

# Intro to R

Data Classes

One dimensional vectors

## Data classes/types

- \* Character: strings or individual characters, quoted
- \* Numeric: any real number(s)
- \* Integer: any integer(s)/whole numbers (1,2,3)
- \* Double: any number with fractional values (1.2, 4.0)
- \* Factor: categorical/qualitative variables
- \* Logical: variables composed of TRUE or FALSE
- \* Date/POSIXct: represents calendar dates and times

## Character and numeric

We have already covered character and numeric types.

```
class(c("tree", "cloud", "stars_&_sky"))
```

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

```
class(c(1, 4, 7))
```

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

## Character and numeric

This can also be a bit tricky.

```
class(c(1, 2, "tree"))
```

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

```
class(c("1", "4", "7"))
```

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

# Logical

`logical` is a type that only has two possible elements: `TRUE` and `FALSE`

```
x <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
class(x)
```

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

Note that `logical` elements are NOT in quotes.

```
z <- c("TRUE", "FALSE", "TRUE", "FALSE")
class(z)
```

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

## General Class Information

There is one useful functions associated with practically all R classes:

`as.CLASS_NAME(x)` **coerces between classes**. It turns x into a certain class.

Examples:

- `as.numeric()`
- `as.character()`
- `as.logical()`
- `as.double()`
- `as.integer()`
- `as.Date()`
- `as.factor()` (More on this one later!)

## General Class Information: Checking

```
class(4)
```

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

```
class(c(1, 4, 7))
```

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

```
class("tree")
```

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

```
class(c("tree", "cloud"))
```

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



# Coercing: seamless transition

Sometimes coercing works great!

```
as.character(4)
```

```
## [1] "4"
```

```
as.numeric(c("1", "4", "7"))
```

```
## [1] 1 4 7
```

```
as.logical(c("TRUE", "FALSE", "FALSE"))
```

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

```
as.logical(0)
```

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

## Coercing: not-so-seamless

When interpretation is ambiguous, R will return **NA** (an R constant representing “**N**ot **A**vailable” i.e. missing value)

```
as.numeric(c("1", "4", "7a"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1] 1 4 NA
```

```
as.logical(c("TRUE", "FALSE", "UNKNOWN"))
```

```
## [1] TRUE FALSE NA
```

```
as.Date(c("2021-06-15", "2021-06-32"))
```

```
## [1] "2021-06-15" NA
```

# Number Subclasses

There are two major number subclasses or types

1. Double
2. Integer

# Double

Double is equivalent to `numeric`. It is a number that contains **fractional values**.

Double stands for [double-precision](#)

```
y <- c(1.1, 2.0, 3.2, 4.5, 5.6)
y
```

```
## [1] 1.1 2.0 3.2 4.5 5.6
```

```
class(y)
```

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

```
typeof(y)
```

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

# Integer

Integer is a special number that contains only **whole numbers**.

```
y
```

```
## [1] 1.1 2.0 3.2 4.5 5.6
```

```
y_int <- as.integer(y)  
y_int
```

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

```
class(y_int)
```

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

```
typeof(y_int)
```

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

# Integer

Need to use `as.integer()` function to create integers (unless they are read in as integers or created as such with `seq` and `sample`). Otherwise, will be double by default.

```
x <- c(1, 2, 3, 4, 5) # technically integers  
class(x)
```

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

```
typeof(x)
```

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

## Checking double vs integer

A `tibble` will show the difference (as does `glimpse()`).

```
my_data <- tibble(double_var = y, int_var = y_int)
my_data
```

```
## # A tibble: 5 × 2
##   double_var int_var
##   <dbl>      <int>
## 1      1.1        1
## 2       2         2
## 3      3.2        3
## 4      4.5        4
## 5      5.6        5
```

```
glimpse(my_data)
```

```
## Rows: 5
## Columns: 2
## $ double_var <dbl> 1.1, 2.0, 3.2, 4.5, 5.6
## $ int_var    <int> 1, 2, 3, 4, 5
```

# Factors

A **factor** is a special **character** vector where the elements have pre-defined groups or 'levels'. You can think of these as qualitative or categorical variables. Order is often important.

Examples:

- red, orange, yellow, green, blue, purple
- breakfast, lunch, dinner
- baby, toddler, child, teen, adult
- Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree
- beginner, novice, intermediate, expert



# Factors

Use the `factor()` function to create factors.

```
x <- c("small", "medium", "large", "medium", "large")  
class(x)
```

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

```
x_fact <- factor(x)  
class(x_fact)
```

```
## [1] "factor"
```

```
x_fact
```

```
## [1] small medium large medium large  
## Levels: large medium small
```

Note that levels are, by default, in **alphanumerical** order!

# Factors

**Q:** Why not use `as.factor()` ?

**A:** You can coerce with `as.factor()`. But you can't specify levels! More on this soon.

# Factors

You can learn what are the unique levels of a factor vector

```
levels(x_fact)
```

```
## [1] "large" "medium" "small"
```

More on how to change the levels ordering in a lecture coming up!

