DATA SOCIETY®

Week 1 day 2 - Fundamentals of R programming

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

Module completion checklist

Objective	Complete
Distinguish data types/structures (`integer`, `character`, `float`, `lists`, `data frames`, etc.)	
Perform different operations on the above data types and structures	
Read/write data	
Clear environment	
Evaluate and address missing values in data	
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	

Type

Data type is a set of values with common characteristics, from which expressions and functions may be formed

It defines **the meaning of data and the way values of that type can be stored**

For instance, a **web page** is a type and any web page has the following basic characteristics:

- Address
- Layout (or absence of thereof)
- Data (or absence of thereof)
- Integration with other web pages into a web site
- Community which allows people to update its content
- Web server where web pages are stored

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 common sources of errors for a person starting to know any programming language, is the data type conversion.

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

Basic data classes: integer

Item	Integer
Value	24, 34L
`typeof()`	integer
`class()`	integer
boolean function	`is.integer()`
`attributes()`	NULL
`length()`	1

*Note: we can use the **L** suffix to qualify any number with the intent of making it an explicit integer*

```
# Create an integer type variable.
integer_var = 234L

# Check type of variable.
typeof(integer_var)
```

```
[1] "integer"
```

```
# Check if the variable is integer.
is.integer(integer_var)
```

```
[1] TRUE
```

```
# Check length of variable
# (i.e. how many entries).
length(integer_var)
```

```
[1] 1
```

Basic data classes: numeric

Item	Numeric
Value	24.34
`typeof()`	double
`class()`	numeric
boolean function	`is.numeric()`
`attributes()`	`NULL`
`length()`	1

```
# Create a numeric class variable.
numeric_var = 24.24
typeof(numeric_var)
```

```
[1] "double"
```

```
# Check length of variable
# (i.e. how many entries).
length(numeric_var)
```

```
[1] 1
```

Basic data classes: character

Item	Character
Value	`"Hello"`
`typeof()`	character
`class()`	character
boolean function	`is.character()`
`attributes()`	NULL
`length()`	1

```
# Create a character class variable.
character_var = "Hello"
```

```
# Check if the variable is character.
is.character(character_var)
```

```
[1] TRUE
```

```
# Check metadata/attributes of variable.
attributes(character_var)
```

```
NULL
```

```
# Check length of variable
# (i.e. how many entries).
length(character_var)
```

```
[1] 1
```

Some useful character operations

```
# Create another character class variable.
case_study = "JUmbLEd CaSE"

# Convert a character string to lower case.
tolower(case_study)
```

```
[1] "jumbled case"
```

```
# Convert a character string to upper case.
toupper(case_study)
```

```
[1] "JUMBLED CASE"
```

```
# Count number of characters in a string.
nchar(case_study)
```

```
[1] 12
```

```
# Compare to the output of the `length` command.
length(case_study)
```

```
[1] 1
```

```
# Get just a part of character string.
substr(case_study, #<- original string
       1,          #<- start index of substring
       7)          #<- end index of substring
```

```
[1] "JUmbLEd"
```

Basic data classes: logical

Item	Logical
Value	`TRUE` or `FALSE`
`typeof()`	logical
`class()`	logical
boolean function	`is.logical()`
`attributes()`	`NULL`
`length()`	1

```
# Create a logical class variable.
logical_var = TRUE

# Check type of variable.
typeof(logical_var)
```

```
[1] "logical"
```

Basic data classes: summary & conversion

Item	Integer	Numeric	Character	Logical
Value	`24, 34L`	`24.34`	Hello	`TRUE` or `FALSE`
`typeof()`	integer	double	character	logical
`class()`	integer	numeric	character	logical
boolean function	`is.integer()`	`is.numeric()`	`is.character()`	`is.logical()`
`attributes()`	`NULL`	`NULL`	`NULL`	`NULL`
`length()`	1	1	1	1
To convert a variable to this type	`as.integer()`	`as.numeric()`	`as.character()`	`as.logical()`

Basic data structures

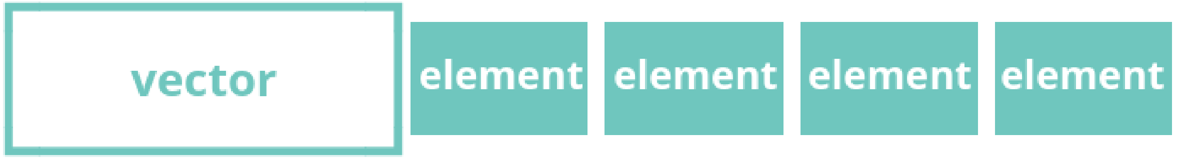
In the past few slides, we have learned some of the most basic as well as common data types

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)	✔	✘
Data frame	2 (rows and columns)	✔	✔

Basic data structures: atomic vectors



`Vector` is a collection of elements that holds the **same** type

- `Mode` of vector means which types of elements it contains
- Most common `modes` of vectors are: `character`, `logical`, `numeric`

Vectors are the most universal, common, and simplest data structure present in nearly all programming languages including low-level programming languages

It is called an array in all other programming languages except R

An array with one dimension is almost the same as a vector so we will not differentiate them here for convenience's sake

Your computer's memory is one giant single-dimensional array!

Basic data structures: atomic vectors

```
# To make an empty vector in R,
# you have a few options:
# Option 1: use `vector()` command.
# The default in R is an empty vector of
# `logical` mode!
vector()
```

```
logical(0)
```

```
# Option 2: use `c()` command
# (`c` stands for concatenate).
# The default empty vector produced by `c()`
# has a single entry `NULL`!
c()
```

```
NULL
```

The length of an empty vector will always be 0 since it has no entries in it!

To make a vector out of a given set of character strings, you can wrap them into `c` and separate by commas

```
# Make a vector from a set of char. strings
c("My", "name", "is", "Vector")
```

```
[1] "My"     "name"   "is"     "Vector"
```

To make a vector out of a given set of numbers you can wrap them into a vector `c` and separate elements by commas

```
# Make a vector out of given set of numbers
c(1, 2, 3, 765, -986, 0.5)
```

```
[1]    1.0    2.0    3.0  765.0 -986.0    0.5
```

Basic data structures: atomic vectors

```
# Create a vector of mode `character` from
# pre-defined set of character strings.
character_vec = c("My", "name", "is", "Vector")
character_vec
```

```
[1] "My"      "name"   "is"      "Vector"
```

```
# Check if the variable is character.
is.character(character_vec)
```

```
[1] TRUE
```

```
# Check metadata/attributes of variable.
attributes(character_vec)
```

```
NULL
```

Item	Vector
Value	`character_vec`
`typeof()`	character
`class()`	character
boolean function	`is.character()`
`attributes()`	NULL
`length()`	4

```
# Check length of variable
# (i.e. how many entries).
length(character_vec)
```

```
[1] 4
```

Basic data structures: access vectors values

```
# To access an element inside of the
# vector use `[]` and the index of the element.
character_vec[1]
```

```
[1] "My"
```

```
# To access multiple elements inside of
# a vector use the start and end indices
# with `:` in-between.
character_vec[1:3]
```

```
[1] "My"   "name" "is"
```

