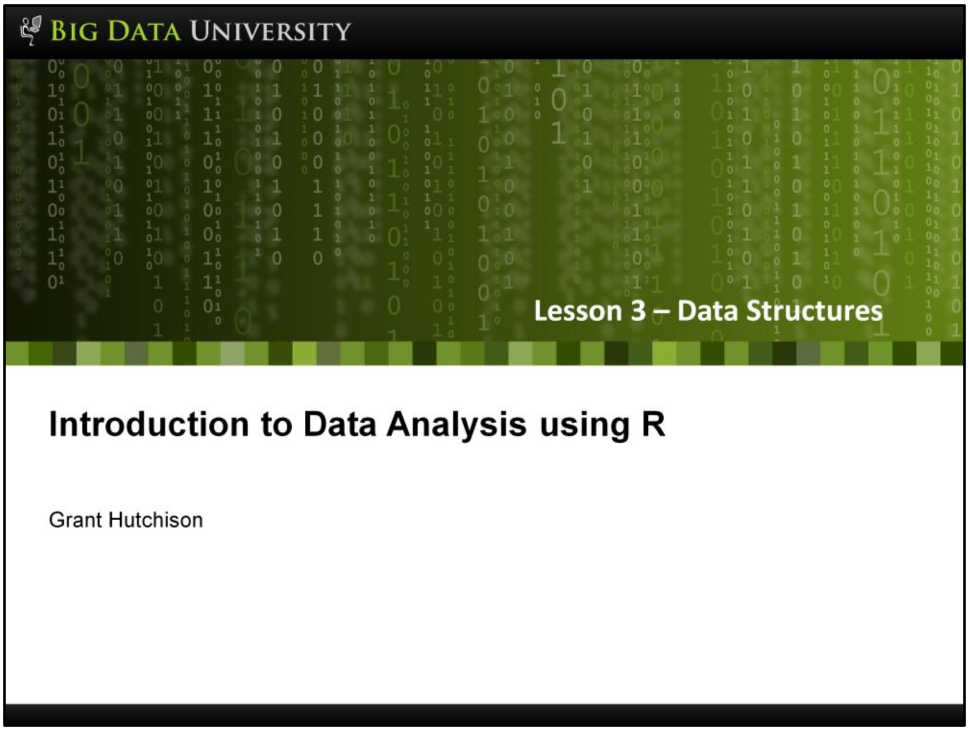


BIG DATA UNIVERSITY

Lesson 3 – Data Structures

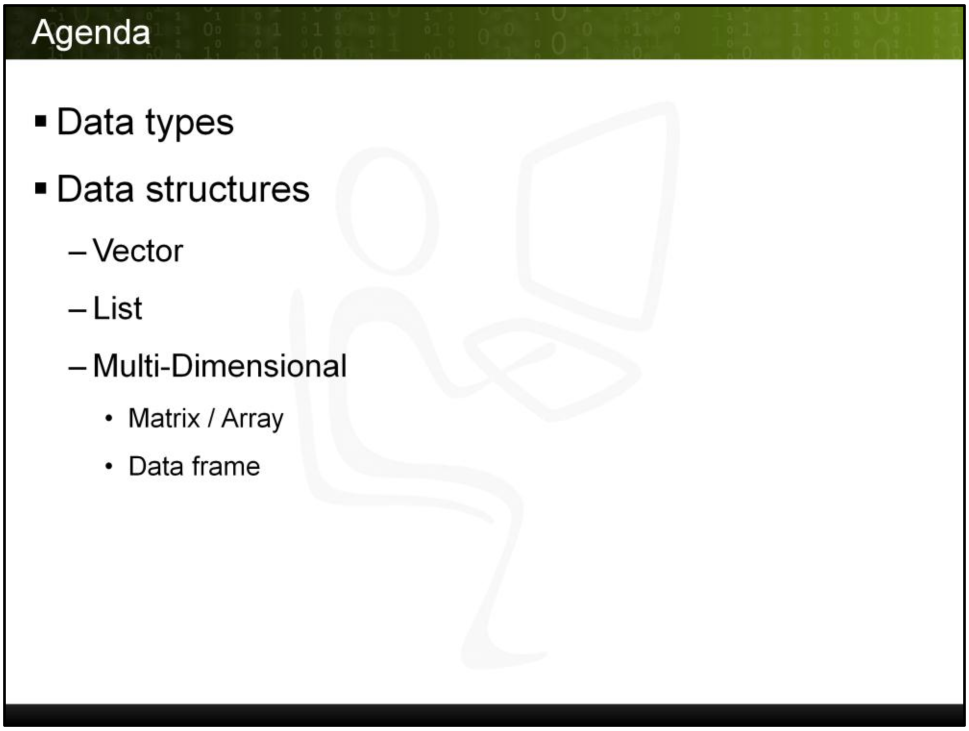
Introduction to Data Analysis using R

Grant Hutchison

Welcome to the lesson on Data Structures.

To perform any meaningful data analysis we need to collect our data into R data structures.

In this lesson we will explore the most frequently used data types and data structures.



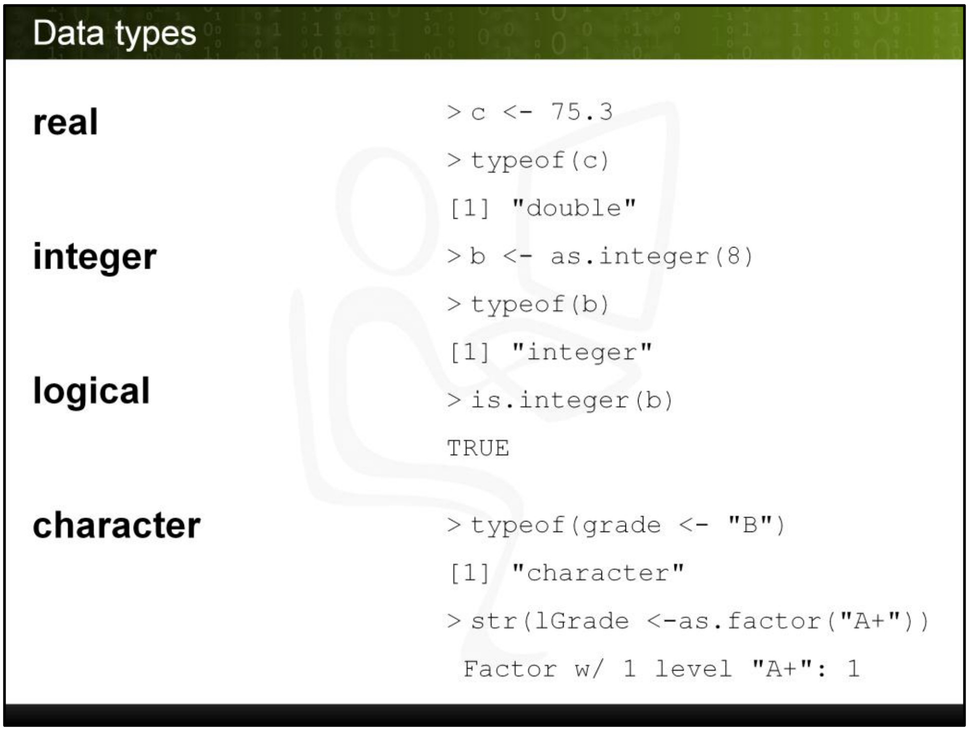
Agenda

- Data types
- Data structures
  - Vector
  - List
  - Multi-Dimensional
    - Matrix / Array
    - Data frame

R can be used to analyze many different forms of data. We will explore the built-in datatypes of R.

Data analysis usually requires an examination of large sets of similar data.

In this lesson we will explore various data structures we can use to hold and manipulate our datasets.



Data types

real

```
> c <- 75.3
> typeof(c)
[1] "double"
```

integer

```
> b <- as.integer(8)
> typeof(b)
[1] "integer"
```

logical

```
> is.integer(b)
TRUE
```

character

```
> typeof(grade <- "B")
[1] "character"
> str(lGrade <-as.factor("A+"))
 Factor w/ 1 level "A+": 1
```

R can handle many different types of data. Here we examine the commonly used: numeric, logical, and character data types.

