Data Types and Data Structures in R

Anthony Chau

UCI Center for Statistical Consulting

2021/01/12 (updated: 2021-02-09)

Learning Objectives

1. Identify and give examples of the data types in R
2. Know how to represent missing values and special values
3. Know how vectorized operations work
4. Know how vector recycling works
5. Know how vector coercion works
6. Compare and contrast each of the data structures
7. Know how to assign names for each data structure

Motivation

- Going back to the **storage** problem: how do we store data in a format that R recognizes
- **Data types** allow us to group related data
- **Data structures** provide an interface to organize, manage, and store our data.
- The data types and data structures define how we interact with data
- Many programming problems occur because of incompatible data types and data structures so mastery of this section is important

Goals moving forward:

- Introduce fundamental data types and data structures
- Build up to the `data frame` data structure
- Recognize the relationship between the different data structures

Data Types

Type	Description	Example
integer	Integers. Suffix an integer with L to create an integer	1L, 2L, 100L
double	Decimals, fractions. Can include integers	1.3, 4/9, 5,
logical	Denote if a condition is true or false. Only two possible values; TRUE, FALSE	TRUE, FALSE, T, F
character	Any kind of text. Wrap the text in "" or ''. Choose one and be consistent.	"hello", "welcome home", 'What's your name?'
NULL	Used to represent an empty vector or an absent vector	NULL

Checking object type

- Check the type of an object with the `typeof()` function.

```
typeof(1L)
#> [1] "integer"
typeof(1.5)
#> [1] "double"
typeof("hello")
#> [1] "character"
typeof(TRUE)
#> [1] "logical"
typeof(NULL)
#> [1] "NULL"
```

Special values: NA

- There are two special values to be aware of in R: `NA` and `NaN`
- `NA` indicates a missing value
- `NA`'s "propagates" when doing performing operations with `NA`
- Many functions have common argument `na.rm=TRUE` to remove `NA` before performing operation
- Check if a value is NA with `is.na()` - will be very useful!

Special values: NA

```r
# NA propagation
1 + NA
#> [1] NA
NA * 5
#> [1] NA

# NA is removed so x = c(0,1,2,3,4)
# then, the mean is computed
mean(x = c(0,1,2,3,4,NA), na.rm = TRUE)
#> [1] 2

# check if something is NA
is.na(NA)
#> [1] TRUE
```

Special values: NaN

- `NaN` indicates an invalid math operation (ie: divide by 0, subtract by infinity)
- Check if a value is `NaN` with `is.nan()`

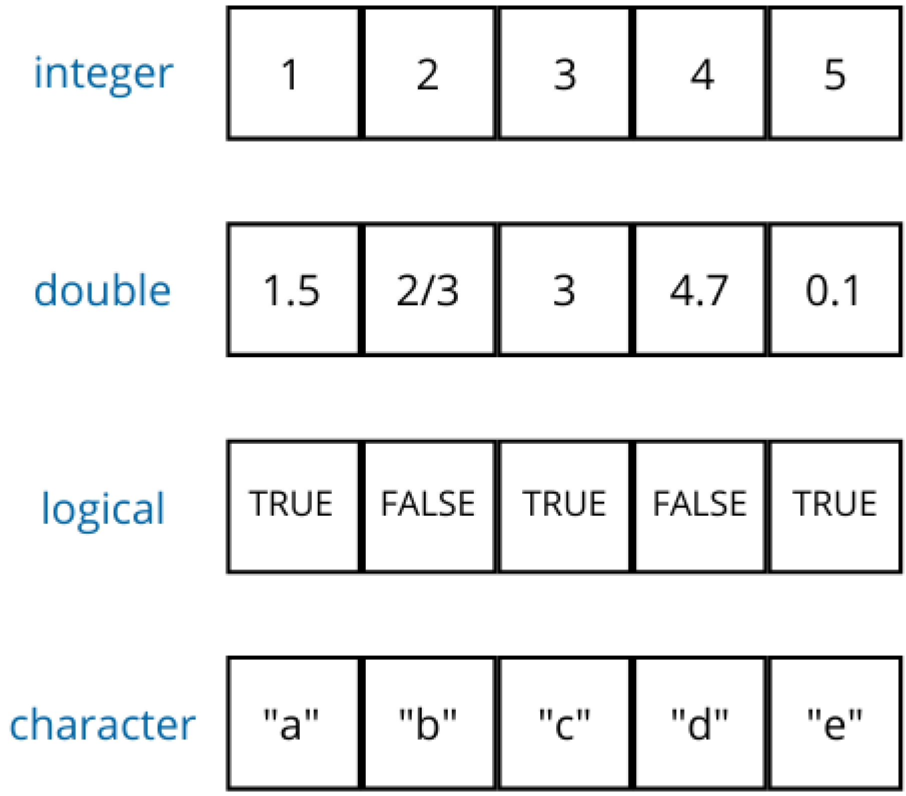
```
# examples where NaN can occur
sqrt(-1)
#> Warning in sqrt(-1): NaNs produced
#> [1] NaN
0/0
#> [1] NaN
Inf - Inf
#> [1] NaN

# Check if something is NaN
is.nan(NaN)
#> [1] TRUE
```

