R Primer

Download at https://github.com/gabrielrvsc/class

Getting Started with R

- You can download R from https://www.r-project.org/
- R is an open-source language mostly used for Statistics and data analysis.
- I is based on a core build with basic functions plus packages that can be installed.
- A package is a set of fuctions/data.
- R community is huge and you should use it! Most of your questions were already answered on Stack Overflow (https://stackoverflow.com).
- Stack Overflow is a forum where people post questions about programming.

RStudio

- R alone is just a console that looks like a notepad. It is not very friendly.
- RStudio is an Integrated development environment (IDE) made for R.
- You can download RStudio from https://rstudio.com/products/rstudio/download/
- Make sure to get the free desktop version.
- RStudio will automatically find your R once installed.

RStudio

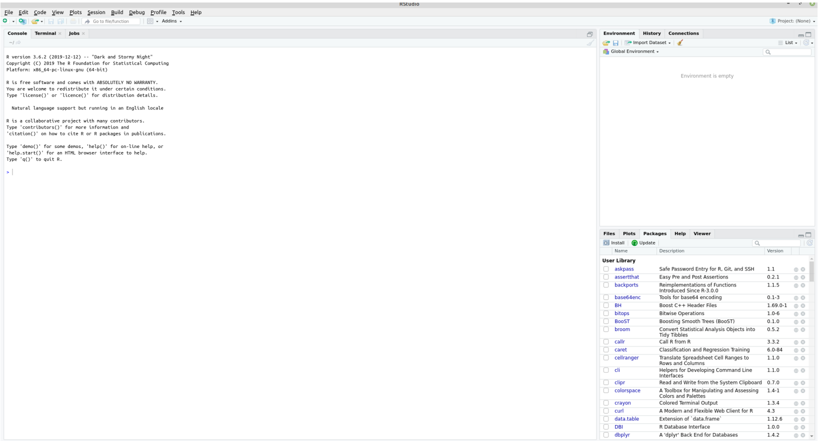
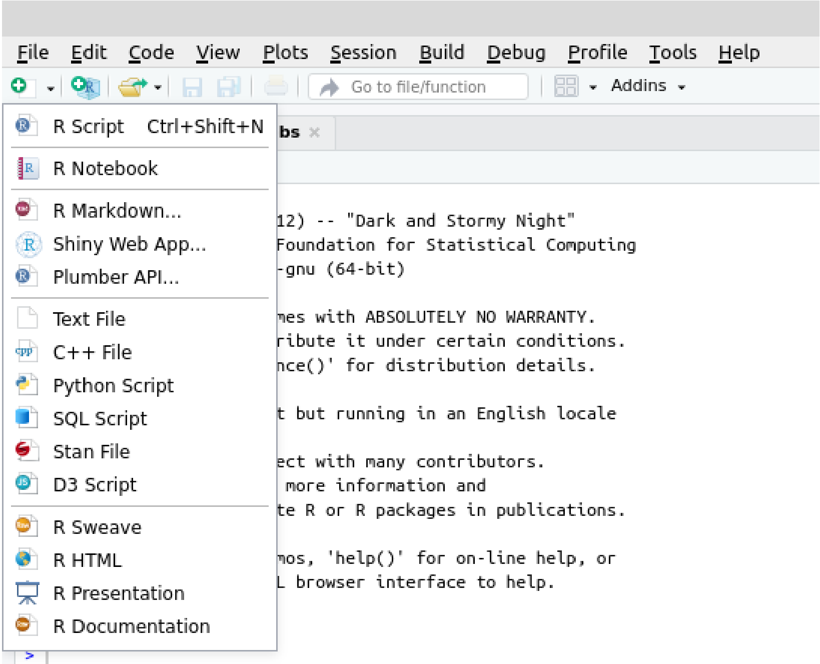


Figure 1: RStudio

RStudio


File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
R Script Ctrl+Shift+N
R Notebook
R Markdown...
Shiny Web App...
Plumber API...
Text File
C++ File
Python Script
SQL Script
Stan File
D3 Script
R Sweave
R HTML
R Presentation
R Documentation
12) -- "Dark and Stormy Night"
Foundation for Statistical Computing
-gnu (64-bit)
nes with ABSOLUTELY NO WARRANTY.
ribute it under certain conditions.
nce()' for distribution details.
t but running in an English locale
ect with many contributors.
more information and
te R or R packages in publications.
mos, 'help()' for on-line help, or
L browser interface to help.

RStudio

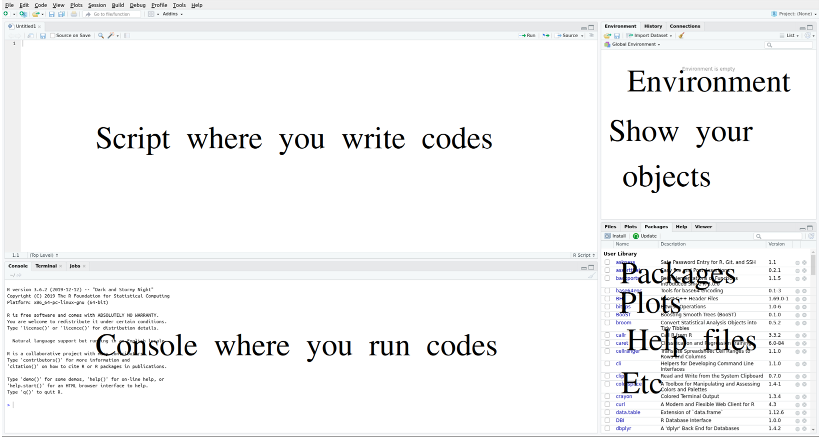


Figure 3: RStudio

RStudio

- You can run commands on the console. For example, try 1+1 and hit enter.
- You can write codes on the script and run in the console by selecting them and pressing Ctrl + Enter (Cmd+Enter for MAC). You can also run the selected codes by pressing the Run button on the top right corner of the script.
- Type 1+1 on the script and run it.