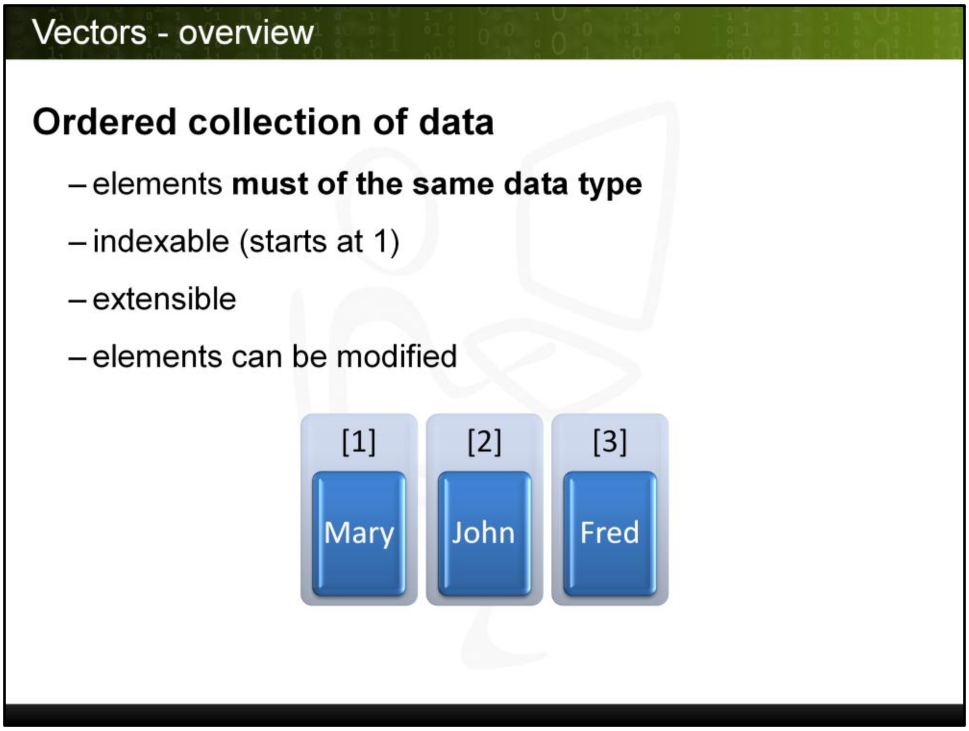
Numeric data is usually classified as either integer and real numbers.

By default R, will store numeric data as a real which is otherwise known as a double precision floating point value.

If we want our data to be stored as an integer value we use the **as.integer()** function.

Logical or boolean data is any data that represents a true or false condition. True or false values may exist within a dataset, or in this example we are using a function which returns a logical value.

Character or string data is often provided within datasets. Character data supplied as a constant can be enclosed within single or double quotation marks. If the character data represents a category then you might want R to treat it as a factor instead of a character string. We will examine the use of factors in future lessons. Here we have decided that a letter grade should be treated as a factor and not as a character string.



Vectors - overview

Ordered collection of data

- elements **must of the same data type**
- indexable (starts at 1)
- extensible
- elements can be modified

