DATA SOCIETY®

Intro to R - Part 2

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

Welcome back!

Before we start, check out this list of projects you can complete with R:
https://www.upgrad.com/blog/data-science-projects-in-r-for-beginners/

Recap: before we can run....basics

- In the last class you **named your own variables** and **performed basic operations** in R to get familiar with the coding environment
- Today, we are going to learn more about vectors, matrices and how to perform various operations on them

Recap: variables and assignment operators

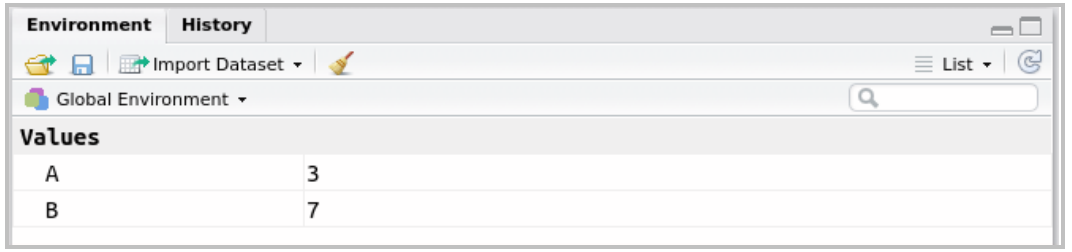
```
# Define a variable using `<-`
# as an assignment operator.
A <- 3
A
```

```
[1] 3
```

```
# Define a variable using `=`
# as an assignment operator.
B = 2 + 5
B
```

```
[1] 7
```

Notice that you not only can assign numbers to variables, you can assign any expression to a variable!



- You can set variables by setting numbers equal to letters or terms. R has **two** assignment operators: <– and =
- When a variable is named (instantiated), R stores it in its "environment"
- R session uses the values stored within its environment for all calculations within that session

Recap: operations with variables

Adding

- You can add variables

```
# Add 2 variables.
C = A + B
C
```

```
[1] 10
```

```
# Add a variable and a number.
D = C + 5
D
```

```
[1] 15
```

The same stands for all other arithmetic operations!

Subtracting

- You can subtract variables

```
# Subtract 2 variables from each other.
D - C
```

```
[1] 5
```

```
# Subtract a variable from number.
33 - D
```

```
[1] 18
```

```
# Or a number from a variable.
D - 33
```

```
[1] -18
```

Recap: Basic data classes and types

- **Data type** describes how internal R language stores our data, while **data class** is more generic and determined by the object-oriented programming mechanism behind R
- In most business cases, we do not distinguish between data types and data classes
- The point is to adopt the data type or data class that fits best

Data class (high level)	Data type (low level)	Example
Integer	Integer	`-1, 5, or 1L, 5L`
Numeric	Double, float	`2.54`
Character	Character	`"Hello"`
Logical	Logical	`TRUE, FALSE`

- *Note: One of the common sources of errors for a person learning to use any programming language is the data type conversion.*

Recap: Basic data classes: what we will use

- To generate more insights within our data, here is a list of functions we can use

Item	Purpose
Value	Example of class
`typeof()`	Finds the type of the variable
`class()`	Returns the class of the variable
boolean function	Specific function that checks class and returns TRUE or FALSE
`attributes()`	Checks the metadata/attribute of the variable
`length()`	Checks the length of the object

Recap: Basic data structures

- In the previous session, we have learned some of the most basic as well as common data types
- Next, we are going to focus on groupings of one or more data types organized in various ways - **data structure**
- A data structure is a method for **describing a certain way to organize pieces of data**, so operations and algorithms can be easily applied

Data structure	Number of dimensions	Single data type	Multiple data types
Vector (Atomic vector)	1 (entries)	✔	✘
Vector (List)	1 (entries)	✔	✔
Matrix	2 (rows and columns)	✔	✘
dataframe	2 (rows and columns)	✔	✔

Module completion checklist

Objective	Complete
Introduce list and dataframe in R	
Perform different operations on the above data types and structures	
Read/write data	
Clear environment	

Basic data structures: lists

- A `list` is a collection of entries that act as a **container**
- It has a **single dimension** at its top level
- It can be called as a `generic vector` because a list can hold items of **different types**
- Lists can be **nested** which means that a `list` can contain elements that are also `lists`

- *Note: If you have ever worked with `JSON` files, they can be translated naturally into the `list` data structure.*

Basic data structures: lists

- Creating `lists`

```
# To make an empty list in R,
# you have a few options:
# Option 1: use `list()` command.
list()
```

```
list()
```

- How is this different from a vector?

```
# Make a list with different entries.
sample_list = list(1, "am", TRUE)
sample_list
```

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

[[2]]
[1] "am"

[[3]]
[1] TRUE
```

Basic data structures: naming list elements

- Lists can have *attributes* such as `names`
- You can name list elements when you **create** a `list`