Notice, all data structures, including vectors, start at index 1!

```
# A special form of a vector in R is a sequence.
number_seq = seq(from = 1, to = 5, by = 1)
number_seq
```

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

```
# Check class.
class(number_seq)
```

```
[1] "numeric"
```

```
# Subset the first 3 elements.
number_seq[1:3]
```

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

Basic data structures: operations on vectors

```
number_seq      #<- Let's take our vector.
```

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

```
number_seq + 5  #<- Add a number to every entry.
```

```
[1]  6  7  8  9 10
```

```
number_seq - 5  #<- Subtract a number from every entry.
```

```
[1] -4 -3 -2 -1  0
```

```
number_seq * 2  #<- Multiply every entry by a number.
```

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

Note: All arithmetic operations in R are element-wise which means data are operated element by element

```
# To sum all elements use `sum`.
sum(number_seq)
```

```
[1] 15
```

```
# To multiply all elements use `prod`.
prod(number_seq)
```

```
[1] 120
```

```
# To get the mean of all vector
# values use `mean`.
mean(number_seq)
```

```
[1] 3
```

```
# To get the smallest value
# in a vector use `min`.
min(number_seq)
```

```
[1] 1
```

Basic data structures: appending & naming

```
# To name each entry in a vector use `names`.
names(number_seq) = c("First", "Second",
                      "Third", "Fourth",
                      "Fifth")

# Check the attributes of vector.
attributes(number_seq)
```

```
$names
[1] "First"  "Second" "Third"  "Fourth" "Fifth"
```

```
# Check the length of vector.
length(number_seq)
```

```
[1] 5
```

Item	Vector
Value	`number_seq`
`typeof()`	double
`class()`	numeric
boolean function	`is.numeric()`
`attributes()`	`names`
`length()`	5

```
# To append elements to a vector, just
# wrap the vector and additional element(s)
# into `c` again!
character_vec = c(character_vec, "!")
character_vec
```

```
[1] "My"      "name"    "is"      "Vector" "!"
```

Basic data structures: why ATOMIC vectors?

What happens if you mix different types of data inside of an atomic vector?

```
# Create a vector with entries of different type.
atomic_vec = c(333, "some text", TRUE, NULL)
atomic_vec
```

```
[1] "333"       "some text" "TRUE"
```

```
# Check class of the resulting vector.
class(atomic_vec)
```

```
[1] "character"
```

R will **cast** (i.e. coerce) all elements of that vector to a type/class that **can most easily accommodate all elements it contains**!
This is why this type of data structure is called `atomic`, which in computer science world is equivalent to homogeneous or unsplittable (although we all know we can split the `atom` ☢)

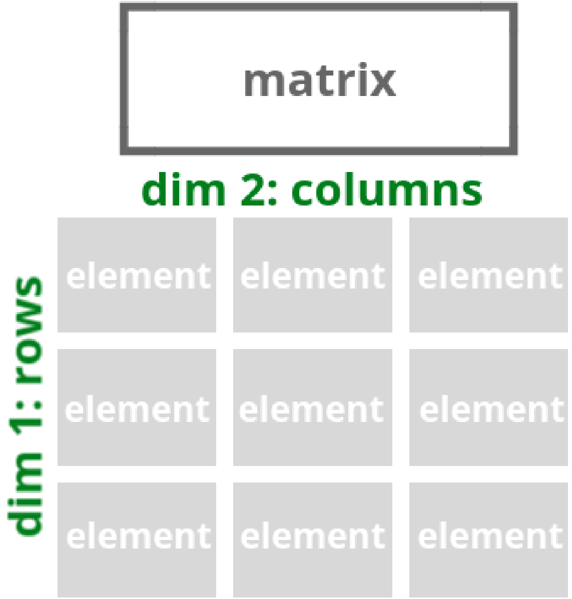
Basic data structures: matrices

A `matrix` is a 2D `vector`, … *say what*?

Yes, a `matrix` is also an array of elements with 2 dimensions instead of 1

Since a `matrix` is a 2-dimensional vector, it only allows elements of the same **type**

`Matrix` data structure shares not only that with a `vector`, working with matrices is very similar to working with 1D vectors



Basic data structures: making matrices

```
# Create a matrix with 3 rows and 3 columns.
sample_matrix1 = matrix(nrow = 3, #<- n rows
                        ncol = 3) #<- m cols
sample_matrix1
```

```
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
[3,]   NA   NA   NA
```

```
# Notice that by default an empty matrix
# will be filled with `NA`s.

# Check matrix dimensions.
dim(sample_matrix1)
```

```
[1] 3 3
```

```
# Notice that the `length` command will produce
# the total number elements in the matrix
# (length = n rows x m cols).
length(sample_matrix1)
```

```
[1] 9
```

```
# Another way to create a matrix is to make
# it out of a vector of numbers.
sample_matrix2 = 1:9 #<- another way to make
                     #    a sequence of numbers!

# Assign dimensions to matrix:
# 1st number is for rows, 2nd is for columns.
dim(sample_matrix2) = c(3, #<- n rows
                        3) #<- m cols

sample_matrix2
```

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

```
# Check matrix dimensions.
dim(sample_matrix1)
```

```
[1] 3 3
```

Basic data structures: making matrices

The shorthand version of the previous 2 commands looks like this

```
# Create a matrix from a sequence of numbers
# with 3 rows & 3 columns.
sample_matrix3 = matrix(1:9,        #<- entries
                        nrow = 3,  #<- n rows
                        ncol = 3)  #<- m cols
sample_matrix3
```

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

Notice that `matrix` command arranges the values by `column` by default!

Create the same matrix but with values arranged by **rows**

```
# Create a matrix from a sequence of numbers
# with 3 rows & 3 columns arranged by row.
sample_matrix4 = matrix(1:9,
                        nrow = 3,
                        ncol = 3,
                        byrow = TRUE)
sample_matrix4
```

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

Basic data structures: working with matrices

```
# Check type of variable.
typeof(sample_matrix4)
```

```
[1] "integer"
```

```
# Check class of variable.
class(sample_matrix4)
```

```
[1] "matrix"
```

```
# Check if the variable of type `integer`.
is.integer(sample_matrix4)
```

```
[1] TRUE
```

```
# Check metadata/attributes of variable.
attributes(sample_matrix4)
```

```
$dim
[1] 3 3
```

Basic data structures: working with matrices

```
# To append rows to a matrix, use `rbind`.
new_matrix1 = rbind(sample_matrix4,
                    10:12)
new_matrix1
```

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

```
# To append columns to a matrix, use `cbind`.
new_matrix2 = cbind(sample_matrix3,
                    10:12)
new_matrix2
```

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

```
# To access an element of a matrix use
# the row and column indices separated
# by a comma inside of `[]`.
new_matrix1[1, 2] #<- element in row 1, col 2
```

```
[1] 2
```

```
# To access a row leave the space in
# column index empty.
new_matrix1[1 , ]
```

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

```
# To access a column leave the space in
# row index empty.
new_matrix1[ , 2]
```

```
[1]  2  5  8 11
```

Basic data structures: operations on matrices

```
# Let's take a sample matrix.
sample_matrix2
```

