Data Frames

R Data Objects

Gaston Sanchez

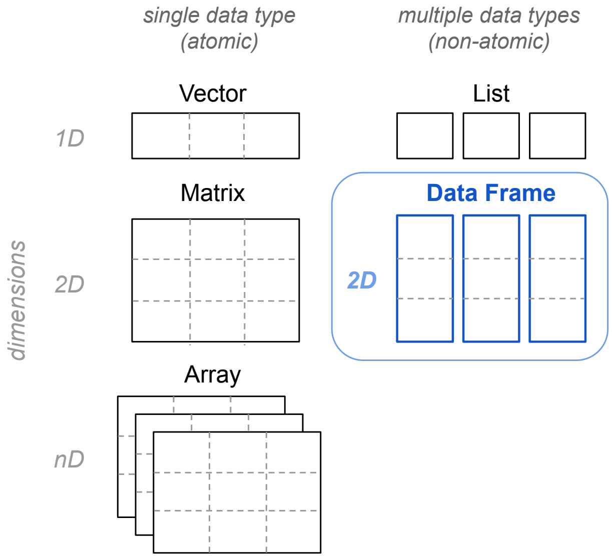
STAT 33B, Fall 2025

About

In this slides we introduce R data objects of class `data.frame`, which provide a nice tabular structure to work with.

Also, we discuss several functions for how to work with data frames from a classic perspective (as opposed to the alternative paradigm from a collection of non-native packages known as `"tidyverse"`).

Basic Data Objects in R



About Data Frames

- R `data.frame`'s are internally stored as lists.
- At the same time, an R `data.frame` behaves like a 2-dimensional object (like a matrix).
- This means that you can manipulate a `data.frame` either like a list but also like a matrix.
- From the list point of view, each element of a `data.frame` is a column.
- A column typically corresponds to an atomic structure (vector or factor), but they can also correspond to a list (this is rare but it's possible).

Inspection Functions

Function	Description
`str()`	Structure
`head()`	First `n` rows
`tail()`	Last `n` rows
`dim()`	Dimensions (# rows, # cols)
`nrow()`	Number of rows
`ncol()`	Number of columns
`names()`	Vector of column names
`colnames()`	Vector of column names
`rownames()`	Vector of row names
`dimnames()`	List of row and column names
`summary()`	Descriptive statistics

Inspection Fuctions

A few rows of `airquality`

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Inspection Fuctions

```r
# display structure
str(airquality)

# display structure (showing few elements)
str(airquality, vec.len = 1)

# first n rows
head(airquality, n = 5)

# last n rows
tail(airquality, n = 5)

# column summaries
summary(airquality)
```

Inspection Fuctions

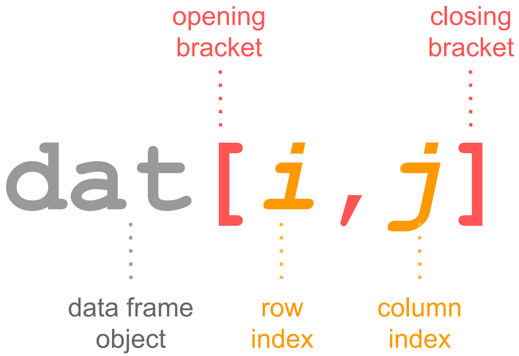
```r
# memory size
object.size(airquality)

# attributes
attributes(airquality)

# data frame dimensions
dim(airquality)

# number of rows
nrow(airquality)

# number of columns
ncol(airquality)
```

Inspection Fuctions

```r
# row names
rownames(airquality)

# column names
colnames(airquality)

# column names
names(airquality)

# object class
class(airquality)

# check if object is data.frame
is.data.frame(airquality)

# data.frame is also a list
is.list(airquality)
```

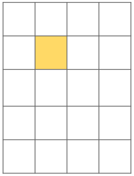
Subsetting and Indexing

Bracket Notation



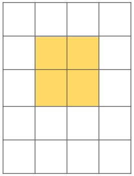
Cells

`dat[2,2]`



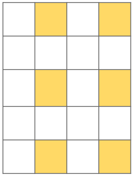
one single
cell

`dat[2:3,2:3]`



consecutive
cells

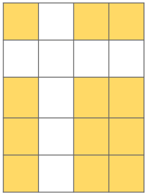
`dat[c(1,3,5),c(2,4)]`



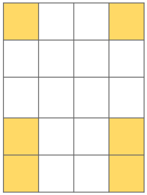
separate
cells

Cells

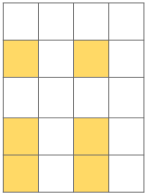
```
dat[-2,-2]
```



```
dat[-(2:3),
    -(2:3)]
```

