Subsetting, missing values and dates in R

From assignment to selection

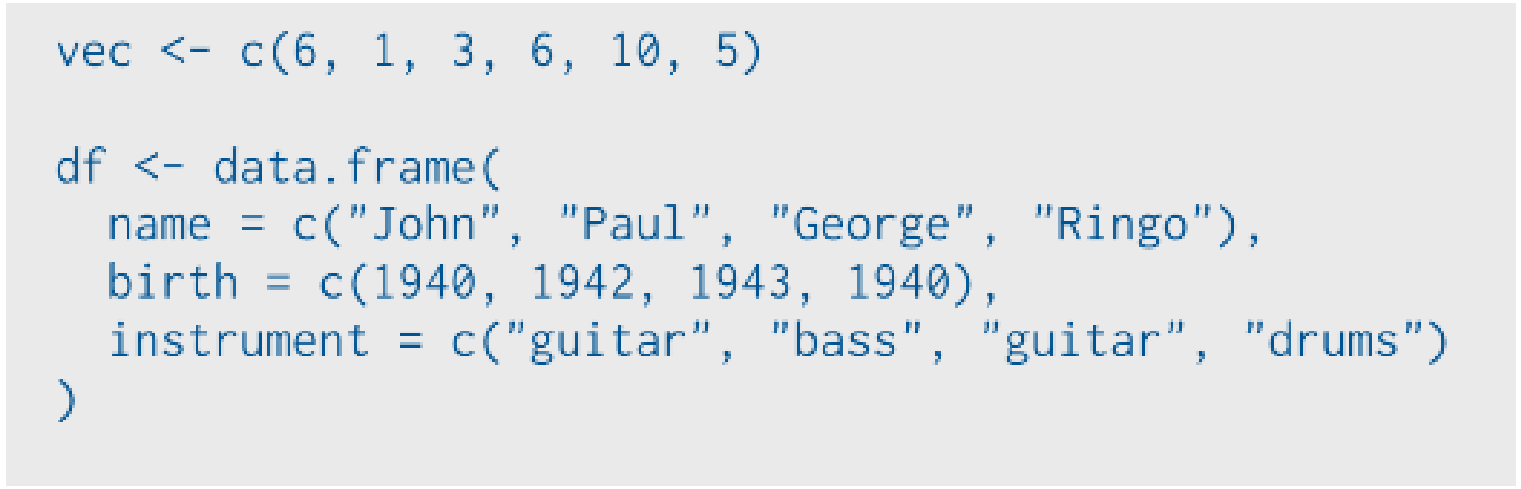
- Remembering vectors such as

```
X <- c(1,2,3,4)
```

- How do we access some of X?
- E.g. if we only want the 3rd element?
- ANSWER: subsetting!

This is an example from Rstudio team

- Run the following code in your head:



```r
vec <- c(6, 1, 3, 6, 10, 5)

df <- data.frame(
  name = c("John", "Paul", "George", "Ringo"),
  birth = c(1940, 1942, 1943, 1940),
  instrument = c("guitar", "bass", "guitar", "drums")
)
```

vec

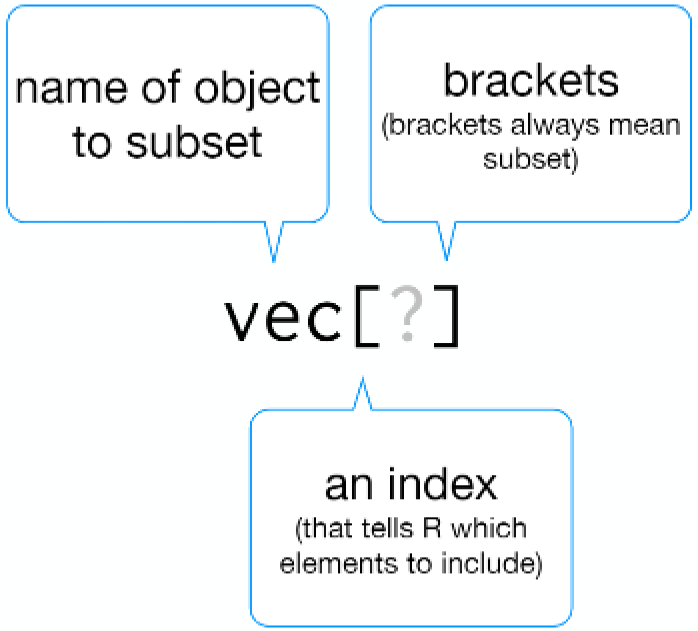
6	1	3	6	10	5

df

name	birth	instrument
John	1940	guitar
Paul	1942	bass
George	1943	guitar
Ringo	1940	drums

