Acting On Variables

Plan for today

- Actions we perform on our objects
    - **operators**
    - **functions**
- Getting help when we need it

Operators

An **operator** is a simple calculation

+	addition
-	subtraction
*	multiplication
/	division
^	taking powers
%%	modulus

Order of Operations

Important note: Order of operations matters!

```
(8-4)/2
```

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

```
8-(4/2)
```

```
## [1] 6
```

Logical Operators

Test whether a statement is TRUE or FALSE

==	equality
!=	inequality
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Logical Operators

- Return a value of `TRUE` or `FALSE`
  - `empire$gender == "female"`
- Which Starwars character is more than 150 cm tall?
  - `empire$height > 150`
- Are any Starwars characters exactly 150cm tall?
  - `empire$height == 150`
  - **WARNING:** `empire$height = 150` will change your data!

Functions

- R is not *just* a calculator. You often want to do something more complex.
- To perform more complicated actions, we use ***functions***
    - **functions** are commands that describe, manipulate, or analyze objects
    - Logical operators & functions are the **verbs** of programming languages
    - This is why we use R! No one wants to calculate a regression by hand...

Functions have 3 parts

Function name

- Each function has one and only one name

```
# The function name is `log`
log(10)
```

```
## [1] 2.302585
```

Functions have 3 parts

Arguments

- One argument is always specified -- the input; this is the object that the function acts on.
- Other arguments control *how* the function acts. For example, do you want the natural log? Or log base 10?
- Each function has defaults for it's arguments. You should know where to find these and how to change them.

```
# The argument here is the input, or `10`
log(10)
```

```
## [1] 2.302585
```

Functions have 3 parts

Output

- The output of a function can be *any* of the object types & and of any class or even a combination of these
- Outputs can be a single value, vector, data.frame, matrix, list, or a plot
- *You can store the output by assigning it to another object!*

```
# The output is `2.302`
log(10)
```

```
## [1] 2.302585
```

```
# If we want to store `2.302` for later
newObject <- log(10)

# Now print out what is contained in `newObject`
newObject
```

```
## [1] 2.302585
```

Mathematical functions

Some obvious ones:

- `sqrt()` square root
- `round()` rounding a number
- `log()` logarithm
- `exp()` exponentiation
- `abs()` absolute value

Example:

```
sqrt(85)
```

```
## [1] 9.219544
```

Functions you'll use a lot!

`c()` - combine or concatenate

`length()` - find out how long a vector is (this is the same as getting the last position)

`factor()` - change a character vector into a factor vector (is there meaning? Ex: treatment vs. control, male vs. female, session 1 vs. session 2 etc.)

`table()` - really nice for getting quick counts (ex: how many males and females are there?)

`cbind()` and `rbind()` - add a vector to an existing data.frame. `cbind()` adds a new column. `rbind()` adds a new row

Multiple arguments

Most functions take more than 1 argument (more than just the input object).

Separate these arguments with commas `,`

```
round(x = 5.86921, digits = 3)
```

```
## [1] 5.869
```

Arguments have names

Use the argument names!

```
# perfect
round(x = 5.86921, digits = 3)
```

```
## [1] 5.869
```

```
# also perfect
round(digits = 3, x = 5.86921)
```

```
## [1] 5.869
```

Arguments have names

Use the argument names!

```
# right answer bc right order
round(5.86921, 3)
```

```
## [1] 5.869
```

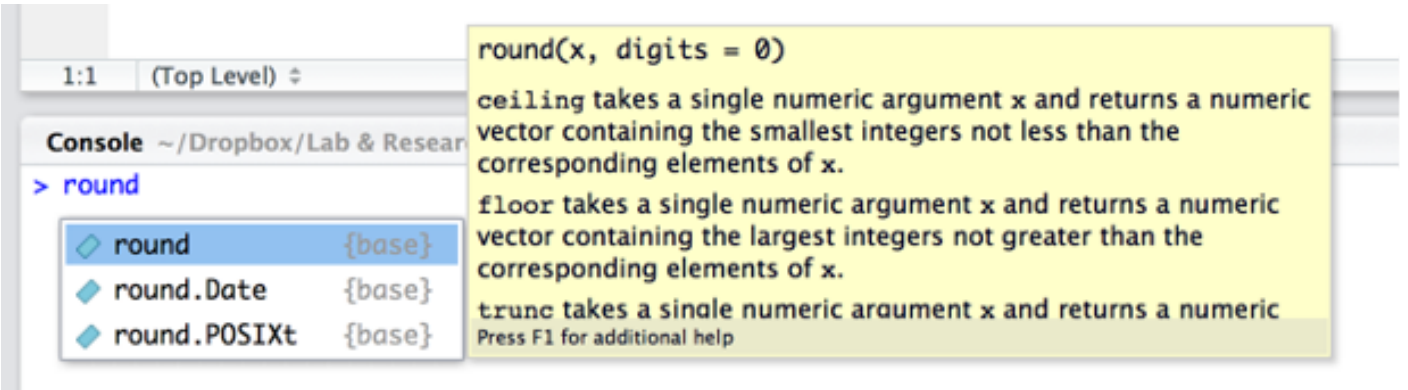
```
# wrong answer bc wrong order
round(3, 5.86921)
```

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

Great, but how do I know what the arguments are for a function?