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

```
# Add a number to every entry.
sample_matrix2 + 5
```

```
     [,1] [,2] [,3]
[1,]    6    9   12
[2,]    7   10   13
[3,]    8   11   14
```

```
# Multiply every entry by a number.
sample_matrix2 * 2
```

```
     [,1] [,2] [,3]
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
```

```
# To sum all elements use `sum`.
sum(sample_matrix2)
```

```
[1] 45
```

```
# To multiply all elements use `prod`.
prod(sample_matrix2)
```

```
[1] 362880
```

```
# To get the mean of all matrix
# values use `mean`.
mean(sample_matrix2)
```

```
[1] 5
```

```
# To get the smallest value
# in a matrix use `min`.
min(sample_matrix2)
```

```
[1] 1
```

Basic data structures: names & attributes

```r
# To name columns of a matrix use `colnames`.
colnames(sample_matrix2) = c("Col1", "Col2",
"Col3")

# To name rows of a matrix use `rownames`.
rownames(sample_matrix2) = c("Row1", "Row2",
"Row3")
sample_matrix2
```

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

```r
# Check the attributes of a matrix.
attributes(sample_matrix2)
```

```
$dim
[1] 3 3

$dimnames
$dimnames[[1]]
[1] "Row1" "Row2" "Row3"

$dimnames[[2]]
[1] "Col1" "Col2" "Col3"
```

Item	Matrix
To create	`matrix()`
Value	`sample_matrix2`
`typeof()`	integer
`class()`	matrix
boolean function	`is.maxtrix()`
`attributes()`	dim, `dimnames[[1]]`, `dimnames[[2]]`
`length()`	9

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 list and dataframe:

- 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
```

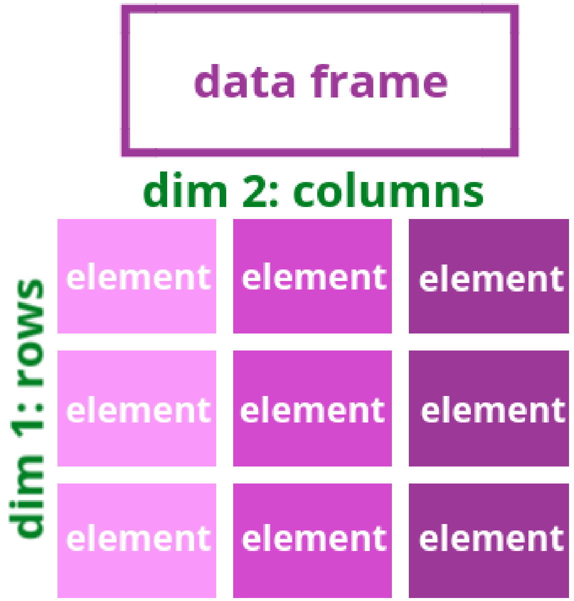
```
sample_list$Two
```

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

```
sample_list$Three
```

```
[1] TRUE
```

Basic data structures: data frames



Note: if you have ever worked with relational databases, you can think of a data frame 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 data frames

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

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

```r
# To make a data frame 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 data frames 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!

Data frames: naming columns

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

```
# Data frane 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 data frame.
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
```

Data frames: naming rows

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

```
# View data frame.
named_df
```

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

```
# Rename data frame 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 data frame *and* **define** row names with method `row.names` at the time of its creation

```
# Define row names explicitely,
# 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
```

Data frames: converting a matrix

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

```
# Make a data frame from matrix.
sample_df1 = as.data.frame(sample_matrix1)
sample_df1
```

```
  V1 V2 V3
1 NA NA NA
2 NA NA NA
3 NA NA NA
```

```
# Make a data frame from matrix with named columns and rows.
sample_df2 = as.data.frame(sample_matrix2)
sample_df2
```

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

Data frames: row and column names of a matrix

```
# Check attributes of a data frame.
attributes(sample_df1)
```

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

$class
[1] "data.frame"

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

Unnamed `matrix` **column names** will default to `V1, V2, ..., Vm`, where `m = num columns` of a matrix

Unnamed `matrix` **row names** will default to `1, 2, ..., n`, where `n = num rows` of a matrix

```
# Check the attributes of data frame.
attributes(sample_df2)
```

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

$class
[1] "data.frame"

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

Named `matrix` **column names** will become `data.frame` column names

Named `matrix` **row names** will become `data.frame` row names

Data frames: 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 data frame
# Option 1: Use `$column_name`.
sample_df2$Col1
```

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

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

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

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

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

Data frames: 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 data frame
# Option 1: use `[row_index, ]`.
sample_df2[1, ]
```

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

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

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

Data frames: 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
```

Data frames: adding new columns

Another common case is adding new columns into an existing data frame

- Use `$new_column_name`
- Use `cbind`

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

Data frames: operations

```
# Let's take our sample data frame.
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: Factor w/ 3 levels "column","new",..: 3 2 1
```

```
# 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
```

Special classes: factors

```
# Let's take a look at the structure of the data frame.
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: Factor w/ 3 levels "column","new",..: 3 2 1
```

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 data frame.
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 data frame
# 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 1

Exercise 1

Module completion checklist

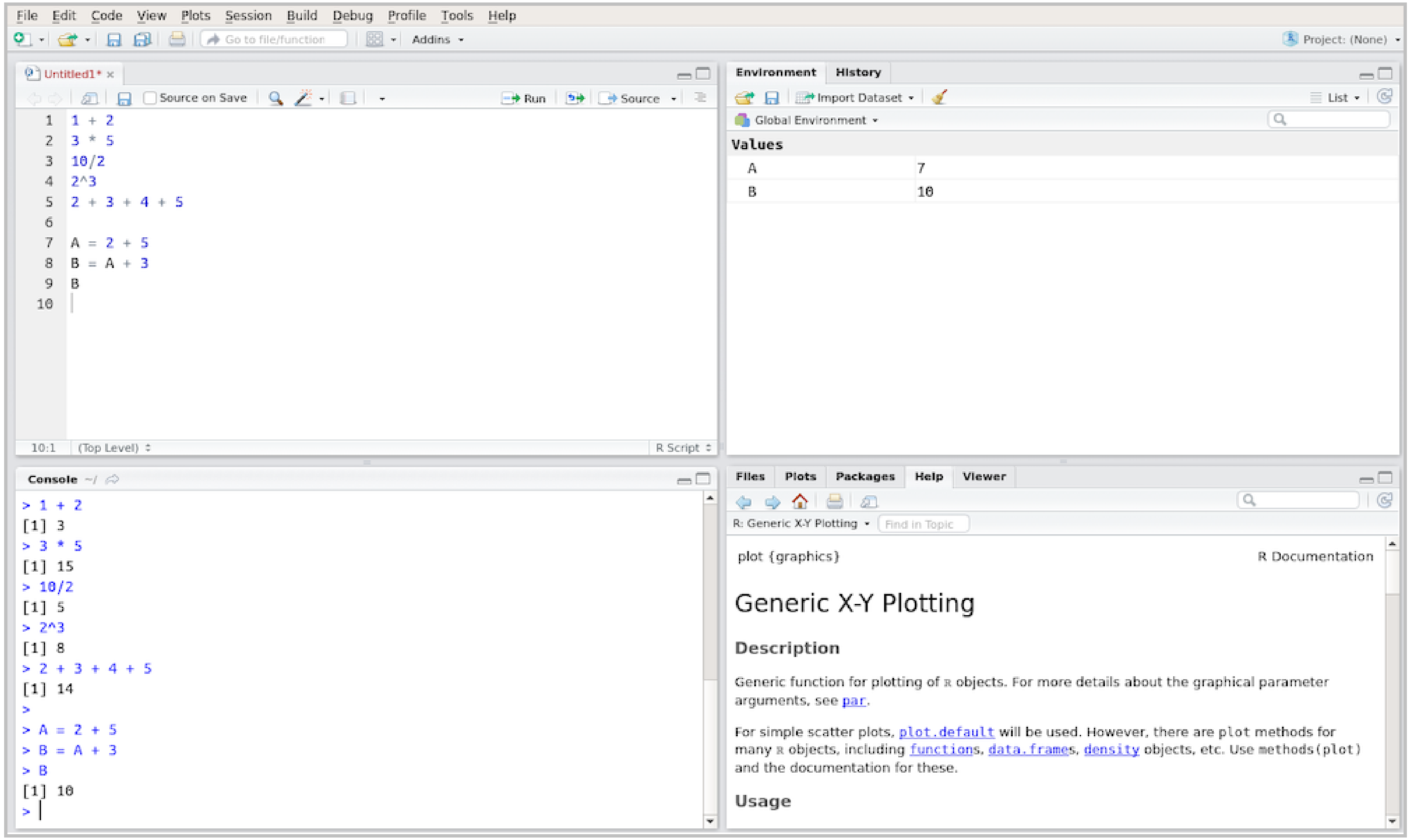
Objective	Complete
Distinguish data types/structures (`integer`, `character`, `float`, `lists`, `data frames`, etc.)	✔
Perform different operations on the above data types and structures	✔
Read/write data	
Clear environment	
Evaluate and address missing values in data	
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	

