Data Science for Economists

Lecture 2: R language basics

Alex Marsh
University of North Carolina | ECON 370

Table of contents

✱ Slides adapted from Grant McDermott's EC 607 at University of Oregon.

Prologue

Checklist

☑ R and RStudio are installed and running on your computer.

☑ Did anyone play around with `ggplot2`?

Checklist

☑ R and RStudio are installed and running on your computer.

☑ Did anyone play around with `ggplot2`?

Packages that you will need for today.

We're going to work almost exclusively in **base** R today.

- I'll also use the **dplyr** package, but only to demonstrate a few additional considerations for working with non-base libraries. Install/update it now, either through RStudio (recommended) or from your R console (`install.packages("dplyr"), dependencies = TRUE`).
- (P.S. If you're on Linux, I recommend installing the pre-compiled binary version of **dplyr** from RSPM. This avoids the need to build the package from source, greatly reducing your installation time. See related example here.)

Agenda

Today and the next lecture are going to be very hands on.

- I'll have slides as per usual, but we're going to spent a lot of time live coding together.

This is deliberate.

- I want you to get comfortable typing R commands yourself — and navigating the RStudio IDE — without resorting to copy+paste.
- Slightly more painful in the beginning, but much better payoff in the long-run.

Introduction

(Some important R concepts)

Basic Arithmetic

R is a powerful calculator and recognizes all of the standard arithmetic operators:

```
1+2 ## Addition
```

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

```
6-7 ## Subtraction
```

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

```
5/2 ## Division
```

```
## [1] 2.5
```

```
2^3 ## Exponentiation
```

```
## [1] 8
```

```
2+4*1^3 ## Standard order of precedence (`*` before `+`, etc.)
```

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

Basic Arithmetic

When possible, do operators in vectors in `R`.

```
first_vec = 1:5   #store our first vector
first_vec         #show vector
```

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

```
first_vec + 0.5   #add 1/2 to each element
```

```
## [1] 1.5 2.5 3.5 4.5 5.5
```

```
first_vec + 6:10 #add another vector to first_vector
```

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

```
first_vec + 6:9   #oops!
```

```
## Warning in first_vec + 6:9: longer object length is not a multiple of shorter
## object length
```

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

Basic Arithmetic (cont.)

We can also invoke modulo operators (quotient & remainder).

- Very useful when dealing with time, for example.

```
100 %/% 60 ## How many whole hours in 100 minutes?
```

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

```
120 %/% 60 ## How many whole hours in 120 minutes?
```

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

```
100 %% 60 ## How many minutes are left over from dividing 100 by 60?
```

```
## [1] 40
```

```
120 %% 60 ## How many minutes are left over from dividing 120 by 60?
```

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

Data Types in R

Data Types

R has 6 basic data types:

1. Character
2. Numeric
3. Integer
4. Logical
5. Complex
6. Raw (we will mostly ignore this type)

Data Types (cont.)

Character

- The character type can be described as "text."
  - It is known as a "string" in other programming languages.

```
my_name    = "Alex Marsh"
first_name = "Alex"
last_name  = "Marsh"     #good style to line up to equals signs
class(my_name)
```

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

```
is.character(my_name)
```

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

Data Types (cont.)

Character: Counting Characters

```
length(my_name)
```

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

```
length(c(first_name,last_name))
```

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

```
nchar(my_name)
```

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

```
nchar(c(first_name,last_name))
```

```
## [1] 4 5
```

Character: Combinding Characters

```
also_my_name = 'Alex Marsh'
my_name == also_my_name
```

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

```
paste(first_name,last_name)
```

```
## [1] "Alex Marsh"
```

```
paste(first_name,last_name,sep="-")
```

```
## [1] "Alex-Marsh"
```

```
paste0(first_name,last_name)
```

```
## [1] "AlexMarsh"
```

Data Types (cont.)

Character: An Aside

- The following code will look *very* similar to a user when examined in the console.

```
my_name
```

```
## [1] "Alex Marsh"
```

```
print(my_name)
```

```
## [1] "Alex Marsh"
```

Data Types (cont.)