```
# Create a named list.
sample_list_named = list(One = 1,
                         Two = "am",
                         Three = TRUE)
sample_list_named
```

```
$One
[1] 1

$Two
[1] "am"

$Three
[1] TRUE
```

```
attributes(sample_list_named)
```

```
$names
[1] "One"   "Two"   "Three"
```

- You can also set element names **after** it has been created

```
# Name existing list.
names(sample_list) = c("One", "Two", "Three")
sample_list
```

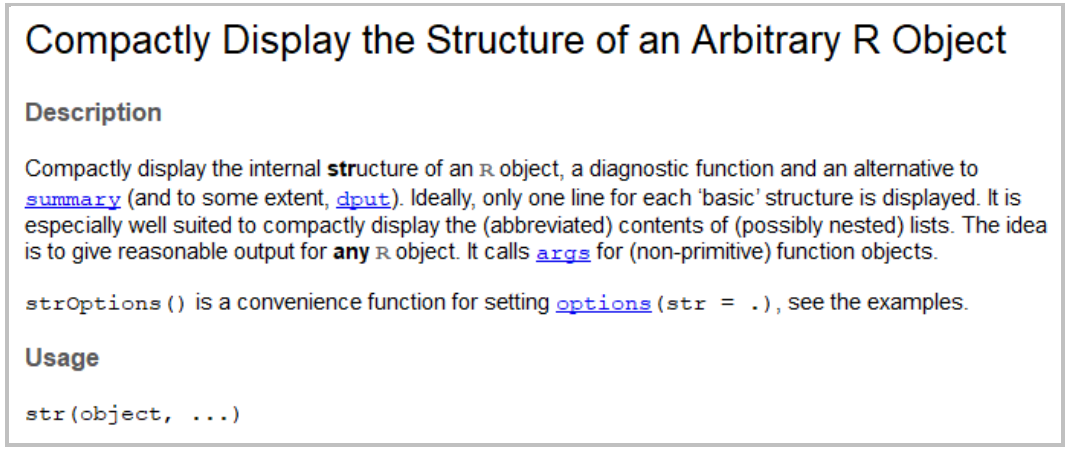
```
$One
[1] 1

$Two
[1] "am"

$Three
[1] TRUE
```

```
attributes(sample_list)
```

```
$names
[1] "One"   "Two"   "Three"
```

Basic data structures: introducing structure

```
?str         #<- Check R documentation
str(object)  #<- Any R object
```



Compactly Display the Structure of an Arbitrary R Object

Description

Compactly display the internal **str**ucture of an R object, a diagnostic function and an alternative to summary (and to some extent, dput). Ideally, only one line for each 'basic' structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists. The idea is to give reasonable output for **any** R object. It calls args for (non-primitive) function objects.

strOptions() is a convenience function for setting options(str = .), see the examples.

Usage

```
str(object, ...)
```

```
# Inspect the list's structure.
str(sample_list)
```

```
List of 3
 $ One  : num 1
 $ Two  : chr "am"
 $ Three: logi TRUE
```

- Command `str` lets you inspect the structure of any R object such as a list or a dataframe
- It returns:
  - The `class` of the object (e.g. `List`)
  - The `length` of the object (e.g. `3`)
  - Snippet of each entry and its `type` (e.g. `One: num 1`, `Two: chr "am"`, `Three: logi TRUE`)

Basic data structures: accessing data within lists

- To access an element in a list, you can use its **index**

```
# Access an element of a list.
sample_list[[2]]
```

```
[1] "am"
```

```
# Access a sub-list with its element(s).
sample_list[2]
```

```
$Two
[1] "am"
```

```
# Access a sub-list with its element(s).
sample_list[2:3]
```

```
$Two
[1] "am"

$Three
[1] TRUE
```

- You can also refer to an element by its **name**, using the `$` operator (as seen in the output of the `str` command)