Vectors

vector: a sequence of values where each value must be of the same type

- Vectors are objects
- Vectors are everywhere!
- Vectors are the building blocks of other data structures
- We can create a vector of each data types: `integer`, `double`, `character`, `logical`

Vector mental model



Vector structure

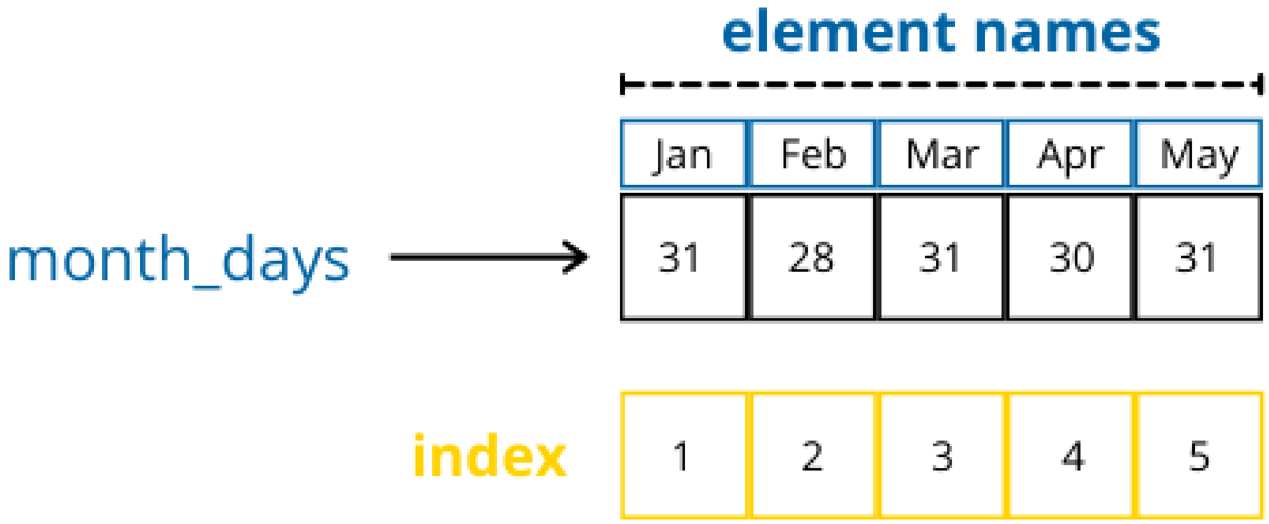
Element names

- Each element of a vector can be assigned a name as well. Call this the **element name** to distinguish from **name**
- A typical use case for this is to label a numerical value with informative text

Indices

- The **indices** give the position of an element
- Indices are integers that always start at 1 and increment by 1 to the length of the vector

Detailed Vector mental model



Setting element names

- Set and access element names for a vector with the `names()` function.
- Note that the element names of a vector is another vector.

```
month_days <- c(31, 28, 31, 30, 31)
month_days
#> [1] 31 28 31 30 31

# set element names
names(month_days) <- c("Jan", "Feb", "Mar", "Apr", "May")
names(month_days)
#> [1] "Jan" "Feb" "Mar" "Apr" "May"

# note the element names associated with each element now
month_days
#> Jan Feb Mar Apr May
#>  31  28  31  30  31

# names(month_days) is a vector
is.vector(names(month_days))
#> [1] TRUE
```