# Factors

Factors can be converted to `numeric` or `character` very easily.

```
x_fact
```

```
## [1] small  medium large  medium large  
## Levels: large medium small
```

```
as.character(x_fact)
```

```
## [1] "small" "medium" "large"  "medium" "large"
```

```
as.numeric(x_fact)
```

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

# Classes Overview

Example	Class	Type	Notes
1.1	Numeric	double	default for numbers
1	integer	integer	Need to coerce to integer with <code>as.integer()</code> or use <code>sample()</code> or <code>seq()</code> with whole numbers
"FALSE", "Ball"	Character	Character	Need quotes
FALSE, TRUE	logical	logical	No quotes
"Small", "Large"	Factor	Factor	Need to coerce to factor with <code>factor()</code>

---

## Summary

- There are two types of number class objects: integer and double
- Logic class objects only have **TRUE** or **FALSE** (without quotes)
- `class()` can be used to test the class of an object `x`
- `as.CLASS_NAME(x)` can be used to change the class of an object `x`
- Factors are a special character class that has levels - more on that soon!
- tibbles show column classes!

# Lab Part 1

[Class Website](#)

[Lab](#)

**Two-dimensional data classes**



## Two-dimensional data classes

Two-dimensional classes are those we would often use to store data read from a file

- a data frame (`data.frame` or `tibble` class)
- a matrix (`matrix` class)
  - also composed of rows and columns
  - unlike `data.frame` or `tibble`, the entire matrix is composed of one R class
  - for example: all entries are `numeric`, or all entries are `character`

# Matrices

`as.matrix()` creates a matrix from a data frame (where all values are the same class). `matrix()` creates a matrix from scratch.

```
head(iris)
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1           3.5           1.4           0.2   setosa
## 2           4.9           3.0           1.4           0.2   setosa
## 3           4.7           3.2           1.3           0.2   setosa
## 4           4.6           3.1           1.5           0.2   setosa
## 5           5.0           3.6           1.4           0.2   setosa
## 6           5.4           3.9           1.7           0.4   setosa
```

```
class(iris)
```

```
## [1] "data.frame"
```

```
iris_mat <- head(tibble(select(iris, -Species)))
as.matrix(iris_mat)
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## [1, ]           5.1           3.5           1.4           0.2
## [2, ]           4.9           3.0           1.4           0.2
## [3, ]           4.7           3.2           1.3           0.2
## [4, ]           4.6           3.1           1.5           0.2
## [5, ]           5.0           3.6           1.4           0.2
## [6, ]           5.4           3.9           1.7           0.4
```

```
matrix(1:6, ncol = 2)
```

# Lists

- One other data type that is the most generic are `lists`.
- Can hold vectors, strings, matrices, models, list of other list!
- Lists are used when you need to do something repeatedly across lots of data
  - for example wrangling several similar files at once
- Lists are a bit more advanced but you may encounter them when you work with others or look up solutions

# Making Lists

- Can be created using `list()`

```
mylist <- list(c("A", "b", "c"), c(1, 2, 3), matrix(1:4, ncol = 2))  
mylist
```

```
## [[1]]  
## [1] "A" "b" "c"  
##  
## [[2]]  
## [1] 1 2 3  
##  
## [[3]]  
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    4
```

```
class(mylist)
```

```
## [1] "list"
```

# Lists

List elements can be named

```
mylist_named <- list(  
  letters = c("A", "b", "c"),  
  numbers = c(1, 2, 3),  
  one_matrix = matrix(1:4, ncol = 2)  
)  
mylist_named
```

```
## $letters  
## [1] "A" "b" "c"  
##  
## $numbers  
## [1] 1 2 3  
##  
## $one_matrix  
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    4
```

Special data classes

# Dates

There are two most popular R classes used when working with dates and times:

- `Date` class representing a calendar date
- `POSIXct` class representing a calendar date with hours, minutes, seconds

We convert data from character to `Date`/`POSIXct` to use functions to manipulate date/date and time

`lubridate` is a powerful, widely used R package from “tidyverse” family to work with `Date` / `POSIXct` class objects

## Creating **Date** class object

```
class("2021-06-15")
```

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

```
library(lubridate)
```

```
ymd("2021-06-15") # lubridate package
```

```
## [1] "2021-06-15"
```

```
class(ymd("2021-06-15")) # lubridate package
```

```
## [1] "Date"
```

```
class(date("2021-06-15")) # lubridate package
```

```
## [1] "Date"
```