```
# Access named list elements.
sample_list$One
```

```
[1] 1
```

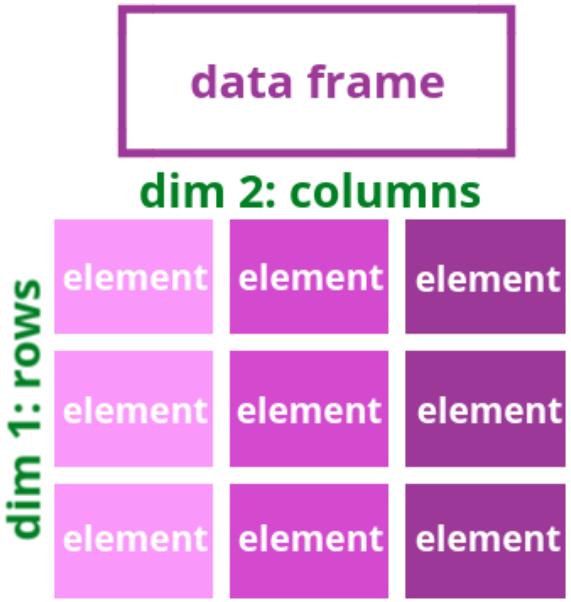
```
sample_list$Two
```

```
[1] "am"
```

```
sample_list$Three
```

```
[1] TRUE
```

Basic data structures: dataframes



- *Note: if you have ever worked with relational databases, you can think of a dataframe as a table in a relational database*

- A `data.frame` is a special kind of `list`, which is limited to a **2D structure**
- Each entry in a `list` is a `column`
- Each `column` has the same number of *entries*
- Columns can be of **different types** (e.g. `character`, `numeric`, `logical`)
- But within each column, the entries are always of the **same type**, which makes each column of a `data.frame` an `atomic vector`
- It combines properties of both `lists` and `atomic vectors`, which makes dataframe a *de facto* standard data structure for use in data analysis

Basic data structures: making dataframes

```
# To make an empty dataframe in R,
# use `data.frame()` command.
data.frame()
```

```
data frame with 0 columns and 0 rows
```

```
# To make a dataframe with several
# columns, pass column values
# to `data.frame()` command just like
# you would do with lists.
data.frame(1:5, 6:10)
```

```
  X1.5 X6.10
1    1     6
2    2     7
3    3     8
4    4     9
5    5    10
```

- As with vectors, matrices, & lists, a `data.frame` can be created empty
- Column values can be passed directly to dataframes when they are created as you would with lists
- You can also combine pre-existing vectors

- *Note: without defined column names `data.frame` auto-generates them. Column names in R cannot have numbers as the first character, which is why R appends `X` to them!*

Dataframes: naming columns

- Use `colnames` to rename columns after `data.frame` is created

```
# Dataframe with unnamed columns.
unnamed_df = data.frame(1:3, 4:6)
unnamed_df
```

```
  X1.3 X4.6
1    1    4
2    2    5
3    3    6
```

```
# Name columns of a dataframe.
colnames(unnamed_df) = c("col1", "col2")
unnamed_df
```

```
  col1 col2
1    1    4
2    2    5
3    3    6
```

- Name columns at the time of creation of the `data.frame`

```
# Pass column names and values to
# `data.frame` command just like you
# would do with named lists.
named_df = data.frame(col1 = 1:3, col2 = 4:6)
named_df
```

```
  col1 col2
1    1    4
2    2    5
3    3    6
```

Dataframes: naming rows

- In addition to column names, you can also **rename** row names of any dataframe with `rownames`

```
# View dataframe.
named_df
```

```
  col1 col2
1    1    4
2    2    5
3    3    6
```

```
# Rename dataframe rows.
rownames(named_df) = c(7:9)
named_df
```

```
  col1 col2
7    1    4
8    2    5
9    3    6
```

- Similarly, you can also create a dataframe *and* **define** row names with method `row.names` at the time of its creation

```
# Define row names explicitly,
# use a `row.names` argument.
data.frame(col1 = 1:3,
           col2 = 4:6,
           row.names = 7:9)
```

```
  col1 col2
7    1    4
8    2    5
9    3    6
```

Dataframes: converting a matrix

- We can make a dataframe from a matrix by casting a `matrix` into a `data.frame` with `as.data.frame` command

```
# Make a dataframe from matrix.
sample_matrix1 = matrix(nrow = 3, ncol = 3)
sample_matrix1 = 1:9
dim(sample_matrix1) = c(3, 3)

sample_df1 = as.data.frame(sample_matrix1)
sample_df1
```

```
  V1 V2 V3
1  1  4  7
2  2  5  8
3  3  6  9
```

```
# Make a dataframe from matrix with named
columns and rows.
sample_df2 = as.data.frame(sample_matrix1,
row.names = c('Row1','Row2','Row3'))
cols = c('Col1','Col2','Col3') # defining the
column names
colnames(sample_df2) = cols   # assigning the
column names to df

sample_df2
```

```
     Col1 Col2 Col3
Row1    1    4    7
Row2    2    5    8
Row3    3    6    9
```

Dataframes: row and column names

```
# Check attributes of a dataframe.
attributes(sample_df1)
```

```
$names
[1] "V1" "V2" "V3"

$class
[1] "data.frame"

$row.names
[1] 1 2 3
```

- **Unnamed** `dataframe` **column names** will default to `V1, V2, ..., Vm`, where `m = num columns` of a dataframe
- **Unnamed** `dataframe` **row names** will default to `1, 2, ..., n`, where `n = num rows` of a dataframe

```
# Check the attributes of dataframe.
attributes(sample_df2)
```

```
$names
[1] "Col1" "Col2" "Col3"

$class
[1] "data.frame"

$row.names
[1] "Row1" "Row2" "Row3"
```

- **Named** `dataframe` **column names** will become `data.frame` column names
- **Named** `dataframe` **row names** will become `data.frame` row names

Dataframes: selecting columns

- Let's explore the different methods we have covered thus far for selecting columns from a `data.frame`
  - Use `$column_name`
  - Use `[[column_index]]`
  - Use `[ , column_index]`