Setting element names

- Alternatively, specify a `name = value` pair when you create the vector

```r
month_days <- c("Jan" = 31,
                "Feb" = 28,
                "Mar" = 31,
                "Apr" = 30,
                "May" = 31)
month_days
#> Jan Feb Mar Apr May
#>  31  28  31  30  31

# names(month_days) is a vector
is.vector(names(month_days))
#> [1] TRUE
```

Useful functions for vectors

- Many functions expect a vector as an argument.

```
x ← c(-2, 0, 2, 4)

# compute sum of all elements in a vector
sum(x)
#> [1] 4
# compute mean of all elements in a vector
mean(x)
#> [1] 1
# compute standard devation of all elements in a vector
sd(x)
#> [1] 2.581989
# get minimum value in a vector
min(x)
#> [1] -2
# get minimum value in a vector
max(x)
#> [1] 4
# get the length of a vector
length(x)
#> [1] 4
```

Useful functions for vectors …

- `seq()` generates a sequence of values
- `rep` repeats elements in a vector

```
# use the colon to get a sequence of numbers
x ← c(1:10)
x
#>  [1]  1  2  3  4  5  6  7  8  9 10

# or use seq() for more flexibility
a ← seq(from = 1, to = 10, by = 1)
a
#>  [1]  1  2  3  4  5  6  7  8  9 10

y_alternate ← rep(x = c(1,2), times = 5)
y_alternate
#>  [1] 1 2 1 2 1 2 1 2 1 2

y_element_wise ← rep(x = c(1,2), each = 5)
y_element_wise
#>  [1] 1 1 1 1 1 2 2 2 2 2

y_same_length ← rep(x = c(1,2), length.out = 5)
y_same_length
#> [1] 1 2 1 2 1
```

Vector Operations

- Arithmetic (`+`, `-`, `*`, `/`) is done element-wise with vectors.

Code Example

```
x <- c(1, 2, 3)
y <- c(1, 4, 9)

x + y
#> [1]  2  6 12

y - x
#> [1] 0 2 6

x * y
#> [1]  1  8 27

y / x
#> [1] 1 2 3
```

- When vectors are the same length (have the same number of elements), arithmetic is intuitive.

Vectorized operations

- Element-wise operations are also called **vectorized** operations
- The idea is that I don't need to explicitly specify an operation on each element of a vector - the operation is applied to each element
- **Vectorized** operations simplify our code

Example - Square root

```
x <- c(1, 4, 9, 16, 25)
x
#> [1]  1  4  9 16 25

sqrt(x)
#> [1] 1 2 3 4 5
```

Vectorized operations

- *Vectorized* operations will save us a lot of time and effort when our operations become complex

Without vectorized operations

```r
x <- c(1, 4, 9, 16, 25)
n <- length(x)
result <- rep(NA_integer_, n)
for (i in seq_len(n)) {
  result[i] <- x[i] ^ (1/2)
}
result
#> [1] 1 2 3 4 5
```

Vector Recycling

- It turns out that you can perform vector operations on vectors of unequal length
- R deals with unequal length by "recycling" the shorter vector to the length of the longer vector

Code Example

```
# note: x is a vector - a length one vector!
x <- 5
y <- c(1, 2, 3)

# behind the scenes, R recycles the value 5 until the
# vector x looks like this: c(5, 5, 5)
# then, it is the usual element-wise operation
x * y
#> [1]  5 10 15

x + y
#> [1] 6 7 8
```

Vector Recycling

- In theory, vector recycling can work when you have any pairs of varying vector lengths.
- But, the behavior is hard to predict and keep track of.
- I suggest to stick with the case where *one vector is length 1 and the other vector is some arbitrary length*

Code Example

```
x ← c(1, 2)
y ← c(2, 4, 6, 8, 10)

# x becomes: c(1, 2, 1, 2, 1)
# so x + y = c(1, 2, 1, 2, 1) + c(2, 4, 6, 8, 10)
x + y
#> Warning in x + y: longer object length is not a multiple of shorter object
#> length
#> [1]  3  6  7 10 11
```