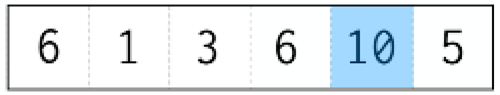
```
# Predict what the following code will do
# DON'T RUN IT!

vec[2]             df[c(2, 4), 3]
vec[c(5, 6)]       df[ , 1]
vec[-c(5,6)]       df[ , "instrument"]
vec[vec > 5]       df$instrument
```

Subset notation


name of object to subset
brackets (brackets always mean subset)
vec[?]
an index (that tells R which elements to include)

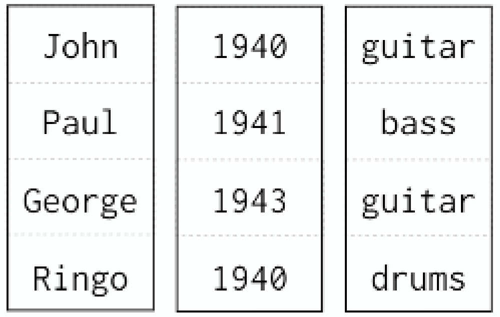
Each dimension needs it own index

```
vec[?]
```



6	1	3	6	10	5

```
vec[?]
 df[?,?]
```



John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

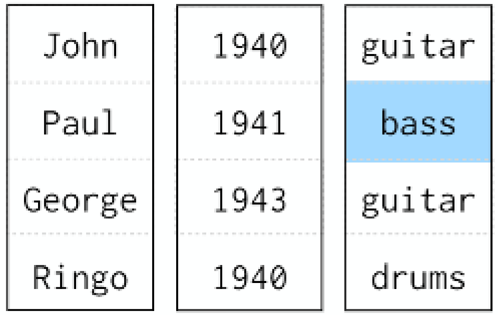
Four ways to subset in R

- Integers
- Blank spaces
- Names
- Logical vectors (i.e. TRUE/FALSE)

Integers

- Just like ij notation from linear algebra

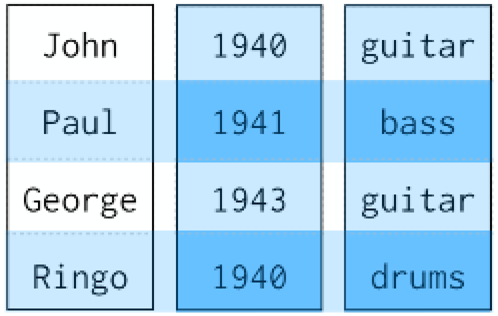
```
df[2,3]
```



John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

- Can pass more than on index

```
df[c(2,4),c(2,3)]
```



John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Special cases

- Colons
  - These give a range from i to j is i:j
  - Useful for creating vectors and repeating outputs

```
df[c(1,1,1,2,2), 1:3]
```

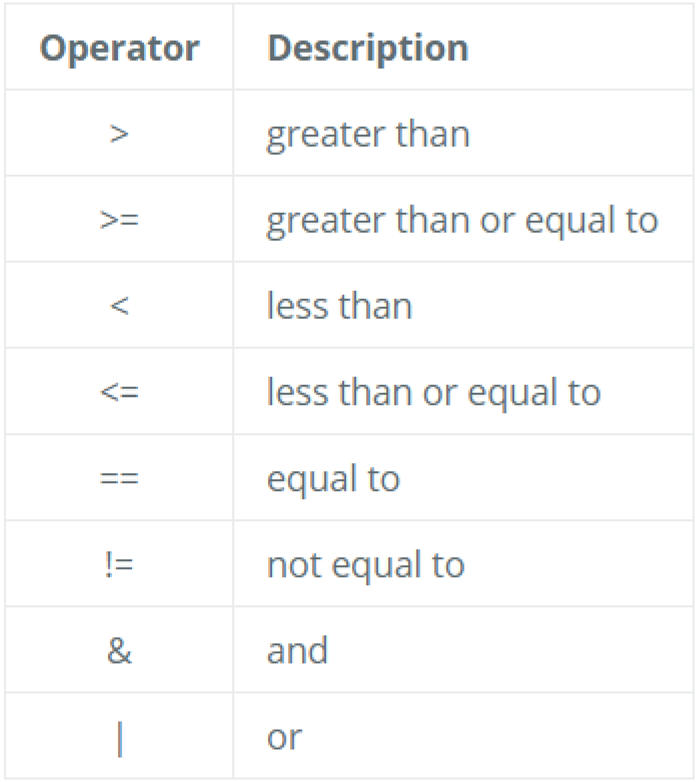
- Zero
  - Returns nothing so empty object

```
vec[0]
# numeric(0)
```

