

---

# PROGRAMMING IN R



# GENERAL LEARNING OBJECTIVE

Gain **hands-on experience with**, and a preliminary understanding of, the following elements of code in **both** R and Python:

- Variables
- Data Structures
- Operators
- Statements and Expressions
- Blocks (and Scope)
- Functions
- Logical (Control) Flow
- Libraries/Packages/Modules
- Inputs/Outputs
- Interpreters/Compilers

# CONTENTS

1. R Basics
2. A Comparison of R and Python
3. Resources for Programming
4. Exercises for Hands-On Learning

# PERSONAL LEARNING OBJECTIVES

Depending on where you are starting from, reaching the learning objectives for this session within the time frame of the session itself may be ambitious.

**Pick a learning objective for yourself that you think is reasonable** given your starting point.

You will have time to continue working on the exercises over the next weeks.



# R BASICS

PROGRAMMING IN R

# PROGRAMMING LANGUAGE REQUIREMENTS (DATA ANALYSIS)

## Data-Centric:

- data analysts seek to **uncover some information** present in the data which is not obviously found during a first pass: **focus on analysis, not on programming**.
- The language should allow high-level concepts and questions related to data analysis to be expressed **naturally**.

## Trustworthy:

- Recipients of analyses are usually **far removed** from the details of implementation, which is why the language used for analysis must be **trustworthy**.

# PERFORMANCE

Data analysis is centered on the analyst figuring out **what questions to ask** and what transformations or algorithms **may provide insights**:

- the computer spends most of its time waiting for user input – it is more valuable to have a language that promotes **human/analyst efficiency** at this stage.

Once a **pipeline to answer the question** has been established, the algorithm can be run again and again every time the data set is updated:

- data processing, not data analysis anymore – the non-creative process is **mechanical** and can be **automated**. This is when computer performance becomes paramount.

# ABOUT R

R is an open source implementation of the (commercial) S language:

- designed for **statistical** and **data analysis**
- supports **functional** and **object-oriented** styles of programming
- **data frame** is a built-in type
- missing values are built-in (WARNING: read the documentation on NAs!!!)
- model formulas are first-class objects
- contains **advanced statistical routines** (external packages that must be loaded)
- state of the art graphics capabilities (ggplot2)
- has dialects (the **tidyverse**, in particular, is quite popular currently)

# ABOUT R

The functional style is **not enforced**, although it is encouraged and natural in R.

The main unit of programming is the **function**.

Data frames are the **natural containers** for data (also, tibbles)

- “tables” whose rows are **observations** and columns are **variables**
- this is a **natural data model** (DBMS, spreadsheets) ...
- ... but there are restrictions when using **tidy data** (more on this later)

## R REFERENCES & BOOKS

R Project: <http://www.r-project.org> (installation)

R Studio: <http://www.rstudio.com> (development environment)

R Cookbook: <http://www.cookbook-r.com>

*The R Inferno*: <http://bit.ly/1mpZabc>

*Advanced R*, by H. Wickham: <https://adv-r.hadley.nz>

*R for Data Science*, by G. Grolemund and H. Wickham: <https://r4ds.had.co.nz>

# BASIC TYPES

**Vector:** logical, integer, double, double complex, string

- each has a missing value literal: NA
- NULL is not the same thing as NA
- scalars are vectors of length 1

**Structured:** list, factor, data frame, array, matrix, etc.

# BASIC TYPES

**Factor:** represents categorical variable

- similar to vector of strings
- stored as an integer vector

**Data Frame:** represents observations of statistical variables

- behaves as a list of vectors
- also behaves as a matrix



# BASIC OPERATORS

Usual arithmetic, comparison, logical operators are elementwise

For matrix multiplication use `\%*\%`

There are 5 assignment operators!!!

- `=`, `<-` (best practice), `->`, `<<-`, `->>`

There are 3 indexing operators:

- `[[_]]` for access to a single element by index or name
- `[_]` for access to multiple elements by index or name
- `$` for access to single element by name

# BASIC FLOW CONTROL

```
if(cond){expr}
```

```
if(cond){expr 1 else {expr2}}
```

```
for(var in seq) {expr}
```

```
while(cond) {expr}
```

```
repeat {expr}
```

```
break
```

```
next
```

Avoid loops if at all possible (they can be VERY SLOW...);  
**vectorize** instead!

# FUNCTIONS

Functions are first-class objects

Function arguments passing can be complicated! (less so with the tidyverse)

- parameters are assigned by position or name
- use = to assign parameter by name
- lazy evaluation (symbol, function only evaluated when needed)

**Hint:** try using one of the `*apply` functions instead of a loop whenever possible

# PRACTICE, PRACTICE, PRACTICE

In what follows we show how to do ... well, more data stuff using R.

The selection of problems is still not intended to be complete, but it provides a gloss of the myriad ways to approach data analysis with R.

Some of it overlaps with the other **Data Analysis Short Course** notebooks.

As always, do not hesitate to consult online resources for more examples (StackOverflow, R-bloggers, etc.).

# A SIMPLE EXAMPLE – CARS

We start by loading one of the standard pedagogical datasets used with R.

```
> data(cars)
```

It's a data frame with two variables, **speed** and **dist**.

```
> str(cars)
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...

> summary(cars)
##           speed           dist
##  Min.      : 4.0      Min.      : 2.00
##  1st Qu.:12.0      1st Qu.: 26.00
##  Median :15.0      Median : 36.00
##  Mean   :15.4      Mean   : 42.98
##  3rd Qu.:19.0      3rd Qu.: 56.00
##  Max.   :25.0      Max.   :120.00
```

## A SIMPLE EXAMPLE – CARS

We can compute the mean of each variable using the pre-built function **colMeans()**.

```
> colMeans(cars)
## speed  dist
## 15.40  42.98
```

But there is no such function to compute the minimum/maximum of each variable. instead, we use loops.

```
# min
> for(i in 1:2){
  print(min(cars[,i]))
}
## [1] 4
## [1] 2
```

## A SIMPLE EXAMPLE – CARS

```
# max
> for(i in 1:2){
  print(max(cars[,i]))
}

## [1] 25
## [1] 120
```

The use of loops was justified because the dataset is **small**, but it would be useful to know how to do the computation using one of the **\*apply** functions.

```
> sapply(cars, min)
## speed dist
##      4    2

> sapply(cars, max)
## speed dist
##     25   120
```

