# Managing and Manipulating Data Using R

## Introduction, part 2

Ozan Jaquette

1. What is R
2. R basics
3. Classification of objects
4. Atomic vectors
5. Lists

# What is R

# What is R

# Why R? Capabilities of R

# Base R vs. R packages

INCLUDE INSTALLING AND LOADING PACKAGES

# Functions; function syntax

[I MAY MOVE THIS SLIDE BELOW; JUST WANT YOU TO DRAFT IT]

describe basic syntax of functions in R; how to get help on a function

# RStudio

# RMarkdown

# R basics

# R as a calculator

```
5
#> [1] 5
5+2
#> [1] 7
10*3
#> [1] 30
```

# Executing commands in R

```
5
#> [1] 5
5+2
#> [1] 7
10*3
#> [1] 30
```

Three ways to execute commands in R

1. Type/copy commands directly into the "console"
2. 'code chunks' in RMarkdown (.Rmd files)
    - Can execute one command at a time, one chunk at a time, or "knit" the entire document
3. R scripts (.R files)
    - This is just a text file full of R commands
    - Can execute one command at a time, several commands at a time, or the entire script

# Shortcuts you should learn for executing commands

```r
5+2
#> [1] 7
10*3
#> [1] 30
```

Three ways to execute commands in R

1. Type/copy commands directly into the "console"
2. 'code chunks' in RMarkdown (.Rmd files)
   - **Cmd/Ctrl + Enter**: execute highlighted line(s) within chunk
   - **Cmd/Ctrl + Shift + k**: "knit" entire document
3. R scripts (.R files)
   - **Cmd/Ctrl + Enter**: execute highlighted line(s)
   - **Cmd/Ctrl + Shift + Enter** (without highlighting any lines): run entire script

# Assignment

**Assignment** means creating a variable – or more generally, an “object” – and assigning values to it

- <- is the assignment operator
  - in other languages = is the assignment operator
- good practice to put a space before and after assignment operator

```r
# Create an object and assign value
a <- 5
a
#> [1] 5

b <- "yay!"
b
#> [1] "yay!"
```

# Classification of objects

# Objects

Most statistical software (e.g., SPSS, Stata) operates on datasets, which consist of rows of observations and columns of variables

- Usually, these packages can open only one dataset at a time

R is an "object-oriented" programming language (like Python, JavaScript). So, what is an "object"?

- formal computer science definitions are confusing because they require knowledge of concepts we haven't introduced yet
- More intuitively, I think objects as anything I assign values to
  - For example, below, "`a`" and "`b`" are objects I assigned values to

```
a <- 5
a
#> [1] 5
b <- "yay!"
b
#> [1] "yay!"
```

- Ben Skinner (my R guru) says "Objects are like boxes in which we can put things: data, functions, and even other objects."

# Objects

- Objects can be categorized by “**type**” (which we will discuss today) and by “**class**” (which we will discuss in later weeks)
  - e.g., a date is an object with a numeric *type* and a date *class*
  - a dataset is an object with a particular type and class
- There is no limit to the number of objects R can hold (except memory)
- R “functions” do different things to different types/classes of objects e.g., date functions are meant to process objects with type=numeric and class=date; these functions don’t work on objects with type=character (e.g., “yay!”)

# Vectors

The fundamental object in R is the "vector"

- A vector is a collection of values
- The individual values within a vector are called "elements"
- Values in a vector can be numeric, character (e.g., "Apple"), or some other *type*

Below we use the combine function `c()` to create a numeric vector that contains three elements

- Help file says that `c()` "combines values into a vector or list"

```
#?c()
x <- c(4, 7, 9)
x
#> [1] 4 7 9
```

Vector where the elements are characters

```
animals <- c("lions", "tigers", "bears", "oh my")
animals
#> [1] "lions"  "tigers" "bears"  "oh my"
```

# Student task (do with the person next to you)

Either in the R console or within the R markdown file, do the following:

1. Create a vector called `v1` with three elements, where all the elements are numbers. Then print the values.
2. Create a vector called `v2` with four elements, where all the elements are characters (i.e., enclosed in single ” or double “” quotes). Then print the values.
3. Create a vector called `v3` with five elements, where some elements are numeric and some elements are characters. Then print the values.