```
# To access a column of a dataframe
# Option 1: Use `$column_name`.
sample_df2$Col1
```

```
[1] 1 2 3
```

```
# To access a column of a dataframe
# Option 2: Use `[[column_index]]`.
sample_df2[[1]]
```

```
[1] 1 2 3
```

```
# To access a column of a dataframe
# Option 3: Use `[ , column_index]`.
sample_df2[, 1]
```

```
[1] 1 2 3
```

Dataframes: subsetting rows

- Let's explore a few methods for selecting a row from a `data.frame`
  - Use `[row_index, ]`
  - Use `["row_name", ]`

```
# To access a row of a dataframe
# Option 1: use `[row_index, ]`.
sample_df2[1, ]
```

```
     Col1 Col2 Col3
Row1    1    4    7
```

```
# To access a row of a dataframe
# Option 2: use `["row_name", ]`.
sample_df2["Row1", ]
```

```
     Col1 Col2 Col3
Row1    1    4    7
```

Dataframes: accessing individual values

- There are four common methods for accessing individual values within a `data.frame`
  - Use `$column_name[row_index]`
  - Use `[[column_index]][row_index]`
  - Use `[row_index, column_index]`
  - Use `["row_name", "column_name"]`

```
# Option 1:
# `data_frame$column_name[row_index]`
sample_df2$Col2[1]
```

```
[1] 4
```

```
# Option 2:
# `data_frame[[column_index]][row_index]`
sample_df2[[2]][1]
```

```
[1] 4
```

```
# Option 3:
# `data_frame[row_index, column_index]`
sample_df2[1, 2]
```

```
[1] 4
```

```
# Option 4:
# `data_frame["row_name", "column_name"]`
sample_df2["Row1", "Col2"]
```

```
[1] 4
```

Dataframes: adding new columns

- Another common case is adding new columns into an existing dataframe
  - Use `$new_column_name`
  - Use `cbind`

```
# To add a new column to a dataframe
# Option 1: use `$new_column_name`.
sample_df2$Col4 = "New column"
sample_df2
```

```
     Col1 Col2 Col3       Col4
Row1    1    4    7 New column
Row2    2    5    8 New column
Row3    3    6    9 New column
```

```
# To add new column(s) to a dataframe
# Option 2: use `cbind`.
sample_df2 = cbind(sample_df2,
                   Col5 = c("Yet another",
                            "new",
                            "column"))
sample_df2
```

```
     Col1 Col2 Col3       Col4        Col5
Row1    1    4    7 New column Yet another
Row2    2    5    8 New column         new
Row3    3    6    9 New column      column
```

Dataframes: operations

```
# Let's take our sample dataframe.
str(sample_df2)
```

```
'data.frame':   3 obs. of  5 variables:
 $ Col1: int  1 2 3
 $ Col2: int  4 5 6
 $ Col3: int  7 8 9
 $ Col4: chr  "New column" "New column" "New column"
 $ Col5: chr  "Yet another" "new" "column"
```

```
# Add a number to each value in a column.
sample_df2$Col1 + 2
```

```
[1] 3 4 5
```

```
# Add a number to each value in a row.
sample_df2[1, ] + 2
```

```
Error in FUN(left, right) : non-numeric argument to binary operator
```

Knowledge Check 1

Special classes: factors

```
# Let's take a look at the structure of the dataframe.
str(sample_df2)
```

```
'data.frame':   3 obs. of  5 variables:
 $ Col1: int  1 2 3
 $ Col2: int  4 5 6
 $ Col3: int  7 8 9
 $ Col4: chr  "New column" "New column" "New column"
 $ Col5: chr  "Yet another" "new" "column"
```

- Our talk about data types and structures in R is not complete without a special class **`factor`**
- A `factor` is a class of variable that is used to quantify **categorical** data
- Both numeric and character variables can be made into factors, but a factor's levels will always be character values
- Every `factor` variable has `levels`, which are unique instances of the values in the column (e.g. `Col5` has 3 unique values, hence 3 levels)
- Use `levels()` to find the number of unique values of a factor

Special classes: dates

```
# Let's make a dataframe.
special_data = data.frame(date_col1 = c("2018-01-01", #<- make a column with character strings
                                        "2018-02-01", #   in the format of date (YYYY-MM-DD)
                                        "2018-03-01"),
                          stringsAsFactors = FALSE)   #<- this option allows us to tell R
                                                      #   to NOT interpret strings as `factors`
special_data
```

```
   date_col1
1 2018-01-01
2 2018-02-01
3 2018-03-01
```

```
# Take a look at the structure.
# Notice both columns appear as `character` and not as `factor`.
str(special_data)
```