## A SIMPLE EXAMPLE – CARS

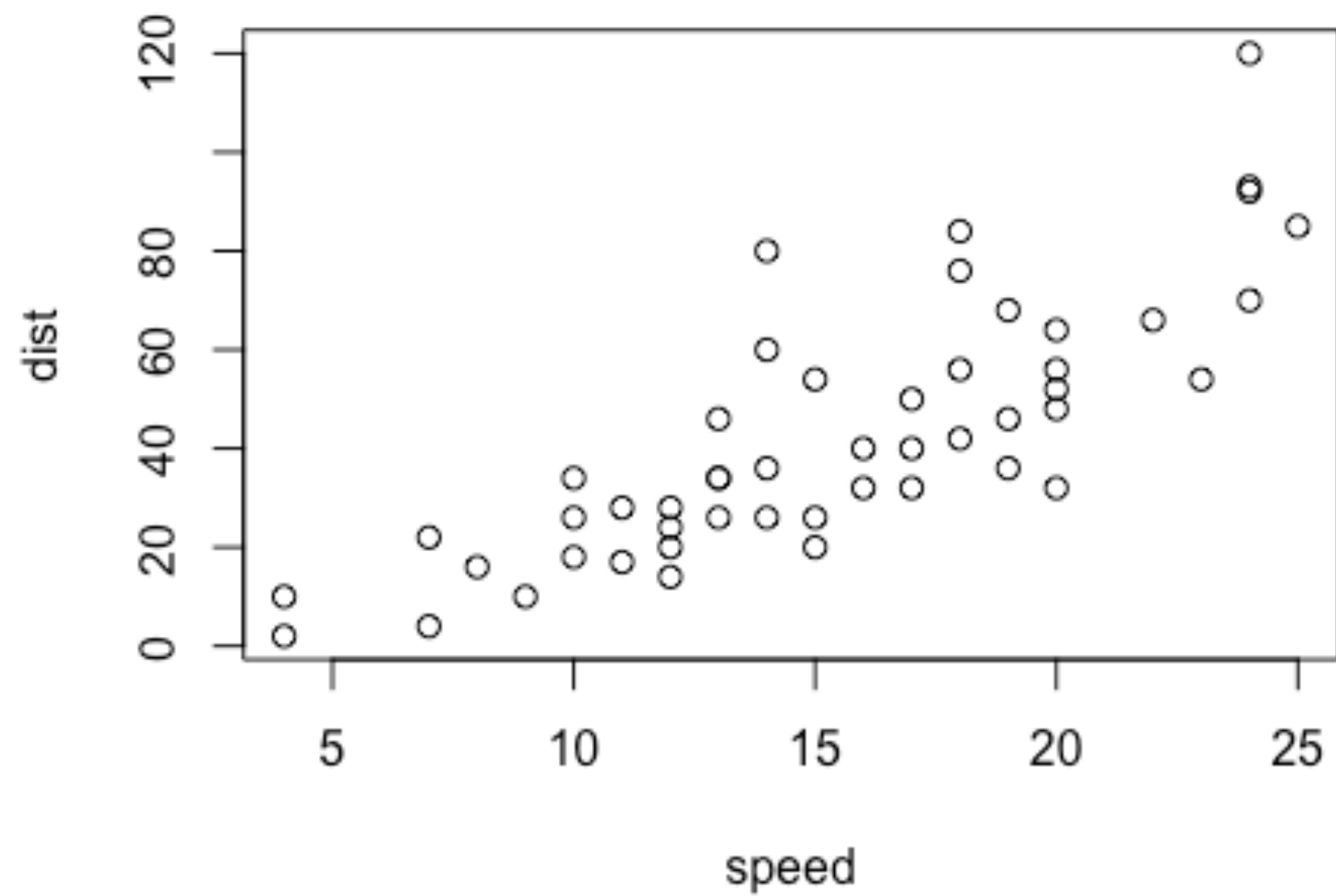
```
> sapply(cars, mean)
## speed  dist
## 15.40  42.98
```

Note that in each of the three cases, the result is a data frame.

A dataset with 2 variables is easy to display.

```
> plot(cars)
```





## A SIMPLE EXAMPLE – CARS

We can see that the relationship between speed and distance has a strong linear component, which we can identify by running a linear regression (implemented in the function `lm`):

```
> reg <- lm(dist ~ speed, data=cars)
```

In this assignment, `reg` is the **object** (or model) obtained when running `lm()` on the **`cars`** dataset.

We can summarize it and see what its attributes are.

# A SIMPLE EXAMPLE – CARS

```
> summary(reg)
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed        3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

## A SIMPLE EXAMPLE – CARS

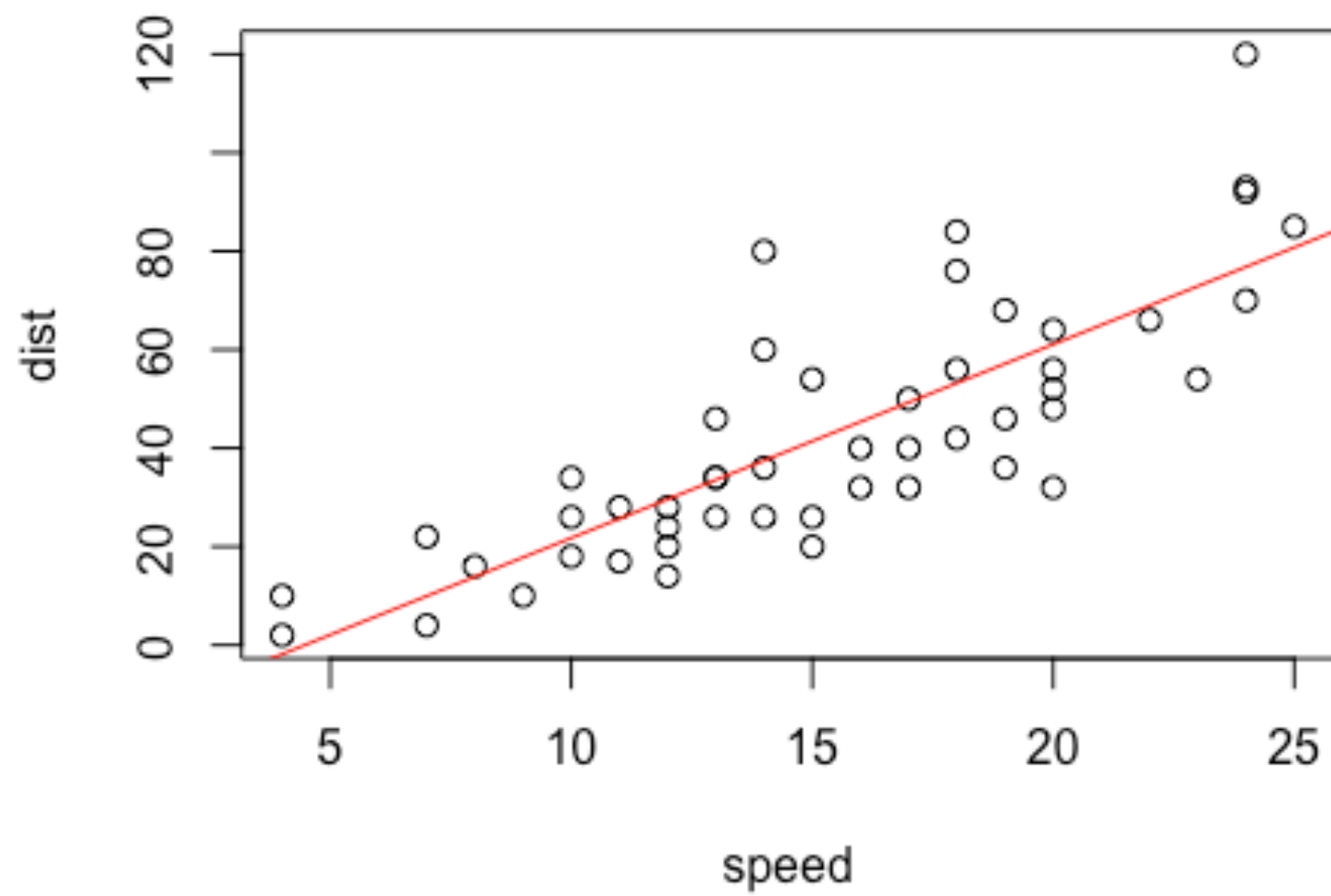
```
> attributes(reg)
## $names
## [1] "coefficients" "residuals"      "effects"        "rank"
## [5] "fitted.values" "assign"         "qr"            "df.residual"
## [9] "xlevels"      "call"          "terms"         "model"
## $class
## [1] "lm"
```

We can extract the coefficients of the linear model as follows:

```
> reg$coefficients
## (Intercept)      speed
## -17.579095      3.932409
```

and use them to plot the line of best fit over the display:

```
> plot(cars)
> abline(reg$coefficients, col="red")
```



# WORKING WITH MATRICES

R's **matrix notation** is not super intuitive, as we will see presently.

Let's generate 2 matrices that we will use in the following code blocks (the brackets around the code mean that the assignment is followed by an object display).

```
> (M <- matrix(1:12, nrow=3, ncol=4))
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

> (V <- matrix( runif(4), 4, 1))
##      [,1]
## [1,] 0.02875853
## [2,] 0.07807830
## [3,] 0.31080912
## [4,] 0.54691573
```

# WORKING WITH MATRICES

When `ncol` and `nrow` are not specified, the default parameter ordering takes over. Since  $M$  is  $3 \times 4$  and  $V$  is  $4 \times 1$ , the product  $MV$  exists (and is  $3 \times 1$ ).

```
# use %*% to multiply matrices
> M %*% V
##           [,1]
## [1,] 7.985893
## [2,] 8.950455
## [3,] 9.915016
```

The product  $VM$  does not exist, however, as the dimensions are not compatible.

```
# uncomment the following line to see the error message
> # V %*% M
```

The choice of `%*%` for matrix multiplication is not ... inspired; most languages use `*`.

# WORKING WITH MATRICES

What DOES \* do in R?

```
> M * 2
##      [,1] [,2] [,3] [,4]
## [1,]    2    8   14   20
## [2,]    4   10   16   22
## [3,]    6   12   18   24
```

Hah! Multiplication by a scalar. Good to know.

One of R's most nefarious habit is that it is not always compatible with reasonable mathematical notation. What do you think the following line of code does?

```
> M * c(2, -2)
```



# WORKING WITH MATRICES

```
> M * c(2, -2)
##      [,1] [,2] [,3] [,4]
## [1,]    2  -8  14 -20
## [2,]   -4  10 -16  22
## [3,]    6 -12  18 -24
```

Apparently, it cycles through the arguments?!?

It's hard to imagine why this construction should yield a result without breaking down, and yet it does. Beware, then.

(It's probably a good idea to verify that code does what it's meant to along the way.)

# WORKING WITH MATRICES

Other familiar operations (like the transpose) are easy to compute:

```
> t(M)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

is indeed the  $4 \times 3$  transpose of  $M$ .

# WORKING WITH MATRICES

**rbind** and **cbind** are used to bind rows and columns, respectively.

```
> cbind(t(M), V)
##           [,1] [,2] [,3]           [,4]
## [1,]         1     2     3 0.02875853
## [2,]         4     5     6 0.07807830
## [3,]         7     8     9 0.31080912
## [4,]        10    11    12 0.54691573

> rbind(M,t(V))
##           [,1]           [,2]           [,3]           [,4]
## [1,] 1.00000000 4.00000000 7.00000000 10.00000000
## [2,] 2.00000000 5.00000000 8.00000000 11.00000000
## [3,] 3.00000000 6.00000000 9.00000000 12.00000000
## [4,] 0.02875853 0.0780783 0.3108091 0.5469157

# but this wont work because of dimension incompatibility; uncomment to test
> # cbind(M, V)
```

# WORKING WITH STRINGS

What is usually called a **string** in other programming languages is a **character object** in R.

Character vectors are created by using double quotes (" , preferred) or single quotes (' , acceptable).

```
> "Come on, everybody!"  
## [1] "Come on, everybody!"
```

In R, strings are scalar values, not vectors of characters.

```
> length("Come on, everybody!")  
## [1] 1
```

# WORKING WITH STRINGS

The combining function **c()** creates a vector of strings.

```
> c("Come", "on", ", " , "everybody", "!")  
## [1] "Come"      "on"      ", "      "everybody"      "!"
```

We use **paste** or **paste0** to **concatenate** strings:

```
> paste("Come", "on", ", " , "everybody", "!")  
## [1] "Come on , everybody !"  
  
> paste("Come", "on", ", " , "everybody", "!", sep=" ")  
## [1] "Come on , everybody !"
```

The function **strsplit()** does the opposite:

```
> strsplit("Come on, everybody!", ", ")  
## [[1]]  
## [1] "Come on"      "everybody!"
```

# R NOTEBOOKS & CHEATSHEETS

These examples, as well as numerous others, can be found in the R Notebooks

- R Basics
- More Data Stuff in R

There is no substitute for writing code, but these examples could be a good starting point for many analysts.

Take the time to run those examples at your console to get a sense for R in action.

RStudio publishes various cheatsheets, which can come in handy (see next slides), but remember that these cheatsheets may become obsolete with new R releases.

# Base R

## Cheat Sheet

### Getting Help

Accessing the help files

#### ?mean

Get help of a particular function.

**help.search('weighted mean')**

Search the help files for a word or phrase.

**help(package = 'dplyr')**

Find help for a package.

More about an object

**str(iris)**

Get a summary of an object's structure.

**class(iris)**

Find the class an object belongs to.

### Using Packages

**install.packages('dplyr')**

Download and install a package from CRAN.

**library(dplyr)**

Load the package into the session, making all its functions available to use.

**dplyr::select**

Use a particular function from a package.

**data(iris)**

Load a built-in dataset into the environment.

### Working Directory

**getwd()**

Find the current working directory (where inputs are found and outputs are sent).

**setwd('C://file/path')**

Change the current working directory.