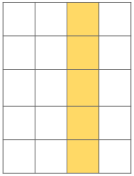


```
dat[-c(1,3),
    -c(2,4)]
```



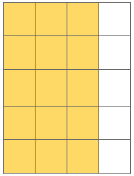
Columns

`dat[ ,3]`



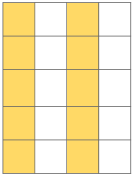
one single column

`dat[ ,1:3]`



consecutive columns

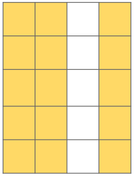
`dat[ ,c(1,3)]`



separate columns

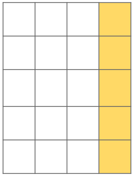
Columns

`dat[ ,-3]`



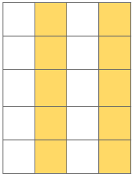
one single column

`dat[ ,-(1:3)]`



consecutive columns

`dat[ ,-c(1,3)]`



separate columns

Columns

```r
# column Ozone
airquality[ ,"Ozone"]

# columns Wind and Temp
airquality[ ,c("Wind","Temp")]
```

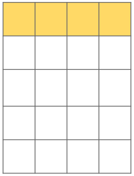
Columns

You can use argument `drop = FALSE` when selecting one column, in order to keep the second dimension of the returned output

```r
# first column (as a one-column data frame)
airquality[ , 1, drop=FALSE]

# column Ozone (as a one-column data frame)
airquality[ , "Ozone", drop=FALSE]
```

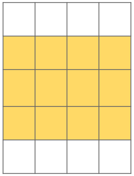
Rows

`dat[1, ]`



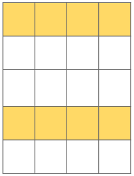
one single row

`dat[2:4, ]`



consecutive rows

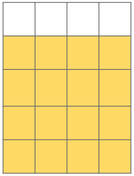
`dat[c(1,4), ]`



separate rows

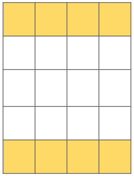
Rows

`dat[-1, ]`



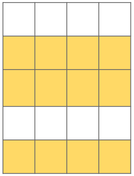
one single
row

`dat[-(2:4), ]`



consecutive
rows

`dat[-c(1,4), ]`



separate
rows

Bracket Notation

single brackets

```
dat[ , name(s)]
```

empty row index

column name(s)

```
dat$name
```

double brackets

```
dat[[j]]
```

single column index (integer, name)

Dollar Notation

```
# column Ozone
airquality$Ozone

# equivalently
airquality$"Ozone"

# equivalently
airquality$'Ozone'
```

Equivalent Calls: brackets and $

```r
airquality[ ,"Wind"]

airquality[ ,"Wind", drop=FALSE]

airquality["Wind"]

airquality[["Wind"]]

airquality$Wind
```

Although the above commands are “equivalent” in the sense that they all allow you to select the data in column `Wind`, keep in mind that there are differences in terms of output format, and performance.

Modifying Data Frames

Modifying Data Frames

We've seen how to access the elements from a `"data.frame"`. Pretty much the same operations to extract the contents can be used to modify them.

Well use a toy data set to see how we can modify the content of a `data.frame`.

Toy Data Frame

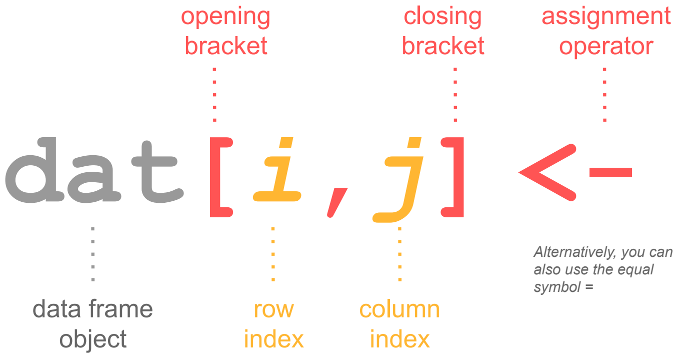
```
    name gender height weight
1 Anakin   male   1.88     84
2  Padme female   1.65     45
3   Luke   male   1.72     77
4   Leia female   1.50     49
```