RStudio overview: recap

A default RStudio layout includes 4 panes:

1. **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
2. **Bottom left** pane has a `Console`, which shows the output of running R commands
3. **Top right** is a helper pane that shows your `Environment` or `History`
4. **Bottom right** is another helper pane that shows `Files`, static `Plots` and interactive plots through `Viewer`, `Help`, and `Packages`

RStudio overview: recap



R's working directory

A folder on your machine (which R treats as your “sandbox”) where R 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`

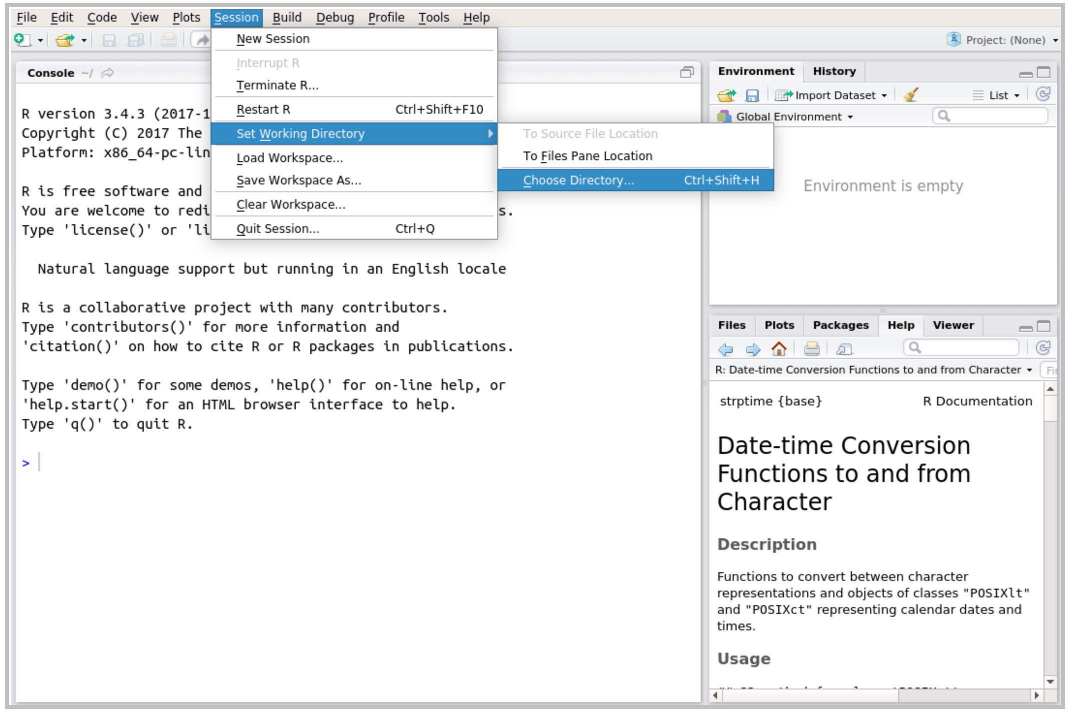
We can `set` the working directory

We can `get` the working directory

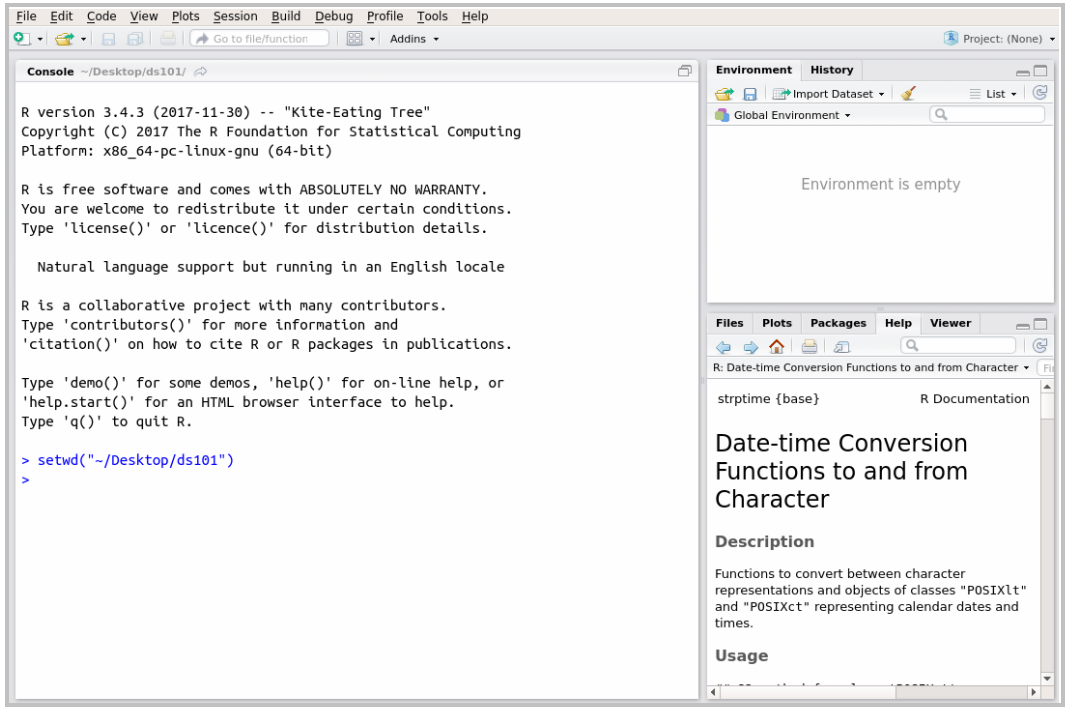
We can `encode` directory paths into `variables` and change them without having to manually type the paths every time