**Use projects in RStudio to set the working directory to the folder you are working in.**

### Vectors

#### Creating Vectors

c(2, 4, 6)	2 4 6	Join elements into a vector
2:6	2 3 4 5 6	An integer sequence
seq(2, 3, by=0.5)	2.0 2.5 3.0	A complex sequence
rep(1:2, times=3)	1 2 1 2 1 2	Repeat a vector
rep(1:2, each=3)	1 1 1 2 2 2	Repeat elements of a vector

#### Vector Functions

<b>sort(x)</b>	<b>rev(x)</b>
Return x sorted.	Return x reversed.
<b>table(x)</b>	<b>unique(x)</b>
See counts of values.	See unique values.

#### Selecting Vector Elements

##### By Position

<b>x[4]</b>	The fourth element.
<b>x[-4]</b>	All but the fourth.
<b>x[2:4]</b>	Elements two to four.
<b>x[-(2:4)]</b>	All elements except two to four.
<b>x[c(1, 5)]</b>	Elements one and five.

##### By Value

<b>x[x == 10]</b>	Elements which are equal to 10.
<b>x[x &lt; 0]</b>	All elements less than zero.
<b>x[x %in% c(1, 2, 5)]</b>	Elements in the set 1, 2, 5.

##### Named Vectors

<b>x['apple']</b>	Element with name 'apple'.
-------------------	----------------------------

### Programming

#### For Loop

```
for (variable in sequence){  
  Do something  
}
```

##### Example

```
for (i in 1:4){  
  j <- i + 10  
  print(j)  
}
```

#### If Statements

```
if (condition){  
  Do something  
} else {  
  Do something different  
}
```

##### Example

```
if (i > 3){  
  print('Yes')  
} else {  
  print('No')  
}
```

#### While Loop

```
while (condition){  
  Do something  
}
```

##### Example

```
while (i < 5){  
  print(i)  
  i <- i + 1  
}
```

#### Functions

```
function_name <- function(var){  
  Do something  
  return(new_variable)  
}
```

##### Example

```
square <- function(x){  
  squared <- x*x  
  return(squared)  
}
```

### Reading and Writing Data

Also see the **readr** package.

Input	Output	Description
df <- read.table('file.txt')	write.table(df, 'file.txt')	Read and write a delimited text file.
df <- read.csv('file.csv')	write.csv(df, 'file.csv')	Read and write a comma separated value file. This is a special case of read.table/write.table.
load('file.RData')	save(df, file = 'file.Rdata')	Read and write an R data file, a file type special for R.

#### Conditions

a == b	Are equal	a > b	Greater than	a >= b	Greater than or equal to	is.na(a)	Is missing
a != b	Not equal	a < b	Less than	a <= b	Less than or equal to	is.null(a)	Is null

## Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

<code>as.logical</code>	TRUE, FALSE, TRUE	Boolean values (TRUE or FALSE).
<code>as.numeric</code>	1, 0, 1	Integers or floating point numbers.
<code>as.character</code>	'1', '0', '1'	Character strings. Generally preferred to factors.
<code>as.factor</code>	'1', '0', '1', levels: '1', '0'	Character strings with preset levels. Needed for some statistical models.

## Maths Functions

<code>log(x)</code>	Natural log.	<code>sum(x)</code>	Sum.
<code>exp(x)</code>	Exponential.	<code>mean(x)</code>	Mean.
<code>max(x)</code>	Largest element.	<code>median(x)</code>	Median.
<code>min(x)</code>	Smallest element.	<code>quantile(x)</code>	Percentage quantiles.
<code>round(x, n)</code>	Round to n decimal places.	<code>rank(x)</code>	Rank of elements.
<code>signif(x, n)</code>	Round to n significant figures.	<code>var(x)</code>	The variance.
<code>cor(x, y)</code>	Correlation.	<code>sd(x)</code>	The standard deviation.

## Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```




## The Environment

<code>ls()</code>	List all variables in the environment.
<code>rm(x)</code>	Remove x from the environment.
<code>rm(list = ls())</code>	Remove all variables from the environment.

**You can use the environment panel in RStudio to browse variables in your environment.**

## Matrices

```
m <- matrix(x, nrow = 3, ncol = 3)
Create a matrix from x.
```

 <code>m[2, ]</code>	- Select a row	<code>t(m)</code>	Transpose
 <code>m[, 1]</code>	- Select a column	<code>m %*% n</code>	Matrix Multiplication
 <code>m[2, 3]</code>	- Select an element	<code>solve(m, n)</code>	Find x in: $m \cdot x = n$

## Lists

```
l <- list(x = 1:5, y = c('a', 'b'))
A list is a collection of elements which can be of different types.
```

<code>l[[2]]</code>	<code>l[1]</code>	<code>l\$x</code>	<code>l['y']</code>
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.




Also see the **dplyr** package.

## Data Frames



```
df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))
A special case of a list where all elements are the same length.
```

x	y
1	a
2	b
3	c

### Matrix subsetting

<code>df[, 2]</code>	
<code>df[2, ]</code>	
<code>df[2, 2]</code>	

### List subsetting

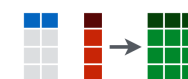
<code>df\$x</code>		<code>df[[2]]</code>	
<i>Understanding a data frame</i>			
<code>View(df)</code>	See the full data frame.		
<code>head(df)</code>	See the first 6 rows.		

`nrow(df)`  
Number of rows.

`ncol(df)`  
Number of columns.

`dim(df)`  
Number of columns and rows.

`cbind` - Bind columns.



`rbind` - Bind rows.



## Strings

Also see the **stringr** package.

<code>paste(x, y, sep = ' ')</code>	Join multiple vectors together.
<code>paste(x, collapse = ' ')</code>	Join elements of a vector together.
<code>grep(pattern, x)</code>	Find regular expression matches in x.
<code>gsub(pattern, replace, x)</code>	Replace matches in x with a string.
<code>toupper(x)</code>	Convert to uppercase.
<code>tolower(x)</code>	Convert to lowercase.
<code>nchar(x)</code>	Number of characters in a string.

## Factors

<code>factor(x)</code>	<code>cut(x, breaks = 4)</code>
Turn a vector into a factor. Can set the levels of the factor and the order.	Turn a numeric vector into a factor by 'cutting' into sections.

## Statistics




<code>lm(y ~ x, data=df)</code> Linear model.	<code>t.test(x, y)</code> Perform a t-test for difference between means.	<code>prop.test</code> Test for a difference between proportions.
<code>glm(y ~ x, data=df)</code> Generalised linear model.	<code>pairwise.t.test</code> Perform a t-test for paired data.	<code>aov</code> Analysis of variance.
<code>summary</code> Get more detailed information out a model.		

## Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	<code>rnorm</code>	<code>dnorm</code>	<code>pnorm</code>	<code>qnorm</code>
Poisson	<code>rpois</code>	<code>dpois</code>	<code>ppois</code>	<code>qpois</code>
Binomial	<code>rbinom</code>	<code>dbinom</code>	<code>pbinom</code>	<code>qbinom</code>
Uniform	<code>runif</code>	<code>dunif</code>	<code>punif</code>	<code>qunif</code>

## Plotting

Also see the **ggplot2** package.