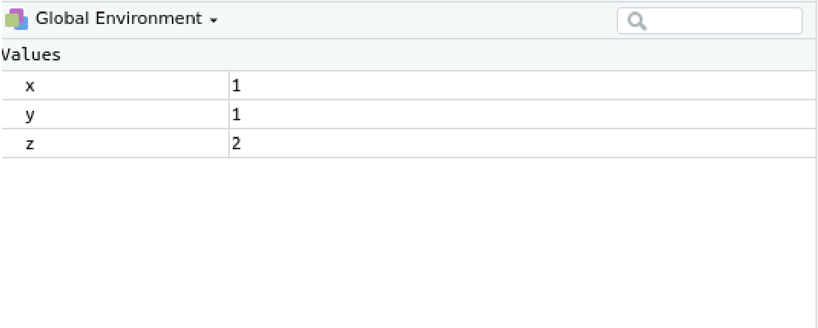
Creating Objects

- Objects are created with the symbols $=$ or $<$- .
- creating an object is equivalent to storing the result.

```
x = 1
y <- 1
z = x + y
```

Creating Objects

- The objects you create will show in your environment.



Objects

- You can see all your objects with:

```
ls()
```

```
## [1] "x" "y" "z"
```

- You can dele an object with:

```
rm(list = c("x", "y"))
```

- You can clear your environment with:

```
rm(list = ls())
```

Working Directory

- The working directory (WD) is the folder where R is working.
- If you read or write a file, the default location will be your WD.
- If you want to read/write from somewhere else you will have to tell R the path.
- Type the following command to find your WD:

```
getwd()
```

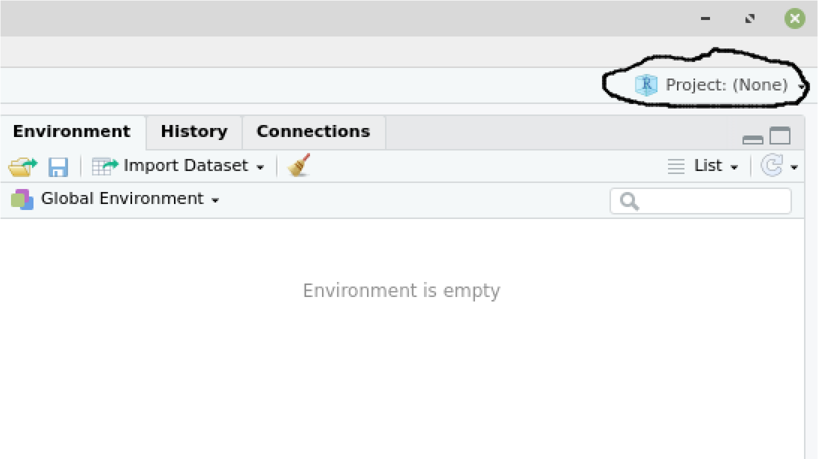
- You can change with the following command:

```
setwd("path")
```

- The path must be between quotation marks.

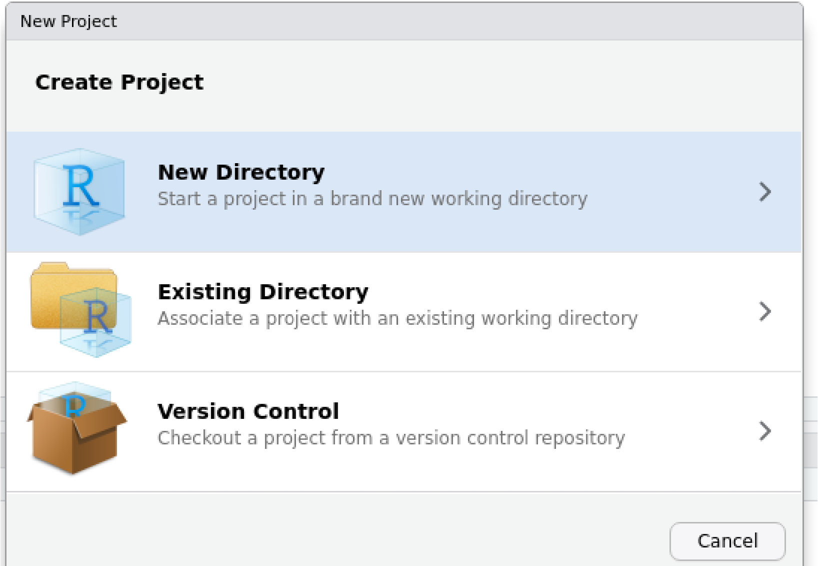
Projects

- Projects are a very good way to keep your work organized.
- Once you setup a project it will be linked to a folder, which will be the project WD.
- You can resume your work by just opening the project and everything will be ready for you.