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/hhs-r-2020")

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

```
[1] "/home/[your-user-name]/Desktop/hhs-r-2020"
```

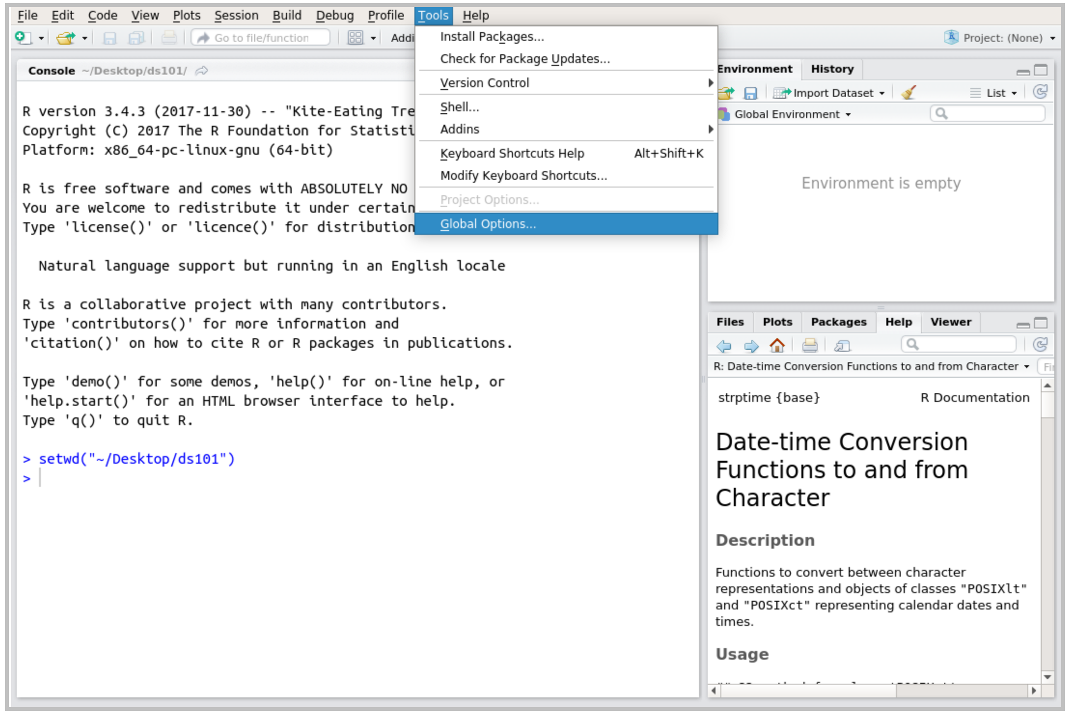
You can set your working directory via command line (on Windows)

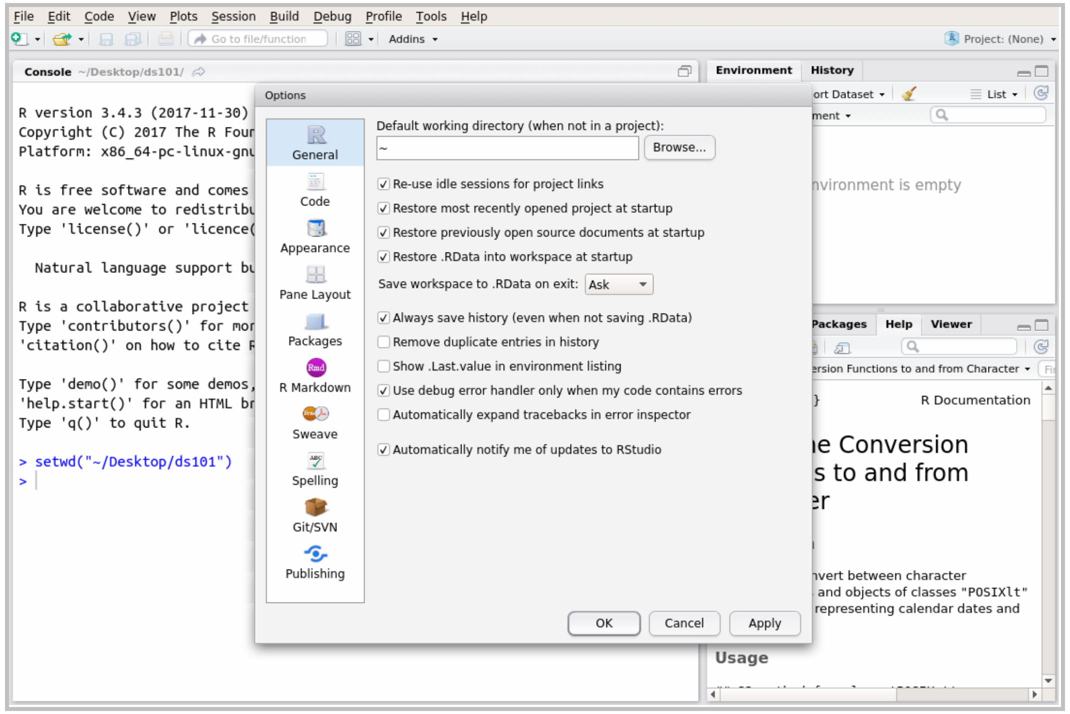
```
# To set working directory call `setwd` with the path to the folder.
setwd("C:/Users/[your-user-name]/Desktop/hhs-r-2020")

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

```
[1] "C:/Users/[your-user-name]/Desktop/hhs-r-2020"
```

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 `hhs-r-2020` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac/Linux).
main_dir = "~/Desktop/hhs-r-2020"

# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/hhs-r-2020"

# 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 `hhs-r-2020` 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 `hhs-r-2020` 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 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/hhs-r-2020/data"
```

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

```
[1] "C:/Users/[your-user-name]/Desktop/hhs-r-2020/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 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

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
```

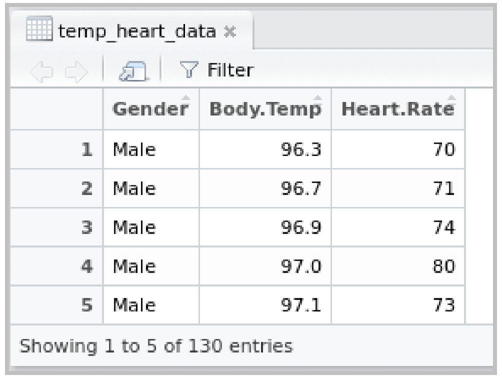
```
tail(temp_heart_data, 4) #<- Inspect the `tail` (last 4 rows).
```

```
    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)
```