```
'data.frame':   3 obs. of  1 variable:
 $ date_col1: chr  "2018-01-01" "2018-02-01" "2018-03-01"
```

Special classes: dates and basic formats

- Given a character string of a particular format, we can convert to a `Date` using `as.Date` function (e.g. `YYYY-MM-DD` format will be automatically detected by R)

```
# Let's make another vector with dates, but in
# a different format.
new_dates = c("January 1, 2018",
              "February 1, 2018",
              "March 1, 2018")

# Let's add another column to the dataframe
# and save it as a Date with a special format.
special_data$date_col2 = as.Date(new_dates,
                                 format = "%B %d, %Y")
special_data
```

```
   date_col1  date_col2
1 2018-01-01 2018-01-01
2 2018-02-01 2018-02-01
3 2018-03-01 2018-03-01
```

- Here is a table of common widgets for dates and their corresponding meanings

Code	Value
`%d`	Day of the month (number)
`%m`	Month (number)
`%b`	Month (abbreviated name)
`%B`	Month (full name)
`%y`	Year (2 digit)
`%Y`	Year (4 digit)

Special values: `NA`

- Missing values is another common issue
- `is.na` helps identify `NA` values
- We will illustrate this now:

```
# Let's add a column with a numeric vector.
special_data$num_col1 = c(1, 555, 3)

# Let's make the 2nd element in that column `NA`.
special_data$num_col1[2] = NA

# To check for `NA`s we use `is.na`.
is.na(special_data$num_col1[2])
```

```
[1] TRUE
```

```
# We can also use it to check the whole column/vector.
# The result will be a vector of `TRUE` or `FALSE` with values corresponding to each element.
is.na(special_data$num_col1)
```

```
[1] FALSE  TRUE FALSE
```

Special values: `NULL`

- Another special value in R is `NULL`
- This value causes an object, or a part of the object, to be `NULL`ified, i.e. removed or cleared

```
# To get rid of a column in a `data.frame` all
# you have to do is set it to `NULL`.
special_data$num_col3 = NULL
special_data
```

```
   date_col1  date_col2 num_col1
1 2018-01-01 2018-01-01        1
2 2018-02-01 2018-02-01       NA
3 2018-03-01 2018-03-01        3
```