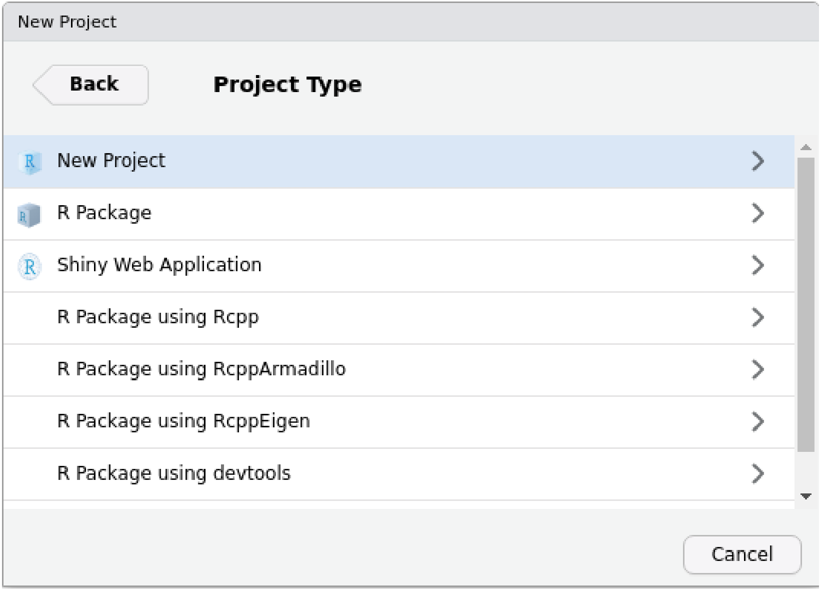


Projects

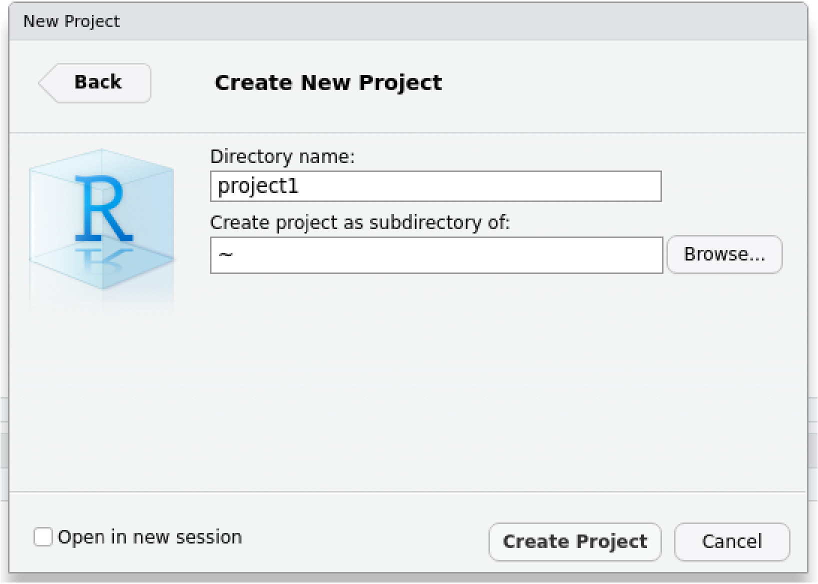
To start a project on a new folder open file > new project and follow these steps:



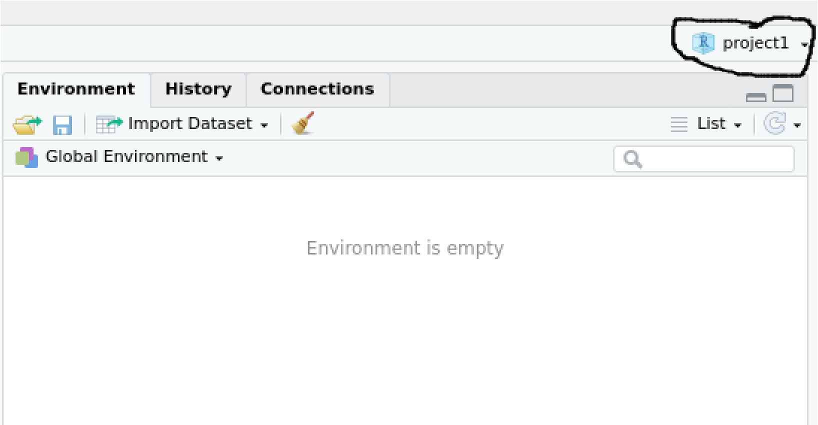
Projects


New Project
Back
Project Type
New Project
R Package
Shiny Web Application
R Package using Rcpp
R Package using RcppArmadillo
R Package using RcppEigen
R Package using devtools
Cancel

Projects


New Project
Back
Create New Project
Directory name:
project1
Create project as subdirectory of:
~
Browse...
Open in new session
Create Project
Cancel

Projects


project1
Environment
History
Connections
Import Dataset
List
Global Environment
Environment is empty

Projects

- The project will create a file project1.Rproj in the selected WD. You can use it to open your project.
- You can also switch between projects by clicking on project1 like the last slide.

Basic Operations

```r
# Addition
3 + 3
```

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

```r
# Subtraction
5 - 4
```

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

```r
# Multiplication
5 * 6
```

```
## [1] 30
```

```r
# Division
10 / 5
```

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

```r
# Exponent
6 ^ 2
```

```
## [1] 36
```

Basic Operations

- The order of priority is Exponent > Division > Multiplication > Addition = Subtraction.
- This is the order R will use in an expression like this:

```r
2 + 2 * 3 + 5 / 2 ^ 2
```

```
## [1] 9.25
```

- You can change the priority using parentheses.

```r
# The first addition should go first and
# the division should go before the exponent
(2 + 2) * 3 + (5 / 2) ^ 2
```

```
## [1] 18.25
```

Logical/Misc Operators

Logical/Misc Operators	
==	equal to
!=	not equal equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
|	OR
&	AND
a%in%b	is 'a' contained in 'b'
%*%	matrix multiplication

Logical/Misc Operators

Some examples:

```r
1 == 2
```

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

```r
1 != 2
```

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

```r
3<4
```

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

We will deal more with this operators latter.