Note for function `ymd`: **y**ear **m**onth **d**ay



## dates

```
a <- ymd("2021-06-15")  
b <- ymd("2021-06-18")  
a - b
```

```
## Time difference of -3 days
```

## Creating **Date** class object

date() is picky...

```
date("06/15/2021") # This doesn't work
```

```
## Error in as.POSIXlt.character(x, tz = tz(x)): character string is not in a
```

```
mdy("06/15/2021") # This works
```

```
## [1] "2021-06-15"
```

```
mdy("06/15/21") # This works
```

```
## [1] "2021-06-15"
```

Note for function mdy: **m**onth **d**ay **y**ear

## Creating **POSIXct** class object

```
class("2013-01-24 19:39:07")
```

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

```
ymd_hms("2013-01-24 19:39:07") # lubridate package
```

```
## [1] "2013-01-24 19:39:07 UTC"
```

```
class(ymd_hms("2013-01-24 19:39:07")) # lubridate package
```

```
## [1] "POSIXct" "POSIXt"
```

UTC represents time zone, by default: Coordinated Universal Time

Note for function `ymd_hms`: **y**ear **m**onth **d**ay **h**our **m**inute **s**econd.

There are functions in case your data have only date, hour and minute (`ymd_hm( )`) or only date and hour (`ymd_h( )`).

# Summary

- two dimensional object classes include: data frames, tibbles, matrices, and lists
- matrix has columns and rows but is all one data class
  - can create a matrix with `matrix()` from scratch or `as.matrix()` from something
- lists can contain multiples of any other class of data including lists!
  - can create lists with `list()`
- calendar dates can be represented with the `Date` class using `ymd()`, `mdy()` functions from `lubridate` package
- `POSIXct` class representing a calendar date with hours, minutes, seconds. Can use `ymd_hms()` or `ymd_hm()` or `ymd_h()` functions from the [lubridate package](#)

## Lab Part 2

[Class Website](#)

[Lab](#)



Image by [Gerd Altmann](#) from [Pixabay](#)

Extra Slides

## Some useful functions from **lubridate** to manipulate **Date** objects

```
x <- ymd(c("2021-06-15", "2021-07-15"))  
x
```

```
## [1] "2021-06-15" "2021-07-15"
```

```
day(x) # see also: month(x) , year(x)
```

```
## [1] 15 15
```

```
x + days(10)
```

```
## [1] "2021-06-25" "2021-07-25"
```

```
x + months(1) + days(10)
```

```
## [1] "2021-07-25" "2021-08-25"
```

```
wday(x, label = TRUE)
```

```
## [1] Tue Thu
```

```
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

## Some useful functions from **lubridate** to manipulate **POSIXct** objects

```
x <- ymd_hms("2013-01-24 19:39:07")  
x
```

```
## [1] "2013-01-24 19:39:07 UTC"
```

```
date(x)
```

```
## [1] "2013-01-24"
```

```
x + hours(3)
```

```
## [1] "2013-01-24 22:39:07 UTC"
```

```
floor_date(x, "1 hour") # see also: ceiling_date()
```

```
## [1] "2013-01-24 19:00:00 UTC"
```



## Differences in dates

```
x1 <- ymd(c("2021-06-15"))  
x2 <- ymd(c("2021-07-15"))
```

```
difftime(x2, x1, units = "weeks")
```

```
## Time difference of 4.285714 weeks
```

```
as.numeric(difftime(x2, x1, units = "weeks"))
```

```
## [1] 4.285714
```

Similar can be done with time (e.g. difference in hours).

# Data Selection

# Matrices

```
n <- 1:9  
n
```

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

```
mat <- matrix(n, nrow = 3)  
mat
```

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

## Vectors: data selection

To get element(s) of a vector (one-dimensional object):

- Type the name of the variable and open the rectangular brackets [ ]
- In the rectangular brackets, type index (/vector of indexes) of element (/elements) you want to pull. **In R, indexes start from 1** (not: 0)

```
x <- c("a", "b", "c", "d", "e", "f", "g", "h")  
x
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h"
```

```
x[2]
```

```
## [1] "b"
```

```
x[c(1, 2, 100)]
```

```
## [1] "a" "b" NA
```

## Matrices: data selection

Note you cannot use `dplyr` functions (like `select`) on matrices. To subset matrix rows and/or columns, use `matrix[row_index, column_index]`.

```
mat
```

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

```
mat[1, 1] # individual entry: row 1, column 1
```

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

```
mat[1, 2] # individual entry: row 1, column 2
```

```
## [1] 4
```

```
mat[1, ] # first row
```

```
## [1] 1 4 7
```

```
mat[, 1] # first column
```

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

```
mat[c(1, 2), c(2, 3)] # subset of original matrix: two rows and two columns
```

## Lists: data selection

You can reference data from list using `$` (if elements are named) or using `[[ ]]`

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

```
## [1] "A" "b" "c"
```

```
mylist_named[["letters"]] # works only for a list with elements' names
```

```
## [1] "A" "b" "c"
```

```
mylist_named$letters # works only for a list with elements' names
```

```
## [1] "A" "b" "c"
```