

# Lecture 4: Data Structures I: Vectors and Factors



UNIVERSITY OF  
SAN FRANCISCO

James D. Wilson

BSDS 100 - Intro to Data Science with R



- Vectors
- Factors



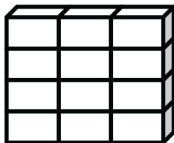
- A **data structure** is a format or organization of data in software that enables efficient use.
- Every programming language has its own types of data structures
- In  $\mathbb{R}$ , you can create your own type of data structure; however, there are some that are automatically recognized by the software.
- **Examples:** list, array, data.frame, vector, matrix, string



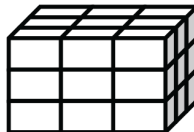
(a) Vector



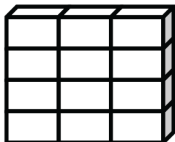
(b) Matrix



(c) Array

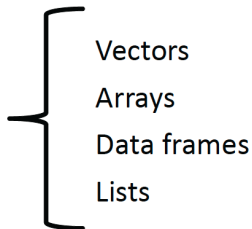


(d) Data frame



Columns can be different modes

(e) List



Vectors

Arrays

Data frames

Lists



Dimension	Homogeneous	Heterogeneous
1	Atomic Vector	List
2	Matrix	Data Frame
$n$	Array	

**Homogeneous:** All contents must be of the same type

**Heterogeneous:** Contents can be of different types

**Note:** There are no 0-dimensional (scalar) types in R, only vectors of length one

# Part I: Vectors



- The basic data structure in R is the vector
- There two types of vectors: atomic vectors and lists

## Properties of Vectors

- Type (`typeof()`)
- Length (`length()`)
- Attributes (`attributes()`)

Use `is.atomic()` or `is.list()` to determine if an object is a vector, **not** `is.vector()`



## Four Common Types of Vectors

- Logical
- Integer
- Double (numeric)
- Character

```
> doubleAtomicVector <- c(1, 3.14, 99.999)
```

```
# use L prefix to get integers instead of doubles
```

```
> integerAtomicVector <- c(1L, 3L, 19L)
```

```
> logicalAtomicVector <- c(TRUE, FALSE, T, F)
```

```
> characterAtomicVector <- c("this", "is a", "string")
```



# Example: Try This



- 1 Create the vector `myFavNum` of you favorite fractional number
- 2 Create the vector `myNums` of your seven favorite numbers
- 3 Create the vector `firstNames` of the first names of two people next to you
- 4 Create the vector `myVec` of the last name and age of someone you know

# Example: Answer these



- 1 Guess and then check what types your vectors are.
- 2 Check the length of each vector.
- 3 Did you write the code in the console window or the editor?
- 4 How do you execute a line of code in the editor?
- 5 How do you execute multiple lines of code simultaneously in the editor?
- 6 Did you leverage the `TAB` button for auto-completion?

# Accessing Elements of a Vector



- To access the individual elements of a vector

```
> (myAtomicVector <- c(1, 2, 3, 4, -99, 5, NA, 4, 22.223))  
[1] 1.000 2.000 3.000 4.000 -99.000 5.000 NA  
[8] 4.000 22.223
```

#look at fifth element of the vector

```
> myAtomicVector[5]  
[1] -99
```

```
> myAtomicVector[c(1, 2, 5, 9)]  
[1] 1.000 2.000 -99.000 22.223
```

```
> myAtomicVector[10]  
[1] NA
```

#look at the third through eighth elements of the vector

```
> myAtomicVector[3:8]  
[1] 3 4 -99 5 NA 4
```



- To look at the first and last 6 elements of a vector

```
> (myAtomicVector <- c(1, 2, 3, 4, -99, 5, NA, 4, 22.223))  
[1] 1.000 2.000 3.000 4.000 -99.000 5.000 NA  
[8] 4.000 22.223
```

#look at the first and last six elements of the vector

```
> head(myAtomicVector)  
[1] 1.000 2.000 3.000 4.000 -99.000 5.000
```

```
> tail(myAtomicVector)  
[1] 4.000 -99.000 5.000 NA 4.000 22.223
```

# Example, continued



- ➊ Add `myFavNum` to the seventh entry of `myNums` and store the result in a variable named `myFirstAddition`
- ➋ Add `myFavNum` to each of the seven entries of `myNums` and store the result in a variable named `mySecondAddition`
- ➌ Add `myFavNum` to **all** of the values in `myNums` and store the result in a variable named `myFirstSum`
- ➍ Add `myFavNum` to the smallest number in `myNums` and store the result in a variable named `thisIsGettingMoreComplex`
- ➎ Add the second entry of `myNums` to the age of the person you select for `myVec` and store the result in a variable named `whatTypeOfVectorIsThis`
  - Does what we did make sense? Did it work? Why?



```
# preamble
myFavNum <- 3.1415
myNums <- c(1, 3, 55, 33, 86, -sqrt(2), -110)
# also works myNums <- 1:7
firstNames <- c("Jeff", "Terence", "David")
myVec <- c("Parr", 99)
```