Toy Data Frame

```r
# creating a data frame
dat <- data.frame(
  name = c('Anakin', 'Padme', 'Luke', 'Leia'),
  gender = c('male', 'female', 'male', 'female'),
  height = c(1.88, 1.65, 1.72, 1.50),
  weight = c(84, 45, 77, 49)
)
```

Modifying elements via index values

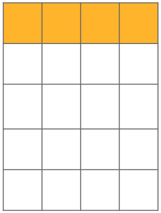
Modifying Elements via Index Values



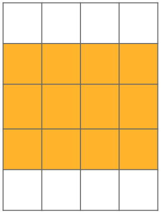
Alternatively, you can also use the equal symbol =

Modifying Rows

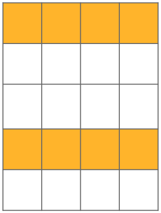
```
dat[i, ] <- x
```



one single row



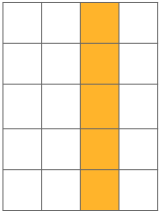
consecutive rows



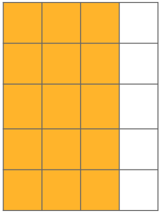
separated rows

Modifying Columns

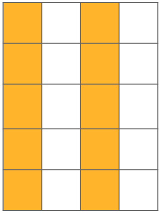
```
dat[ ,j] <- x
```



one single column



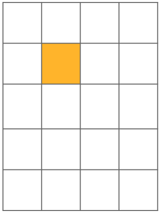
consecutive columns

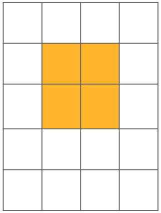


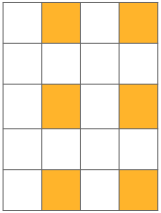
separated columns

Modifying Cells

```
dat[i,j] <- x
```


one single
cell


consecutive
cells


separated
cells

Example: modifying dat

```
# affecting cell 1,1
dat[1,1] <- 'ANAKIN'

# gender as factor
dat[,2] = factor(c('male',
'female', 'male', 'female'))
```

More options to modify columns

Example: modifying columns of `dat`

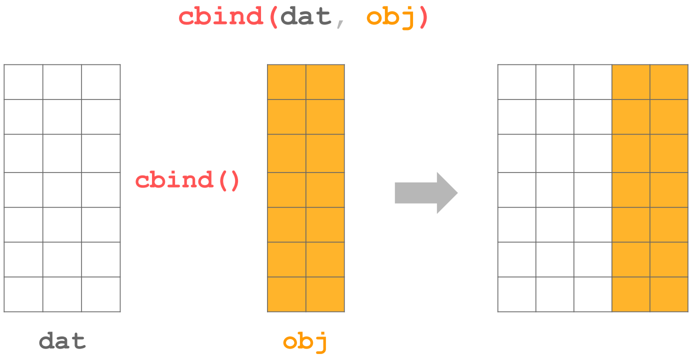
```r
# name in upper case
dat[ ,"name"] <- toupper(dat[ ,"name"])

# height in cms
dat$height <- dat$height * 100

# weight in pounds
dat[['weight']] <- dat[['weight']] * 2.20462
```

Example: logical subsetting

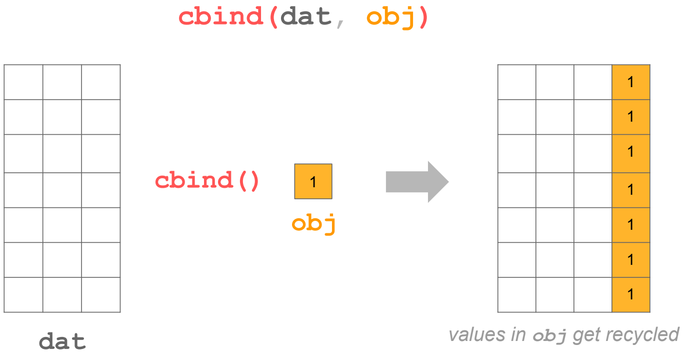
```r
# change male names
dat[dat$gender == 'male', 'name'] <- c('Anakin', 'Luke')

# multiple conditions
conds <- (dat$gender == 'female' & dat$height > 160)

dat[conds, 'name'] <- "PADME"
```