- Notice the warning - it's encouraging us to try to keep the longer vector a multiple of the shorter vector

Type Coercion

- Recall that all elements in a vector must be of the same type
- If we try to circumvent this property, R converts all elements to the same type through **coercion**.

```
# integer and double
x <- c(1L, 2.3)
x
#> [1] 1.0 2.3
typeof(x)
#> [1] "double"

# character and double
y <- c("1", 1)
y
#> [1] "1" "1"
typeof(y)
#> [1] "character"

# double and logical
z <- c(1, TRUE)
z
#> [1] 1 1
typeof(z)
#> [1] "double"
```

Type Coercion Rule

- One rule summarizes what happens when combining different types

Type Coercion rule: character → double → integer → logical

- Types downstream on the chain are converted to the highest type on the chain

Type Coercion Rule

- Notice that the most general type (`character`) takes precedence - the character type can sensibly represent data of the `double`, `integer`, or `logical` class

```
# character
"uci"
# double as character
"1.5"
# integer as character
"1"
# logical as character
"TRUE"
```

Caution for Type Coercion

- Be mindful of type coercion - it may happen silently without your awareness

Common situations where type coercion can occur

- You use data from multiple sources - certain variables may be stored differently
- Some functions may need to convert to a specific type to perform some task

Why use a vector?

- *Consistency*: data is all of the same type; allowable operations are defined accordingly
- For example, how is arithmetic defined for vectors with a mix of character and numeric values?
- Enforcing homogeneous type will make our code more predictable and manageable

Checkpoint Question 1 - Vectors

Consider the following R code. Which of the following is **not** a vector (if any)?

```
x <- 10
y <- c(10, 20, 30)
y <- y - x
names(y) <- c("zero", "ten", "twenty")
```

A. `x`

B. `y`

C. `names(y)`

D. `x`, `y`, `names(y)` are all vectors

Checkpoint Question 2 - Vectors

Consider the following R code. What is `z`?

```
x <- 100
y <- c(1, 2, 3)
z <- x * y
z
```

A. `c(1, 2, 3)`

B. `c(100, 2, 3)`

C. `c(100, 200, 300)`

D. `c(1, 2, 300)`

Checkpoint Question 3 - Vectors

Consider the following R code. What type is `x`?

```r
x <- c(1, "1", TRUE)
typeof(x)
```

A. `integer`

B. `double`

C. `character`

D. `logical`

Checkpoint Question 4 - Vectors

Consider the following R code. What is `y`?

```
subtract_five <- function(v){
  v - 5
}

x <- c(5, 10, 15)
y <- subtract_five(x)
y
```

A. `v - 5`

B. `c(0, 5, 10)`

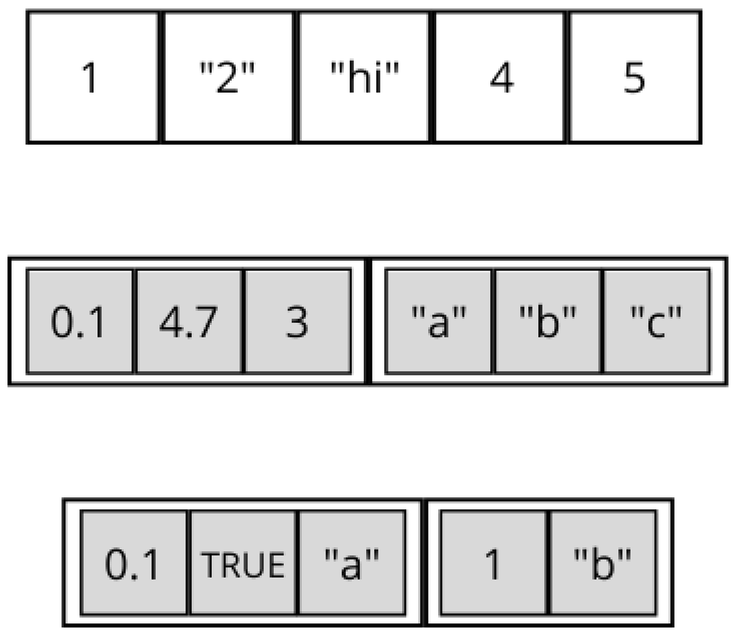
C. `c(0, 10, 15)`

D. `c(5, 10, 15)`

Lists

- **list**: a sequence of values where each value can have different types
- Lists are objects
- Lists are the most flexible data structure
- Think of lists as generalizations of vectors
    - **Vectors** hold *homogeneous* data
    - **Lists** hold *heterogeneous* data
- Since lists are more general and heterogeneous, it is harder to classify them like with vectors