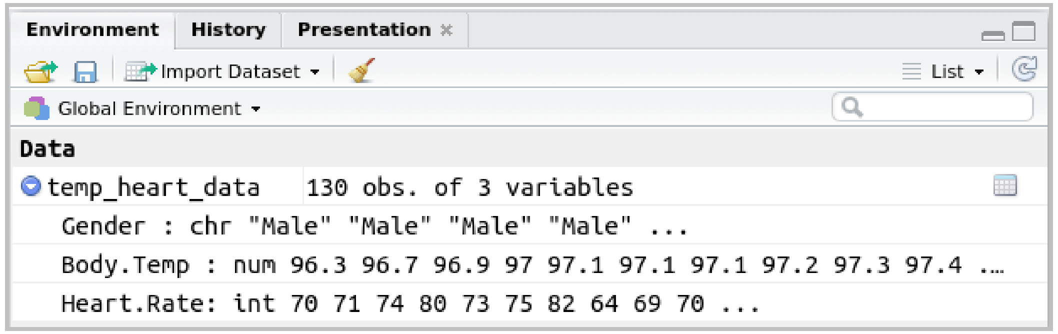
temp_heart_data

Filter

	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

Showing 1 to 5 of 130 entries

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
Distinguish data types/structures (`integer`, `character`, `float`, `lists`, `data frames`, etc.)	✔
Perform different operations on the above data types and structures	✔
Read/write data	✔
Clear environment	
Evaluate and address missing values in data	
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	

Clearing objects from environment

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

```
 [1] "atomic_vec"         "case_study"         "character_var"
 [4] "character_vec"      "data_dir"           "directory"
 [7] "head"               "highlight_js"       "integer_var"
[10] "logical_var"        "main_dir"           "named_df"
[13] "new_dates"          "new_matrix1"        "new_matrix2"
[16] "number_seq"         "numeric_var"        "platform"
[19] "plot_dir"           "sample_df1"         "sample_df2"
[22] "sample_list"        "sample_list_named"  "sample_matrix1"
[25] "sample_matrix2"     "sample_matrix3"     "sample_matrix4"
[28] "session_info"       "special_data"       "temp_heart_data"
[31] "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
```

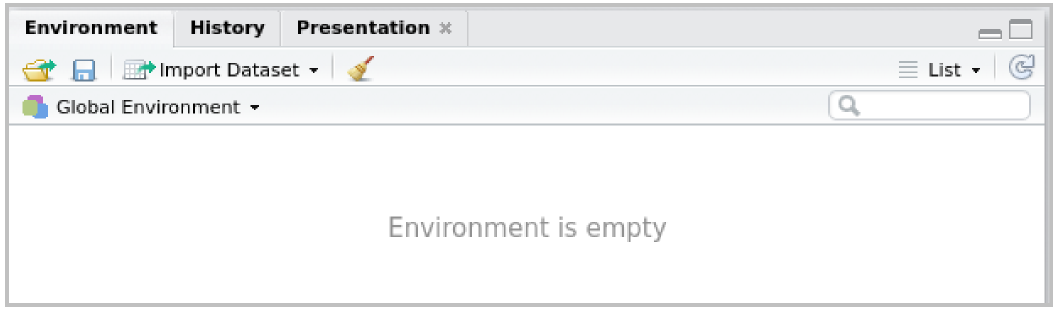
```
# List all objects again to check.
ls()
```

```
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 2

Exercise 2

Module completion checklist

Objective	Complete
Distinguish data types/structures (`integer`, `character`, `float`, `lists`, `data frames`, etc.)	✔
Perform different operations on the above data types and structures	✔
Read/write data	✔
Clear environment	✔
Evaluate and address missing values in data	
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	

Introducing CMP data set

We are going to explore a new data set called `ChemicalManufacturingProcess` from `AppliedPredictiveModeling` package in R

This dataset contains information about a chemical manufacturing **process**

The goal is to understand the relationship between the process and the resulting **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

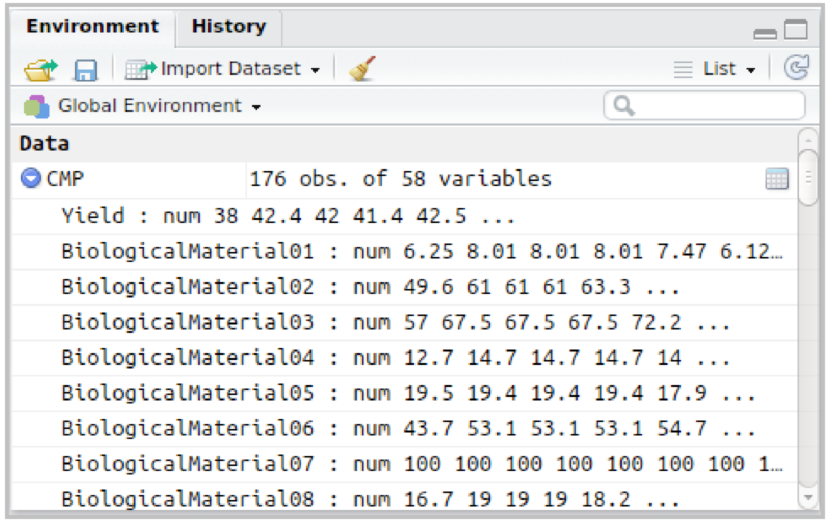
The **process** variables include measurements such as temperature, drying time, washing time, and concentrations of byproducts at various steps

Loading data set

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

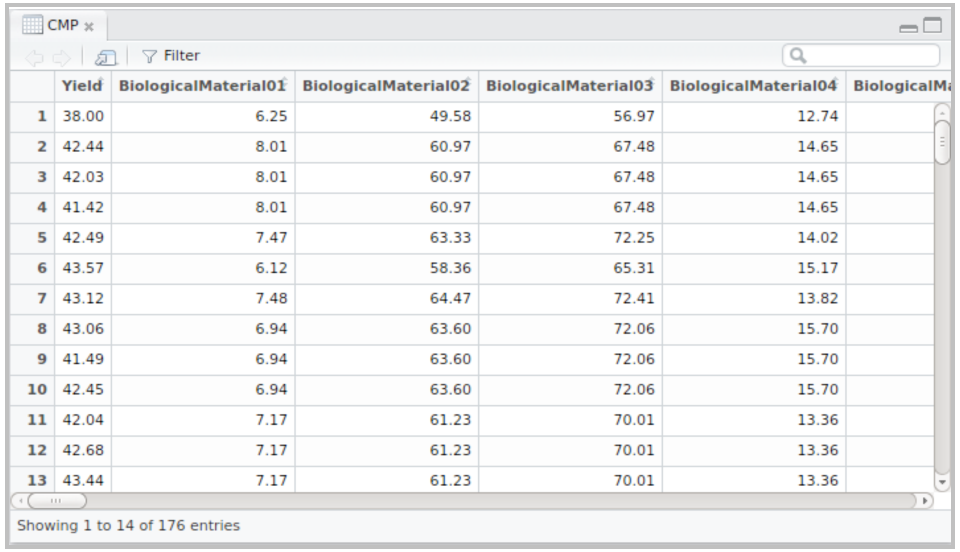
```
# Set working directory to where we store data.
setwd(data_dir)