Data Types, Classes, and Structures

Up to this point, we have been working with numbers. There are actually six data types in R:

- Double: Numeric, real.
- Integer: Numeric, integer.
- Character: Name.
- logical: TRUE, FALSE
- complex: $a + bi$ - A complex number.
- raw: a byte.

We will work only with the first four types.

Data Types, Classes, and Structures

- You can ask R the type of the object:

```r
typeof(2)
```

```
## [1] "double"
```

```r
typeof(2L)
```

```
## [1] "integer"
```

```r
typeof("a")
```

```
## [1] "character"
```

```r
typeof(TRUE)
```

```
## [1] "logical"
```

Data Types, Classes, and Structures

- You can also ask R if an object is of a particular type:

```r
is.double(2)
```

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

```r
is.integer(2L)
```

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

```r
is.character("a")
```

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

```r
is.logical(TRUE)
```

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

```r
is.character(2)
```

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

Data Types, Classes, and Structures

- You can also ask for the class of the object.
- Class is not the same as type. For example, a matrix is a class and its type can be double, integer, character, etc...

```
# creates a 2x2 matrix of zeros.
A = matrix(0, 2, 2)
class(A)
```

```
## [1] "matrix"
```

```
typeof(A)
```

```
## [1] "double"
```

Data Types, Classes, and Structures

- A particular class we will be using is called factor.
- They are used for categorical variables.
- They only accept a particular set of values called levels.

```
fact = c("a","b","b")
fact = as.factor(fact)

levels(fact)
```

```
## [1] "a" "b"
```

```
class(fact)
```

```
## [1] "factor"
```

Data Types, Classes, and Structures

- So far we mostly worked with objects of a single element.
- R have several data structures:

	Homogeneous	Heterogeneous
1-dimensional	vector	list
2-dimensional	matrix	data frame
more than 2 dimensions	array	

Data Types, Classes, and Structures

- Vectors are created with the following command:

```
vec1 = c(2,4,6,8,10)
is.vector(vec1) # is it a vector?
```

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

```
is.double(vec1) # is it type double
```

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

```
is.numeric(vec1) # is it class numeric?
```

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

```
is.atomic(vec1) # is it homogeneous?
```

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

Data Types, Classes, and Structures

- There are other ways to create vectors:

```
vec2 = seq(2,10,2) #seq(from,to,by)
vec2
```

```
## [1]  2  4  6  8 10
```

```
vec3 = 1:10 #seq with by = 1
vec3
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
length(vec3)
```

```
## [1] 10
```

Data Types, Classes, and Structures

- Vector operations

```
vec4 = 1:5
vec5 = 6:10

vec4 + vec5
```

```
## [1]  7  9 11 13 15
```

```
vec4 - vec5
```

```
## [1] -5 -5 -5 -5 -5
```

```
vec4 * vec5 #inner product
```

```
## [1]  6 14 24 36 50
```

Data Types, Classes, and Structures

- We can transpose vectors with the **t** function.

```
t(vec4)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
```

```
vec4 %*% t(vec5) #5X1 times 1X5
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    6    7    8    9   10
## [2,]   12   14   16   18   20
## [3,]   18   21   24   27   30
## [4,]   24   28   32   36   40
## [5,]   30   35   40   45   50
```

```
t(vec4) %*% vec5 #1X5 times 5X1
```

```
##      [,1]
## [1,]  130
```

Data Types, Classes, and Structures

We can also do logical operations with vectors:

```
vec4 == vec5
```

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

```
1 %in% vec4 #ask if 1 is in vec4
```

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

```
vec5 > vec4
```

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

Data Types, Classes, and Structures

```r
A = matrix(c(1,2,3,4,5,6),nrow=2,byrow=TRUE)
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```r
is.atomic(A)
```

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

```r
is.matrix(A)
```

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

```r
B = matrix(c(1,6,3,5,7,2),nrow=3)
B
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    6    7
## [3,]    3    2
```

Data Types, Classes, and Structures

- Here are some other matrix operations:

```
dim(A) # Dimension
```

```
## [1] 2 3
```

```
nrow(A) # Number of rows
```

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

```
ncol(A) # Number of columns
```

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

```
C = A%*%B # Multiplication (vectors also work)
C
```

```
##      [,1] [,2]
## [1,]   22   25
## [2,]   52   67
```

```
A*A # Inner product
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]   16   25   36
```

Data Types, Classes, and Structures

- You can also try:

```r
det(C) # Determinant
```

```
## [1] 174
```

```r
eigen(C) # Eigenvalues/ vectors
```

```
## eigen() decomposition
## $values
## [1] 87  2
##
## $vectors
##            [,1]       [,2]
## [1,] -0.3589791 -0.7808688
## [2,] -0.9333456  0.6246950
```

```r
solve(C) # Inverse
```

```
##            [,1]       [,2]
## [1,]  0.3850575 -0.1436782
## [2,] -0.2988506  0.1264368
```