 <code>plot(x)</code> Values of x in order.	 <code>plot(x, y)</code> Values of x against y.	 <code>hist(x)</code> Histogram of x.
---	---	---

## Dates

See the **lubridate** package.



# Advanced R

## Cheat Sheet

Created by: Arianne Colton and Sean Chen

### Environment Basics

Environment – **Data structure** (with two components below) that powers lexical scoping

```
Create environment: env1<-new.env()
```

1. **Named list** (“Bag of names”) – each name points to an object stored elsewhere in memory.

If an object has no names pointing to it, it gets automatically deleted by the garbage collector.

- Access with: **ls('env1')**
- 2. **Parent environment** – used to implement lexical scoping. If a name is not found in an environment, then R will look in its parent (and so on).
- Access with: **parent.env('env1')**

### Four special environments

1. **Empty environment** – ultimate ancestor of all environments
  - Parent: none
  - Access with: **emptyenv()**
2. **Base environment** - environment of the base package
  - Parent: empty environment
  - Access with: **baseenv()**
3. **Global environment** – the interactive workspace that you normally work in
  - Parent: environment of last attached package
  - Access with: **globalenv()**
4. **Current environment** – environment that R is currently working in (may be any of the above and others)
  - Parent: empty environment
  - Access with: **environment()**

## Environments

### Search Path

**Search path** – mechanism to look up objects, particularly functions.

- Access with : **search()** – lists all parents of the global environment (see Figure 1)
- Access any environment on the search path:  
**as.environment('package:base')**

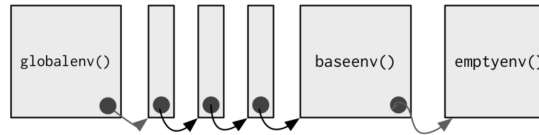


Figure 1 – The Search Path

- Mechanism : always start the search from global environment, then inside the latest attached package environment.
- New package loading with **library()/require()** : new package is attached right after global environment. (See Figure 2)
- Name conflict in two different package : functions with the same name, latest package function will get called.

```
search() :  
'GlobalEnv' ... 'Autoloads' 'package:base'  
  
library(reshape2); search()  
'GlobalEnv' 'package:reshape2' ... 'Autoloads' 'package:base'  
  
NOTE: Autoloads : special environment used for saving memory by  
only loading package objects (like big datasets) when needed
```

Figure 2 – Package Attachment

### Binding Names to Values

**Assignment** – act of binding (or rebinding) a name to a value in an environment.

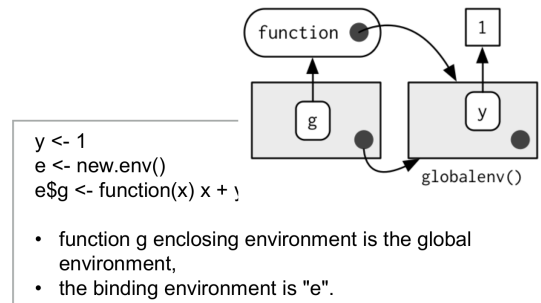
1. **<-** (Regular assignment arrow) – always creates a variable in the current environment
2. **<<-** (Deep assignment arrow) - modifies an existing variable found by walking up the parent environments

**Warning:** If **<<-** doesn't find an existing variable, it will create one in the global environment.

### Function Environments

1. **Enclosing environment** - an environment where the function is created. It determines how function finds value.
  - Enclosing environment never changes, even if the function is moved to a different environment.
  - Access with: **environment('func1')**
2. **Binding environment** - all environments that the function has a binding to. It determines how we find the function.
  - Access with: **pryr::where('func1')**

**Example** (for enclosing and binding environment):



3. **Execution environment** - new created environments to host a function call execution.
  - Two parents :
    - I. Enclosing environment of the function
    - II. Calling environment of the function
  - Execution environment is thrown away once the function has completed.
4. **Calling environment** - environments where the function was called.
  - Access with: **parent.frame('func1')**
  - Dynamic scoping :
    - About : look up variables in the calling environment rather than in the enclosing environment
    - Usage : most useful for developing functions that aid interactive data analysis

## Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

**Note:** R has no 0-dimensional or scalar types. Individual numbers or strings, are actually vectors of length one, NOT scalars.

Human readable description of any R data structure :

```
str(variable)
```

Every **Object** has a mode and a class

1. **Mode:** represents how an object is stored in memory
  - 'type' of the object from R's point of view
  - Access with: **typeof()**
2. **Class:** represents the object's abstract type
  - 'type' of the object from R's object-oriented programming point of view
  - Access with: **class()**

	typeof()	class()
strings or vector of strings	character	character
numbers or vector of numbers	numeric	numeric
list	list	list
data.frame	list	data.frame

## Factors

1. Factors are built on top of integer vectors using two attributes :

```
class(x) -> 'factor'
```

```
levels(x) # defines the set of allowed values
```

2. Useful when you know the possible values a variable may take, even if you don't see all values in a given dataset.

### Warning on Factor Usage:

1. Factors look and often behave like character vectors, they are actually integers. Be careful when treating them like strings.
2. Most data loading functions automatically convert character vectors to factors. (Use argument `stringAsFactors = FALSE` to suppress this behavior)

## Object Oriented (OO) Field Guide

### Object Oriented Systems

R has three object oriented systems :

1. **S3** is a very casual system. It has no formal definition of classes. It implements generic function OO.
  - **Generic-function OO** - a special type of function called a generic function decides which method to call.

Example:	drawRect(canvas, 'blue')
Language:	R

- **Message-passing OO** - messages (methods) are sent to objects and the object determines which function to call.

Example:	canvas.drawRect('blue')
Language:	Java, C++, and C#

2. **S4** works similarly to S3, but is more formal. Two major differences to S3 :
  - **Formal class definitions** - describe the representation and inheritance for each class, and has special helper functions for defining generics and methods.
  - **Multiple dispatch** - generic functions can pick methods based on the class of any number of arguments, not just one.
3. **Reference classes** are very different from S3 and S4:
  - **Implements message-passing OO** - methods belong to classes, not functions.
  - **Notation** - \$ is used to separate objects and methods, so method calls look like **canvas\$drawRect('blue')**.

### S3

#### 1. About S3 :

- R's first and simplest OO system
- Only OO system used in the base and stats package
- Methods belong to functions, not to objects or classes.

#### 2. Notation :

- **generic.class()**

mean.Date()	Date method for the generic - mean()
-------------	--------------------------------------

#### 3. Useful 'Generic' Operations

- Get all methods that belong to the 'mean' generic:
  - **Methods('mean')**
- List all generics that have a method for the 'Date' class :
  - **methods(class = 'Date')**

4. **S3 objects** are usually built on top of lists, or atomic vectors with attributes.