Character: An Aside

- The following code will look *very* similar to a user when examined in the console.

```
my_name
```

```
## [1] "Alex Marsh"
```

```
print(my_name)
```

```
## [1] "Alex Marsh"
```

- However, what's happening is ***very different.***
  - The first thing returns the *object* `my_name`
  - The second only prints the *contents* of the object `my_name`

Character: An Aside

- The following code will look *very* similar to a user when examined in the console.

```
my_name
```

```
## [1] "Alex Marsh"
```

```
print(my_name)
```

```
## [1] "Alex Marsh"
```

- However, what's happening is ***very different.***
    - The first thing returns the *object* `my_name`
    - The second only prints the *contents* of the object `my_name`
- Imagine I have something written on a piece of paper. The first line is if I handed you the piece of paper and you read it. The second line is if I told you aloud what was written on the paper.
    - This difference will matter a lot when it comes to functions.

Data Types (cont.)

Character

```
first_name + " " + last_name
```

```
## Error in first_name + " ": non-numeric argument to binary operator
```

```
toupper(paste(first_name,last_name))
```

```
## [1] "ALEX MARSH"
```

```
tolower(paste(first_name,last_name))
```

```
## [1] "alex marsh"
```

Data Types (cont.)

Character: Special Characters

- Special characters "not native" to English can still be used but must be encoded so that `R` handles them correctly.
- Encoding guarantees that the computer will handle the special characters correctly.
- We will not spend much time on this; however, to read more about encoding text, see this post.

Data Types (cont.)

Numeric

Numeric data are "numbers"

- In a lot of programming languages, there is this distinction between floats and integers. This is also true in `R` but to a much smaller degree.
- Unless explicitly stated as an integer, all "numbers" are numeric data.

Data Types (cont.)

Numeric

Numeric data are "numbers"

- In a lot of programming languages, there is this distinction between floats and integers. This is also true in `R` but to a much smaller degree.
- Unless explicitly stated as an integer, all "numbers" are numeric data.

```
my_age    = 28
my_height = 5 + 11/12
#my_height_apps = 6 + 0

c(my_age,my_height)
```

```
## [1] 28.000000  5.916667
```

```
class(my_age)
```

```
## [1] "numeric"
```

```
class(my_height)
```

Data Types (cont.)

Special Numeric Data

- There are a handful of special numeric values including Inf, -Inf and NaN
- Be careful when working with these

Data Types (cont.)

Special Numeric Data

- There are a handful of special numeric values including Inf, -Inf and NaN
- Be careful when working with these

```
1/0
```

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

```
log(0)
```

```
## [1] -Inf
```

```
log(-1)
```

```
## Warning in log(-1): NaNs produced
```

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

Data Types (cont.)

Integers

"Whole numbers"

- Integers in R are distinguished from numeric data by having an L after the number part.
- The distinction between numeric and integers is not that important here in R.
  - We will mostly ignore the distinction.

Data Types (cont.)

Integers

"Whole numbers"

- Integers in R are distinguished from numeric data by having an L after the number part.
- The distinction between numeric and integers is not that important here in R.
    - We will mostly ignore the distinction.

```
also_my_age = 28L
class(my_age)
```

```
## [1] "numeric"
```

```
class(also_my_age)
```

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

Data Types (cont.)

Logical

Logical data are either `TRUE` or `FALSE`.

- `T` and `F` are equivalent.

Data Types (cont.)

Logical

Logical data are either `TRUE` or `FALSE`.

- `T` and `F` are equivalent.

```
R_is_fun  = TRUE
R_is_hard = FALSE
true      = T
false     = F

R_is_fun  == true
```

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

```
R_is_hard == false
```

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

Data Types (cont.)

Logical: Operations

The most common use of logical data is for testing conditions and control flow.

- `>`, `<`, `>=`, and `<=` test for greater/less than and/or equal to.
- & and | are the logical operators for "and" and "or" respectively.
    - Order of operations: & are evaluated before |.
- ! negates a logical: `!TRUE` becomes `FALSE` and vice versa,
- To test if equal, use two equal signs ==. Not equal !=.

Data Types (cont.)

Logical: Operations

Data Types (cont.)

Logical: Operations

```r
2 > 1
```

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

```r
1 > 2 & 1 > 1/2
```

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

```r
1 > 2 | 1 > 1/2
```

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

```r
2 > 1 | 1 > 1/2
```

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

Data Types (cont.)