# Formal classification of vectors in R

Here, I introduce the classification of vectors by Grolemund and Wickham

There are two broad types of vectors

1. **Atomic vectors**. An object that contains elements. There are six types of atomic vectors:
    - **logical**, **integer**, **double**, **character**, **complex**, and **raw**.
        - **Integer** and **double** vectors are collectively known as **numeric** vectors.
2. **Lists**. Like atomic vectors, lists are objects that contain elements
    - elements within a list may be atomic vectors
    - elements within a list may also be other lists; that is lists can contain other lists
    - This sounds vague and confusing; I'll explain and give examples below

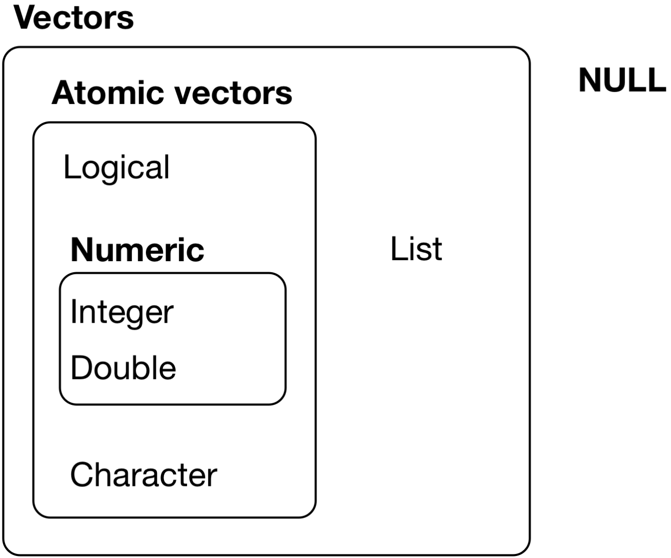
One difference between atomic vectors and lists: **homogeneous** vs. **heterogeneous** elements

- atomic vectors are **homogeneous**: all elements within atomic vector must be of the same type
- lists can be **heterogeneous**: e.g., one element can be an integer and another element can be character

# Formal classification of vectors in R

Here is a visual representation of the Grolemund and Wickham classification

PATRICIA - PUT PIC ON WEBSITE AND PROVIDE LINK



# Developing an intuitive understanding of vector types

**Grolemund and Wickham classification**:

1. **Atomic vectors**. six types: logical, integer, double, character, complex, raw.
2. **Lists**

Problem with this classification:

- Not conceptually intutive
- Technically, lists are a type of vector, but people often think of atomic vectors and lists as fundamentally different things

**Classification used by my R Guru Ben Skinner**:

- data **type**: logical, numeric (integer and double), character, etc.
- data **structure**: vector, list, matrix, etc.

I find Skinner's classification more intuitive conceptually. However, it isn't consistent with R functions or the way R thinks about objects.

If you find this classification of data *type* and data *structure* helpful, totally fine to think of objects in this way while you start to learn R.

# Atomic vectors

# “Length” of an atomic vector is the number of elements

For remainder of lecture, I’ll use the term **vector** to refer to atomic vectors

Use `length()` function to examine vector length

```
x <- c(4, 7, 9)
x
#> [1] 4 7 9
length(x)
#> [1] 3

animals <- c("lions", "tigers", "bears", "oh my")
animals
#> [1] "lions"  "tigers" "bears"  "oh my"
length(animals)
#> [1] 4
```

A single number (or a single string/character) is a vector with `length==1`

```
z <- 5
length(z)
#> [1] 1
length("Tommy")
#> [1] 1
```

# Data type of a vector

The “type” of an atomic vector refers to the elements within the vector.