List mental model



Lists in R

- Create lists with the `list()` function. Separate values with a comma
- Check that an object is a list with `is.list()`
- Just like with vectors, we can name each element of a list with `names()`

```
my_list <- list(1L, "hello", TRUE, 1.5)
my_list
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] "hello"
#>
#> [[3]]
#> [1] TRUE
#>
#> [[4]]
#> [1] 1.5

# names(my_list) is still a vector
names(my_list) <- c("integer", "character", "logical", "double")
names(my_list)
#> [1] "integer"   "character" "logical"   "double"
```

Why use a list?

- The raw data you receive is "hierarchical"

```
{
name: Anthony
academic_year: "2018-2019"
term: "fall"
courses: [
    {
      course_name: "English 1"
      units: 4
      grade: "B"
    },
    {
      course_name: "Economics 1"
      units: 4
      grade: "C"
    },
    {
      course_name: "Statistics 1"
      units: 4
      grade: "B+"
    }
  ]
}
```

Why use a list?

- Apply common operations to data from different time periods

```
# read in data
lab_Jan2020 <- read.csv(file = "lab_results_Jan-2020.csv")
lab_Feb2020 <- read.csv(file = "lab_results_Feb-2020.csv")
lab_Mar2020 <- read.csv(file = "lab_results_Mar-2020.csv")

lab_data_all <- list(lab_Jan2020, lab_Feb2020, lab_Mar2020)

clean_data(lab_data_all)
plot_data(lab_data_all)
build_model(lab_data_all)
```

Checkpoint Question 1 - Lists

What are some reasons you would you use a list over a vector?

A. Lists can hold heterogeneous data

B. You want to process a group of related data

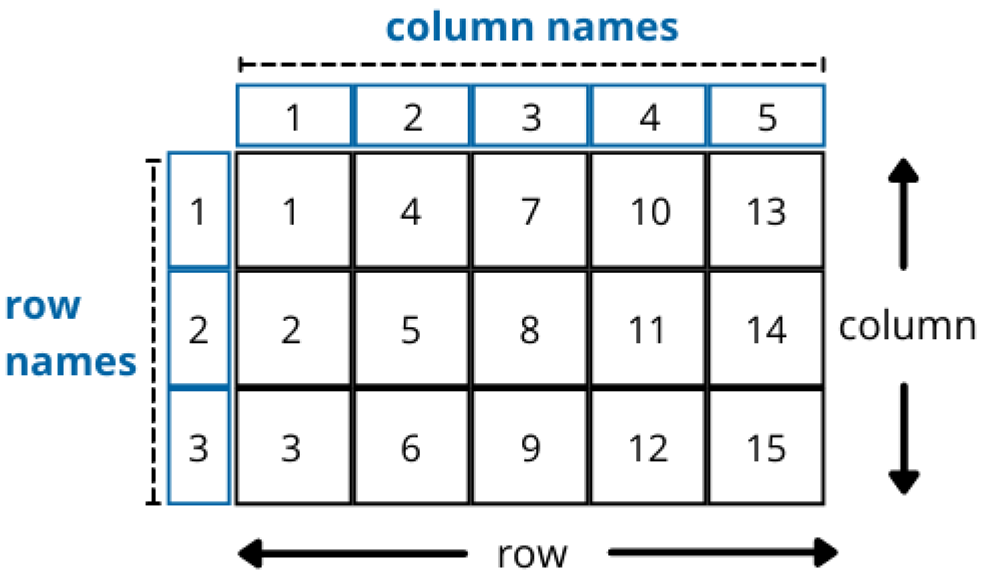
C. Your data is naturally hierarchical

D. All of the above

Matrices

- **matrix**: a 2-dimensional rectangular table of values where every value must be the same type
- Matrices are objects
- Matrices hold **homogeneous** data
- Commonly, you use matrices with numbers
- Every row and column in a matrix is a vector
- Since we are in 2D, we use **rows** and **columns** to index a matrix