Logical: Operations

Data Types (cont.)

Logical: Operations

```r
1 > 1/2 & 1 > 2 | 3 > 2   #order of operations are important
```

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

```r
1 > 2 & (1 > 1/2 | 3 > 2) #use parenthesis when in doubt
```

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

```r
0.5 == 1/2
```

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

```r
3 != 2                    #that is ! followed by a =
```

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

Data Types (cont.)

Logical: Operations

%in% tests for containment

Logical: Operations

%in% tests for containment

```
R_is_fun
```

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

```
!R_is_fun
```

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

```
2 %in% c(1,2,3,4)
```

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

```
!(5 %in% c(1,2,3,4))
```

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

Aside: Binary operators

Operators like +, -, /, %%, %in%, etc are special functions in R that are binary operators.

Aside: Binary operators

Operators like +, -, /, %%, %in%, etc are special functions in R that are binary operators.

Notice that above to do "not in," it was a bit clunky:

```r
!(5 %in% c(1,2,3,4))
```

Aside: Binary operators

Operators like +, -, /, %%, %in%, etc are special functions in R that are binary operators. Notice that above to do "not in," it was a bit clunky:

```
!(5 %in% c(1,2,3,4))
```

You can define a new binary operator like so:

```
"%!in%" = function(a,b){!(a %in% b)}
5 %!in% c(1,2,3,4)
```

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

Data Types (cont.)

Special Logicals

- `NA` is a special type of logical data
- Cannot be compared the same way.
- `NA` is different than `NaN`

Data Types (cont.)

Special Logicals

- `NA` is a special type of logical data
- Cannot be compared the same way.
- `NA` is different than `NaN`

```
NA == TRUE
```

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

```
is.na(NA)
```

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

```
1 == NaN
```

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

```
is.nan(NA)
```

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

Data Types (cont.)

Logical: Operations

If "mathematical operations" are performed on logicals, they will be coerced into 1s and 0s.

Data Types (cont.)

Logical: Operations

If "mathematical operations" are performed on logicals, they will be coerced into 1s and 0s.

```
as.numeric(c(TRUE,FALSE))
```

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

```
TRUE + FALSE
```

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

```
sum(c(TRUE,FALSE,TRUE))
```

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

```
mean(c(TRUE,FALSE,TRUE))
```

```
## [1] 0.6666667
```

Data Types (cont.)

Logical: Comparing Floats

Must be careful when testing for equality of floating point numbers ("decimals")

Data Types (cont.)

Logical: Comparing Floats

Must be careful when testing for equality of floating point numbers ("decimals")

```r
0.1 + 0.2 == 0.3
```

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

What happened?

Data Types (cont.)

Logical: Comparing Floats

Must be careful when testing for equality of floating point numbers ("decimals")

```
0.1 + 0.2 == 0.3
```

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

What happened?

- Finite precision: floating point numbers aren't exact due to how numbers are stored in a computer.

```
all.equal(0.1+0.2,0.3)
```

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

Data Types (cont.)

Logical

You can read more about logical operators and types here and here.

Data Types (cont.)

Complex

Complex data are data that have complex numbers.

- We probably won't use these much but can pop-up if you're not careful.

```
2+3*sqrt(-1) #doesn't work
```

```
## Warning in sqrt(-1): NaNs produced
```

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

```
2+3i         #works
```

```
## [1] 2+3i
```

```
class(2+3i)  #complex
```

```
## [1] "complex"
```

Assignment

In `R`, we can use either `<-` or `=` to handle assignment.[1]

[1] The `<-` is really a `<` followed by a `-`. It just looks like one thing b/c of the font I'm using here.

Assignment

In `R`, we can use either `←` or `=` to handle assignment.[1]

Assignment with `←`

`←` is normally read aloud as "gets". You can think of it as a (left-facing) arrow saying *assign in this direction*.

```
a ← 10 + 5
a
```

```
## [1] 15
```

[1] The `←` is really a `<` followed by a `-`. It just looks like one thing b/c of the font I'm using here.

