Stat 705
Class 1
Statistical computing with R

Richard P. Waterman

Wharton

Table of contents I

Today's module

Topics to be covered in this module:

- Course prerequisites and goals
- A quick look at the syllabus
- Top 10 reasons for wanting to learn R
- Getting started with R and RStudio
- Summary
- What you should be able to do now
- Next time

Prerequisites and goals

- Prereqs: Stat 613/621 (101/102, 111/112). Familiarity with regression
- No prior coding experience required
- Course goals
  1. Gain familiarity with R. What is it good for, bad for and its limitations
  2. You will have seen many of the sorts of things that R can do: graphics, reports etc.
  3. Exposure to the R *ecosystem*
  4. Good programming practice
- This isn't a class to learn more Stats, but rather a class to learn about programming for data and analytics. There are an increasing number of other Wharton classes that use R, so you will be filling those pre-reqs

Syllabus discussion

Syllabus discussion

Top 10 reasons for learning R

1. Free
2. Reproducibility of the analysis (full programming language)
3. Fast prototyping
4. Synthetic data creation/proof of concept
5. Immediate access to new analytics via add-on packages
6. Open source community/ecosystem/support
7. Parallelization and big data platforms
8. Work flow integration
9. Everyone else is using it/de facto standard
10. Tangible big data/productivity skill (good for the resume)

The RStudio studio IDE

Integrated **D**evelopment **E**nvironment

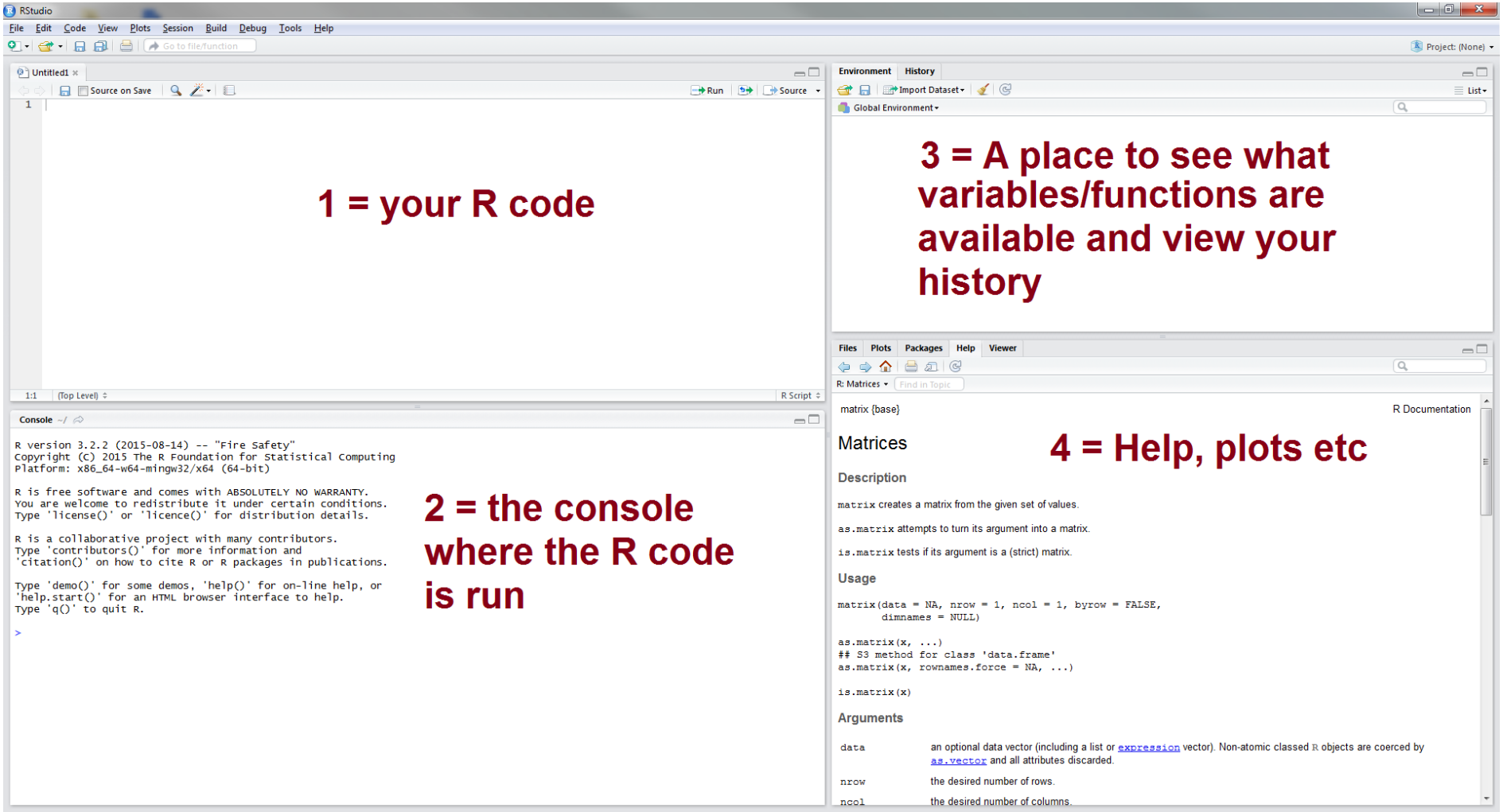
An IDE can increase productivity by rationalizing workflow. It allows you to manage a project through a single program. It provides lots of *hints*: a `syntax` guide, code highlighting, function completion, access to the help facility etc.

