Why  ?

Rei Sanchez-Arias, Ph.D.

Introducing R

The R Project for Statistical Computing

Why R?

- **Free:** R and Rstudio are free and *open sourced*!
- **Reproducible:** Analyses done using R are easy to run again with updated data.
- **Collaboration:** Results are easy to share.
- **Manipulation:** Data manipulation is much easier due to cool packages.
- **Understandable:** You can follow how new variables are created in someone else's code and to remember what past-you did.

Really, why R?

- **Visualizations:** Play with visualizations before throwing into a dashboard to see if it looks interesting.
- **Help:** Finding answers to your programming questions is a quick Google away.
- **Community:** Actively developed and a very active user community

R History

R is a programming language. Specifically it's a statistical programming language very popular in the data science community.

R is a dialect of the S language which was developed by John Chambers at Bell Labs in 1976.

The philosophy behind S (and R) was to allow users to begin in an *interactive environment*.

As their needs and skills grew they could move into more of the programming aspects.

Packages 

Packages

- Packages are simply bits of code, external to the core R code that are designed to perform a specific function.
- The vast majority of the usefulness and functionality of R resides in *packages*.
- These packages live in online repositories and can be installed on your own system to be used.

Installing packages

- Packages need only be **installed once**, although you may have to reinstall when upgrading R or when you want to use a newer version of a package.
- To install from CRAN (The Comprehensive R Archive Network) all one needs to do is:

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

Using packages

Once installed all the functions (and data) in a package are available to be used. *Load* a package using the `library()` function

```
library(tidyverse)                      # load the tidyverse package
iris %>%                                # iris is a built-in dataset
  filter(Species == "setosa") %>%       # filter observations
  head(10)                              # print the first 10 rows
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

Under the hood

- `library()` loads the package into memory and allows you to use the functions within without naming the package directly every time (allowing you to use `function_name()` instead of `package::function_name()`)
- Technically what is happening here is that when attaching a package R puts those functions in your *search path*, the place R looks first for objects and functions.
- This may cause problems if packages have functions with the same name. R will choose the version for the package loaded last.
- Packages are *attached* in your current session and need to be attached every time you start a new session.

Finding help and dealing with errors

R-package authors are *required to document* their functions although this happens at a various levels of usefulness.

- Simply type `?function_name()` to get help on a function. For example: `?paste`
- Look carefully what parameters the function requires and what type they are.
- Some are required (listed first, no default) and some are optional (a default value is usually listed).
- Most function help will also indicate what the function returns.
- Good documentation also has more information on what the function is doing.

Elsewhere

- Sometimes authors will provide more detailed documentation online.
- This is more common for more recent packages where the authors may have a GitHub repository and associated webpage.
- Often discussion pages (Google groups, Stackoverflow, community.rstudio.com) can also be a useful source of help

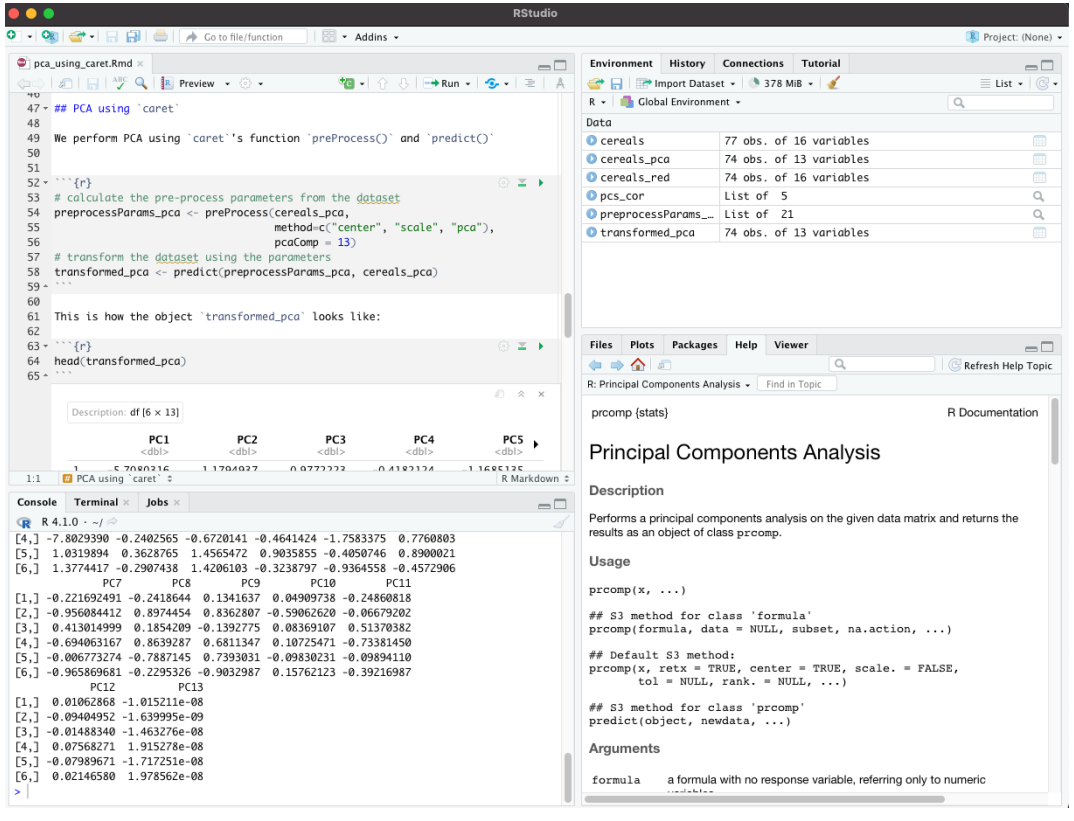
Dealing with errors

What to do?

1. **Re-read** the error message and then think about it for a minute. See if you can't *get a grasp on what's really going wrong*.
2. Check your code for errors. **Spelling** errors, misplaced commas, **forgotten** parenthesis can all cause problems
3. Look it up - I very, very rarely get an error that someone else hasn't seen before.

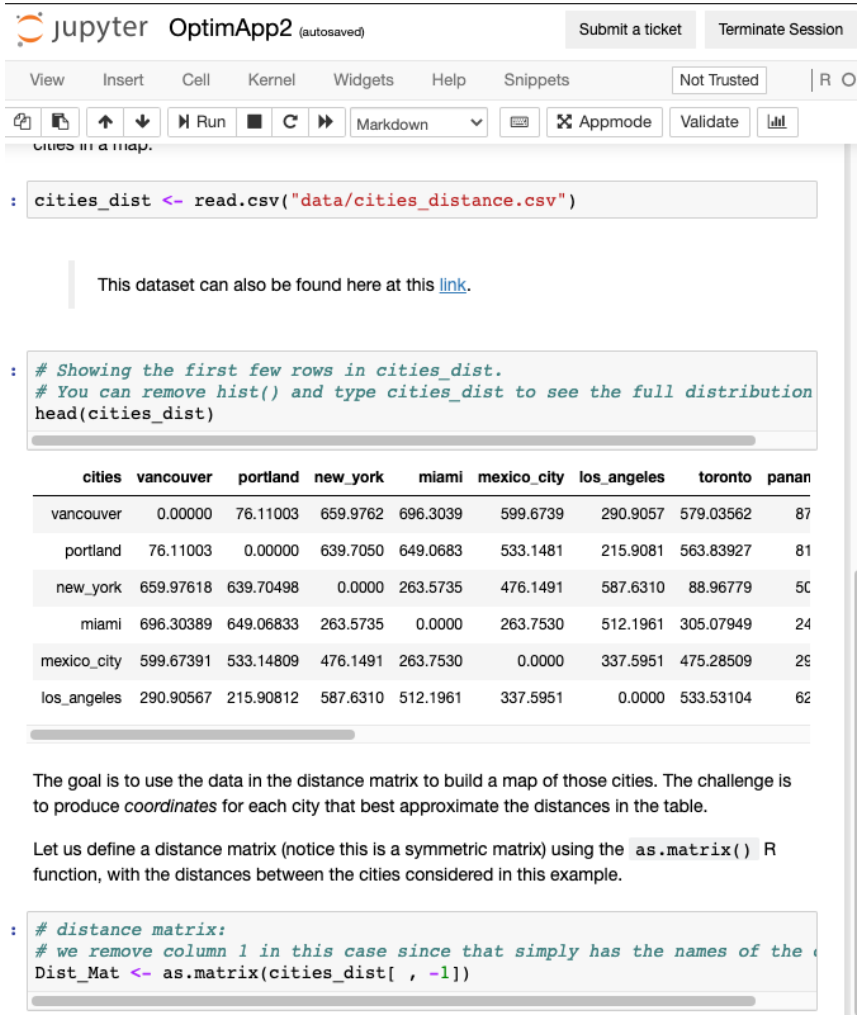
Using R with RStudio

Popular option: use R with the RStudio IDE



Using R with Project Jupyter

Popular option: use R within Project Jupyter



Arithmetic

Basic operations

- R uses the usual symbols for addition `+`, subtraction `-`, multiplication `*`, division `/`, and exponentiation `^`. Parentheses `( )` can be used to specify the order of operations.
- R also provides `%%` for taking the modulus and `%/%` for integer division.