Assignment

In `R`, we can use either `←` or `=` to handle assignment.[1]

Assignment with `←`

`←` is normally read aloud as "gets". You can think of it as a (left-facing) arrow saying *assign in this direction*.

```
a ← 10 + 5
a
```

```
## [1] 15
```

Of course, an arrow can point in the other direction too (i.e. `→`). So, the following code chunk is equivalent to the previous one, although used much less frequently.

```
10 + 5 → a
```

[1] The `←` is really a `<` followed by a `-`. It just looks like one thing b/c of the font I'm using here.

Assignment (cont.)

Assignment with `=`

You can also use `=` for assignment.

```
b = 10 + 10 ## Note that the assigned object *must* be on the left with "=".
b
```

```
## [1] 20
```

Assignment (cont.)

Assignment with `=`

You can also use `=` for assignment.

```
b = 10 + 10 ## Note that the assigned object *must* be on the left with "=".
b
```

```
## [1] 20
```

Which assignment operator to use?

Most R users (purists?) seem to prefer `<-` for assignment, since `=` also has specific role for evaluation *within* functions.

- We'll see lots of examples of this later.
- But I don't think it matters; `=` is quicker to type and is more intuitive if you're coming from another programming language. (More discussion here and here.)
- I prefer `=` as it makes for easier to read and cleaner code.

Bottom line: Use whichever you prefer. Just be consistent.

Data Structures in R

Data Structures

In base `R`, there are 5 main data structures (not exhaustive).

1. atomic vector
2. list
3. matrix/array
4. data.frame
5. factors

Data Structures (cont.)

Vectors

- Vectors are a collection of multiple objects.
- There are two types of vectors:
    1. atomic vectors
    2. lists.
- Atomic vectors must be all of the same type of data.
- To create an atomic vector, put multiple objects within `c()` separated by commas.
    - **Never store an object as c!**
    - Will break your code and it might take hours to figure out why.
- A list is a collection of atomic vectors.
- Atomic vectors are one dimensional

Data Structures (cont.)

Vectors

- Vectors are a collection of multiple objects.
- There are two types of vectors:
  1. atomic vectors
  2. lists.
- Atomic vectors must be all of the same type of data.
- To create an atomic vector, put multiple objects within `c()` separated by commas.
  - **Never store an object as c!**
  - Will break your code and it might take hours to figure out why.
- A list is a collection of atomic vectors.
- Atomic vectors are one dimensional

```
numeric_grades = c(90,75,95,85,100,60,76)
letter_grades  = c("A-","C","A","B","A","D","C")
mixed_grades   = c("A", 95,"B",85,"C",75)
```

Data Structures (cont.)

Vectors

- Vectors are a collection of multiple objects.
- There are two types of vectors:
  1. atomic vectors
  2. lists.
- Atomic vectors must be all of the same type of data.
- To create an atomic vector, put multiple objects within `c()` separated by commas.
  - **Never store an object as c!**
  - Will break your code and it might take hours to figure out why.
- A list is a collection of atomic vectors.
- Atomic vectors are one dimensional

```
numeric_grades = c(90,75,95,85,100,60,76)
letter_grades  = c("A-","C","A","B","A","D","C")
mixed_grades   = c("A", 95,"B",85,"C",75)
```

I said atomic vectors must be all of the same type. However, the last line ran without an error. What do you think happened?

Data Structures (cont.)

Vectors

```
class(numeric_grades)
```

```
## [1] "numeric"
```

```
class(letter_grades)
```

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

```
mixed_grades
```

```
## [1] "A"  "95" "B"  "85" "C"  "75"
```

```
class(mixed_grades)
```

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

Data Structures (cont.)

Vectors: Attributes

- Objects in R can have attributes. Each type of data structure (and objects more generally) will have different attributes.
- Types of attributes include (but not limited to):
    1. names
    2. dimnames
    3. class

Data Structures (cont.)

Vectors: Attributes

```r
names(numeric_grades)
```

```
## NULL
```

```r
names(numeric_grades) = c("Student 1","Student 2","Student 3","Student 4",
                          "Student 5","Student 6","Student 7") #this is bad code!
names(numeric_grades)
```

```
## [1] "Student 1" "Student 2" "Student 3" "Student 4" "Student 5" "Student 6"
## [7] "Student 7"
```

Data Structures (cont.)