While there are six “types” of actomic vectors, we’ll focus on the following types:

- numeric:
  - “integer” (e.g., 5)
  - “double” (e.g., 5.5)
- character (e.g., “ozan”)
- logical (e.g., `TRUE`, `FALSE`)

Use `typeof()` function to examine vector type

```
x
#> [1] 4 7 9
typeof(x)
#> [1] "double"

p <- c(1.5, 1.6)
p
#> [1] 1.5 1.6
typeof(p)
#> [1] "double"

animals
#> [1] "lions"   "tigers" "bears"   "oh my"
typeof(animals)
#> [1] "character"
```

# Data type of a vector, numeric

Numeric vectors can be “integer” (e.g., 5) or “double” (e.g., 5.5)

```
typeof(1.5)
#> [1] "double"
```

R stores numbers as doubles by default.

```
x
#> [1] 4 7 9
typeof(x)
#> [1] "double"
```

To make an integer, place an `L` after the number:

```
typeof(5)
#> [1] "double"
typeof(5L)
#> [1] "integer"
```

# Data type of a vector, character

In contrast to “numeric” data types wich are used to store numbers, the “character” data type is used to store **strings** of text.

- Strings may contain any combination of numbers, letters, symbols, etc.
- Character vectors are sometimes referred to as string vectors

When creating a vector where elements have `type==character` (or when referring to the value of a string), place single ‘ or double “” quotes around text

- the text within quotes is the “string”

```
c1 <- c("cat",'cash','candy cane')
c1
#> [1] "cat"        "cash"       "candy cane"
typeof(c1)
#> [1] "character"
length(c1)
#> [1] 3
```

Numeric values can also be stored as strings

```
c2 <- c("1","2","3")
c2
#> [1] "1" "2" "3"
typeof(c2)
#> [1] "character"
```

# Data type of a vector, logical

Logical vectors can take three possible values: `TRUE`, `FALSE`, `NA`

- `TRUE`, `FALSE`, `NA` are special keywords; they are different from the character strings `"TRUE"`, `"FALSE"`, `"NA"`
- Don't worry about `"NA"` for now

```r
typeof(TRUE)
#> [1] "logical"
typeof("TRUE")
#> [1] "character"

typeof(c(TRUE,FALSE,NA))
#> [1] "logical"
typeof(c(TRUE,FALSE,NA,"FALSE"))
#> [1] "character"

log <- c(TRUE,TRUE,FALSE,NA,FALSE)
typeof(log)
#> [1] "logical"
length(log)
#> [1] 5
```

We'll learn more about logical vectors later

# All elements in (atomic) vector must have same data type.

Atomic vectors are **homogenous**;

- An atomic vector has one data type
- all elements within an atomic vector must have the same data "type"

If a vector contains elements of different type, the vector type will be type of the most "complex" element

Atomic vector types from simplest to most complex:

- logical < integer < double < character

```
typeof(c(TRUE,TRUE,NA))
#> [1] "logical"
typeof(c(TRUE,TRUE,NA,1L)) # recall L after an integer forces type to be integer
#> [1] "integer"
typeof(c(TRUE,TRUE,NA,1.5))
#> [1] "double"
typeof(c(TRUE,TRUE,NA,1.5,"howdy!"))
#> [1] "character"
```

# Named vectors

All vectors can be “named” (i.e., name individual elements within vector)

Example of creating an unamed vector

- the `str()` function “compactly display[s] the internal structure of an R object” [from help file]; very useful for describing objects

```
#?str
x <- c(1,2,3,"hi!")
x
#> [1] "1"   "2"   "3"   "hi!"
str(x)
#>  chr [1:4] "1" "2" "3" "hi!"
```

Example of creating a named vector

```
y <- c(a=1,b=2,3,c="hi!")
y
#>     a     b           c
#>   "1"   "2"   "3" "hi!"
str(y)
#>  Named chr [1:4] "1" "2" "3" "hi!"
#>  - attr(*, "names")= chr [1:4] "a" "b" "" "c"
```