Other Built-in Functions

- R has several built-in functions:

```r
mean(vec1)
```

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

```r
sd(vec1) # standard deviation
```

```
## [1] 3.162278
```

```r
sum(vec1)
```

```
## [1] 30
```

```r
prod(vec1)
```

```
## [1] 3840
```

```r
log(vec1) #ln
```

```
## [1] 0.6931472 1.3862944 1.7917595 2.0794415 2.3025851
```

```r
exp(vec1) # Exponential
```

```
## [1]     7.389056    54.598150   403.428793  2980.957987 22026.465795
```

```r
summary(log(vec1))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.6931  1.3863  1.7918  1.6506  2.0794  2.3026
```

Indexing

- You can access a particular element of a vector/matrix using idexes:

```
vec1[2] # second element of vec1
```

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

```
vec1[1:3] # elements 1 2 and 3 of vec1
```

```
## [1] 2 4 6
```

```
C[2,1] # row 2 column 1 of matrix C
```

```
## [1] 52
```

```
which(vec1 > 3)#position of elements in vec1 bigger than 3
```

```
## [1] 2 3 4 5
```

```
vec1[which(vec1 > 3)] # elements in vec1 bigger than 3
```

```
## [1]  4  6  8 10
```

Creating and combining strings

The command **paste** is used to combine strings

```
names<-paste("samp",1:4,sep="")
names
```

```
## [1] "samp1" "samp2" "samp3" "samp4"
```

```
namesWithSp<-paste("samp",1:4,sep=" ")
namesWithSp
```

```
## [1] "samp 1" "samp 2" "samp 3" "samp 4"
```

```
namesByMach<-paste("samp",1:4,sep="Mach")
namesByMach
```

```
## [1] "sampMach1" "sampMach2" "sampMach3" "sampMach4"
```

Getting Help

- You can se how a function works with ?functionName.
- This will give you access to the function documentation, which is standardized for all R functions.
- If you get an error or warning message, take a deep breath, read the error or warning, and try to figure out your error.
- If all else fails, Google the error.

Working with data

- The most usual way to read data in R is from csv files with **read.csv**.
- However, R has tools to read many types of data like xlsx, dta, etc.
- R also has its own way of storing data (.rda and .RData files).
- This is how you can read a csv file:

```r
data = read.csv("cars.csv", row.names = 1)
head(data) # shows the first 6 entries.
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

- The **rownames = 1** is to set the first column in the csv file as the names of the rows in the data. In this case we have the names of the cars.
- You can also write csv files.

```r
write.csv(data, file = "path/name.csv")
```

Working with data

- The R data extensions (.rda and .RData files) are very useful if you are dealing with large datasets.
- However, they will only work in R (other software may be compatible with adjustments).
- The advantage of these tools is that they compile the data in to a binary file, which is much faster to load an save.
- .rda is for a single object:

```
save(data,file = "path/data.rda")
```

- You can also save your entire workspace with .RData:

```
save.image("path/name.RData")
```

Working with data

- The **summary** function gives descriptive information on the data. The statistics will be calculated for each individual column.

```
summary(data)
```

- The **View** function opens the data on a sheet similar to excel, but in read only mode.

```
View(data)
```

- You can open the same sheet by clicking at the data object in the environment (top right part of RStudio).

Data Frames

- When we loaded the data, a new class of object was introduced:

```r
class(data)
```

```
## [1] "data.frame"
```

- Data Frames are a special class made to deal with data. Each column is a variable and each row is an observation.
- It falls in the heterogeneus classes. Each column can have a different type.

Data Frames

- Variable names play an important role in Data Frames. You can get a particular variable with the **$** element.

```r
head(data$cyl)
```

```
## [1] 6 6 4 6 8 6
```

```r
mean(data$cyl)
```

```
## [1] 6.1875
```

- You can also use indexes like in a vector/matrix:

```r
data[1,2]
```

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

```r
data$disp[3]
```

```
## [1] 108
```

Lists

- Lists are the most heterogeneous data in R. They can store anything.
- For example, you can have a list with a matrix, a vector, a string and even a function.

```
l1 = list(matrix = C, vector = vec1, string = "hello")
```

- Note that the elements in the list were named. This is a good practice that allow us to access the elements using their names.

```
l1$vector
```

```
## [1]  2  4  6  8 10
```

- You can also use numbers to index elements:

```
l1[[3]]
```

```
## [1] "hello"
```

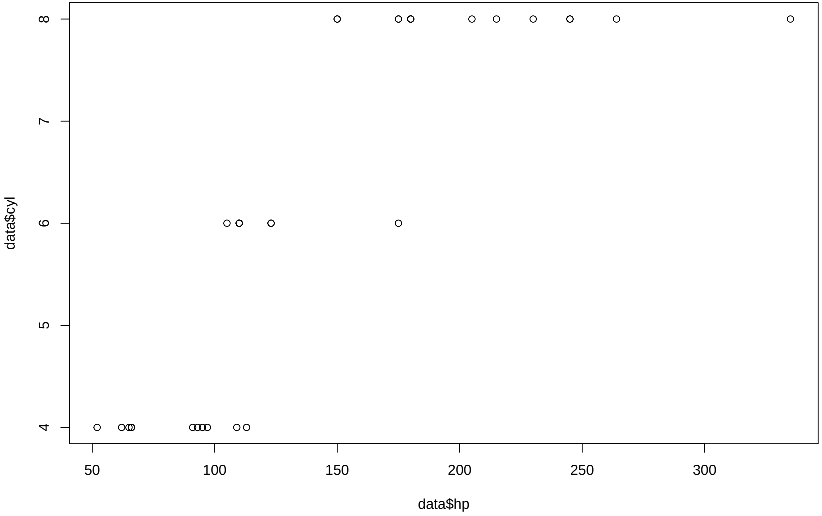
- Data Frames can be seen as a special case of lists where each element is a vector of the same size.
- Lists may seem weird now, but latter you will see that the output of many functions are lists.

Basic Graphical Tools

- Graphical tools in R can be the topico for a whole course.
- The next slides will show just some basic examples.
- R has severa built in plot tools. However, if you want to go deep you should study a package called ggplot.
- ggplot is so wide that people wrote entire books justs for it.

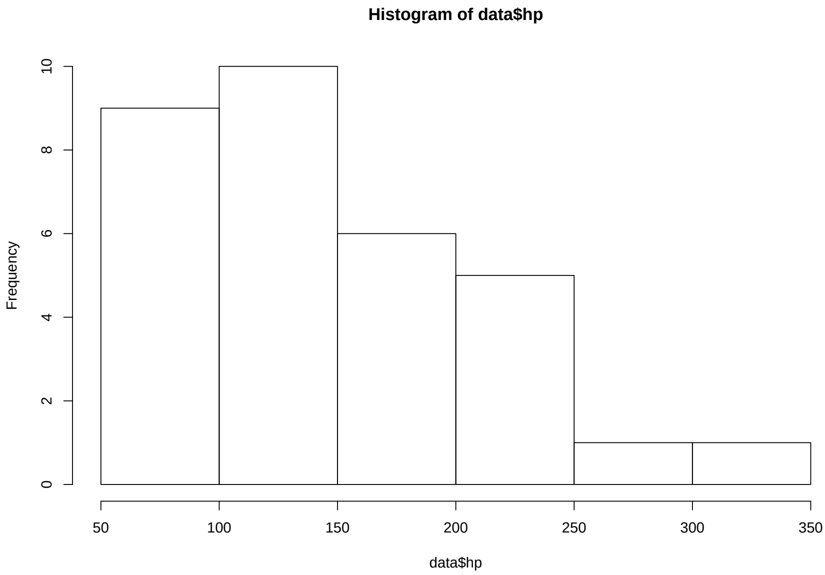
Basic Graphical Tools

```r
plot(data$hp,data$cyl)
```



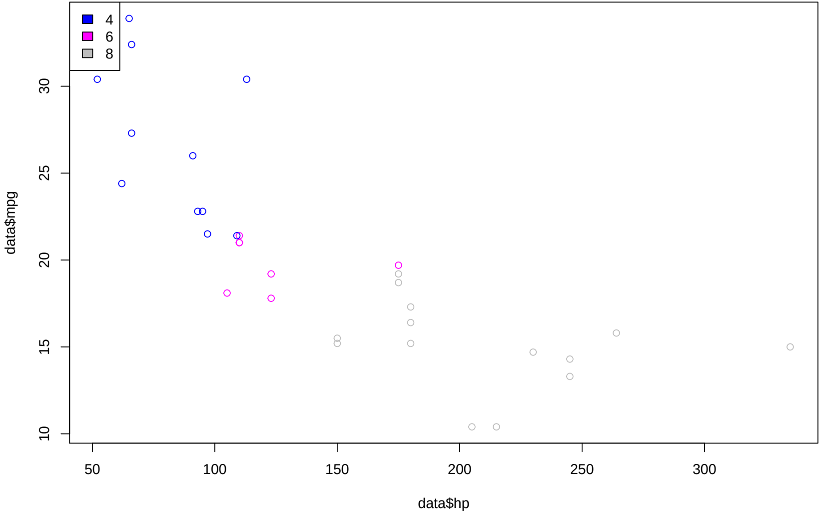
Basic Graphical Tools

```
hist(data$hp)
```



Basic Graphical Tools

```
plot(data$hp,data$mpg, col=data$cyl)
legend("topleft",fill=c(4,6,8),legend=levels(as.factor(data$cyl)))
```



Loop, If and Else

- Loops are codes for iterative processes. The most basic way to do it is using the **for** interface.
- A good example of something that can only be made in a loop is to create data that follow a random walk.
- A random walk is when we have

$$y_{t+1} = y_t + \varepsilon_{t+1}$$

where ε_t is an error term that we will sample from a normal distribution with mean 0 and variance 1.

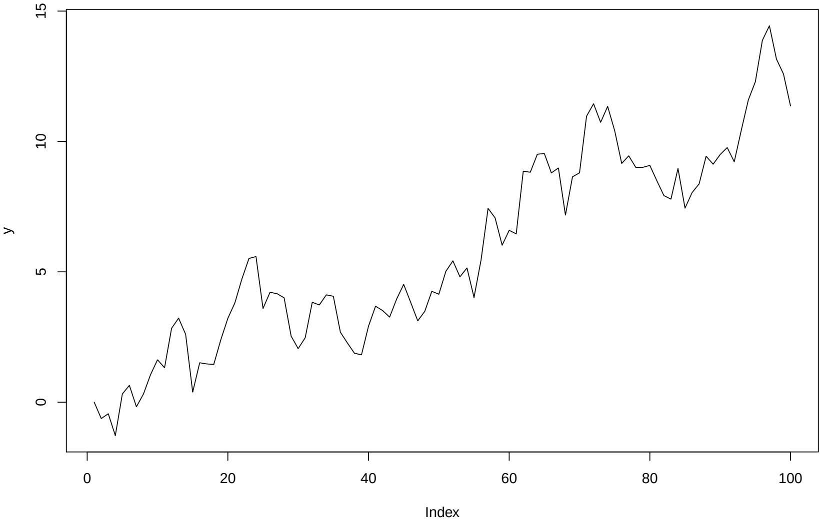
```
N = 100
y = rep(NA, N)
y[1] = 0

set.seed(1)
for(i in 1:(N-1)){
  y[i+1] = y[i] + rnorm(1)
}
head(y)
```

```
## [1]  0.0000000 -0.6264538 -0.4428105 -1.2784391  0.3168417  0.6463495
```

Loop, If and Else