Vectors: Attributes

```r
names(numeric_grades)
```

```
## NULL
```

```r
names(numeric_grades) = c("Student 1","Student 2","Student 3","Student 4",
                          "Student 5","Student 6","Student 7") #this is bad code!
names(numeric_grades)
```

```
## [1] "Student 1" "Student 2" "Student 3" "Student 4" "Student 5" "Student 6"
## [7] "Student 7"
```

```r
names(numeric_grades) = NULL
names(numeric_grades) = paste("Student",1:length(numeric_grades))
names(numeric_grades)
```

```
## [1] "Student 1" "Student 2" "Student 3" "Student 4" "Student 5" "Student 6"
## [7] "Student 7"
```

Vectors: Indexing

- Indexing in `R` is very simple: it starts at 1 and count up by 1.
  - This is different than indexing in other programming languages which starts at 0.
  - Most "mathematical" languages will start at 1 e.g. `R`, `Matlab`, `Julia`.
- To index a vector, put `[i]` after the name of the vector where `i` is the ith position.

```
some_numbers = c(27,22,94)
some_numbers[1]
```

```
## [1] 27
```

```
some_numbers[3]
```

```
## [1] 94
```

```
some_numbers[1:2]
```

```
## [1] 27 22
```

Data Structures (cont.)

Vectors: Indexing

- Can use indexing to change values in an object

Data Structures (cont.)

Vectors: Indexing

- Can use indexing to change values in an object

```
some_numbers[2]
```

```
## [1] 22
```

```
some_numbers[2] = 23
some_numbers[2]
```

```
## [1] 23
```

Data Structures (cont.)

Vectors: Indexing

- Can use indexing to change values in an object

```
some_numbers[2]
```

```
## [1] 22
```

```
some_numbers[2] = 23
some_numbers[2]
```

```
## [1] 23
```

- Can sequentially index

Data Structures (cont.)

Vectors: Indexing

- Can use indexing to change values in an object

```
some_numbers[2]
```

```
## [1] 22
```

```
some_numbers[2] = 23
some_numbers[2]
```

```
## [1] 23
```

- Can sequentially index

```
some_numbers[1:2][2]
```

```
## [1] 23
```

Data Structures (cont.)

Lists

- Lists are collections of atomic vectors.
- Since each atomic vector is still an atomic vector, within vector types have to be the same but across vectors can differ in type.

Lists

- Lists are collections of atomic vectors.
- Since each atomic vector is still an atomic vector, within vector types have to be the same but across vectors can differ in type.

```
names(letter_grades) = paste("Student",1:length(letter_grades))
grade_list = list(names(numeric_grades),numeric_grades,letter_grades)
grade_list
```

```
## [[1]]
## [1] "Student 1" "Student 2" "Student 3" "Student 4" "Student 5" "Student 6"
## [7] "Student 7"
##
## [[2]]
## Student 1 Student 2 Student 3 Student 4 Student 5 Student 6 Student 7
##        90        75        95        85       100        60        76
##
## [[3]]
## Student 1 Student 2 Student 3 Student 4 Student 5 Student 6 Student 7
##      "A-"       "C"       "A"       "B"       "A"       "D"       "C"
```

Data Structures (cont.)

Lists: Indexing

- Indexing for lists is almost the same as vectors, you just have another layer to deal with.
- To return an element of the list as a list, use single brackets: `[i]`
- To return an element of the list as a vector, use double brackets: `[[i]]`

Lists: Indexing

- Indexing for lists is almost the same as vectors, you just have another layer to deal with.
- To return an element of the list as a list, use single brackets: `[i]`
- To return an element of the list as a vector, use double brackets: `[[i]]`

```
grade_list[1]
```

```
## [[1]]
## [1] "Student 1" "Student 2" "Student 3" "Student 4" "Student 5" "Student 6"
## [7] "Student 7"
```

```
grade_list[[1]]
```

```
## [1] "Student 1" "Student 2" "Student 3" "Student 4" "Student 5" "Student 6"
## [7] "Student 7"
```

Data Structures (cont.)

Lists: Indexing

- Elements of a list can still have names

Data Structures (cont.)