```
# To check for `NULL`s use `is.null`.
is.null(special_data$num_col3)
```

```
[1] TRUE
```

```
# To check for `NULL`s use `is.null`.
is.null(special_data$num_col2)
```

```
[1] TRUE
```

Knowledge Check 2

Exercise 1

Module completion checklist

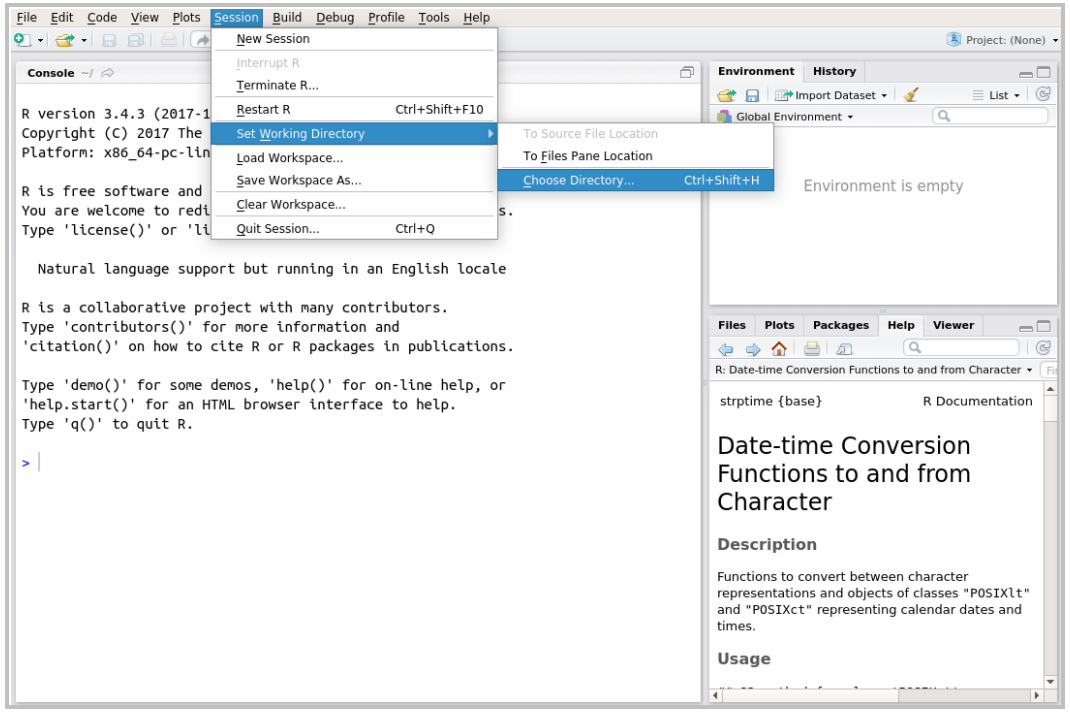
Objective	Complete
Introduce list and dataframe in R	✔
Perform different operations on the above data types and structures	✔
Read/write data	
Clear environment	

R's working directory

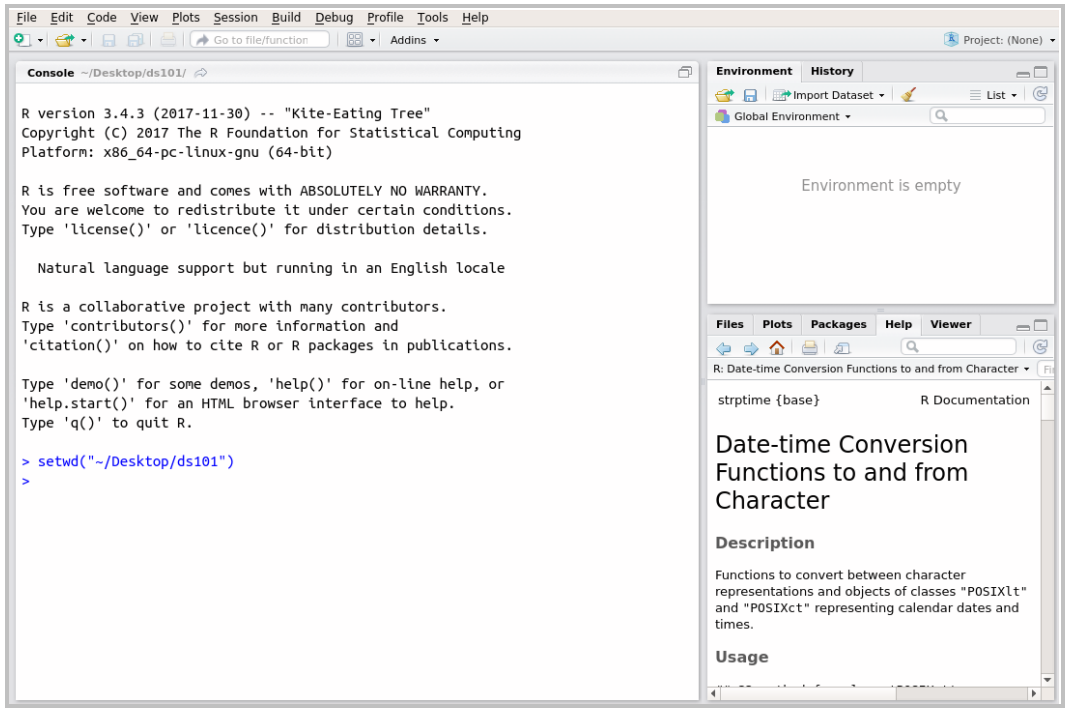
- The **working directory** is a folder on your machine (which R treats as your "sandbox") where R saves your files and from which it loads your data
- R has a **default** working directory, which can be found and set through RStudio's `Global Options`
- We can `set` the working directory
- We can `get`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

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`



R's working directory

- 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/skillsoft-2021")

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

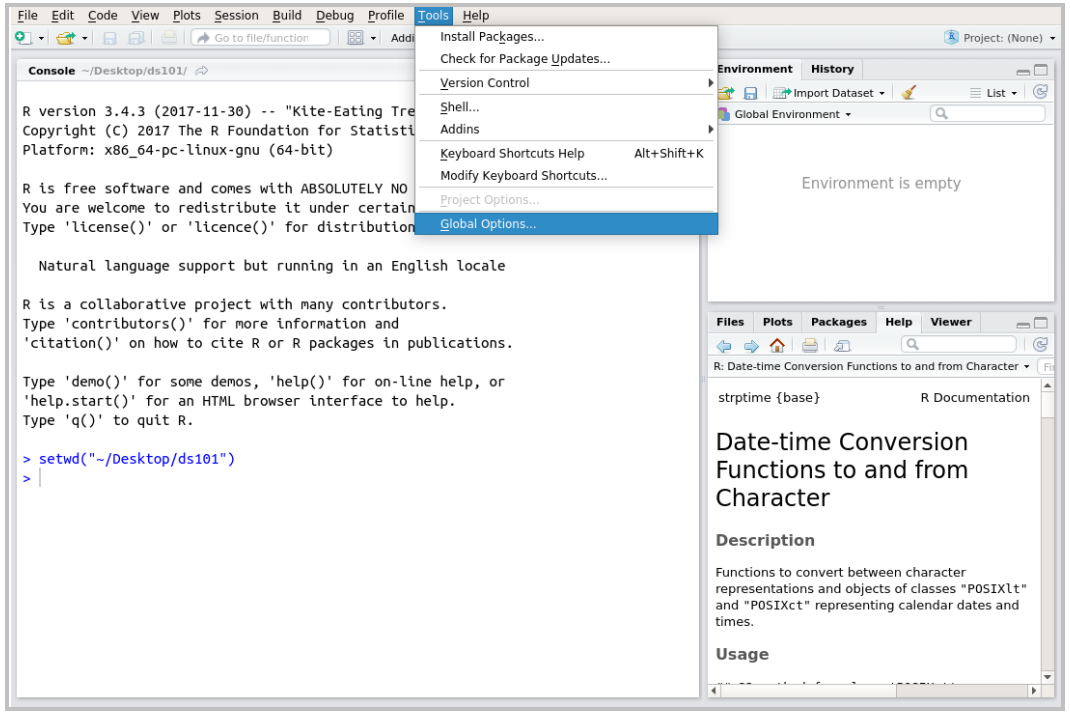
```
[1] "/home/[your-user-name]/Desktop/skillsoft-2021"
```

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

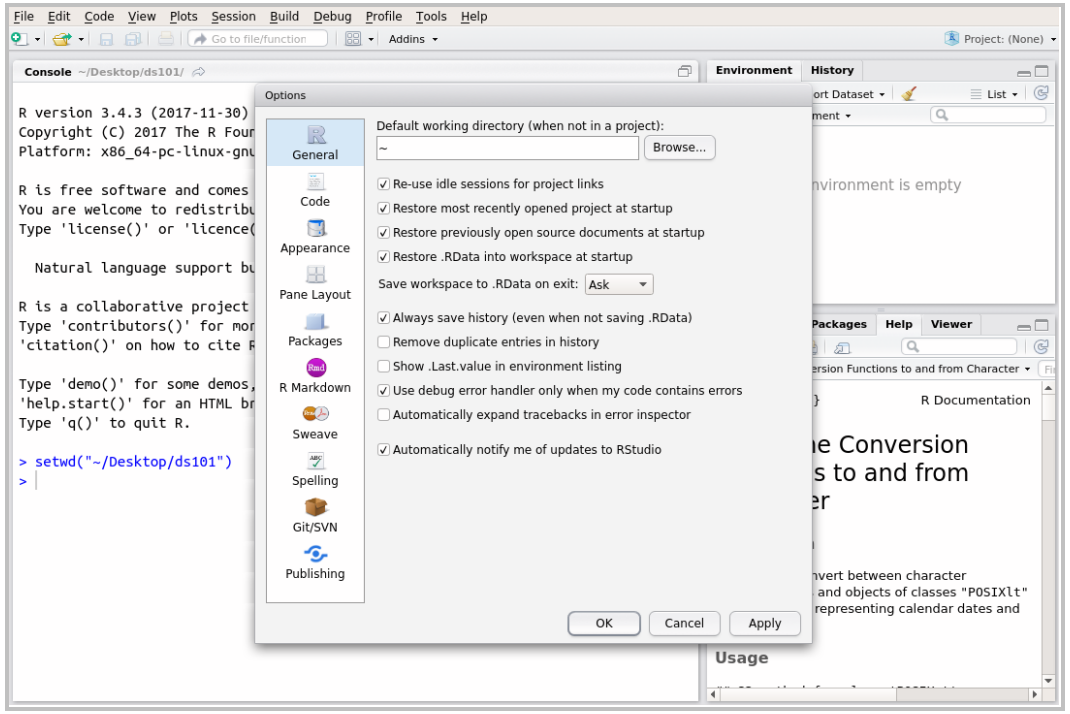
```
# To set working directory call `setwd` with the path to the folder.
setwd("C:/Users/[your-user-name]/Desktop/skillsoft-2021")