Adding new elements

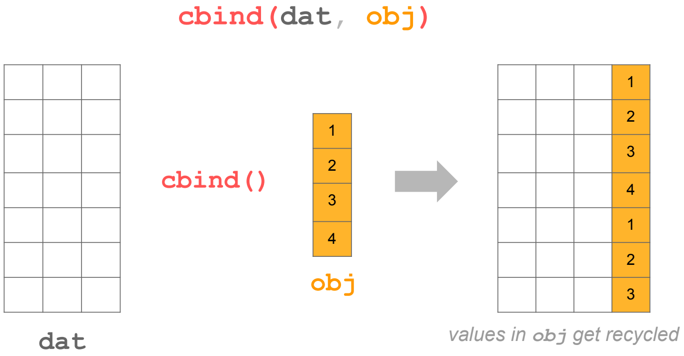
Adding columns with `cbind()`



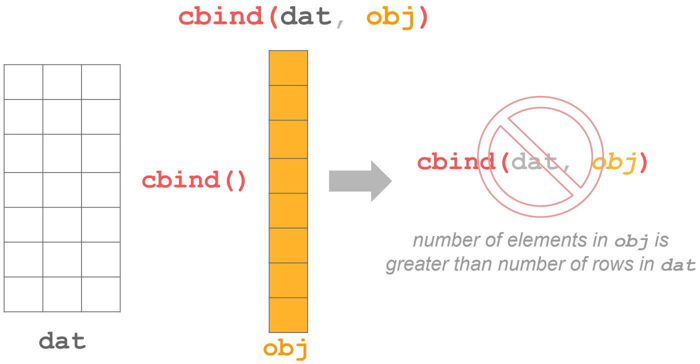
Adding columns with `cbind()`



Adding columns with `cbind()`



Adding columns with `cbind()`


cbind(dat, obj)
cbind()
dat
obj
cbind(dat, obj)
number of elements in obj is
greater than number of rows in dat

Adding columns with `cbind()`

```r
# adding constant
dat = cbind(dat, constant = "force")

# add random column
dat = cbind(dat, random = runif(nrow(dat)))
```

Other ways to add columns

Equivalent ways to add a `new` column to data frame `dat`:

```r
# add random column
dat$new <- runif(nrow(dat))

# add random column
dat[["new"]] <- runif(nrow(dat))

# add random column
dat["new"] <- runif(nrow(dat))
```

Combining dat[ ] and cbind()

```r
# number of columns
last = ncol(dat)
# increasing column indices
tmp = c(last+1, last+2)

# add new columns
dat[tmp] <- cbind(
  1:nrow(dat),
  rnorm(nrow(dat)))
```

Moving and Removing Columns

Moving columns

The common approach to move columns is to define a vector with the column names in the desired order, and then redefine the current data frame

```
# rearranged column names
rename = c('name', 'weight', 'height', "gender")

# moving columns
dat = dat[ ,rename]

# equivalently (list syntax)
dat[rename]
```

Removing a single column

To remove a column, you can indicate its name, and then *nullify it*

```
# equivalent calls
dat$weight <- NULL

dat[["weight"]] <- NULL

dat["weight"] <- NULL
```

Removing various columns

To remove various columns, you can indicate their names (via a character vector) or their column indices (via a numeric vector) and then *nullify them*

```
# equivalent calls
dat[c("weight", "height")] <- NULL

dat[ ,c("weight", "height")] <- NULL

dat[ ,c(3, 4)] <- NULL
```

Sorting Elements

Sorting Elements

Some sorting functions:

- `sort()`
- `order()`

```r
# Wind and Temp values
# in Month 5, ordered by Wind
wind5 = with(airquality,
order(Wind[Month == 5]))

airquality[wind5,
c('Wind','Temp')]
```

Merging Tables

Merging Data Frames with `merge()`

Merging two or more data tables is another frequent type of operation. This can be done using the **`merge()`** function.