- The old, old days: create code in one file
- Source the code into the R terminal
- Debug
- Repeat

The new days, fire up RStudio and ...

RStudio

Fire up RStudio and get:



R-project activity flow

- Choose a directory to house the project
- Create a new R document
  - Write code into the R document
  - Run in the console
  - Debug, repeat
- Save R-code file with a sensible name into working directory
- Close RStudio and save project and workspace

Next day: fire up RStudio and read in your R project file and continue working.


Video tutorial on the R-project activity process

A first look at R

R can be used a calculator

```
# Notice that comments are started with the "#" character
# The basic arithmetic operators (+, - , * , /, ^ [power], %% [modulus])
# The functions will operate element-wise on vectors too
# This code will add these two numbers together
2 + 2
```

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

R has pretty much every mathematical function you could need

```
# Here's the exponential and log functions (base e by default)
exp(1) # Exponential function
```

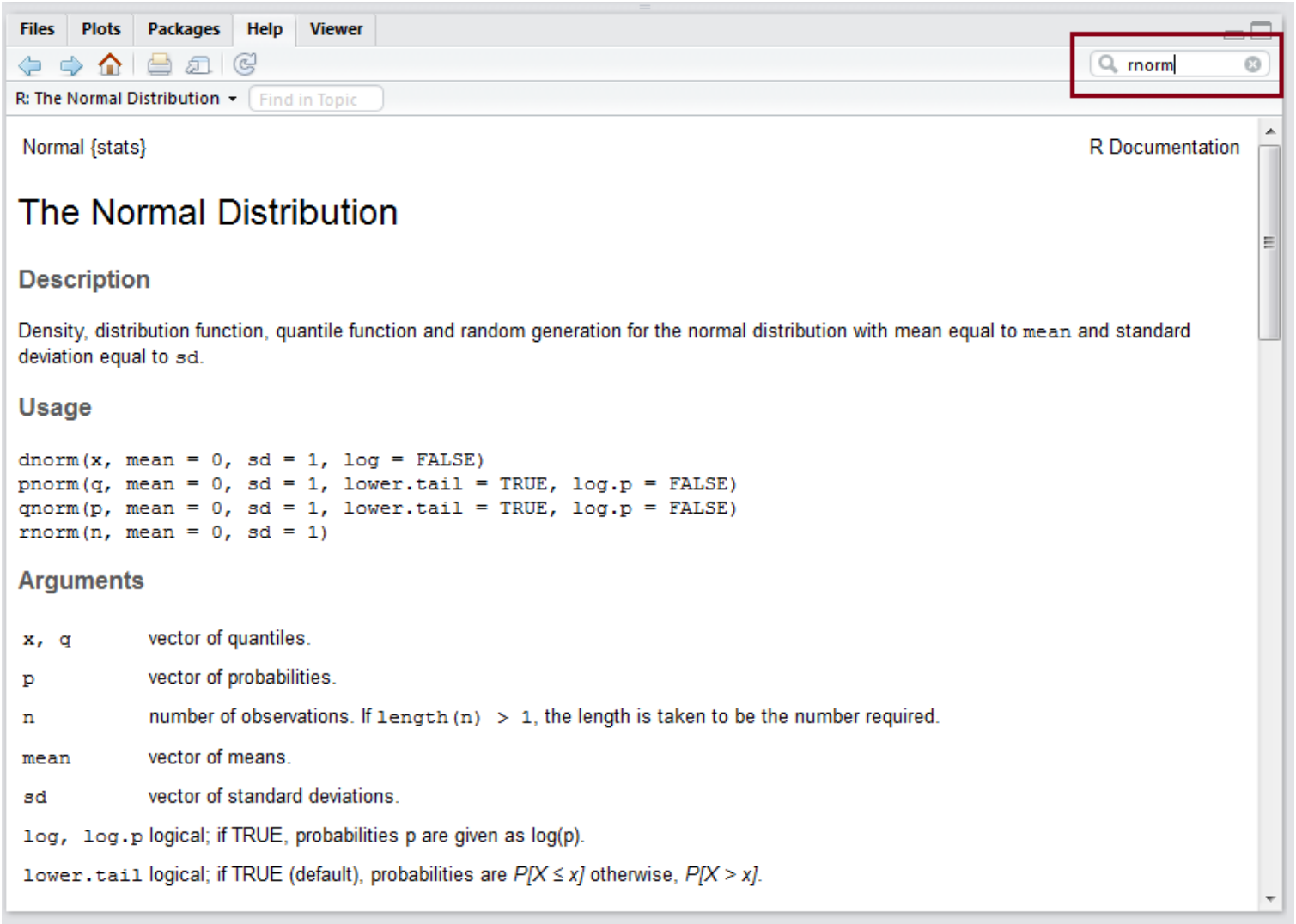
```
## [1] 2.718282
```

```
log(3) # Natural log
```

```
## [1] 1.098612
```

The Help facility

From the RStudio help window, search for commands



Help from the command line

Help can also be accessed from the command line in various ways:

```
help(rt) # Learn about the command for the t-distribution

help.search("rt") # Objects matching rt (note the quotes)

                  # apropos returns a vector of objects that fuzzy match
apropos("which")  # "which" in the search list (note the quotes)

## [1] "Sys.which" "which"     "which.max" "which.min"

?glm  # A short cut to the help command for glm

??glm # Everything matching glm on the search path
```

Monte-Carlo simulations

- Monte-Carlo (probabilistic scenario generation) is all about generating random variables.
- R has all the random variable distributions you could want:
  1. Normal
  2. t-distribution
  3. Bernouilli
  4. Binomial
  5. Poisson and so on ....
- It is typically a good idea to set the **seed** of the random number generator for reproducibility. The command to do this is `set.seed()`.