[1]	[2]	[3]
Mary	John	Fred

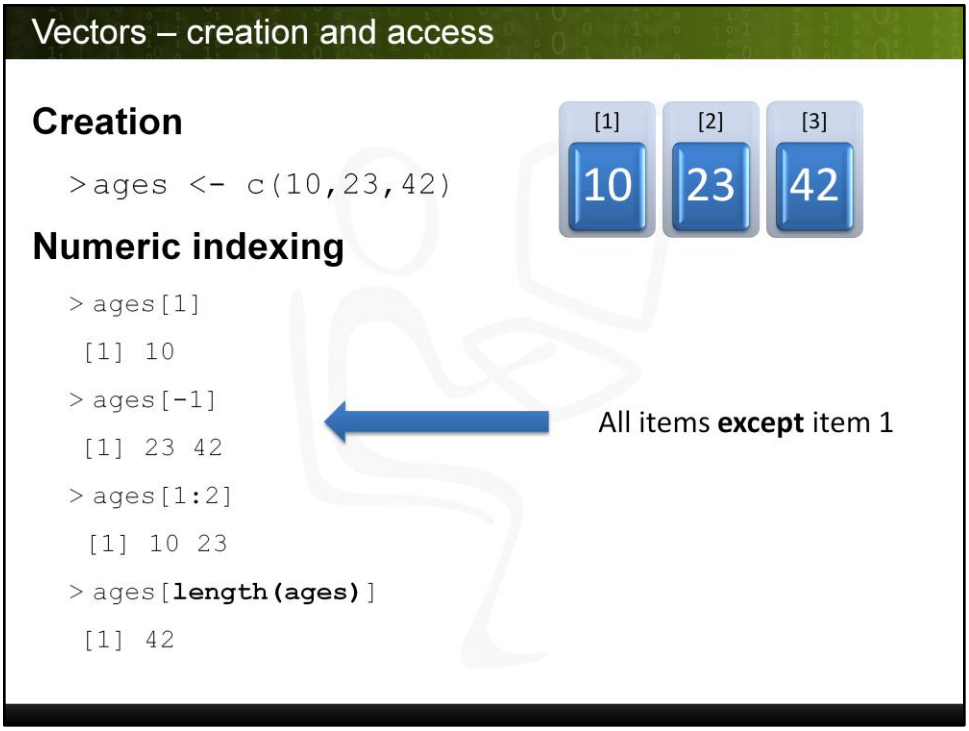
A vector is used to maintain an ordered collection of values of the same data type. Any R data type can be used for a vectors.

The elements within the vector can be accessed using indexes. The indexing starts with the value 1.

Here we have an example of a character vector of three names.

Vectors can also be extended with new elements if required.

The elements in a vector can be modified.



Vectors – creation and access

Creation

```
>ages <- c(10,23,42)
```

Numeric indexing

```
> ages[1]
 [1] 10
> ages[-1]
 [1] 23 42
> ages[1:2]
 [1] 10 23
> ages[length(ages)]
 [1] 42
```

All items **except** item 1

The most common technique of creating a vector is to use the c() function. This function will **combine** its arguments to form a new vector.

There are many methods of accessing items within a vector.

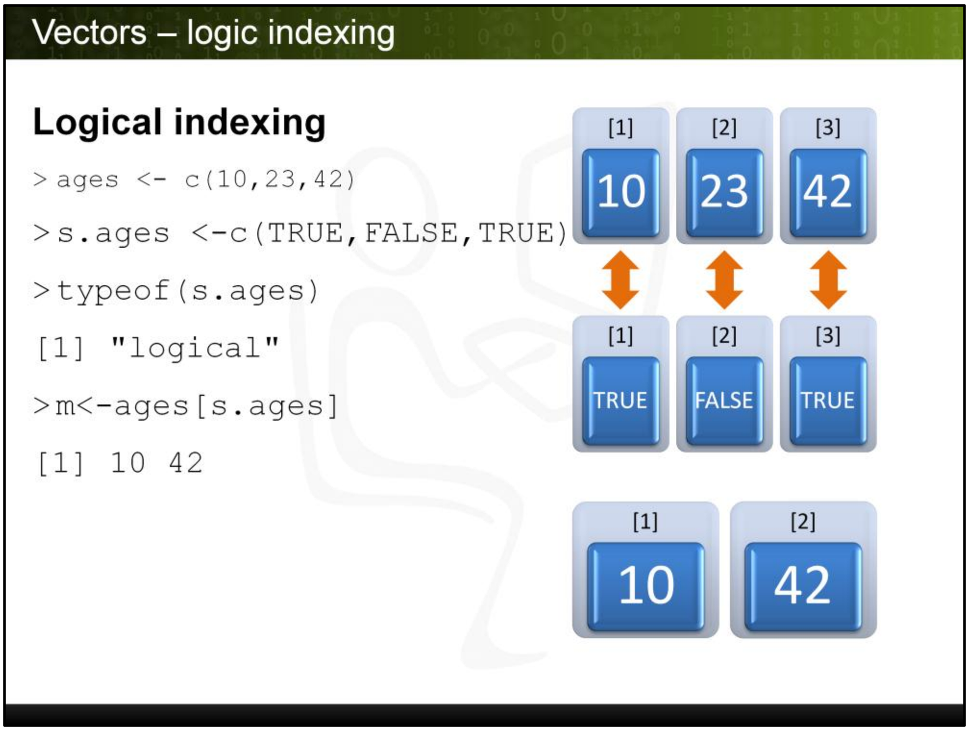
In this first example we are using a single integer as the indexing method to retrieve the first element of the ages vector. The square brackets are used to access the elements from the data structure.