The behavior of `merge()` depends on a combination of several arguments. We'll see some of the frequent scenarios.

Merging: Simplest case

```
merge(x, y, by = id)
```

id	x1	x2
a		
b		
c		
d		

&

id	y1	y2
a	1	5
b	2	6
c	3	7
d	4	8

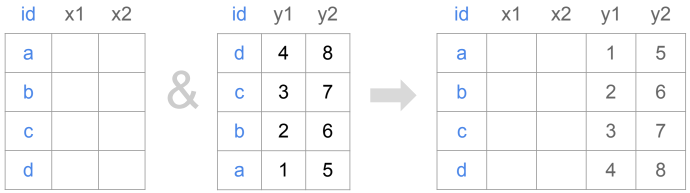
$\Rightarrow$

id	x1	x2	y1	y2
a			1	5
b			2	6
c			3	7
d			4	8

Same `id` columns (with same values) in both data frames.
In this case we could also use `cbind()`

Merging: Less simple case

```
merge(x, y, by = id)
```



id	x1	x2
a		
b		
c		
d		

&

id	y1	y2
d	4	8
c	3	7
b	2	6
a	1	5

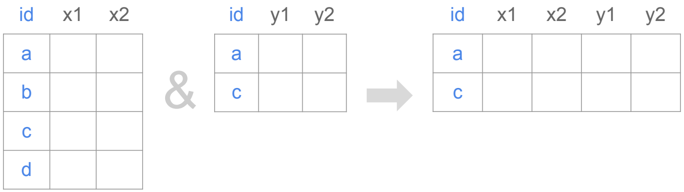
id	x1	x2	y1	y2
a			1	5
b			2	6
c			3	7
d			4	8

Same `id` columns in both data frames.
Same `id` values but in different order.

Data frames merging (case 1)

Merging: Default merging

```
merge(x, y, by = id, all = FALSE)
```



id	x1	x2
a		
b		
c		
d		

&

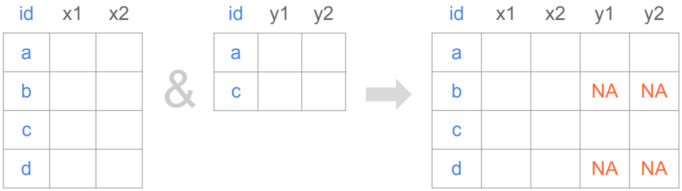
id	y1	y2
a		
c		

id	x1	x2	y1	y2
a				
c				

same **`id`** columns in both data frames.
`id` in **`y`** is a subset of **`id`** in **`x`**

Merging `all = TRUE`

```
merge(x, y, by = id, all = TRUE)
```



id	x1	x2
a		
b		
c		
d		

&

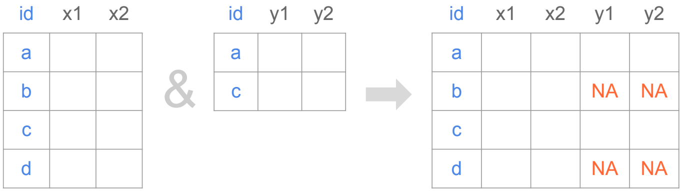
id	y1	y2
a		
c		

id	x1	x2	y1	y2
a				
b			NA	NA
c				
d			NA	NA

same `id` columns in both data frames.
`id` in `y` is a subset of `id` in `x`

Merging `all.x = TRUE`

```
merge(x, y, by = id, all.x = TRUE)
```



id	x1	x2
a		
b		
c		
d		

&

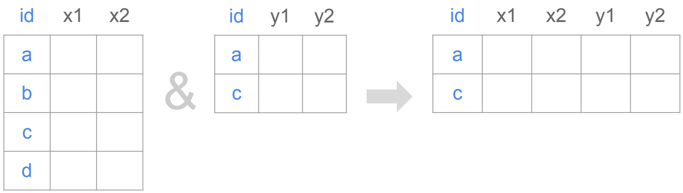
id	y1	y2
a		
c		

id	x1	x2	y1	y2
a				
b			NA	NA
c				
d			NA	NA

same `id` columns in both data frames.
`id` in `y` is a subset of `id` in `x`

Merging `all.y = TRUE`

```
merge(x, y, by = id, all.y = TRUE)
```

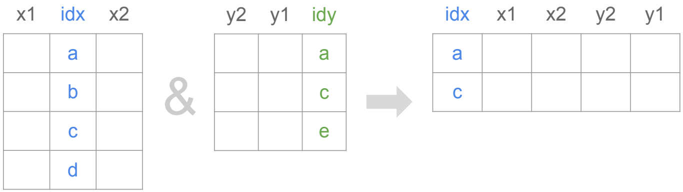


same **`id`** columns in both data frames.
`id` in **`y`** is a subset of **`id`** in **`x`**