Matrix mental model



Matrices in R

- Create a matrix with the `matrix()` function
- For a matrix, we need to provide some data to fill the matrix
- Check that an object is a matrix with `is.matrix()`

Let's create a `3 x 5` matrix and populate with values from 1 to 15.

```
# recall the colon shortcut to create a sequence of numbers
x ← c(1:15)
x
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

# with the vector x, create a matrix with 3 rows and 5 columns
m ← matrix(data = x, nrow = 3, ncol = 5)

# notice how the values are filled up
m
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    4    7   10   13
#> [2,]    2    5    8   11   14
#> [3,]    3    6    9   12   15
```

Specify how values are filled in matrices

- Note how the values are filled with `matrix()`
- Add `byrow=TRUE` as an argument to `matrix()` to fill the values by row. The default is to fill by column

```
x <- c(1:15)
# note how I don't need to specify byrow=FALSE
m_by_column <- matrix(data = x, nrow = 3, ncol = 5)
m_by_row <- matrix(data = x, nrow = 3, ncol = 5, byrow = TRUE)

# values filled by column
m_by_column
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    4    7   10   13
#> [2,]    2    5    8   11   14
#> [3,]    3    6    9   12   15
# values filled by row
m_by_row
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
#> [2,]    6    7    8    9   10
#> [3,]   11   12   13   14   15
```

Fill by row vs fill by column

byrow = TRUE

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

fill matrix by row

byrow = FALSE

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

fill matrix by column

Matrix row names and column names

- Set and view the row names with `rownames()`
- Set and view the column names with `colnames()`
- View both row names and column names with `dimnames()`
- Alternatively, set dimension names when creating the matrix

Matrix row names and column names

```r
m <- matrix(data = c(1:15), nrow = 3, ncol = 5, byrow = TRUE)

rownames(m) <- c("r1", "r2", "r3")
rownames(m)
#> [1] "r1" "r2" "r3"

colnames(m) <- c("c1", "c2", "c3", "c4", "c5")
colnames(m)
#> [1] "c1" "c2" "c3" "c4" "c5"

dimnames(m)
#> [[1]]
#> [1] "r1" "r2" "r3"
#>
#> [[2]]
#> [1] "c1" "c2" "c3" "c4" "c5"

m
#>    c1 c2 c3 c4 c5
#> r1  1  2  3  4  5
#> r2  6  7  8  9 10
#> r3 11 12 13 14 15
```

Matrix dimensions

- Get number of rows with `nrow()`
- Get number of columns with `ncol()`
- Get dimension of matrix with `dim()`

```
m <- matrix(data = c(1:15), nrow = 3, ncol = 5, byrow = TRUE)
m
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
#> [2,]    6    7    8    9   10
#> [3,]   11   12   13   14   15

nrow(m)
#> [1] 3

ncol(m)
#> [1] 5

dim(m)
#> [1] 3 5
```

Useful functions for matrices

Function	Description
t()	Transpose a matrix
rowMeans()	Compute the mean for each row
rowSums()	Compute the sum for each row
colMeans()	Compute the mean for each column
colSums()	Compute the sum for each column
rbind()	Combine objects by row
cbind()	Combine objects by column

`rbind()` and `cbind()`

- Add more rows and columns to matrix with `rbind()` and `cbind()`
- Check that the number of rows (columns) are the same for your objects to prevent unexpected behavior

```
m <- matrix(data = c(1:15), nrow = 3, ncol = 5, byrow = TRUE)

# recall vector recycling
rbind(m, c(4))
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
#> [2,]    6    7    8    9   10
#> [3,]   11   12   13   14   15
#> [4,]    4    4    4    4    4

cbind(m, c(6))
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    1    2    3    4    5    6
#> [2,]    6    7    8    9   10    6
#> [3,]   11   12   13   14   15    6
```

Checkpoint Question 1 - Matrices

What is `dim(m)`?

```
m <- matrix(c(1:8), nrow = 4, ncol = 2, byrow = TRUE)
dim(m)
```