```
plot(y,type = "l")
```



Loop, If and Else

- You can also write loops that will only stop when a criterion is met with the **while** interface.
- In the example below we will keep adding a random number between 0 and 1 to x until x becomes bigger or equal to 100.
- Then we will evaluate the variable i, which tells us how many iterations were needed.

```
x = 0
i = 0
set.seed(1)
while(x < 100){
  x = x + runif(1)
  i = i+1
}
i
```

```
## [1] 194
```

Loop, If and Else

- When one iteration depends on the previous it is hard to escape from the **for** and **while** loops.
- However, when the iterations are independent, we can use a set of functions from the **apply** family.
- For example, suppose we want to calculate the standard deviation of all columns in the car dataset;

```
# apply(data, dim, function, ... extra arguments)
apply(data,2,sd)
```

```
##        mpg         cyl        disp          hp        drat          wt
##   6.0269481   1.7859216 123.9386938  68.5628685   0.5346787   0.9784574
##        qsec          vs          am        gear        carb
##   1.7869432   0.5040161   0.4989909   0.7378041   1.6152000
```

Loop, If and Else

- If we are dealing with lists we can use the **lapply** function:

```
# A list with the tree matrices we created
matlist = list(A = A, B = B, C = C)
# This will get the first line of all matrices
lapply(matlist, function(x) x[1,] )
```

```
## $A
## [1] 1 2 3
##
## $B
## [1] 1 5
##
## $C
## [1] 22 25
```

- The function **sapply** is for the cases where the input is a list (or vector) and the output is a single element:

```
sapply(matlist, sum)
```

```
##   A   B   C
##  21  24 166
```

- **sapply** also works for vectors.

Loop, If and Else

- **if** statements are used when we want R to do someting once a certain condition is met.
- The sintax is **if (condition) {what to do}**
- We can use **if** to do the samething we did in the **while** code, but with a for:

```
x = 0
set.seed(1)
for(i in 1:100000){
  x = x + runif(1)
  if(x>100){
    break
  }
}
i
```

```
## [1] 194
```

Loop, If and Else

- It is possible to include an **else** statement with the **if**.
- In this case, if the condition is not met, R will run the code inside the **else**.

```
x = 1
if(x>2){
  z = x
}else{
  z = 0
}
z
```

```
## [1] 0
```

Loop, If and Else

- Finally, in certain situations we can simplify the code with the **ifelse** function.

```
x = 1:10
# ifelse(condition, if true, if false)
ifelse(x>5 , 5, x)
```

```
##  [1] 1 2 3 4 5 5 5 5 5 5
```

User Defined Functions (UDF)

A UDF is a function created by the user. It follows the structure

functionName = function(arguments){calculation}

For Example:

```
subtract<-function(x,y){
  result = x-y
  return(result)
}
```

- A function always stops when it reaches the **return** command.

User Defined Functions (UDF)

- Now we can use our function:

```
subtract(10,5)
```

```
## [1] 5
```

```
subtract(x = 20, y = 10)
```

```
## [1] 10
```

User Defined Functions (UDF)

- Now let's write a more interesting function.
- This function will create data from autoregressive models like:

$$y_{t+1} = \rho y_t + \varepsilon_{t+1}$$

- where $0 \leq \rho \leq 1$

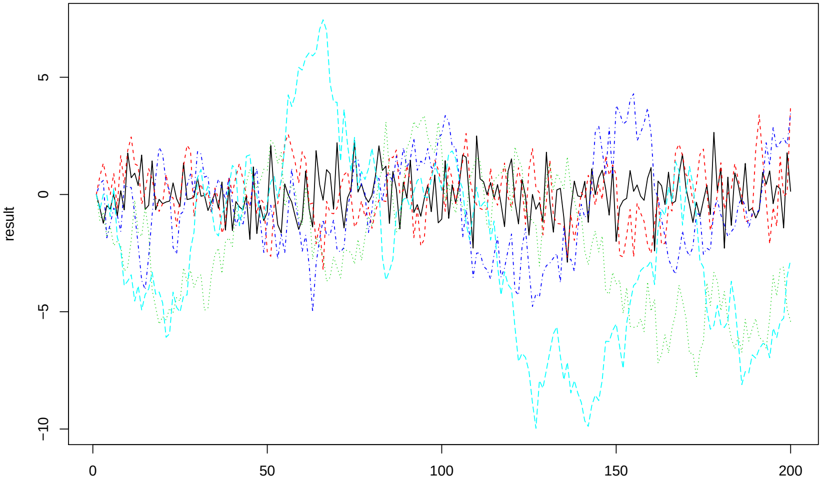
```
AR = function(N, rho){
  y = rep(NA, N)
  y[1] = 0
  for(i in 1:(N-1)){
    y[i+1] = rho*y[i] + rnorm(1)
  }
  return(y)
}
```

- We can use our UDF in an sapply loop with several values of ρ

```
rho_vector = c(0, 0.5, 0.95, 0.99, 1)
result = sapply(rho_vector, function(x) AR(200,x))
```

User Defined Functions (UDF)

```
matplot(result, type = "l")
```



Generation of Random Numbers

- Several codes so far had a **set.seed** before the functions we were running.
- This seed is used when we are dealing with random numbers. It allows some other user to replicate the exact same experiment.
- For example, we are going to sample 3 numbers from a vector with a seed set to 1.
- This is a random experiment, but if you use the sabe seed in your computer you should get the same result.

