Objects and data types

Session overview

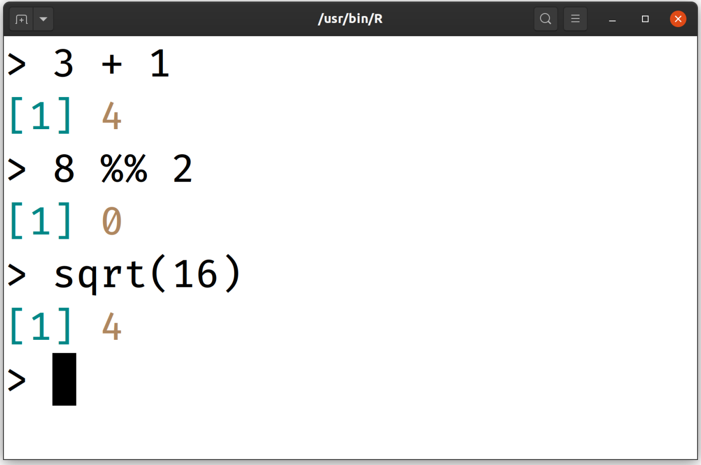
1. Objects and object assignment
2. Data types
3. More complex objects

These topics are fairly abstract.

It will make sense later.

Objects and object assignment

We've seen how R can be used interactively, as a calculator.



```
> 3 + 1
[1] 4
> 8 %% 2
[1] 0
> sqrt(16)
[1] 4
> 
```

This is great, but we often need to store things in memory.

- To use the result of one calculation as the input for another.
- To load some data and carry out an analysis.

So, we need some way of referring to these saved objects.

The assignment arrow

We can store something by giving it a name:

```
x <- 2
y <- 4
```

We can then use stored objects in subsequent calculations:

```
z <- x * y
```

We’ll return to this later...

Data types

You’ll come across many types of data

- Numeric (e.g., `1.0`, `2e12`)
- Integer (e.g., `1L`)
- Character (`"like this"`)
- Logical (`TRUE`, `FALSE`)
- Factors
- Missing values (e.g., `NA`)
- Date, times, intervals
- …

We need ways of representing these in R.

Numeric values

```r
x <- 1
y <- 1.42
```

Numeric values

```
x <- 1
y <- 1.42
```

Types and type conversion

- We can query the type of an object with `str` or `typeof`.
- We can check for specific types too, e.g. `is.numeric`, `is.integer`.
- We can convert between types with `as.numeric`, `as.integer`, etc.

Characters (or ‘strings’)

```
> first <- "Joe"
> last <- "Bloggs"
> age <- "40"

> is.character(first)
[1] TRUE

> paste(first, last)
[1] "Joe Bloggs"

> age + 10
Error in age + 10 : non-numeric argument...
> age <- as.numeric(age)
> age + 10
[1] 50
```

Logical (or boolean) values

```
> 5 > 4
[1] TRUE
> "Joe" == "Bloggs"
[1] FALSE
> "Joe" == "Joe"
[1] TRUE

> typeof(TRUE)
[1] "logical"

> str(TRUE)
 logi TRUE

> TRUE == FALSE
[1] FALSE
```

```
> !(TRUE)
[1] FALSE

> TRUE & FALSE
[1] FALSE

> TRUE | FALSE
[1] TRUE

> any(TRUE, FALSE, FALSE)
[1] TRUE

> all(TRUE, FALSE, FALSE)
[1] FALSE
```

Categorical values

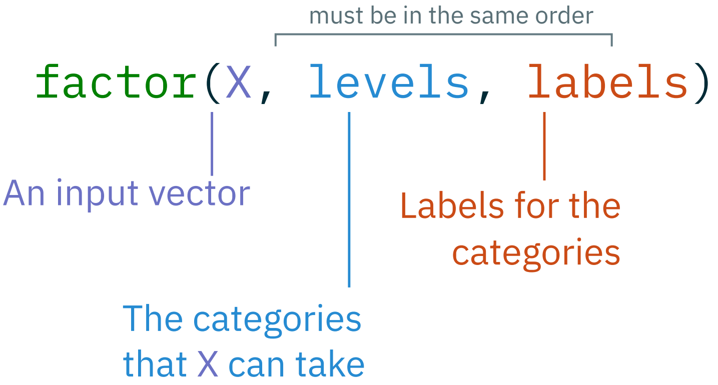
- Binary (e.g. sex)
- Nominal (e.g. ethnicity)
- Ordinal (e.g. education)

Binary values can be represented with TRUE/FALSE or 0/1:

```
> mtcars$ineff <- ifelse(mtcars$mpg < 15,
>                      TRUE, FALSE)
> mtcars$ineff
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[13] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> as.numeric(mtcars$ineff)
 [1] 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 [...]
```

Nominal or ordinal values can be represented as factors.