A. `4 rows by 2 columns`

B. `2 rows by 4 columns`

Checkpoint Question 2 - Matrices

Which row is the value `6` in?

```
m <- matrix(c(1:8), nrow = 4, ncol = 2, byrow = TRUE)
m
```

A. 1st row

B. 2nd row

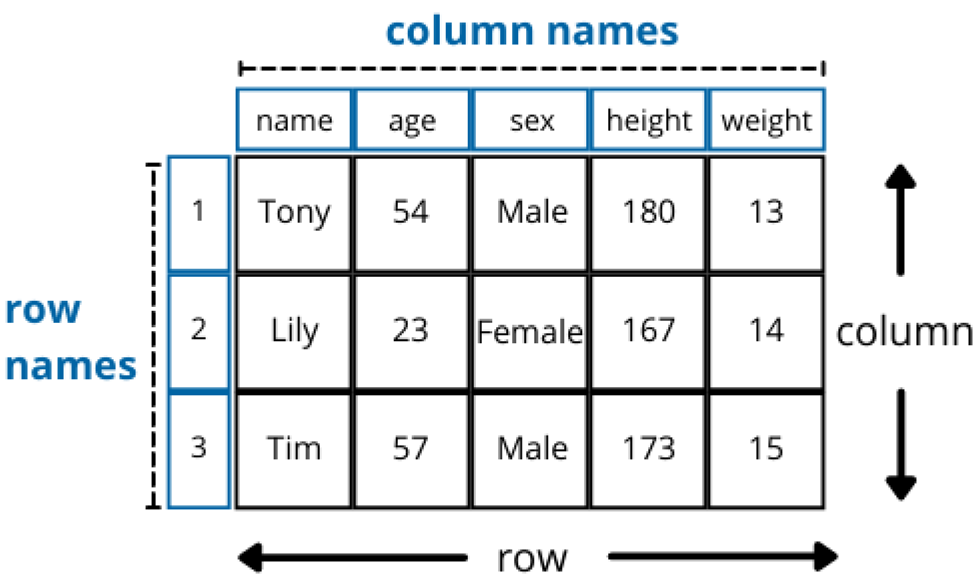
C. 3rd row

D. 4th row

Data Frames

- **data frame**: a 2-dimensional rectangular table of values where every value in a column must be the same type
- Data frames are objects
- Data frame columns hold **homogeneous** data
- The entire data frame holds **heterogeneous** data
- It turns out that a *data frame is a list of vectors*
- Data frame format is familiar - a typical csv/Excel file in the wild

Data frame mental model



Data frames in R

- Create a data frame with the `data.frame()` function
- Specify each column as `column name = vector of values`
- Check that an object is a data frame with `is.data.frame()`

```
mascots <- data.frame(name = c("Peter Anteater", "Josephine Bruin",
                               "King Triton","Tommy Trojan"),
                      age = c(56, 101, 60, 140),
                      residence = c("Irvine", "Los Angeles",
                                    "San Diego", "Los Angeles"))

mascots
#>               name age   residence
#> 1   Peter Anteater  56      Irvine
#> 2 Josephine Bruin 101 Los Angeles
#> 3      King Triton  60   San Diego
#> 4     Tommy Trojan 140 Los Angeles
```

Useful functions for data frames

- Data frames share many functions with matrices

```r
mascots <- data.frame(name = c("Peter Anteater", "Josephine Bruin",
                               "King Triton", "Tommy Trojan"),
                      age = c(56, 101, 60, 140),
                      residence = c("Irvine", "Los Angeles",
                                    "San Diego", "Los Angeles"))

nrow(mascots)
#> [1] 4
rownames(mascots)
#> [1] "1" "2" "3" "4"
ncol(mascots)
#> [1] 3
colnames(mascots)
#> [1] "name"      "age"       "residence"
dim(mascots)
#> [1] 4 3
```

Data frame and other data structures

Notice how the data frame has properties from other data structures

1. Data frame columns are vectors
2. The data frame is a list (of vectors)
3. Data frame is a 2-dimensional rectangular structure like a matrix

> Important to know the simpler data structures since the data frame is a mix and match of all of them.

Checkpoint Question 1 - Data Frames

What is the relationship between data frames, lists, and vectors?

A. All vectors are data frames

B. A data frame is a vector of lists

C. A data frame is a list of vectors

D. All lists are data frames