Generating random variables

```
# Set the seed of the random number generator to my Mum's birthday
set.seed(19390909)
# Generate some standard normals (in this case 25 Z's)
rnorm(25)
```

```
##  [1]  1.18442070 -0.17245683  1.26561144 -0.17012955
##  [5]  0.46851901 -0.76307874  0.71152068 -0.49668572
##  [9] -2.98218322  0.87247250 -0.21468746 -0.18043349
## [13]  0.65739175  0.21312101 -0.41492639 -0.01054358
## [17]  0.54461312  1.57315883  0.02126714 -0.03929818
## [21] -0.84758874  0.97450104  1.77367649  0.18858813
## [25] -1.12328407
```

```
# You can assign (save) the results of a calculation into a variable
# The variable "a" will be a vector with 25 elements
a <-  rnorm(25)
```

Notice the *assignment operator* `<-`

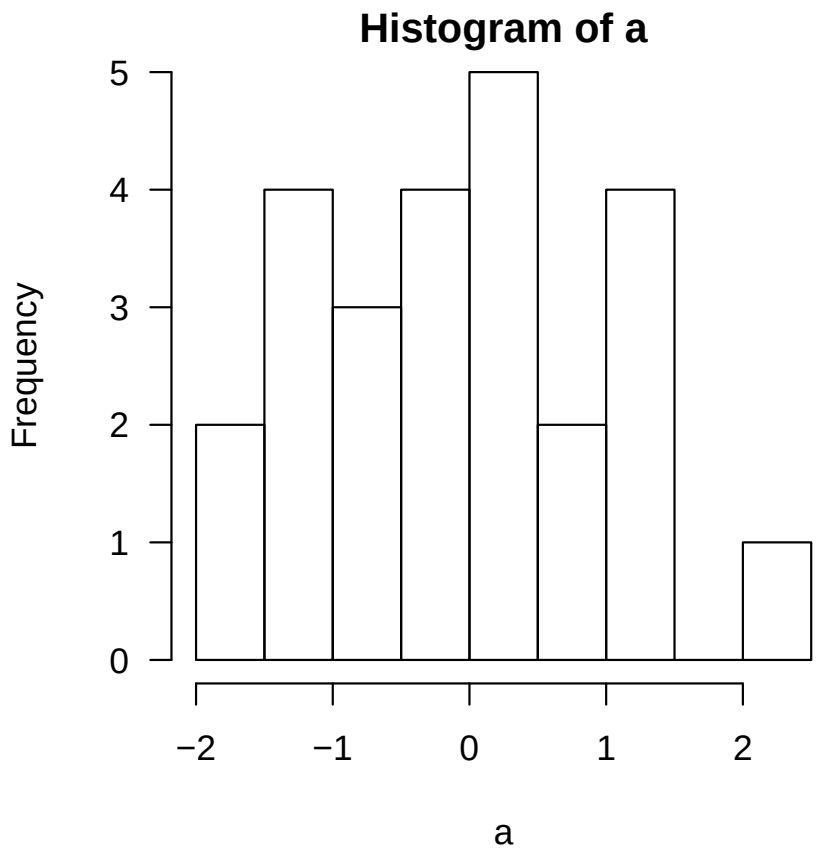
It is how you assign a `value` to a `variable`.