Two ways:

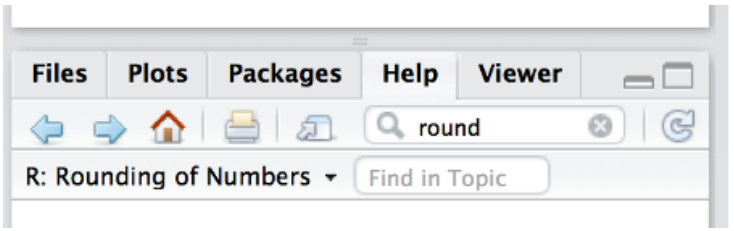
1. In RStudio, press the **tab** key to see the names of arguments and descriptions. (*note, this might not work in the online practice assignments, but it should definitely work when running RStudio locally*)



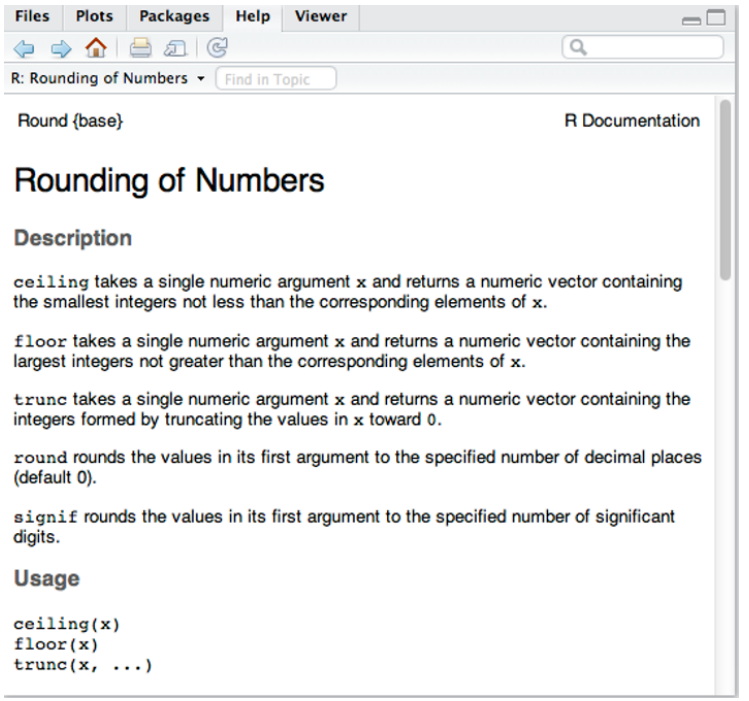
2. Look in the R Documentation

Looking at the documentation for help

Go to the `help` tab



Or type `?round` into the console



Breakdown of help documentation

Try `typing code` to look up the R documentation for the correlation function, which is called `cor`.

This will be the example we use.

R: Correlation, Variance and Covariance (Matrices)

cor {stats} R Documentation

Correlation, Variance and Covariance (Matrices)

Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

Arguments

`x`	a numeric vector, matrix or data frame.
`y`	`NULL` (default) or a vector, matrix or data frame with compatible dimensions to `x`. The default is equivalent to `y = x` (but more efficient).
`na.rm`	logical. Should missing values be removed?
`use`	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings `"everything"`, `"all.obs"`, `"complete.obs"`, `"na.or.complete"`, or `"pairwise.complete.obs"`.
`method`	a character string indicating which correlation coefficient (or covariance) is to be computed. One of `"pearson"` (default), `"kendall"`, or `"spearman"`, can be abbreviated.
`V`	symmetric numeric matrix, usually positive definite such as a covariance matrix.

Details

cor {stats}

Correlation, Variance and Covariance (Matrices)

Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

Arguments

`x`	a numeric vector, matrix or data frame.
`y`	`NULL` (default) or a vector, matrix or data frame with compatible dimensions to `x`. The default is equivalent to `y = x` (but more efficient).
`na.rm`	logical. Should missing values be removed?
`use`	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings `"everything"`, `"all.obs"`, `"complete.obs"`, `"na.or.complete"`, or `"pairwise.complete.obs"`.
`method`	a character string indicating which correlation coefficient (or covariance) is to be computed. One of `"pearson"` (default), `"kendall"`, or `"spearman"`, can be abbreviated.
`V`	symmetric numeric matrix, usually positive definite such as a covariance matrix.

Details

For `cov` and `cor` one must *either* give a matrix or data frame for `x` *or* give both `x` and `y`.

The inputs must be numeric (as determined by `is.numeric`: logical values are also allowed for historical compatibility): the `"kendall"` and `"spearman"` methods make sense for ordered inputs but `xtfrm` can be used to find a suitable prior transformation to numbers.

Value

For `r <- cor(*, use = "all.obs")`, it is now guaranteed that `all(r <= 1)`.

Examples

```
var(1:10)  # 9.166667

var(1:5, 1:5) # 2.5

## Two simple vectors
cor(1:10, 2:11) # == 1

## Correlation Matrix of Multivariate sample:
(Cl <- cor(longley))
## Graphical Correlation Matrix:
symnum(Cl) # highly correlated
```

All together

Logical operators evaluate `TRUE` or `FALSE`

- In `data$gender == "female"` the `==` is the **logical operator**
- However, `gender == "female"` doesn't work! R doesn't know where to look!

Indexing allows you to get a subset of your data

- For a 2-dimensional data.frame, `data[rows, columns]`
- If you want *all* the rows, `data[ ,columns]` (& vice versa)

COMBINING THESE is powerful!

- `data[data$gender == "female", ]` is correct!
- `data[gender == "female"]` is *incorrect*! Can you find the 2 reasons why?

Note: we will go through other ways of doing this; but understanding the logic is really, really important!