Data frames merging (case 2)

Merging: Default `all = FALSE`

```
merge(x, y, by.x = idx, by.y = idy)
```



Different `id` columns in the data frames.

Merging: `all = TRUE`

```
merge(x, y, by.x = idx, by.y = idy,
      all = TRUE)
```

x1	idx	x2
	a	
	b	
	c	
	d	

&

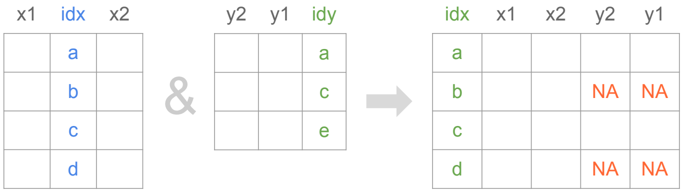
y2	y1	idy
		a
		c
		e

idx	x1	x2	y2	y1
a				
b			NA	NA
c				
d			NA	NA
e	NA	NA		

Different `id` columns in the data frames.

Merging: `all.x = TRUE`

```
merge(x, y, by.x = idx, by.y = idy,
      all.x = TRUE)
```



x1	idx	x2
	a	
	b	
	c	
	d	

&

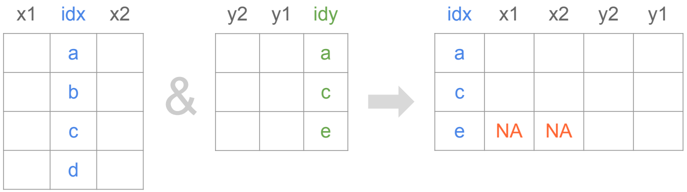
y2	y1	idy
		a
		c
		e

idx	x1	x2	y2	y1
a				
b			NA	NA
c				
d			NA	NA

Different `id` columns in the data frames.

Merging: `all.y = TRUE`

```
merge(x, y, by.x = idx, by.y = idy,
      all.y = TRUE)
```



x1	idx	x2
	a	
	b	
	c	
	d	

&

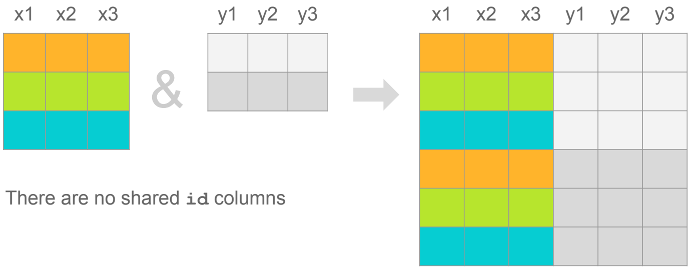
y2	y1	idy
		a
		c
		e

idx	x1	x2	y2	y1
a				
c				
e	NA	NA		

Different `id` columns in the data frames.

Merging with no matching column names

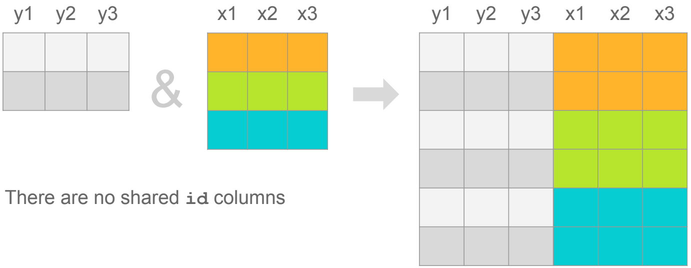
```
merge(x, y)
```



There are no shared `id` columns

Merging with no matching column names

```
merge(y, x)
```



There are no shared `id` columns

Tidyverse Approach

Later on we'll talk about another different approach for working with data.frames and similar tabular structures via the ecosystem of packages known as the `"tidyverse"`