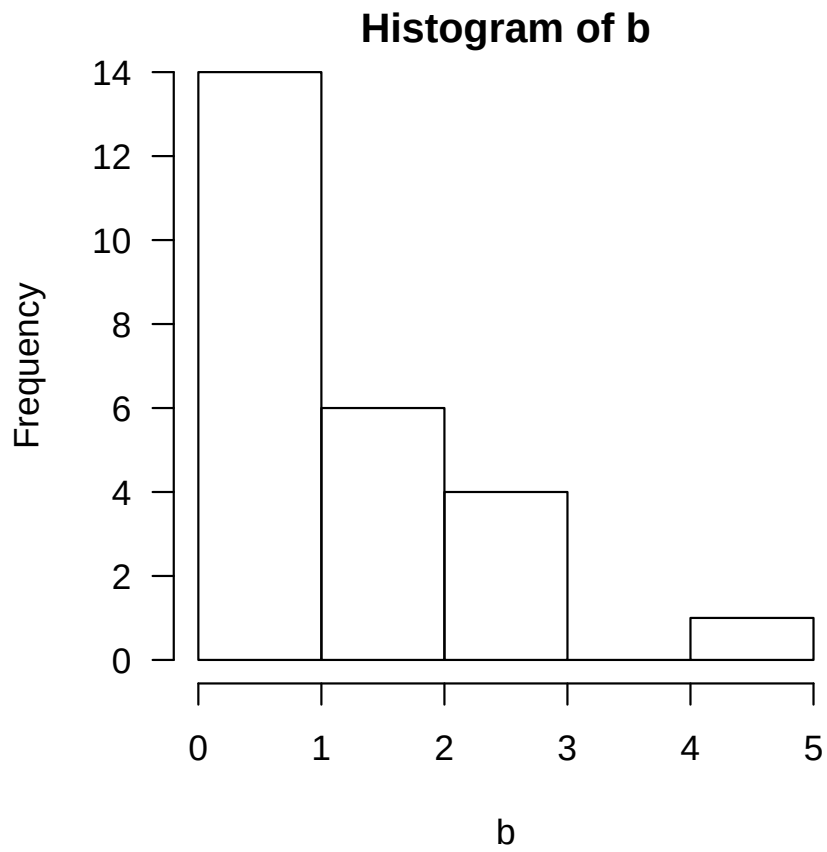
Generating random variables

```
# Have a look at what is in "a" now
print(a)

##  [1] -0.327525406 -0.422747106 -1.451195097 -1.584309765
##  [5]  0.267044038  0.558342576 -0.755558684  0.453240882
##  [9] -1.236154973  0.160023502 -0.661922583  1.412729092
## [13]  0.211450261 -0.002119005  1.124724970 -0.516598621
## [17]  1.246209228  1.385130745 -0.189339695  0.860492749
## [21]  0.169789733 -1.044686804 -1.429534729 -1.678716139
## [25]  2.169785393

# Basic statistical summaries of "a"
summary(a)

##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.678716 -0.755559 -0.002119 -0.051258  0.558343  2.169785

# Manipulate a: here we are squaring it
b <- a^2
```

Basic graphics

```
# Create histograms of "a" and "b"
hist(a)
hist(b)
```





Types of data

R stores data as different data `types`. A data `type` essentially tells the R program how the data should be stored and used.

- Numeric data
- Character data (needs quotes about it)
- Logical values

```
# Make a vector of values by using the "c" (combine) function
var1 <- c(21.2,15.6)
var2 <- c("Ford F-150", "Corvette")
var3 <- c(TRUE,FALSE)
# You can ask what type of data R thinks any variable is with
# the "class" function.
# You can  have multiple commands on the same line separated by ";"
class(var1); class(var2); class(var3)

## [1] "numeric"
## [1] "character"
## [1] "logical"
```

Named vectors

The elements of a vector can be named, which is useful for displaying results and accessing elements of the vector.

Vectors contain elements that all have the **same** type.
If you want to mix types, we will see a different structure, called a `list` for doing that.

```
# A named vector of GPA's

gpas <- c("Math" = 3.4, "Verbal" = 3.7, "Analytics" = 3.9)

print(gpas) # Note the output contains the names

##      Math    Verbal Analytics
##       3.4       3.7       3.9
```

Using basic operators on vectors

```
# Create two vectors, x and y
x <- c(1,2,3,4); y <- c(1,4,2,6)

# Add the numbers element-wise
x + y

## [1]  2  6  5 10

# Multiply the numbers element-wise
x * y

## [1]  1  8  6 24

# Divide x by y element-wise
x / y

## [1] 1.0000000 0.5000000 1.5000000 0.6666667
```

Applying a function (sum) to a vector

```
# We will apply the "sum" function to each of the three vectors
sum(var1)

## [1] 36.8

sum(var2)

## Error in sum(var2):  invalid 'type' (character) of argument

# Notice that you can't add up character data
 sum(var3)

## [1] 1

# Notice that TRUE/FALSE is converted/coerced to 1/0 for summing
```

Applying a function (mean) to a vector

```
mean(var1); mean(var2) ;mean(var3)

## [1] 18.4

## Warning in mean.default(var2):  argument is not numeric or logical:
returning NA

## [1] NA
## [1] 0.5
```

The `NA` value is the missing value code in R. It is **not** quoted.
The mean of the logical vector is found after coercing the logicals to numeric data.