```
(1 + 1/100)^100
```

```
## [1] 2.704814
```

```
17 %/% 5
```

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

Some built-in functions

```
exp(1)
```

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

```
sin(pi/6)
```

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

```
floor(4.3456)
```

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

```
ceiling(9.50)
```

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

Variables

Defining variables

To assign a value to a variable we use the **assignment command** `<-`

- Variables are created the first time you assign a value to them. You can give a variable any name made up of letters, numbers, and `.` or `_`, provided it starts with a letter, or `.` then a letter. Note that names are **case sensitive**.

```
x <- 100

(1 + 1/x)^x
```

```
## [1] 2.704814
```

When assigning a value to a variable, the expression on the right-hand side is evaluated first, then that value is placed in the variable on the left-hand side.

Function calls

In mathematics a function takes one or more arguments (or inputs) and produces one or more outputs (or return values). Functions in R work in an analogous way.

To call or invoke a built-in (or *user-defined*) function in R you write the name of the function followed by its argument values enclosed in parentheses and separated by commas.

```
seq(from = 1, to = 9, by = 2)
```

```
## [1] 1 3 5 7 9
```

Function calls (cont.)

Some arguments are optional, and have *predefined default values*, for example, if we omit `by`, then R assumes `by = 1`:

```
seq(1, 9)
```

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

To find out about default values and alternative usages of the built-in function `fname`, you can access the **built-in help** by typing `help(fname)` or `?fname` in the console.

Function arguments

Every function has a default order for the arguments. If you provide arguments in this order, then they do not need to be named, but you can choose to give the arguments out of order provided you give them names in the format `argument_name = expression`.

```
seq(by = -2, 9, 1)
```

```
## [1] 9 7 5 3 1
```

```
z <- 9
seq(1, z, z/3)
```

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

Vectors

Vector

A vector is an indexed list of variables. You can think of a vector as a drawer in a filing cabinet: the drawer has a name on the outside and within it are files labelled sequentially 1,2,3,... from the front.

Each file is a simple variable whose name is made up from the name of the vector and the number of the label/index:

> the name of the `i`-th element of vector `x` is `x[i]`.

Creating vectors

To create vectors of length greater than 1, we use functions that produce vector-valued output.

Three basic functions for constructing vectors are

- `c(...)` (combine)
- `seq(from, to, by)` (sequence)
- `rep(x, times)` (repeat).

```
(x <- seq(1, 20, by = 2))
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

Vector entries

> To refer to element `i` of vector `x`, we use `x[i]`.

If `i` is a vector of positive integers, then `x[i]` is the corresponding subvector of `x`. If the elements of `i` are negative, then the corresponding values are *omitted*.

```
(x <- 100:115)
```

```
##  [1] 100 101 102 103 104 105 106 107 108
```

```
i <- c(1, 3, 2)
x[i]
```

```
## [1] 100 102 101
```

The function `length(x)` gives the number of elements of `x`. It is possible to have a vector with no elements.

```
length(sin(c(-pi, 0, pi, pi/2, -pi/2)))
```

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

Vector Operations

Algebraic operations on vectors act on each element separately, that is, **element-wise**.

```
x <- c(1, 2, 3)
y <- c(4, 5, 6)
x*y
```

```
## [1]  4 10 18
```

Recycling

When you apply an algebraic expression to two vectors of unequal length, R automatically repeats the shorter vector until it has something the same length as the longer vector.

```
c(1, 2, 3, 4) + c(1, 2)
```

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

```
# Another example
c(1,2,3) + c(1,2)
```

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

Statistics

A useful set of functions that take vector arguments are `sum(...)`, `prod(...)`, `max(...)`, `min(...)`, `sqrt(...)`, `sort(x)`, `mean(x)`, and `var(x)`.