Lists: Indexing

- Elements of a list can still have names

```
names(grade_list) = c("student names","numeric grade","letter grade")
grade_list
```

```
## $`student names`
## [1] "Student 1" "Student 2" "Student 3" "Student 4" "Student 5" "Student 6"
## [7] "Student 7"
##
## $`numeric grade`
## Student 1 Student 2 Student 3 Student 4 Student 5 Student 6 Student 7
##        90        75        95        85       100        60        76
##
## $`letter grade`
## Student 1 Student 2 Student 3 Student 4 Student 5 Student 6 Student 7
##      "A-"       "C"       "A"       "B"       "A"       "D"       "C"
```

Data Structures (cont.)

Lists: Indexing

- You can access an element of a list using it's name and the `$`

Data Structures (cont.)

Lists: Indexing

- You can access an element of a list using it's name and the `$`

```
grade_list$`numeric grade` #`` are used because of the space
```

```
## Student 1 Student 2 Student 3 Student 4 Student 5 Student 6 Student 7
##        90        75        95        85       100        60        76
```

Data Structures (cont.)

Lists: Indexing

- You can access an element of a list using it's name and the `$`

```
grade_list$`numeric grade` #`` are used because of the space
```

```
## Student 1 Student 2 Student 3 Student 4 Student 5 Student 6 Student 7
##        90        75        95        85       100        60        76
```

- Double indexing still works

Data Structures (cont.)

Lists: Indexing

- You can access an element of a list using it's name and the `$`

```
grade_list$`numeric grade` #`` are used because of the space
```

```
## Student 1 Student 2 Student 3 Student 4 Student 5 Student 6 Student 7
##        90        75        95        85       100        60        76
```

- Double indexing still works

```
grade_list[[1]][4]
```

```
## [1] "Student 4"
```

```
grade_list$`student names`[4]
```

```
## [1] "Student 4"
```

Data Structures (cont.)

Matrix

- Matrices are essentially two dimensional atomic vectors.
- Most things that apply to atomic vectors are true for matrices
- While we can make character matrices, they aren't very useful as matrices are most useful for mathematical operations.
- Due to the nature of this course, we will probably not use matrices much as we are not coding things by hand.

Data Structures (cont.)

Matrix

- Matrices are essentially two dimensional atomic vectors.
- Most things that apply to atomic vectors are true for matrices
- While we can make character matrices, they aren't very useful as matrices are most useful for mathematical operations.
- Due to the nature of this course, we will probably not use matrices much as we are not coding things by hand.

```
num_mat = matrix(1:9,ncol=3)
num_mat
```

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

Data Structures (cont.)

Matrix: Indexing

- Indexing a matrix is similar to atomic vectors; two dimensions to index.
- If one dimension is omitted, returns that entire dimension.
- First index is for rows; second index is for columns.

```
num_mat[1,2]
```

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

```
num_mat[1,]
```

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

```
num_mat[,2]
```

```
## [1] 4 5 6
```

```
num_mat[5]
```

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

Array

- Arrays are 2 dimensional *or greater* matrices.
- Again, while arrays can be useful, we will likely not be working with them much.

```
num_array = array(1:(3*3*2),dim = c(3,3,2))
num_array
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
```

Data Structures (cont.)

Array: Indexing

- Indexing is same as matrices, just the same number of indexes as dimensions

Data Structures (cont.)

Array: Indexing

- Indexing is same as matrices, just the same number of indexes as dimensions

```
num_array[1,2,1]
```

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

```
num_array[,,1]
```

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

```
num_array[1,2,]
```

```
## [1]  4 13
```

```
num_array[12]
```

Data Structures (cont.)

Data.frame

- data.frames are basically "data sets."
- Using established terminology, they are collections of atomic vectors in a rectangular format.
    - This may sound similar to lists; however, lists don't have a restriction on the geometric format.
    - Could have a lists of lists of lists etc.
- Each column of a data.frame must be of the same type.
- Each row is an observation.

```
data(mtcars)
head(mtcars,5)
```

```
##                    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
```

Data Structures (cont.)

