

# Data Structures I: Vectors, Matrices, and Arrays



UNIVERSITY OF  
SAN FRANCISCO

Abbie M. Popa

BSDS 100 - Intro to Data Science with R



- Vectors
- Matrices



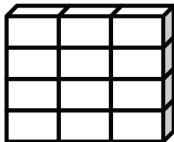
- A **data structure** is a format or organization of data in software that enables efficient use.
- Every programming language has its own types of data structures
- In  $\mathbb{R}$ , you can create your own type of data structure; however, there are some that are automatically recognized by the software.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



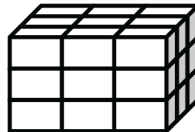
(a) Vector



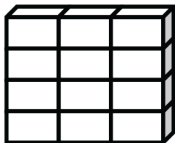
(b) Matrix



(c) Array

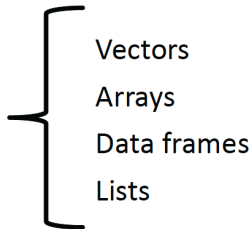


(d) Data frame



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 suffix to get integers instead of doubles
```

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

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

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





- Coding/Debugging, continue to practice
- If you need more  $\mathbb{R}$  Markdown Practice try the second activity (let me know how it goes!):

[https://github.com/abbiepopa/BSDS100/blob/master/class\\_code/More\\_Practice\\_Knitting.pdf](https://github.com/abbiepopa/BSDS100/blob/master/class_code/More_Practice_Knitting.pdf)

- Will continue with vectors and data structures today!



- Data structures hold data (numbers, strings, logicals...)
- A vector is one-dimensional (think of it like a shopping list), and can only hold one type of data
- We build a vector by concatenating (connecting) items of data with `c()`
  - e.g., `number_vector <- c(1, 2.4, -88)`

# 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 use 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
```



- 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("Sarah", "Terence", "Habibi")
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





Why did (2) `myFavNum + myNums` work?

R "recycles" the shorter vector to get the length of the longer vector

Try this:

```
my_n1_vector <- 5
my_n3_vector <- 1:3
my_n7_vector <- c(3, 7, 28, 43, 1, 5.5, 19)
my_n9_vector <- sample(1:100, 9)
```

- 1 What happens if you add `my_n1_vector` to `my_n3_vector`?
- 2 What happens if you add `my_n3_vector` to `my_n9_vector`?
- 3 What happens if you add `my_n7_vector` to `my_n9_vector`?



```
# What happens when you add a vector of length 1 to  
# a vector of length 3?
```

```
> my_n1_vector + my_n3_vector  
[1] 6 7 8
```

```
# The single item is added to each of the three  
# elements of the vector of length 3.
```



```
# What happens when you add a vector of length 3 to  
# a vector of length 9?
```

```
> my_n3_vector + my_n9_vector  
[1] 62  4 50 27 17 47 87 43 99
```

```
# The three elements are added (in order) to the first  
# three, second three, and third three elements of the  
# length nine vector
```



```
# What happens when you add a vector of length 7 to a vector of  
# length 9?
```

```
> my_n7_vector + my_n9_vector  
[1] 64.0 9.0 75.0 69.0 16.0 49.5 105.0 44.0 103.0
```

Warning message:

```
In my_n7_vector + my_n9_vector :  
  longer object length is not a multiple of shorter object length
```

```
# The first two elements of the length-7 vector will be recycled,  
# the last five will be used once
```



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: Matrices and Arrays



- By giving an atomic vector a dimension attribute, it behaves like a multi-dimensional array
- A special case of the array is a matrix, a two-dimensional array
- A matrix has 2 dimensions, and an array has  $n \geq 2$  - dimensions.
- Matrices and arrays are created with `matrix()` and `array()`



```
> x <- matrix(1:10, nrow = 2, ncol = 5)
# can drop nrow and ncol to shorten but keep in this order

> x
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	3	5	7	9
[2,]	2	4	6	8	10



```
> y <- array(1:12, c(2, 3, 2))
```

```
> y
```

```
, , 1
```

```
      [,1] [,2] [,3]
```

```
[1,]      1      3      5
```

```
[2,]      2      4      6
```

```
, , 2
```

```
      [,1] [,2] [,3]
```

```
[1,]      7      9     11
```

```
[2,]      8     10     12
```



---

## 1-D Function    n-D Functions

---

<code>length()</code>	<code>nrow()</code> , <code>ncol()</code> , <code>dim()</code>
<code>names()</code>	<code>rownames()</code> , <code>colnames()</code> , <code>dimnames()</code>
<code>c()</code>	<code>cbind()</code> , <code>rbind()</code>

---

**Note:** a matrix or array can also be one-dimensional, e.g., an object that is defined as a matrix is permitted to only have one column or one row; although they may look and behave alike, a vector and a one-dimensional matrix behave differently and may generate strange output when using certain functions, e.g., `tapply()`

# Common R Functions for Working with Data



Function	Purpose
----------	---------

<code>length(object)</code>	Number of elements
<code>dim(object)</code>	Dimensions of an object
<code>str(object)</code>	Structure of an object
<code>class(object)</code>	Type of an object
<code>c(object, object, ....)</code>	Combines objects into a vector
<code>cbind(object, object, ....)</code>	Combines objects as columns
<code>rbind(object, object, ....)</code>	Combines objects as rows
<code>object</code>	Prints the object
<code>head(object)</code>	Prints the first part of the object
<code>tail(object)</code>	Prints the last part of the object
<code>ls()</code>	Lists current objects
<code>rm(object, object, ...)</code>	Deletes one or more objects





- When subsetting matrices, we specify row, then column, (e.g., `my_matrix[2, 3]` will get you the second row of the 3rd column.
- We can also select multiple rows or columns, as with vector indices
- Leaving one of the positions blank will retrieve all of that dimension (e.g., `my_matrix[, 3]` will give you all of the rows for the 3rd column.
- Omitting the comma is not recommended, and will get you values based on a different indexing system



- For a 3-dimensional array you would order by row, column, layer
- More than 3 dimensions is hard to visualize, but you will call the index based on the order you specified when making the array
- A useful feature of indexing, is if you give indexes out of order you can rearrange the output (for example put the 3rd column before the 1st column)