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

```
[1] "C:/Users/[your-user-name]/Desktop/skillsoft-2021"
```

R's default working directory



- You can also set a default working directory for whenever R is launched



- Look at the very first option in the `General` section of the `Global Options` to see what the current working directory is
- To change it just click on `Browse` and select a default working directory

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 `skillsoft-2021` folder

```
# Set `main_dir` to the location of your `skillsoft-2021` folder (for Mac/Linux).
main_dir = "~/Desktop/skillsoft-2021"

# Set `main_dir` to the location of your `skillsoft-2021` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/skillsoft-2021"

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

Directory settings

1. We will store all data sets in the `data` directory inside of the `skillsoft-2021` 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 `skillsoft-2021` 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")

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

# Set directory to data_dir.
setwd(data_dir)
```

Directory settings

- Now all you have to do to switch between working directories is 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/skillsoft-2021/data"
```

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

```
[1] "C:/Users/[your-user-name]/Desktop/skillsoft-2021/data"
```

Loading dataset into R: read CSV files

- Most of the time you will be working with data that was generated elsewhere which you will then need to load into 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

- First, we can take a general look into our dataset structure with `str()`

```
# 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 ...
```

Viewing data in R

- Then, we can inspect the head or tail of our data with `head()` or `tail()` function
- By default, `head()` will give you the **first six** rows and `tail()` will give you the **last six**
- However, you can also adjust the number of rows as the following example illustrates

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