# Sequences

(Loose) definition: a sequence is a set of numbers in ascending or descending order

A vector containing a "sequence" of numbers (e.g., 1, 2, 3) can be created using the colon operator `:` with the notation `start:end`

```
-5:5
#>  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
5:-5
#>  [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
s<- 1:10 #same as this: s<- c(1:10)
s
#>  [1]  1  2  3  4  5  6  7  8  9 10
length(s)
#> [1] 10
```

Creating sequences using `seq()` function

- basic syntax [with default values]:

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

```
seq(10,15)
#> [1] 10 11 12 13 14 15
seq(from=10,to=15,by=1)
#> [1] 10 11 12 13 14 15
seq(from=100,to=150,by=10)
#> [1] 100 110 120 130 140 150
```

# Vectorized math

Most mathematical operations operate on each element of the vector

- e.g., add a single value to a vector and that value added to each element of the vector

```
1:3
#> [1] 1 2 3
1:3+.5
#> [1] 1.5 2.5 3.5
(1:3)*2
#> [1] 2 4 6
```

Mathematical operations involving two vectors with the same length behave differently

- e.g., for addition: add element 1 of vector 1 to element 1 of vector 2, add element 2 of vector 1 to element 2 of vector 2, etc.

```
c(1,1,1)+c(1,0,2)
#> [1] 2 1 3
c(1,1,1)*c(1,0,2)
#> [1] 1 0 2
```

# Lists

# Lists

What is a **list**?

- Like (atomic) vectors, a list is an object that contains **elements**
- Unlike vectors, data types can differ across elements within a list
- An element within a list can be another list
  - this characteristic makes lists more complicated than vectors
  - suitable for representing hierarchical data

Lists are more complicated than vectors; today we'll just provide a basic introduction

# Create lists using `list()` function

Create a vector (for comparison purposes)

```
a <- c(1,2,3)
typeof(a)
#> [1] "double"
length(a)
#> [1] 3
```

Create a list

```
b <- list(1,2,3)
typeof(b)
#> [1] "list"
length(b)
#> [1] 3
b # print list is awkward
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

# Investigate structure of lists using `str()` function

When investigating lists, `str()` is better than printing the list

```
b <- list(1,2,3)
typeof(b)
#> [1] "list"
length(b)
#> [1] 3
str(b) # 3 elements, each element is a numeric vector w/ length=1
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
```

```
c <- list(c(3,4),c(-5,1,3))
typeof(c)
#> [1] "list"
length(c)
#> [1] 2
str(c) # 2 elements; element 1=vector w/ length=2; element 2=vector w/length=3
#> List of 2
#>  $ : num [1:2] 3 4
#>  $ : num [1:3] -5 1 3
```

# Elements within list can have different data types

Lists are **heterogeneous**

- data types can differ across elements within a list

```
b <- list(1,2,"apple")
typeof(b)
#> [1] "list"
length(b)
#> [1] 3
str(b)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : chr "apple"
```

Vector are **homogeneous**

```
a <- c(1,2,"apple")
typeof(a)
#> [1] "character"
str(a)
#>  chr [1:3] "1" "2" "apple"
```

# Lists can contain other lists

```
x1 <- list(c(1,2), list("apple", "orange"), list(1, 2, 3))
typeof(x1)
#> [1] "list"
length(x1)
#> [1] 3
str(x1)
#> List of 3
#>  $ : num [1:2] 1 2
#>  $ :List of 2
#>   ..$ : chr "apple"
#>   ..$ : chr "orange"
#>  $ :List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

Note that:

- first element of list is a numeric vector with length=2
- second element is a list with length=2
  - first element is character vector with length=1
  - second element is character vector with length=1
- third element is with length=3
  - first element is numeric vector with length=1
  - second element is numeric vector with length=1
  - third element is numeric vector with length=1