- ❶ myFirstAddition <- myFavNum + myNums[7]
- ❷ mySecondAddition <- myFavNum + myNums
- ❸ myFirstSum <- myFavNum + sum(myNums)
- ❹ thisIsGettingMoreComplex <- myFavNum + min(myNums)
- ❺ whatTypeOfVectorIsThis <- sum(c(myNums[2], myVec[2]))  
Error in sum(c(myNums[2], myVec[2])) :  
invalid 'type' (character) of argument



Missing values are specified with `NA`, a logical vector of length one.

- `NA` will always be coerced to the correct type if used inside `c()`

```
> c(1, 2, 3, NA)
[1] 1 2 3 NA
```

```
> x[1]
[1] 1
```

```
> x <- c(1, 2, 3, NA)
```

```
> x[4]
[1] NA
```

```
> typeof(x)
[1] "double"
```

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



- Certain functions will fail when applied to vectors with an `NA`

```
> myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4, NA)
[1] 99.1 98.2 97.3 96.4    NA
```

```
> sum(myAtomicVector_01)
[1] NA
```

```
> mean(myAtomicVector_01)
[1] NA
```





- You can avoid this by providing the argument `na.rm = TRUE`

```
> sum(myAtomicVector_01, na.rm = TRUE)
[1] 391
```

```
> mean(myAtomicVector_01, na.rm = TRUE)
[1] 97.75
```



To check the type of a vector, use `typeof()`, or more specifically

- `is.character()`
- `is.double()`
- `is.integer()`
- `is.logical()`
- `is.na()`



Coercion is a great feature in  $\mathbb{R}$  which can make coding easy, but may also have unintended consequences.

- All elements in an atomic vector must be the same type
- If you attempt to combine different types in an atomic vector they will be coerced to the most flexible type
- **Most to least flexible types** ↓
  - character
  - double
  - integer
  - logical

- When a logical vector is coerced to numeric (double or integer), TRUE = 1 and FALSE = 0

```
> x <- c("abc", 123)
```

```
> typeof(x)
```

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

You can explicitly coerce using `as.character()`, `as.double()`, `as.integer()`, and `as.logical()`



- A quick way to figure out what data structure an object is composed of is to use `str()`, which is short for structure
- `str()` provides a concise description for any R data structure



- The syntax is awkward and takes some time to get used to
- Once you understand the sequence of events in conditional subsetting, it will feel more natural
- Try to figure out what is happening in the following example:

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))  
[1] 99.1 98.2 97.3 96.4
```

```
> myAtomicVector_01[myAtomicVector_01 > 98]  
[1] 99.1 98.2
```

What is actually happening in the last slide:

- 1 The `myAtomicVector_01 > 98` part of the statement tests each element of the vector to see whether it is `> 98` and returns a `LOGICAL` value for each test which, in this case, returns the logical vector `(T T F F)`
- 2 The vector `(T T F F)` is passed to `myAtomicVector_01`, which returns the first two elements and omits the final two
  - An equivalent statement would be  
`myAtomicVector_01[c(T, T, F, F)]`



---

Function	Action
----------	--------

---

<code>seq(from, to, by)</code>	Creates a vector of numbers from <code>from</code> to <code>to</code> in increments of <code>by</code>
--------------------------------	--

<code>rep(x, times)</code>	Creates a vector that repeats the values in <code>x</code> exactly <code>times</code> number of times
----------------------------	---

<code>x + (-, /, *) y</code>	For <code>x</code> and <code>y</code> of the <i>same length</i> , calculates a vector of the same length where each entry is the <b>entry-wise</b> summation (subtraction, division, or product) of <code>x</code> and <code>y</code>
------------------------------	---

---





- If you would like to create a vector that is a sequence of numbers from  $x$  to  $y$  that increase by exactly one, then you can simply write

$x:y$

- `rep()` can be applied to a `seq()`, providing a flexible means to create sequences with repeating patterns.

## Example:

```
> rep(seq(1, 1.3, .1), 2)
[1] 1.0 1.1 1.2 1.3 1.0 1.1 1.2 1.3
```

# Example



```
> x <- rep(c(1,2), 3)
```

```
> y <- seq(from = .5, to = 3, by = .5)
```

```
> x
```

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

```
> y
```

```
[1] 0.5 1.0 1.5 2.0 2.5 3.0
```

```
> x+y
```

```
[1] 1.5 3.0 2.5 4.0 3.5 5.0
```

```
> x/y
```

```
[1] 2.0000000 2.0000000 0.6666667 1.0000000 0.4000000 0.6666667
```

# A List of Logical Operators



Operator	Description
<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal to
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal to
<code>==</code>	Exactly equal to
<code>!=</code>	Not equal to
<code>!x</code>	Not $x$
<code>x   y</code>	$x$ or $y$
<code>x &amp; y</code>	$x$ and $y$
<code>isTRUE(x)</code>	Test if $x$ is TRUE



- A name is a vector **attribute**
- Can be identified using the `names()` function