```
  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
```

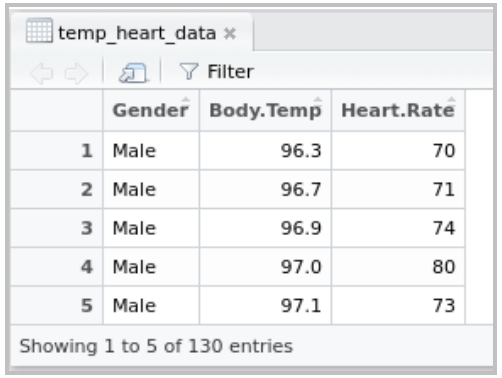
```
tail(temp_heart_data, 4) #<- Inspect the `tail` (last 4 rows).
```

```
    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
```

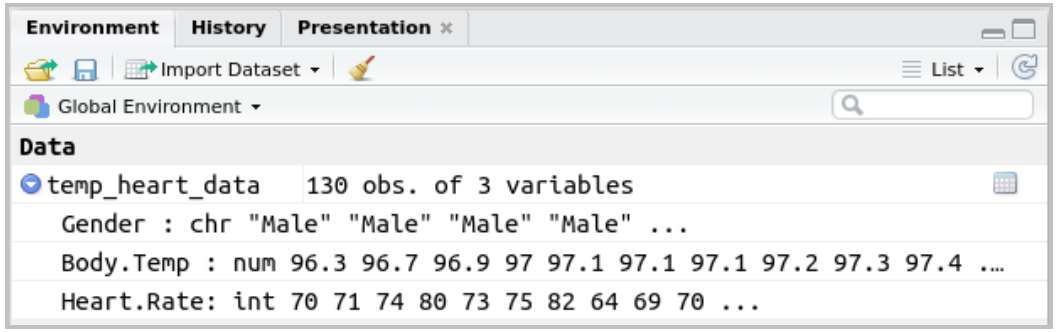
Viewing data in R

- View in the tabular data explorer

```
View(temp_heart_data)
```



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



Other file types and commands in R

- The following is a list of commands to read data in other file types

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
`read.JPEG("filename.jpg")`	Read JPEG image files

Saving data: write CSV files

- The most common way to share tabular data is by 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
Introduce list and dataframe in R	✔
Perform different operations on the above data types and structures	✔
Read/write data	✔
Clear environment	

Clearing objects from environment

```
# List all objects in environment.
ls()
```

```
 [1] "A"                "B"                "C"
 [4] "cols"             "D"                "data_dir"
 [7] "directory"        "head"             "highlight_js"
[10] "main_dir"         "named_df"         "new_dates"
[13] "platform"         "plot_dir"         "sample_df1"
[16] "sample_df2"       "sample_list"      "sample_list_named"
[19] "sample_matrix1"   "session_info"     "special_data"
[22] "temp_heart_data"  "temp_heart_subset" "unnamed_df"
```

```
# Remove individual variable(s).
rm(X, x, this_is_a_valid_name, This.Is.Also.A.Valid.Name, unnamed_list) #<- example
rm(list=ls()) #<- actual command
```

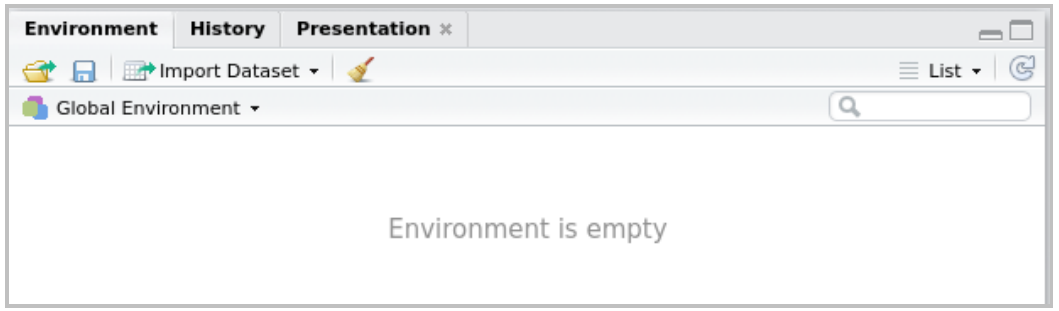
```
# List all objects again to check.
ls()
```

```
character(0)
```

Notice the variables we have removed are gone!

Clearing the entire environment

- The clear environment will always appear 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.

Knowledge Check 3

Exercise 2

Module completion checklist

Objective	Complete
Introduce list and dataframe in R	✔
Perform different operations on the above data types and structures	✔
Read/write data	✔
Clear environment	✔

Summary

- We have successfully started our journey with R and gotten familiar with the concept of variables which we will use throughout this course
- You have **learned more about data structures like lists, dataframes and how to perform various operations on them**
- **In our next module**, we are going to learn more about performing basic data manipulation operations on datasets. Stay excited!

This completes our module

Congratulations!