- Negative integer
  - Returns everything EXCEPT the index specified

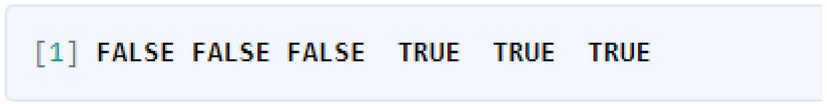
Using logical operators

- We can also use indices with logical operators. Logical operators include greater than (>), less than (<), and equal to (==). A full list of logical operators in R is displayed below:



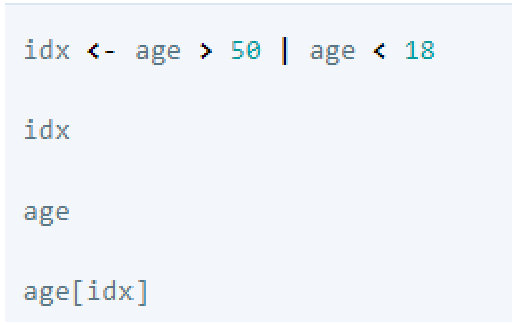
Operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
&	and
|	or

- If we want to know if each element in our vector is, say greater than 50, we can write
- vec > 50
- Which would return



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

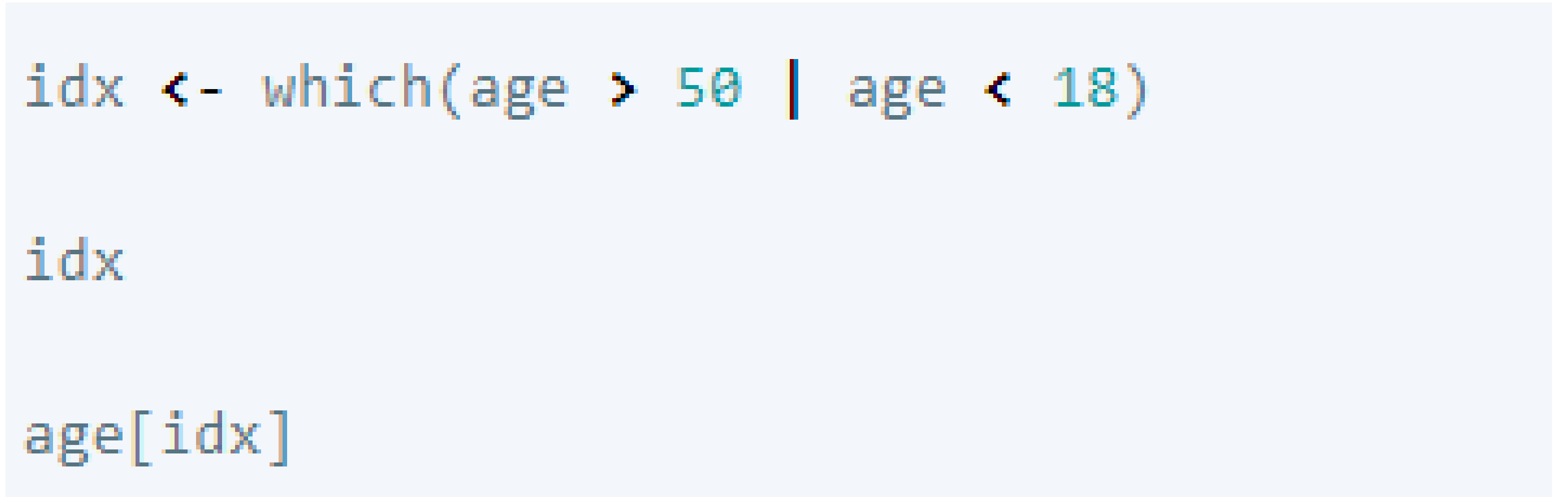
- We can extend this to multiple logical conditions such as



```
idx <- age > 50 | age < 18

idx

age

age[idx]
```