Factor variables

When a variable will be used in a model as a categorical (discrete) variable, we code it as a `factor`. It looks like a character variable, but is represented internally in a different (more efficient) format.
It has

- A set of `levels`: the values that the variable can take
- A set of `labels` for each of the levels.
- It can be made `ordinal` by using the *ordered* argument

I surveyed 10 people and asked them if they would recommend the Black Tie Tailgate Gala at the Auto Show to a friend:

> *On a scale of 1 to 5 would you recommend this Black Tie Tailgate to a friend or colleague? (1 = definitely no, 2 = probably no, 3 = maybe, 4 = probably yes, 5 = definitely yes)*

Factor variables

Here are the results: `(4,5,3,4,4,5,2,4,1,3)`
We will call the results vector `opinions` and create it as a factor

```
# Entering this data as a factor
# Notice below that you can add comments at the end of a line too
opinions <- factor(
        x = c(4,5,3,4,4,5,2,4,1,3),   # the data
        levels = c(1,2,3,4,5),        # the possible values
        labels = c("definitely no","probably no",
                   "maybe", "probably yes",
                   "definitely yes"),# labels for each level
        ordered = TRUE)
print(opinions)

##  [1] probably yes   definitely yes maybe
##  [4] probably yes   probably yes   definitely yes
##  [7] probably no    probably yes   definitely no
## [10] maybe
## 5 Levels: definitely no < probably no < ... < definitely yes
```

Logical and comparison operators

We often want to compare numbers, and the logical and comparison operators let us do that. The core functions:

```
<   less than
<=  less than or equal to
>   greater than
>=  greater than or equal to
==  exactly equal to
!=  not equal to
!x  Not x
x | y  x OR y
x & y  x AND y
```

These can be applied to vectors and will operate element-wise on the vector. They return logical values.

Examples – comparison operators

```
x <- c(1,2,3); y <- c(3,2,1)
x < y ; x <= y

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

x > y ; x >= y

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

x == y; x != y

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

Examples – logical operators

```
x <- c(TRUE,TRUE,FALSE,FALSE); y <- c(TRUE,FALSE,TRUE,FALSE)
x | y

## [1]  TRUE  TRUE  TRUE FALSE

x & y

## [1]  TRUE FALSE FALSE FALSE

!y

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

Good coding practice

- Use the RStudio project feature to manage your workflow
- Copiously document your code
- Use variable names that are self descriptive
- Indenting code can greatly enhance readability
- Don't hardcode values that remain constant, but make repeat appearances. Store them into variables and reference the variable name
- Check out Google's style guide for R coding, posted on the Canvas *useful links* page

Module summary

Topics covered today include:

- Reviewed the syllabus
- Previewed RStudio
- Variables types
- Factor variables
- Vectors (`c()`)
- Random numbers
- Good coding practice

Next time

- Data structures (vectors/matrices/arrays/lists and data tables)
- Subsetting, slicing and dicing
- Applying functions to arrays and lists

Gotchas

- Mixing data types in the same vector will cause all elements to be coerced to the same type.
- Testing for logical equality with = rather than the correct ==.
- Quoting the missing value code. It should **not** be quoted. `NA` is not the same as `"NA"`.
- Don't put commas in numbers, as in 999,999.

Today's function list

Do you know what each of these functions does?

```
<-
c
class
exp
factor
hist
log
mean
print
rnorm
set.seed
sum
summary
```