A `factor` represents categorical data in terms of **a numeric value** and **an associated label.**



must be in the same order

```
factor(X, levels, labels)
```

An input vector

The categories that X can take

Labels for the categories

If you’re familiar with Stata, this is similar to ‘values’ and ‘value labels’.

```
> marital
 [1] "Never married" "Divorced"      "Widowed"
 [4] "Never married" "Divorced"      "Married"
 [7] "Never married" "Divorced"      "Married"
[10] "Married"       "Married"       "Married"
[13] "Married"       "Married"       "Divorced"
[...]

> table(marital)
marital
     Divorced       Married Never married
         3383         10117          5416
    No answer     Separated       Widowed
           17           743          1807

> typeof(marital)
[1] "character"
```

```
> marital_f <- factor(marital)
> marital_f
[1] Never married Divorced      Widowed       Never
[5] Divorced      Married       Never married Divor
[9] Married       Married       Married       Marri
    [...]
Levels: Divorced Married Never married No answer
        Separated Widowed

> typeof(marital_f)
[1] "integer"

> as.numeric(marital_f)
[1]  3 1 6 3 1 2 3 1 2 2 2 2 2 2 1
[16] 2 6 3 2 2 2 2 3 6 6 6 6 6 1 6
[31] 6 2 2 3 2 3 3 3 3 3 2 2 1 3 3
[46] 3 2 2 2 2 3 2 2 2 2 1 1 1 3 3
```

```
> marital_n
[1]  3 1 6 3 1 2 3 1 2 2 2 2 2 2 1
[16] 2 6 3 2 2 2 2 3 6 6 6 6 6 1 6
[31] 6 2 2 3 2 3 3 3 3 3 2 2 1 3 3

> categories <- c("Divorced",
>                 "Married",
>                 "Never married",
>                 "No answer",
>                 "Separated",
>                 "Widowed")

> marital_f <- factor(marital_n,
>                     levels = 1:6,
>                     labels = categories)
```

Missing values

We can represent missing values with `NA`.

You may require a more informative representation of missing values.

For example:

- Not applicable
- Don’t know
- Refused

For this, I would use integers:

```
    -777    Not applicable
    -888    Don't know
    -999    Refused
```

e.g., `as.integer(-777)`.

We’ve covered several type conversions, but there are many more…

```
as.Date
as.character
as.numeric
as.ordered
as.difftime
as.double
as.complex
...
```

```
as.difftime
as.double
as.complex
...
```

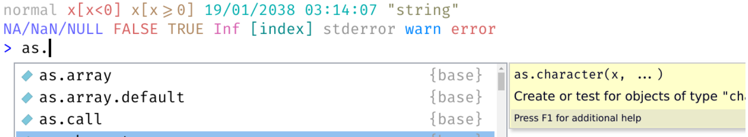
We've covered several type conversions, but there are many more…

```
as.Date
as.character
as.numeric
as.ordered
as.difftime
as.double
as.complex
...
```

```
as.difftime
as.double
as.complex
...
```

RStudio tip: Tab expansion

You can use tab expansion to see a list of available commands.



```
normal x[x<0] x[x⩾0] 19/01/2038 03:14:07 "string"
NA/NaN/NULL FALSE TRUE Inf [index] stderror warn error
> as.
```

```
as.array                {base}    as.character(x, ... )
as.array.default        {base}    Create or test for objects of type "ch
as.call                 {base}    Press F1 for additional help
```

Objects

Objects

> *An object is anything we want to store in memory.*

To store an object, we use the assignment operator.

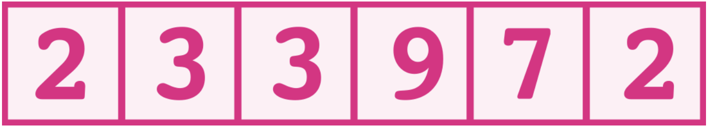
```r
x <- 1
y <- "A string"
z <- TRUE
```

If you don’t assign the result to an object, R will print the result and instantly forget what happened.

More complex objects

Vectors

- Vectors can be thought of as contiguous cells containing data.



2	3	3	9	7	2

- Vectors can contain any data type (e.g. logical, integer, string).
- However, a given vector can only contain one type (i.e., you can't mix them).
- Vectors can be defined with the `c` or `seq` commands.

Defining vectors

```r
# By hand
one_to_five <- c(1, 2, 3, 4, 5)

# Using the 'seq' function
lazy <- seq(from = 1,
            to   = 5,
            by   = 1)

# Same, but without naming the arguments
lazier <- seq(1, 5, 1)

# Using ':'
laziest <- 1:5
```

Matrices

A `matrix` is a rectangular array of data.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

They can be created with the `matrix` or `array` functions.

Matrices

```
> # Define a vector of integers.
> x <- 1:20

> # Fill matrix columns with 'x'
> matrix(x, ncol = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

By default, `matrix` fills by column. We can instead fill by row with the `byrow` option:

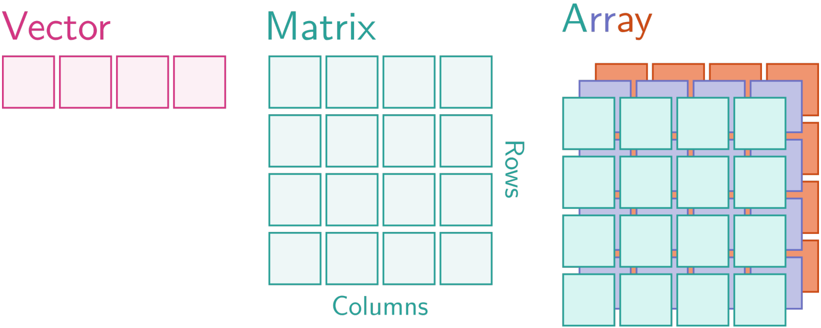
```
matrix(x, ncol = 5, byrow = TRUE)
```

```
> matrix(x, ncol = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20

> matrix(x, ncol = 5, byrow = TRUE)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
```

Arrays

An `array` is a vector with one or more dimensions.



We don’t often use them.