Indexing with logical operators using the which() function

- While logical expressions will return a vector of TRUE and FALSE values of the same length, we could use the which() function to output the indices where the values are TRUE. Indexing with either method generates the same results, and personal preference determines which method you choose to use. For example:



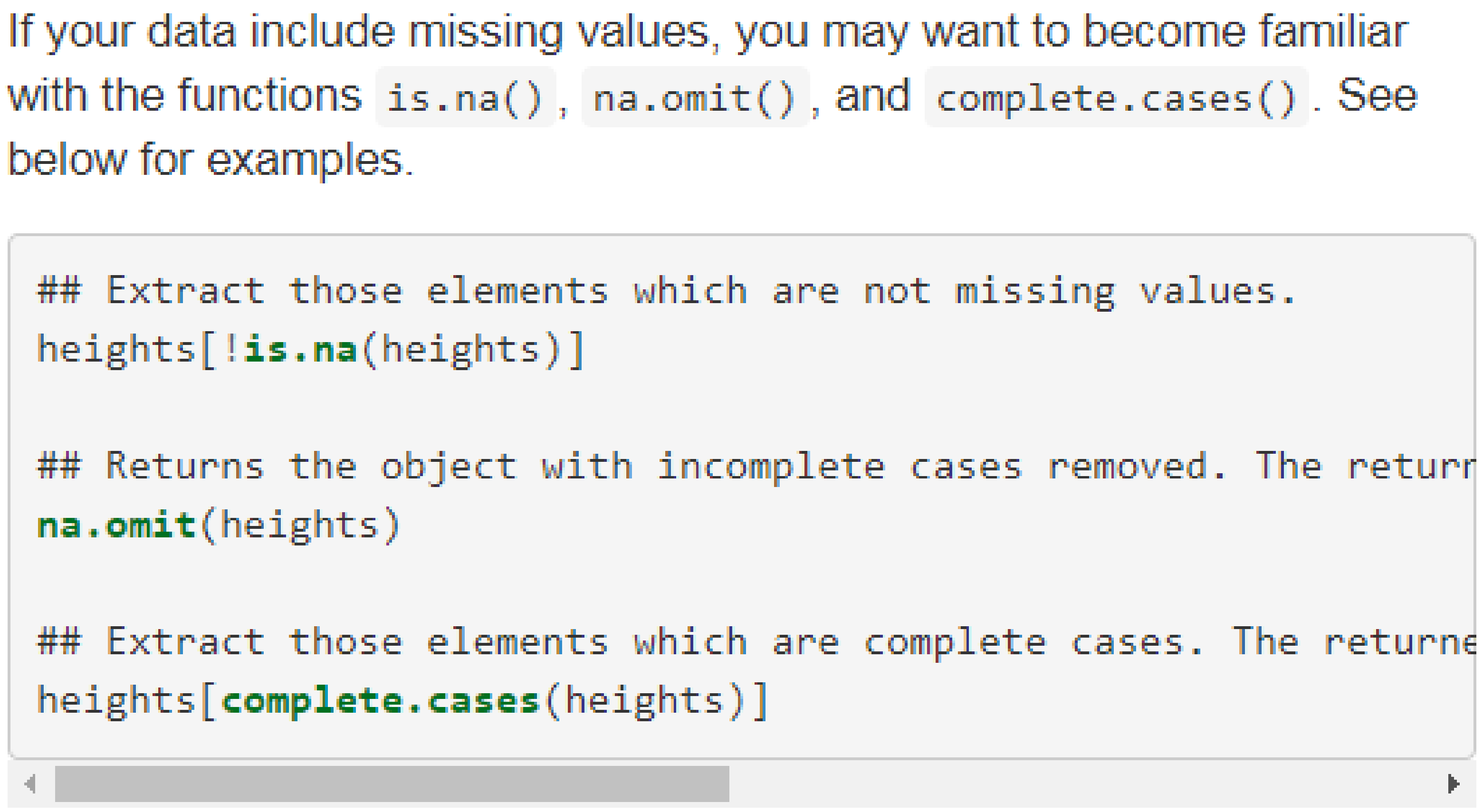
```
idx <- which(age > 50 | age < 18)

idx

age[idx]
```