Note that functions applied to a vector may be defined to act element-wise or may act on the whole vector input to return a result:

```
mean(1:6)
```

```
## [1] 3.5
```

```
sqrt(20:25)
```

```
## [1] 4.472136 4.582576 4.690416 4.795832 4.898979 5.000000
```

```
var(seq(from = 3, to = 10, by = 0.5))
```

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

Matrices

Define a matrix

A matrix is created from a vector using the function matrix, which has the form

```
matrix(data, nrow = 1, ncol = 1, byrow = FALSE)
```

Here `data` is a vector of length at most `nrow*ncol`, `nrow` and `ncol` are the number of rows and columns, respectively (with default values of 1), and `byrow` can be either `TRUE` or `FALSE` (defaults to `FALSE`) and indicates whether you would like to fill the matrix up row-by-row or column-by-column, using the elements of `data`.

If `length(data)` is less than `nrow*ncol` (for example, the length is 1), then `data` is reused as many times as is needed.

Matrix example

```
A <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
A
```

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

To retrieve the dimension of a matrix use `dim()`:

```
dim(A)
```

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

Diagonal Matrix

To create a diagonal matrix we use `diag(x)`.

```
diag(c(2, 4, 6, 8))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    0    0    0
## [2,]    0    4    0    0
## [3,]    0    0    6    0
## [4,]    0    0    0    8
```

- To join matrices with rows of the same length (stacking vertically) use `rbind(...)`.

Matrix operations

The usual algebraic operations, including `*`, act element-wise on matrices.

- To perform matrix multiplication we use the operator `%*%`.
- We also have a number of functions for using with matrices, for example `nrow(x)`, `ncol(x)`, `det(x)` (the determinant), `t(x)` (the transpose), and `solve(A, B)`, which returns `x` such that `A %*% x == B`. If `A` is invertible then `solve(A)` returns the matrix inverse of `A`.

Matrix product

```
A <- matrix(c(3, 5, 2, 3, -2, pi),
            nrow = 3, ncol = 2)
A
```

```
##      [,1]      [,2]
## [1,]    3  3.000000
## [2,]    5 -2.000000
## [3,]    2  3.141593
```

```
B <- matrix(c(1, 1, 0, 1),
            nrow = 2, ncol = 2)
B
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    1    1
```

Matrix product:

```
A %*% B
```

```
##          [,1]      [,2]
## [1,] 6.000000  3.000000
## [2,] 3.000000 -2.000000
## [3,] 5.141593  3.141593
```

Matrix/Vector object

If you wish to find out if an object is a matrix or vector, then you use `is.matrix(x)` and `is.vector(x)`.

Of course mathematically speaking, a vector is equivalent to a matrix with one row or column, but they are treated as different types of objects in R.

To create a matrix `A` with one column from a vector `x`, we use `A <- as.matrix(x)`. Note that this does not change the values of `x`.

To create a vector from the columns of a matrix `A` we use `as.vector(A)`; this just strips the dimension attribute from `A` and leaves the elements as they are (stored columnwise).

Missing Data

NA

It is often the case, that certain observations are missing. Depending on the statistical analysis involved, missing data can be ignored or invented (a process called imputation). R represents missing observations through the data value `NA`.

Think of `NA` values as *place holders* for data that should have been there, but, for some reason, are not. We can detect whether variables are missing values using `is.na()`.

> Note that `NA` and `NULL` are not equivalent. `NA` is a place-holder for something that exists but is missing. `NULL` stands for something that never existed at all.

is.na()

```
# assign NA to a variable
a <- NA
# is a missing?
is.na(a)
```

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

```
myvector <- c(11, NA, 3, 7)
```

```
# NAs can propagate
mean(myvector)
```

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

```
# NAs can be ignored
mean(myvector, na.rm = TRUE)
```

```
## [1] 7
```