need to `return` the updated `dataset` at the end of the function

Create a function for imputing NAs

```
ImputeNAsWithMean = function(dataset){

  for(i in 1:ncol(dataset)){
    is_na = is.na(dataset[, i])
    if(any(is_na)){
      na_ids = which(is_na)
      var_mean = mean(dataset[, i],
                      na.rm = TRUE)
      dataset[na_ids, i] = var_mean
      message = paste0(
        "NAs substituted with mean in ",
        colnames(dataset)[i])
      print(message)
    }
  }
  return(dataset)
}
```

Congratulations on creating your first function in R!

Impute NAs with mean using custom function

```
# Let's re-generate our subset again.
CMP_subset = CMP[, c(1:4, 14:16)]

# Let's test the function giving the
# `CMP_subset` as the argument.
CMP_subset_imputed = ImputeNAsWithMean(CMP_subset)
```

```
[1] "NAs substituted with mean in ManufacturingProcess01"
[1] "NAs substituted with mean in ManufacturingProcess02"
[1] "NAs substituted with mean in ManufacturingProcess03"
```

```
# Inspect the structure.
str(CMP_subset_imputed)
```

```
'data.frame':   176 obs. of  7 variables:
 $ Yield                 : num  38 42.4 42 41.4 42.5 ...
 $ BiologicalMaterial01  : num  6.25 8.01 8.01 8.01 7.47 6.12 7.48 6.94 6.94 6.94 ...
 $ BiologicalMaterial02  : num  49.6 61 61 61 63.3 ...
 $ BiologicalMaterial03  : num  57 67.5 67.5 67.5 72.2 ...
 $ ManufacturingProcess01: num  11.2 0 0 0 10.7 ...
 $ ManufacturingProcess02: num  16.7 0 0 0 0 ...
 $ ManufacturingProcess03: num  1.54 1.54 1.54 1.54 1.54 ...
```

Module completion checklist

Objective	Complete
Identify the various ways to read, write and view data	✔
Save the environment and environment variables	✔
Clear environment	✔
Load environment and environment variable	✔
Detect and address missing values in data	✔
Manipulate data types and structures using flow control structures (for loops, conditionals, etc)	✔
Save and load custom functions into R environment as a module	

Creating a module with functions

- Since we have learned how to create functions to help us optimize our programming style, we now need a consolidated way to load them into the environment without having to copy paste the code into each script
- We will create a simple function **module**, which is a regular `R` script that only contains the functions we would like to use
- We've copied some of the functions that we created in class into `custom_functions.R` file that is located in `main_dir` (your main class folder)

Creating a module with functions

custom_functions.R

Source on Save | Run | Source

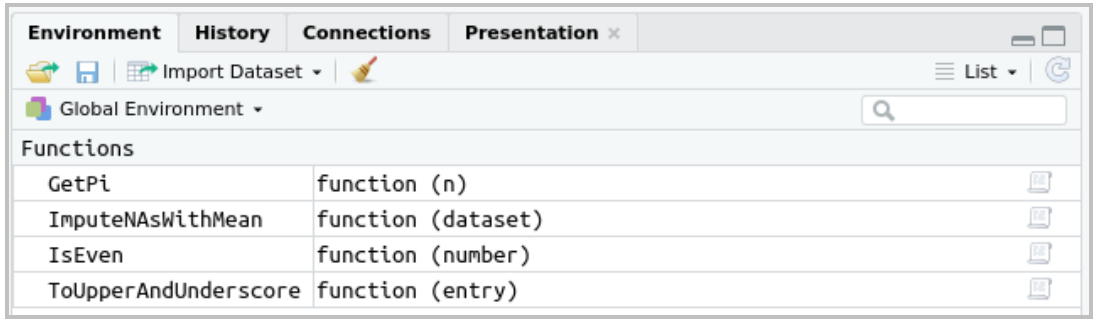
```r
# Make function that rounds to the first `n` digits of `pi`.
GetPi = function(n){           #<- Add `n` argument to function declaration

  pi_num = round(3.14159265359, #<- Round `pi`
                 n)             #<-   to `n` digits
  return(pi_num)
}

# Impute NAs for numeric dataset.
ImputeNAsWithMean = function(dataset){

  for(i in 1:ncol(dataset)){
    is_na = is.na(dataset[, i])
    if(any(is_na)){
      na_ids = which(is_na)
      var_mean = mean(dataset[, i],
                      na.rm = TRUE)
      dataset[na_ids, i] = var_mean
      message = paste0(
        "NAs substituted with mean in ",
        colnames(dataset)[i])
      print(message)
    }
  }
  return(dataset)
}

# A function that converts characters to upper case
# and replaces space with underscore.
ToUpperAndUnderscore = function(entry) {
  gsub(" ", "_",
       toupper(entry))
}

# A function that determines whether the number is even and
# returns either TRUE or FALSE.
IsEven = function(number){  #<- the function takes a number as input
  if(number %% 2 == 0){     #<- checks if the remainder after division by 2 is zero
    TRUE                    #<- returns `TRUE` if it is
  }else{                    #<- otherwise
    FALSE                   #<- it returns `FALSE`
```

Loading a module with functions

- In order to load the module with R code into our environment, we need to use a function `source`

```
# Set your working directory to `main_dir`.
setwd(main_dir)

# Call `source` function with the file name of the module.
source("custom_functions.R")
```

- If the file name was correct, the content of the module (i.e. your custom functions) should now appear in the R environment under `Functions` section



Test your custom functions

- Let's test some of the functions to see if it's loaded and works correctly!

```
# Get Pi rounded to 2 decimal points.
GetPi(2)
```

```
[1] 3.14
```

```
# Is 2345 an even number?
IsEven(2345)
```

```
[1] FALSE
```

- Now you can define custom functions and easily load them into your environment form a single module file!

Knowledge check 3

Exercise 3

Module completion checklist

Objective	Complete
Identify the various ways to read, write and view data	✔
Save the environment and environment variables	✔
Clear environment	✔
Load environment and environment variable	✔
Detect and address missing values in data	✔
Manipulate data types and structures using flow control structures (for loops, conditionals, etc)	✔
Save and load custom functions into R environment as a module	✔

Workshop!

- **Today will be your first *after class* workshop**
- Workshops are to be completed outside of class and emailed to the instructor by the beginning of class tomorrow
- Make sure to comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Workshop objectives:
  - Read in chicago_census.csv dataset into a new dataframe and perform various operations on a dataframe
  - Work with environment by storing, clearing and loading the environment
  - Practice working on control flows

This completes our module

Congratulations!