Missing data

- As R was designed to analyse datasets, it includes the concept of missing data (which is uncommon in other programming languages)
- Missing data are represented in vectors as NA.
- When doing operations on numbers, most functions will return NA if the data you are working with include missing values.
- This feature makes it harder to overlook the cases where you are dealing with missing data.
- You can add the argument ***na.rm = TRUE*** to calculate the result while ignoring the missing values.

```r
heights <- c(2, 4, 4, NA, 6)
mean(heights)
max(heights)
mean(heights, na.rm = TRUE)
max(heights, na.rm = TRUE)
```

Useful NA functions



If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
## Extract those elements which are not missing values.
heights[!is.na(heights)]

## Returns the object with incomplete cases removed. The retur
na.omit(heights)

## Extract those elements which are complete cases. The returne
heights[complete.cases(heights)]
```

Factors

- Factors are very useful and actually contribute to making R particularly well suited to working with data. So we are going to spend a little time introducing them.
- Factors represent categorical data. They are stored as integers associated with labels and they can be ordered or unordered.
- While factors look (and often behave) like character vectors, they are actually treated as integer vectors by R. So you need to be very careful when treating them as strings
- Once created, factors can only contain a pre-defined set of values, known as *levels*. By default, R always sorts levels in alphabetical order.

Examples

```r
sex <- factor(c("male", "female", "female", "male"))
```

- R will assign 1 to the level "female" and 2 to the level "male" (because f comes before m, even though the first element in this vector is "male"). You can see this by using the function levels() and you can find the number of levels using `nlevels()`:

```r
levels(sex)
nlevels(sex)
```

Level order

- Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., “low”, “medium”, “high”), it improves your visualization, or it is required by a particular type of analysis. Here, one way to reorder our levels in the sex vector would be:

```r
sex # current order
```

```
#> [1] male   female female male
#> Levels: female male
```

```r
sex <- factor(sex, levels = c("male", "female"))
sex # after re-ordering
```

```
#> [1] male   female female male
#> Levels: male female
```

Converting factors

- To convert to a character vector use

```
as.character(sex)
```

- In some cases, you may have to convert factors where the levels appear as numbers (such as concentration levels or years) to a numeric vector.
- eg, in one part of your analysis the years might need to be encoded as factors (e.g., comparing average weights across years) but in another part of your analysis they may need to be stored as numeric values (e.g., doing math operations on the years).
- `as.numeric()` returns the index values of the factor, not its levels, so it will result in an entirely new (and unwanted in this case) set of numbers.
- One method to avoid this is to convert factors to characters, and then to numbers.

Renaming factors

- When your data is stored as a factor, you can use the `plot()` function to get a quick glance at the number of observations represented by each factor level.
- Example

```
sex <- surveys$sex
head(sex)
```

```
#> [1] M M
#> Levels:  F M
```

```
levels(sex)
```

```
#> [1] ""  "F" "M"
```

```
levels(sex)[1] <- "undetermined"
levels(sex)
```

```
#> [1] "undetermined" "F"            "M"
```

```
head(sex)
```

```
#> [1] M            M            undetermined undetermined un
determined
#> [6] undetermined
#> Levels: undetermined F M
```

Dates

- One of the most common issues that new (and experienced!) R users have is converting date and time information into a variable that is appropriate and usable during analyses.
- Simple practice for dealing with date data is to ensure that each component of your date is stored as a separate variable.
- Using str(), we can confirm that our data frame has a separate column for day, month, and year, and that each contains integer values

Lubridate

- We are going to use the ymd() function from the package lubridate (which belongs to the tidyverse; learn more here).
- lubridate gets installed as part as the tidyverse installation.
- When you load the tidyverse (library(tidyverse)), the core packages (the packages used in most data analyses) get loaded.
- lubridate however does not belong to the core tidyverse, so you have to load it explicitly with library(lubridate)

Date formats

- `ymd()` takes a vector representing year, month, and day, and converts it to a Date vector.
- Date is a class of data recognized by R as being a date and can be manipulated as such.
- The argument that the function requires is flexible, but, as a best practice, is a character vector formatted as “YYYY-MM-DD”.
- Alternatives are YYYY/MM/DD or YYYY/DD/MM etc

Let's create a date object and inspect the structure:

```r
my_date <- ymd("2015-01-01")
str(my_date)
```

Now let's paste the year, month, and day separately - we get the same result:

```r
# sep indicates the character to use to separate each component
my_date <- ymd(paste("2015", "1", "1", sep = "-"))
str(my_date)
```