```
> x <- c(1, 2, 3)
> names(x)
NULL
```

```
> x <- c(1, 2, 3); names(x) <- c("a", "b", "c")
> names(x)
[1] "a" "b" "c"
```

```
> x <- c(a = 1, b = 2, c = 3)
> names(x)
[1] "a" "b" "c"
```

```
> x <- c(a = 1, b = 2, 3)
> names(x)
[1] "a" "b" ""
```

## Part II: Factors



- A factor is a vector of elements from a discrete set, and is used to store categorical (ordinal or nominal) data
- Factors are built on top of **integer vectors** using two attributes:
  - 1 The `class()` 'factor': makes them behave differently from regular integer vectors
  - 2 The `levels()`: define the discrete set of permissible values



```
> x <- factor(c("M", "F", "F", "M"))
```

```
> x
```

```
[1] M F F M
```

```
Levels: F M
```

```
> class(x)
```

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

```
> typeof(x)
```

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



- Although we (intelligent humans) have an inherent ability to understand the ordering of the ordinal categories below,  $\mathbb{R}$  does not, and unless told, will treat them as nominal categorical variables
- Nominal (unordered) factors are sorted automatically by  $\mathbb{R}$ , e.g., alphabetically, numerically, etc.
- **Note:** The terms *ordered* and *sorted* are **not** synonymous here





```
> bodyType <- factor(c("healthy", "healthy", "healthy", "obese", ...  
"overweight", "overweight", "skinny"))  
> bodyType  
[1] healthy    healthy    healthy    obese      overweight overweight skinny  
Levels: healthy obese overweight skinny  
  
> levels(bodyType)  
[1] "healthy"    "obese"      "overweight" "skinny"  
  
> str(bodyType)  
Factor w/ 4 levels "healthy","obese",...: 1 1 1 2 3 3 4  
  
> bodyType < "obese"  
[1] NA NA NA NA NA NA NA  
Warning message:  
In Ops.factor(bodyType, "obese") : < not meaningful for factors
```



As seen in the previous example,

- Even though nominal factors are ordered due to the underlying integer mapping, logical comparisons based on levels fail
- Nominal factors *can* be filtered if we access the underlying integer mapping, but weird results may arise



- We can create ordinal factors by including the option `ordered = TRUE`
- By creating an ordinal set of factors, we are telling R to explicitly use the ordering we are providing
- Let's examine a messier version of `bodyType`, where instead of the body type being explicit (e.g., “obese”), the body types are coded for brevity

```
(bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),  
  levels = c("s", "h", "ov", "ob"),  
  labels = c("Skinny", "Healthy", "Overweight", "Obese"),  
  ordered = TRUE))
```



Let's examine exactly what is being executed

- ❶ `c("h", "h", "h", "ob", "ov", "ov", "s")` is what we want to classify as a factor
- ❷ `levels = c("s", "h", "ov", "ob")` provides the levels
  - Omitting this enables R to order the levels itself
- ❸ `labels = c("Skinny", "Healthy", "Overweight", "Obese")` are the *nice* labels we want to see instead of the more obscurely-coded factors
  - `labels` are mapped directly to `levels`
- ❹ `ordered = TRUE` instructs R to order the factors according to `levels`



```
(bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),
                    levels = c("s", "h", "ov", "ob"),
                    labels = c("Skinny", "Healthy", "Overweight", "Obese")
                    ordered = TRUE))

> levels(bodyType)
[1] "Skinny"      "Healthy"      "Overweight"   "Obese"

> str(bodyType)
Ord.factor w/ 4 levels "Skinny"<"Healthy"<..: 2 2 2 4 3 3 1

> bodyType < "Obese"
[1] TRUE TRUE TRUE FALSE TRUE TRUE TRUE

> bodyType[bodyType < "Obese"]
[1] Healthy Healthy Healthy Overweight Overweight Skinny
Levels: Skinny < Healthy < Overweight < Obese
```



Use the following code to create the variable `myCyl` using the dataset `mtcars`

```
myCyl <- mtcars$cyl
```

- 1 Create an ordered factor from `myCyl`, mapping the levels to 'Small', 'Medium' and 'Large'
- 2 How many observations have cylinders  $\leq$  'Medium'?



- Complete the Computational Assignment [here](#).

**Due:** Next Tuesday at the beginning of class.