# You can name each element in the list

```r
x2 <- list(a=c(1,2), b=list("apple", "orange"), c=list(1, 2, 3))
str(x2)
#> List of 3
#>  $ a: num [1:2] 1 2
#>  $ b:List of 2
#>   ..$ : chr "apple"
#>   ..$ : chr "orange"
#>  $ c:List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

`names()` function shows names of elements in the list

```r
names(x2) # has names
#> [1] "a" "b" "c"
names(x1) # no names
#> NULL
```

# Access individual elements in a “named” list

Syntax: `list_name$element_name`

```
x2 <- list(a=1, b=list("apple", "orange"), c=list(1, 2, 3))
typeof(x2$a)
#> [1] "double"
length(x2$a)
#> [1] 1

typeof(x2$b)
#> [1] "list"
length(x2$b)
#> [1] 2

typeof(x2$c)
#> [1] "list"
length(x2$c)
#> [1] 3
```

Note: We’ll spend more time practicing “accessing elements of a list” in upcoming weeks

# Compare structure of list to structure of element within a list

```
str(x2)
#> List of 3
#>  $ a: num 1
#>  $ b:List of 2
#>   ..$ : chr "apple"
#>   ..$ : chr "orange"
#>  $ c:List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3

str(x2$c)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
```

# A dataset is just a list!

A data frame is a list with the following characteristics:

- Data type can differ across elements (like all lists)
- Each **element** in data frame must be a **vector**, not a **list**
  - Each element (column) is a variable
- Each element in a data frame must have the same length
  - The length of an element is the number of observations (rows)
  - so each variable in data frame must have same number of observations
- Each element is named
  - these element names are the variable names

```
names(df)
#> [1] "mpg" "cyl" "hp"
head(df, n=5) # print first 5 rows
#> # A tibble: 5 x 3
#>     mpg   cyl    hp
#> * <dbl> <dbl> <dbl>
#> 1  21       6   110
#> 2  21       6   110
#> 3  22.8     4    93
#> 4  21.4     6   110
#> 5  18.7     8   175
```

Additionally, data frames have “attributes”; we’ll discuss those in upcoming weeks

# A data frame is a named list

```
typeof(df)
#> [1] "list"
names(df)
#> [1] "mpg" "cyl" "hp"
length(df) # length=number of variables
#> [1] 3
str(df)
#> 'data.frame':    32 obs. of  3 variables:
#>  $ mpg: num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
#>  $ cyl: num  6 6 4 6 8 6 8 4 4 6 ...
#>  $ hp : num  110 110 93 110 175 105 245 62 95 123 ...
```

Like any named list, can examine the elements

- Individual elements of a data frame are the variables
- these variables are vectors with length equal to the number of rows/observations

```
typeof(df$mpg)
#> [1] "double"
length(df$mpg) # length=number of rows/obs
#> [1] 32
str(df$mpg)
#>  num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
```

# Main takeaways about atomic vectors and lists

Basic data stuctures

1. **(Atomic) vectors**: **logical**, **integer**, **double**, **character**.
   - each element in vector must have same data type
2. **Lists**:
   - Data type can differ across elements

Takeaways

- These concepts are difficult; ok to feel confused
- I will reinforce these concepts throughout the course
- Good practice: run simple diagnostics on any new object
  - `length()` : how many **elements** in the object
  - `typeof()` : what **type** of data is the object
  - `str()` : hierarchical structure of the object

# Main takeaways about atomic vectors and lists

Basic data stuctures

1. **(Atomic) vectors**: **logical**, **integer**, **double**, **character**.
   - each element in vector must have same data type
2. **Lists**:
   - Data type can differ across elements

Takeaways

- These data structures (vectors, lists) and data types (e.g., character, numeric, logical) are the basic building blocks of all object oriented programming languages
- Application to statistical analysis
  - Datasets are just lists
  - The individual elements – columns/variables – within a dataset are just vectors
- These structures and data types are foundational for all “data science” applications
  - e.g., mapping, webscraping, network analysis, etc.