- Factor and data frame are S3 class
- Useful operations:

Check if object is an S3 object	is.object(x) & !isS4(x) or pryr::otype()
Check if object inherits from a specific class	inherits(x, 'classname')
Determine class of any object	<b>class(x)</b>

### Base Type (C Structure)

R base types - the internal C-level types that underlie the above OO systems.

- **Includes** : atomic vectors, list, functions, environments, etc.
- **Useful operation** : Determine if an object is a base type (Not S3, S4 or RC) **is.object(x)** returns FALSE

- **Internal representation** : C structure (or struct) that includes :

- Contents of the object
- Memory Management Information
- Type
  - Access with: **typeof()**

# Functions

## Function Basics

**Functions** – objects in their own right  
All R functions have three parts:

body()	code inside the function
formals()	list of arguments which controls how you can call the function
environment()	“map” of the location of the function’s variables (see “Enclosing Environment”)

Every operation is a function call

- +, for, if, [, \$, { ...
- x + y is the same as +(x, y)

**Note:** the backtick (`), lets you refer to functions or variables that have otherwise reserved or illegal names.

## Lexical Scoping

### What is Lexical Scoping?

- Looks up value of a symbol. (see “Enclosing Environment”)
- **findGlobals()** - lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+' 'x'
```

```
environment(f) <- emptyenv()
f()
# error in f(): could not find function “+”
```

- R relies on lexical scoping to find everything, even the + operator.

## Function Arguments

**Arguments** – passed by reference and copied on modify

1. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.
2. Check if an argument was supplied : **missing()**

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

3. Lazy evaluation – since x is not used **stop("This is an error!")** never get evaluated.

```
f <- function(x) {
  10
}
f(stop('This is an error!')) -> 10
```

4. Force evaluation

```
f <- function(x) {
  force(x)
  10
}
```

5. Default arguments evaluation

```
f <- function(x = ls()) {
  a <- 1
  x
}
```

f() -> 'a' 'x'	ls() evaluated inside f
f(ls())	ls() evaluated in global environment

## Return Values

- **Last expression evaluated or explicit return().**  
Only use explicit return() when returning early.
- **Return ONLY single object.**  
Workaround is to return a list containing any number of objects.
- **Invisible return object value** - not printed out by default when you call the function.

```
f1 <- function() invisible(1)
```

## Primitive Functions

### What are Primitive Functions?

1. Call C code directly with **.Primitive()** and contain no R code

```
print(sum) :
> function (..., na.rm = FALSE) .Primitive('sum')
```

2. **formals()**, **body()**, and **environment()** are all NULL
3. Only found in base package
4. More efficient since they operate at a low level

## Infix Functions

### What are Infix Functions?

1. Function name comes in between its arguments, like + or –
2. All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

3. Useful way of providing a default value in case the output of another function is NULL:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

## Replacement Functions

### What are Replacement Functions?

1. Act like they modify their arguments in place, and have the special name xxx <-
2. Actually create a modified copy. Can use **pryr::address()** to find the memory address of the underlying object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```

# Subsetting

**Subsetting returns a copy of the original data, NOT copy-on modified**

## Simplifying vs. Preserving Subsetting

### 1. Simplifying subsetting

- Returns the **simplest** possible data structure that can represent the output

### 2. Preserving subsetting

- Keeps the structure of the output the **same** as the input.
- When you use `drop = FALSE`, it's preserving

	Simplifying*	Preserving
Vector	<code>x[[1]]</code>	<code>x[1]</code>
List	<code>x[[1]]</code>	<code>x[1]</code>
Factor	<code>x[1:4, drop = T]</code>	<code>x[1:4]</code>
Array	<code>x[1, ]</code> or <code>x[, 1]</code>	<code>x[1, , drop = F]</code> or <code>x[, 1, drop = F]</code>
Data frame	<code>x[, 1]</code> or <code>x[[1]]</code>	<code>x[, 1, drop = F]</code> or <code>x[[1]]</code>

Simplifying behavior varies slightly between different data types:

#### 1. Atomic Vector

- `x[[1]]` is the same as `x[1]`

#### 2. List

- `[ ]` always returns a list
- Use `[[ ]]` to get list contents, this returns a single value piece out of a list

#### 3. Factor

- Drops any unused levels but it remains a factor class

#### 4. Matrix or Array

- If any of the dimensions has length 1, that dimension is dropped

#### 5. Data Frame

- If output is a single column, it returns a vector instead of a data frame

## Data Frame Subsetting

**Data Frame** – possesses the **characteristics of both lists and matrices**. If you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices

#### 1. Subset with a single vector : Behave like lists

```
df1[c('col1', 'col2')]
```

#### 2. Subset with two vectors : Behave like matrices

```
df1[, c('col1', 'col2')]
```

The results are the same in the above examples, however, results are different if subsetting with only one column. (see below)

#### 1. Behave like matrices

```
str(df1[, 'col1']) -> int [1:3]
```

- Result: the result is a vector

#### 2. Behave like lists

```
str(df1[['col1']]) -> 'data.frame'
```

- Result: the result remains a data frame of 1 column

## \$ Subsetting Operator

#### 1. About Subsetting Operator

- Useful shorthand for `[[` combined with character subsetting

```
x$y is equivalent to x[['y', exact = FALSE]]
```

#### 2. Difference vs. `[[`

- `$` does partial matching, `[[` does not

```
x <- list(abc = 1)
x$a -> 1      # since "exact = FALSE"
x[['a']] ->    # would be an error
```

#### 3. Common mistake with `$`

- Using it when you have the name of a column stored in a variable

```
var <- 'cyl'
x$var
# doesn't work, translated to x[['var']]
# Instead use x[[var]]
```

## Examples

#### 1. Lookup tables (character subsetting)

```
x <- c('m', 'f', 'u', 'f', 'f', 'm', 'm')
lookup <- c(m = 'Male', f = 'Female', u = NA)
lookup[x]
> m f u f f m m
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
unname(lookup[x])
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
```

#### 2. Matching and merging by hand (integer subsetting)

Lookup table which has multiple columns of information:

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c('Excellent', 'Good', 'Poor'),
  fail = c(F, F, T)
)
```

First Method

```
id <- match(grades, info$grade)
info[id, ]
```

Second Method

```
rownames(info) <- info$grade
info[as.character(grades), ]
```

#### 3. Expanding aggregated counts (integer subsetting)

- Problem:** a data frame where identical rows have been collapsed into one and a count column has been added
- Solution:** `rep()` and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index: `rep(x, y)` `rep` replicates the values in `x`, `y` times.

```
df1$countCol is c(3, 5, 1)
rep(1:nrow(df1), df1$countCol)
> 1 1 1 2 2 2 2 2 3
```

#### 4. Removing columns from data frames (character subsetting)

There are two ways to remove columns from a data frame:

Set individual columns to NULL	<code>df1\$count3 &lt;- NULL</code>
Subset to return only columns you want	<code>df1[c('col1', 'col2')]</code>

#### 5. Selecting rows based on a condition (logical subsetting)

- This is the most commonly used technique for extracting rows out of a data frame.

```
df1[df1$count1 == 5 & df1$count2 == 4, ]
```