```r
vec6 = 1:10
# sample(x, size, replace = FALSE)
set.seed(1)
sample(vec6, 3, replace = FALSE)
```

```
## [1] 9 4 7
```

Generation of Random Numbers

- R has build in functions to generate data from several distributions like normal, uniform, Student t, etc.

```
# 10 numbers from a normal distribution with mean 0 and sd 1
rnorm(10,0,1)
```

```
##  [1]  1.329799263  1.272429321  0.414641434 -1.539950042 -0.928567035
##  [6] -0.294720447 -0.005767173  2.404653389  0.763593461 -0.799009249
```

```
# 10 numbers from a uniform distribution with min 0 max 1
runif(10,-1,1)
```

```
##  [1] -0.74888981 -0.46555866 -0.22777181 -0.97321933 -0.23522409  0.73938169
##  [7] -0.31930201 -0.03583977  0.19913165 -0.01291739
```

```
# 10 numbers from a t distribution 5 degrees of freedom
rt(10,5)
```

```
##  [1] -0.86409523 -1.50276529  0.85199410 -1.82436807 -0.06641194 -1.41288461
##  [7] -0.32612422  0.44183505  0.95343054 -0.33398807
```

Generation of Random Numbers

- Note that all functions that we just used started with a **r**, which comes from random.
- If you use a **d** you will calculate the density, **p** for the distribution and **q** for the quantile.

Packages

- Registered are packages are stores in the Comprehensive R Achive Network (CRAN).
- These packages can be installed with install.packages("pkg name")
- And they can be loaded with **library(pkg name)**.
- Once you load the package you have access to all its functions, data and documentation.

```
install.packages("glmnet")
library(glmnet)
?glmnet
```

The Tidyverse

- The Tidyverse is a set of dozens of packages, all compatible with each other, made for data treatment and analysis.
- It is the state of the art for data treatment. You can install and load it with:

```r
install.packages("tidyverse")
library(tidyverse)
```

The Tidyverse

- One of the most important features of Tidyverse is the pipe operator **%>%**.
- Consider the code:

```
log(exp(sin(2^2)))
```

```
## [1] -0.7568025
```

- Once we keep using functions inside functions the code may look very confusing and it is very easy to get lost in the parentheses.
- Also, it is important that you code can be read and understood by other humans, and not only machines.
- The same results can be obtained with the pipe operator:

```
2^2 %>% sin() %>% exp() %>% log()
```

```
## [1] -0.7568025
```

The Tidyverse

- The pipe operator can be used in the same way with data frames. It is very fast and you can perform a lot of operations in a single step.
- For example, we can select columns:

```
df1 = data %>% select(cyl,mpg,hp)
head(df1)
```

```
##                    cyl  mpg  hp
## Mazda RX4            6 21.0 110
## Mazda RX4 Wag        6 21.0 110
## Datsun 710           4 22.8  93
## Hornet 4 Drive       6 21.4 110
## Hornet Sportabout    8 18.7 175
## Valiant              6 18.1 105
```

The Tidyverse

- We can filter the data with some criterion:

```
df2 = df1 %>% filter(cyl == 4)
head(df2)
```

```
##   cyl  mpg hp
## 1   4 22.8 93
## 2   4 24.4 62
## 3   4 22.8 95
## 4   4 32.4 66
## 5   4 30.4 52
## 6   4 33.9 65
```

The Tidyverse

- It is also possible to create new variables:

```
df3 = df2 %>% mutate(newvar = hp/mpg)
head(df3)
```

```
##   cyl  mpg hp   newvar
## 1   4 22.8 93 4.078947
## 2   4 24.4 62 2.540984
## 3   4 22.8 95 4.166667
## 4   4 32.4 66 2.037037
## 5   4 30.4 52 1.710526
## 6   4 33.9 65 1.917404
```

Tidyverse

- The **group_by** and the **summarise** functions together are used to group the observations by some variable.

```
df4 = df1  %>% group_by(cyl) %>%
  summarise(hp = mean(hp), mpg = mean(mpg))
head(df4)
```

```
## # A tibble: 3 x 3
##     cyl    hp   mpg
##   <int> <dbl> <dbl>
## 1     4  82.6  26.7
## 2     6 122.   19.7
## 3     8 209.   15.1
```

Tidyverse

- Finally, we can combine as many operations as we want in one chain of pipe codes.

```
df5 = data %>% select(mpg, hp, cyl) %>%
  filter(cyl > 4) %>%
  group_by(cyl) %>%
  summarise(hp = mean(hp), mpg = mean(mpg)) %>%
  mutate(newvar = hp/mpg)

df5
```

```
## # A tibble: 2 x 4
##     cyl    hp   mpg newvar
##   <int> <dbl> <dbl>  <dbl>
## 1     6  122.  19.7   6.19
## 2     8  209.  15.1  13.9
```