Data.frame

- `names()` on a data.frame returns the names of the variables.
- `length()` and `ncol()` return the number of variables in the data.frame
- `nrow()` returns the number of rows in the data.frame.
- Note: if you don't use `head()`, `tail()` or another way to restrict which rows are returned, returning a data.frame will return all rows, which is annoying.
    - This is fixed in a popular library called `data.table` that we will learn later.

```
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
```

Data.frame: Indexing

- Indexing a data.frame is similar to indexing a matrix; the columns can have names

```
mtcars[1:5,"mpg"]
```

```
## [1] 21.0 21.0 22.8 21.4 18.7
```

```
mtcars$mpg[1:5]
```

```
## [1] 21.0 21.0 22.8 21.4 18.7
```

```
mtcars[1:5,1]
```

```
## [1] 21.0 21.0 22.8 21.4 18.7
```

```
row.names(mtcars)[1:5]
```

```
## [1] "Mazda RX4"         "Mazda RX4 Wag"     "Datsun 710"
## [4] "Hornet 4 Drive"    "Hornet Sportabout"
```

Data Structures (cont.)

Factors:

- Factors are essentially categorical variables.
- Factors can organized text into different groups.
- Are only really useful to data analysis and modeling.
- Factors have different "levels" which are the groups.
- Can be tricky when first learning, but ultimately rather simple.

```
fact_groups = letters[sample(1:26,10,replace=T)]
fact_groups
```

```
##  [1] "a" "j" "z" "y" "w" "f" "m" "g" "n" "j"
```

```
fact_groups = factor(fact_groups,levels=letters)
fact_groups
```

```
##  [1] a j z y w f m g n j
## Levels: 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
```

Getting help

Help

For more information on a (named) function or object in R, consult the "help" documentation. For example:

```
help(plot)
```

Or, more simply, just use `?`:

```
# This is what most people use.
?plot
```

Help

For more information on a (named) function or object in R, consult the "help" documentation. For example:

```r
help(plot)
```

Or, more simply, just use `?`:

```r
# This is what most people use.
?plot
```

Aside 1: Comments in R are demarcated by `#`.

- Hit `Ctrl+Shift+c` in RStudio to (un)comment whole sections of highlighted code.

Help

For more information on a (named) function or object in R, consult the "help" documentation. For example:

```
help(plot)
```

Or, more simply, just use `?`:

```
# This is what most people use.
?plot
```

Aside 1: Comments in R are demarcated by `#`.

- Hit `Ctrl+Shift+c` in RStudio to (un)comment whole sections of highlighted code.

Aside 2: See the *Examples* section at the bottom of the help file?

- You can run them with the `example()` function. Try it: `example(plot)`.

Help (cont.)

Vignettes

For many packages, you can also try the `vignette()` function, which will provide an introduction to a package and it's purpose through a series of helpful examples.

- Try running `vignette("dplyr")` in your console now.

Help (cont.)

Vignettes

For many packages, you can also try the `vignette()` function, which will provide an introduction to a package and it's purpose through a series of helpful examples.

- Try running `vignette("dplyr")` in your console now.

I highly encourage reading package vignettes if they are available.

- They are often the best way to learn how to use a package.

Help (cont.)

Vignettes

For many packages, you can also try the `vignette()` function, which will provide an introduction to a package and it's purpose through a series of helpful examples.

- Try running `vignette("dplyr")` in your console now.

I highly encourage reading package vignettes if they are available.

- They are often the best way to learn how to use a package.

One complication is that you need to know the exact name of the package vignette(s).

- E.g. The `dplyr` package actually has several vignettes associated with it: "dplyr", "window-functions", "programming", etc.
- You can run `vignette()` (i.e. without any arguments) to list the available vignettes of every *installed* package installed on your system.
- Or, run `vignette(all = FALSE)` if you only want to see the vignettes of any *loaded* packages.

Help (cont.)

Demos

Similar to vignettes, many packages come with built-in, interactive demos.

To list all available demos on your system:[1]

```
demo(package = .packages(all.available = TRUE))
```

[1] How would you limit the demos to one particular package?

Help (cont.)

Demos

Similar to vignettes, many packages come with built-in, interactive demos.

To list all available demos on your system:[1]

```
demo(package = .packages(all.available = TRUE))
```

To run a specific demo, just tell R which one and the name of the parent package. For example:

```
demo("graphics", package = "graphics")
```

[1] How would you limit the demos to one particular package?

Next lecture(s): Objects and the OOP approach