# 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 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	

Showing 1 to 14 of 176 entries

Subsetting data

In this module we will explore a subset of this data set, 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            #<- verify that we have the correct set of columns
```

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

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

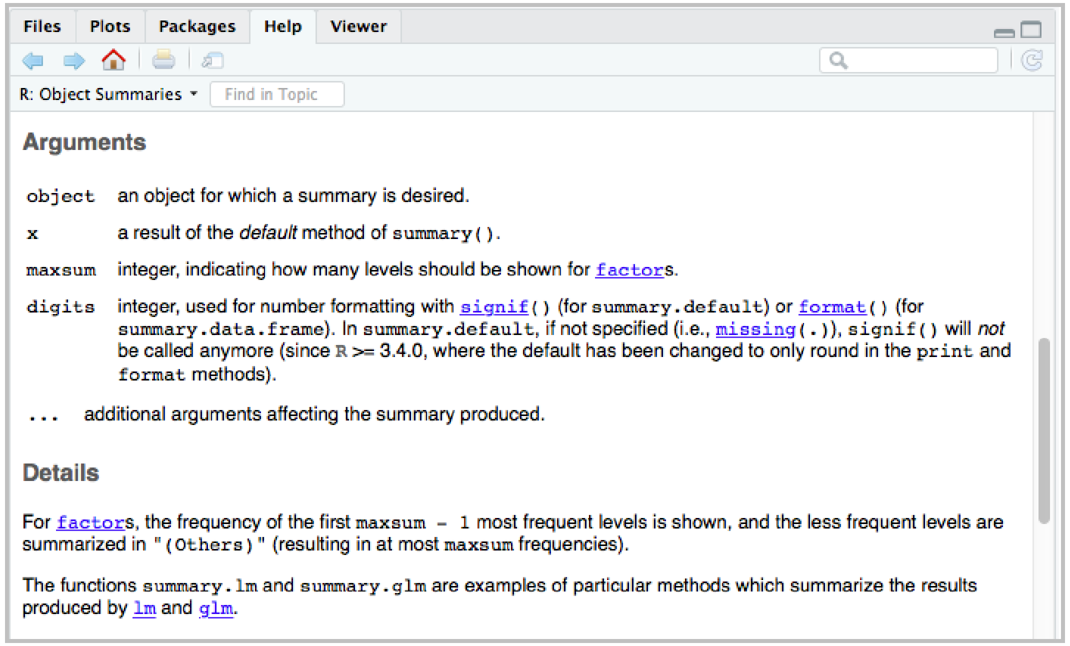
```
'data.frame':   176 obs. of  7 variables:
 $ Yield                 : num  38 42.4 42 41.4 42.5 ...
 $ BiologicalMaterial01  : num  6.25 8.01 8.01 8.01 7.47 6.12 7.48 6.94 6.94 6.94 ...
 $ BiologicalMaterial02  : num  49.6 61 61 61 63.3 ...
 $ BiologicalMaterial03  : num  57 67.5 67.5 67.5 72.2 ...
 $ 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 data frame, or one single column within the data frame, use `summary`

```
?summary

summary(data) #<- Either the data frame or
single column
```



Arguments

`object`	an object for which a summary is desired.
`x`	a result of the *default* method of `summary()`.
`maxsum`	integer, indicating how many levels should be shown for `factors`.
`digits`	integer, used for number formatting with `signif()` (for `summary.default`) or `format()` (for `summary.data.frame`). In `summary.default`, if not specified (i.e., `missing(.)`), `signif()` will *not* be called anymore (since R >= 3.4.0, where the default has been changed to only round in the `print` and `format` methods).
`...`	additional arguments affecting the summary produced.

Details

For `factors`, the frequency of the first `maxsum - 1` most frequent levels is shown, and the less frequent levels are summarized in `"(Others)"` (resulting in at most `maxsum` frequencies).

The functions `summary.lm` and `summary.glm` are examples of particular methods which summarize the results produced by `lm` and `glm`.

Summary statistics: 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 1st 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
```

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 the method `na.rm`? Or what if removing `NA`s skews 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)

Replacing `NA`s with **`mean`** may not work well if the data contains outliers

Working with missing data

Function `is.na` will provide a vector of `TRUE` or `FALSE` for each element of a given vector

It is hard to track elements that are indeed `NA` for datasets containing even a moderate number of data points

```
# 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
```

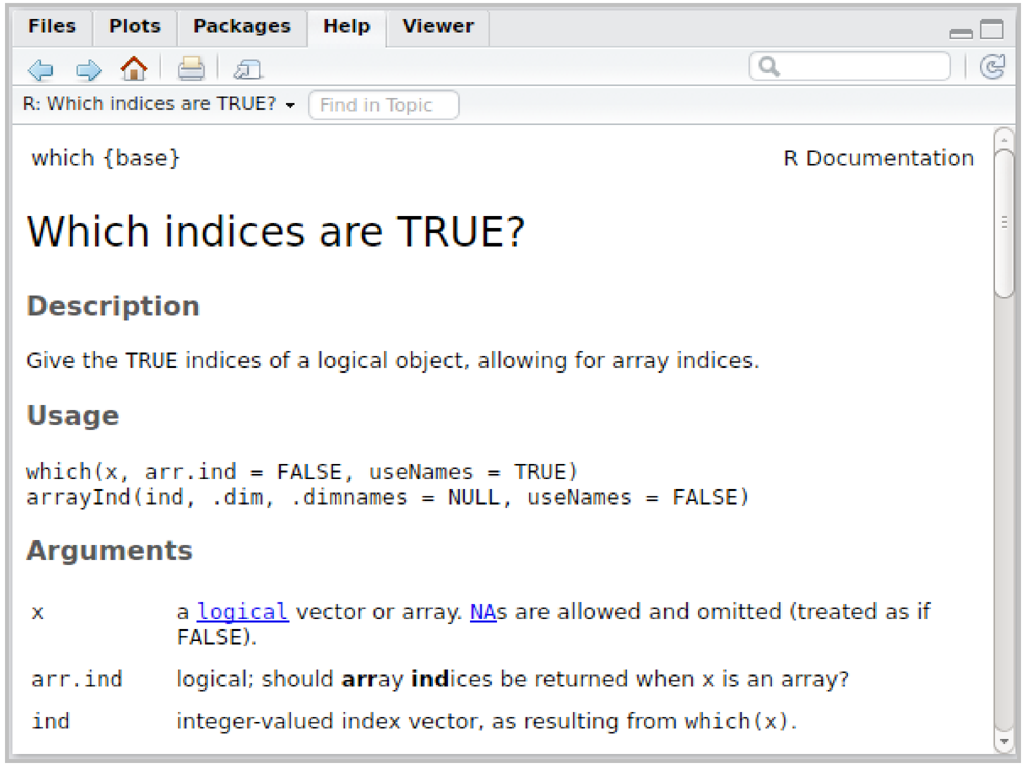
Working with missing data

```
?which
```

`which` function is an invaluable utility function in R's `base` package

It takes either a vector/array of `logical` values OR a vector/array of any values with a comparison statement in one of the comparison operators (==, != , >, <, >=, <=) and a value to which we are comparing to

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



Working with missing data: identifying NA values

```
# Let's save this vector of logical values to a variable.
is_na_MP01 = is.na(CMP_subset$ManufacturingProcess01)

# To determine WHICH elements in the vector are `TRUE`and are NA, 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_MP01)
```

```
[1] 1
```

```
# This is also a correct way to set it up.
which(is_na_MP01 == TRUE)
```

```
[1] 1
```

Working with missing data: 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_MP01)
na_id
```

```
[1] 1
```

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

```
[1] NA
```

Working with missing data: mean replacement

We need to compute a value suitable for replacing 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
```

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

We can now take the mean and assign it to the missing value within 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
```

Now instead of the `NA` we have the `mean` value of this column!