### Boolean Algebra vs. Sets (Logical and Integer Subsetting)

1. **Using integer subsetting** is more effective when:
  - You want to find the first (or last) TRUE.
  - You have very few TRUEs and very many FALSEs; a set representation may be faster and require less storage.
2. **which()** - conversion from boolean representation to integer representation

```
which(c(T, F, T, F)) -> 1 3
```

- Integer representation length : is always  $\leq$  boolean representation length
- Common mistakes :
  - I. Use **x[which(y)]** instead of **x[y]**
  - II. **x[-which(y)]** is not equivalent to **x[!y]**

#### Recommendation:

Avoid switching from logical to integer subsetting unless you want, for example, the first or last TRUE value

### Subsetting with Assignment

1. All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
df1$col1[df1$col1 < 8] <- 0
```

2. Subsetting with nothing in conjunction with assignment :

- Why : Preserve original object class and structure

```
df1[] <- lapply(df1, as.integer)
```

### Debugging Methods

1. **traceback()** or **RStudio's error inspector**
  - Lists the sequence of calls that lead to the error
2. **browser()** or **RStudio's breakpoints tool**
  - Opens an interactive debug session at an arbitrary location in the code
3. **options(error = browser)** or **RStudio's "Rerun with Debug" tool**
  - Opens an interactive debug session where the error occurred
  - Error Options:

#### options(error = recover)

- Difference vs. 'browser': can enter environment of any of the calls in the stack

#### options(error = dump\_and\_quit)

- Equivalent to 'recover' for non-interactive mode
- Creates **last.dump.rda** in the current working directory

In batch R process :

```
dump_and_quit <- function() {
  # Save debugging info to file
  last.dump.rda
  dump.frames(to.file = TRUE)

  # Quit R with error status
  q(status = 1)
}
options(error = dump_and_quit)
```

In a later interactive session :

```
load("last.dump.rda")
debugger()
```

### Condition Handling of Expected Errors

1. **Communicating potential problems to users:**

#### I. stop()

- Action : raise fatal error and force all execution to terminate
- Example usage : when there is no way for a function to continue

#### II. warning()

- Action : generate warnings to display potential problems
- Example usage : when some of elements of a vectorized input are invalid

#### III. message()

- Action : generate messages to give informative output
- Example usage : when you would like to print the steps of a program execution

2. **Handling conditions programmatically:**

#### I. try()

- Action : gives you the ability to continue execution even when an error occurs

#### II. tryCatch()

- Action : lets you specify handler functions that control what happens when a condition is signaled

```
result = tryCatch(code,
  error = function(c) "error",
  warning = function(c) "warning",
  message = function(c) "message"
)
```

Use `conditionMessage(c)` or `c$message` to extract the message associated with the original error.

### Defensive Programming

**Basic principle** : "fail fast", to raise an error as soon as something goes wrong

1. **stopifnot()** or use 'assertthat' package - check inputs are correct
2. **Avoid subset(), transform() and with()** - these are non-standard evaluation, when they fail, often fail with uninformative error messages.
3. **Avoid [ and supply()]** - functions that can return different types of output.
  - Recommendation : Whenever subsetting a data frame in a function, you should always use **drop = FALSE**

---

# A COMPARISON OF R AND PYTHON

PROGRAMMING IN R



# A BIT OF HISTORY

## R:

- a successor to S
- developed by statisticians as a 'statistical programming language'
- built-in data structures and functionality intended to make working with data easier
- gained prominence as a free and open source alternative to expensive statistical software

## Python:

- created in the early 90's but popularized in the 00's
- intended to be easy to read, easy to understand and easy to learn, relative to other OOLs
- has a massive base of open-source modules

# COMPARISON

## R:

- technically object oriented, but this tends to be a bit hidden in practice
- lends itself to quick interactive scripting, data exploration
- has special built-in notation for statistical models
- has a special data type – the data frame – for handling datasets

## Python:

- object oriented
- lends itself to writing structured, pre-designed computer code.
- intended to be a general programming language
- designed to create code that is easy to read



## A NOTE : VECTORIZATION IN INTERPRETED LANGUAGES

High-level interpreted languages are slower than low-level/ compiled languages.

To get around this, these languages will sometimes hand off (behind the scenes) certain types of operations to functions written in lower-level languages (like C).

In order to take advantage of this, the R and Python, communities emphasize a certain programming strategy when using lists/vectors/arrays.

In particular, they avoid cycling through each item of a list, and instead often use special functions that **map** a chosen function or operation to every item in the list.

This can run counter to habits gained when learning other languages.

# SO MANY PACKAGES/MODULES!

The strength of both R and Python lies in their many technical packages and modules.

These allow a programmer to implement very sophisticated functionality simply by making a few function calls.

Let's open the RPackagesDemo and PythonPackagesDemo notebooks to see some of this in action.

## Available CRAN Packages By Name

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

<a href="#">A3</a>	Accurate, Adaptable, and Accessible Error Metrics for Predictive Models
<a href="#">abbyyR</a>	Access to Abbyy Optical Character Recognition (OCR) API
<a href="#">abc</a>	Tools for Approximate Bayesian Computation (ABC)
<a href="#">abc.data</a>	Data Only: Tools for Approximate Bayesian Computation (ABC)
<a href="#">ABC.RAP</a>	Array Based CpG Region Analysis Pipeline
<a href="#">ABCanalysis</a>	Computed ABC Analysis
<a href="#">abcdeFBA</a>	ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this package
<a href="#">ABCOptim</a>	Implementation of Artificial Bee Colony (ABC) Optimization
<a href="#">ABCp2</a>	Approximate Bayesian Computational Model for Estimating P2
<a href="#">abcrf</a>	Approximate Bayesian Computation via Random Forests

---

# RESOURCES FOR PROGRAMMING

PROGRAMMING IN R

# R STUDIO

The screenshot displays the R Studio desktop environment. The top toolbar includes icons for file operations and a 'Go to file/function' search bar. The main editor window shows two tabs: 'Untitled1\*' and 'nodo\_dataset\_igraphlab'. Below the editor is the 'Console' pane, which contains the following text:

```
R version 3.2.1 (2015-06-18) -- "World-Famous Astronaut"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]
```

Below the console, a red status bar indicates: 'Loading required package: RMySQL'.

On the right side, the 'Environment' pane shows the 'Global Environment' with a search bar. It lists variables in the workspace:

Variable	Value
data	813 obs. of 2 variables
mydb	Formal class MySQLConnection
namecounts	'table' int [1:256(1d)] 2 8 4 1 1 2 6 1 2 1 ...
namelist	chr [1:256] "Aaron" "Aboriginal" "Adam" "Adrianna" ...
rs	Formal class MySQLResult

Below the environment pane is the 'Files' pane, which shows a file explorer view of the current directory. It includes a toolbar with 'New Folder', 'Delete', 'Rename', and 'More' options. The file list is as follows:

Name	Size	Modified
.RData	6.7 KB	Jul 11, 2018, 4:09 PM
.Rhistory	178 B	Sep 17, 2019, 5:31 PM
Applications		
CHLPA_proc_20150323.pdf	190.4 KB	Nov 1, 2015, 10:02 PM
CHLPA_proc_20150323.vdx	109.6 KB	Nov 1, 2015, 10:02 PM
CHLPA_report20150323.docx	1.4 MB	Nov 1, 2015, 10:02 PM
CHLPA_simple_simulation_20150323.xlsx	84.4 KB	Nov 1, 2015, 10:02 PM
CHLPAdocumentflow_20150323.pdf	109.5 KB	Nov 1, 2015, 10:02 PM
CHLPAdocumentflow_20150323.vdx	95.3 KB	Nov 1, 2015, 10:02 PM

# JUPYTER NOTEBOOKS

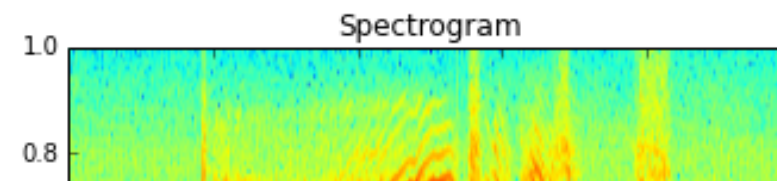
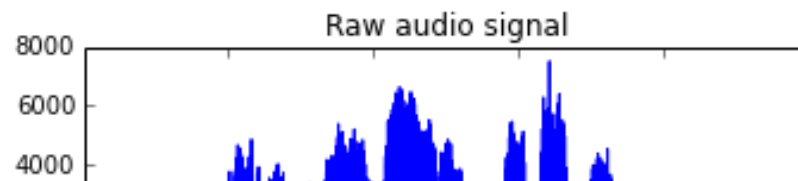
$$X_k = \sum_{n=0} x_n e^{-\frac{j2\pi}{N}kn} \quad k = 0, \dots, N-1$$

We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile
rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin specgram routine:

```
In [2]: %matplotlib inline
from matplotlib import pyplot as plt
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(x); ax1.set_title('Raw audio signal')
ax2.specgram(x); ax2.set_title('Spectrogram');
```



# R NOTEBOOKS

You can use the provided R notebooks to:

- get a sense of what can be done
- gain exposure to many examples of the language syntax
- help you write your own code
- learn why the code works the way it does, and some theory behind the code

## HCLUST()

Let's start by clustering the entire `mtcars` dataset, using the Euclidean distance metric, and plot the result. Hierarchical clustering is implemented in the `cluster` function `hclust()`.

```
(hclustcars <- hclust(dist(mtcars)))  
plot(hclustcars)
```

Call:  
`hclust(d = dist(mtcars))`

Cluster method : complete  
Distance : euclidean  
Number of objects: 32



The output of `hclust` gives us some information about the parameters being used to create the hierarchy. In this case the distance is Euclidean (as expected) and the cluster formation strategy (the **linkage**) is complete (these are the default settings).

# ON-LINE RESOURCES

Stack Exchange/Stack Overflow/Cross Validated

Blogs (e.g. R Bloggers)

Official Sites:

- Python Software Foundation: <https://www.python.org>
- Comprehensive R Archive Network (CRAN): <https://cran.r-project.org>



CRAN  
[Mirrors](#)  
[What's new?](#)  
[Task Views](#)  
[Search](#)

## The Comprehensive R Archive Network

### Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

---

# EXERCISES FOR HANDS-ON LEARNING

PROGRAMMING IN R



# GETTING INTO PROGRAMMING

Develop/assess R skills by carrying out the following exercises (order unimportant):

You may choose to carry out each of the exercises separately, or to write a single program that carries out all of the individual exercises.

**You will find much of the base code you need in the course notebooks and R markdown files**, but you will need to tweak and add to this code to carry out the exercises. You will also find a lot of helpful information and code on the internet!

# EXPRESSIONS, VARIABLES, DATA STRUCTURES, OPERATORS (1)

Create three variables and assign numerical values to each of these variables.

Then write one or more statements that carry out the following types of operations using these variables: addition, subtraction, multiplication, division, raising to a power.

## EXPRESSIONS, VARIABLES, DATA STRUCTURES, OPERATORS (2)

Create three variables and assign string values to each of these variables.

Write a statement that joins the three strings into a single string. Write some code that prints the string.

Write some code that tests to see if a substring of your choosing is contained within the larger string.

## EXPRESSIONS, VARIABLES, DATA STRUCTURES, OPERATORS (3)

Create three variables and assign lists to each of these variables. Join the three lists into a new list containing three distinct sub-lists (a list of three lists).

Create a list without sub-lists (all original list elements are part of a single larger list).

Create a fourth list by splitting this resulting list in half and assigning the second half of the list to a new variable.

Extract the last item of this list (it can either stay in the original list or be removed from it) and assign this element to a variable.

# STATEMENTS, BLOCKS, CONTROL FLOW, LOGICAL OPERATORS

Write a statement that contains at least three nested blocks.

Use at least three of the following control flow options: `if`, `if ... else`, `while`, `for`, `break`, `next`, `switch`.

# FUNCTIONS

Write a function that takes three arguments as input and returns one value.

Call the function with arguments of your choosing.

# LIBRARIES/PACKAGES/MODULES

Execute the relevant command that shows a list of the packages that are currently installed in your R environment.

- hint: use the internet, example notebooks and handouts to help you find the relevant command.

Use available documentation to determine what some of these do.

- hint: take a look at the R markdown files that are available – you may notice some relevant information there.
- choose a module/package from this list and load the relevant package/module if necessary.

Write some code that uses functions and objects supplied by this package.

# INPUTS/OUTPUTS (1)

Print to the standard output of R three sentences of your choosing, on three separate lines, using a single statement of code.



## INPUTS/OUTPUTS (2)

Locate a comma separated values (CSV) file stored on your computer

- (Hint - there should be a folder called Data in the main course directory).

Read this file into the notebook and store the results in one or more variables.

## INPUTS/OUTPUTS (3)

Create a new file and write four lines in CSV format to this file.

In a separate statement, write four more lines to this existing file, without overwriting the original file.

# INTERPRETERS/COMPILERS

Write enough code to generate at least five different error messages from R.

Copy these error messages into a markdown cell, and write a short note under each explaining the meaning of the error message, and how the code was fixed.

## OPTIONAL EXERCISES

1. Using a language of your choice, write a function that, when passed a dataset, reports 5 interesting pieces of information about the dataset. Load a dataset and run the function on this dataset.
2. Using a language of your choice, write two functions. The output of the first function should work as the input to the second function. The first function should read in a dataset and generate a subset of the dataset based on some chosen criteria. The second function should read in a dataset and provide summary data of some type for each column in the dataset. Load a dataset and run both functions on the dataset.