Let's compute the `max` of the column without `na.rm` specified to see if it works:

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

```
[1] 14.1
```

Working with missing data

Next 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 3

Exercise 3

Module completion checklist

Objective	Complete
Distinguish data types/structures (`integer`, `character`, `float`, `lists`, `data frames`, etc.)	✔
Perform different operations on the above data types and structures	✔
Read/write data	✔
Clear environment	✔
Evaluate and address missing values in data	✔
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	

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

- Writing conditional statements using `if`, `if...else`, and `ifelse`
- Writing loops using `for`
- Writing function definitions using `function`

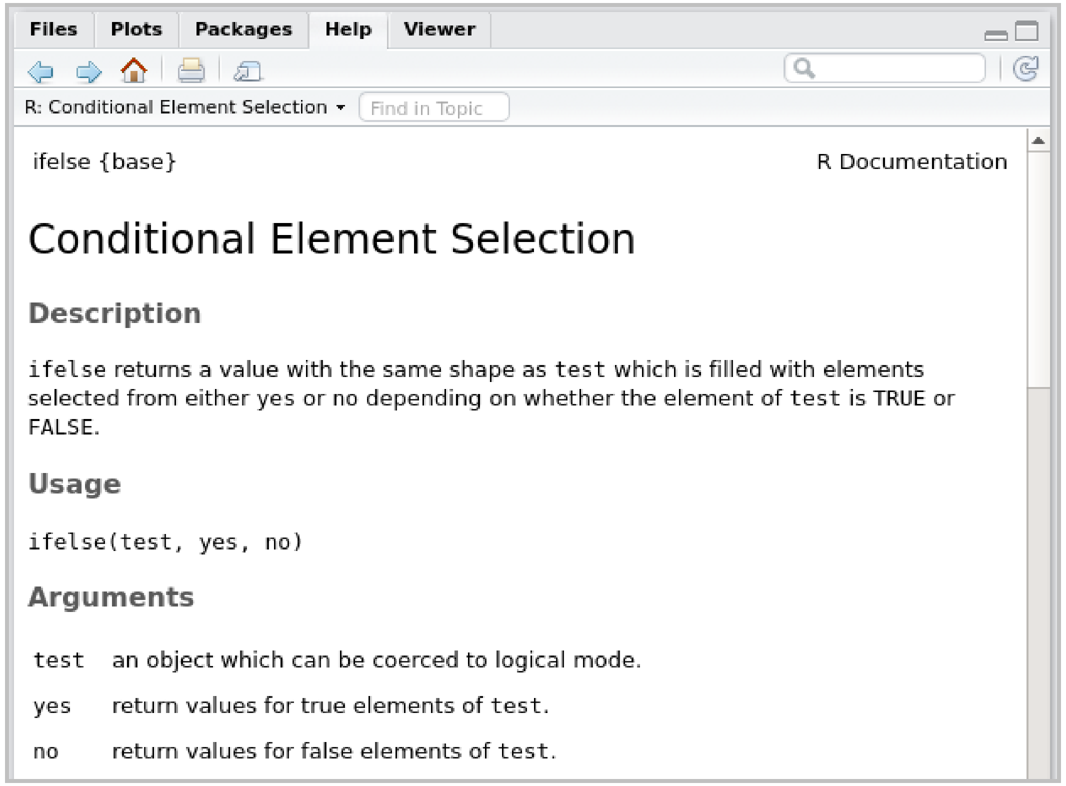
Conditionals: `ifelse` function

```
?ifelse
```

The simplest conditional is `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

`ifelse` function must return a value



Ifelse example

Let's say we want to take `Yield` from the `CMP` dataset and convert it to either `above average` or `below average`

We can use `ifelse` here. Let's demonstrate

```
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

It has an explicit `start` and `end`

Arguments for a `for` loop can take several forms, the most common 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

Loops: `for` loop

Here is a basic example of a `for` loop

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

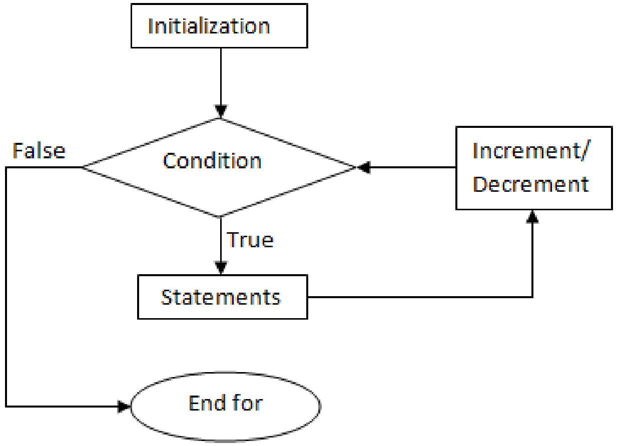


fig: Flowchart for for loop

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

Loops: `for` loop

We can identify the start and end points of the loop in several 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 wanted to print the variable names that start at index `3` and end at index `6`, so we only needed 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 just a subset
# of the variable names.
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 it can be re-used later

They make your code:

- More **abstract** so it can be used with different data and/or parameters
- More **modular** so 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
}
```

Functions in R: 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 most basic function is one that takes no arguments.

The function definition consists of 2 main components:

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 `{}` can either:
   i. Perform just an **action**
   ii. Return a specific value

Functions in R: 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!"
```

Functions in R: function with arguments

```
# 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
```

Functions in R: call function without arguments

What happens if you try to invoke a function that requires arguments without passing an 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
```

Functions in R: 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

The full step by step process of creating this function will be detailed in a supplemental deck

Functions in R

```r
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!

```r
# 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"
```

```r
# 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
...
```

Knowledge check 4

Exercise 4

Module completion checklist

Objective	Complete
Distinguish data types/structures (`integer`, `character`, `float`, `lists`, `data frames`, etc.)	✔
Perform different operations on the above data types and structures	✔
Read/write data	✔
Clear environment	✔
Evaluate and address missing values in data	✔
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	✔

Summary

Topics		
Week 1-2	Intro to R programming	♪
Week 3-5	Machine Learning - Regression and Unsupervised Learning	
Week 6-8	Machine Learning - Classification	

In today's module we learned and operated on several **data structures** in R such as matrics and data frames

We also create our first **loop** function together in R

After class, you can try to manipulate your data in different data structures

So far, we are all dealing with base R

In the next module, we will look at **`packages`** in R which contains various functions and will make our programming process more efficient. Stay excited!

This completes our module

Congratulations!