Negative integer values can be used to exclude elements from the selection.

A range of values, or another vector of indexes, can be specified within the square brackets to select specific items from the data structure.

Another very useful function that works with many different data structures is the length() function.

If you pass a vector to the length() function it will return the number of elements or items in the list.



Vectors – logic indexing

Logical indexing

```
> ages <- c(10,23,42)
>s.ages <-c(TRUE,FALSE,TRUE)
>typeof(s.ages)
[1] "logical"
>m<-ages[s.ages]
[1] 10 42
```

Vectors can be created using any of the supported R datatypes.

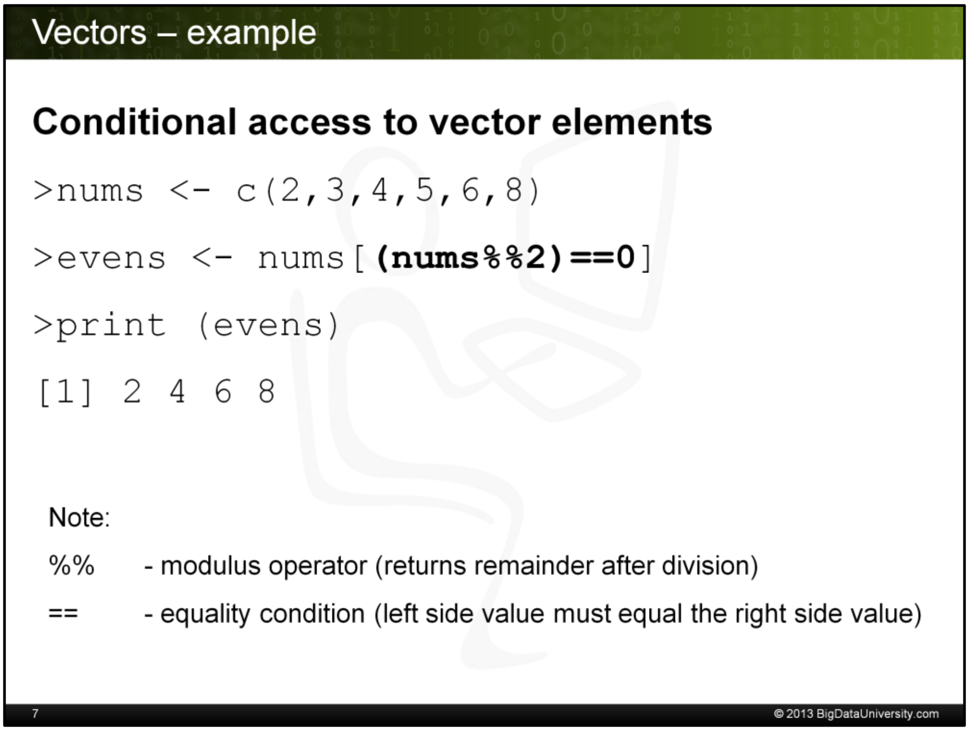
Here we have decided to create a vector of called ages.

Instead of using numeric indexing we use a logical vector to access elements of the vector.

We create a logical vector with the same number of elements called s.ages.

When we access the elements of the ages vector using a logical vector, only the values from the ages vector with corresponding TRUE values in the logical vector are returned. Our resulting vector 'm' contains the values 10 and 42 only.

We will soon discover how the conditional access methods for analyzing data structuers is a very powerful feature of R.



Vectors – example

Conditional access to vector elements

```
>nums <- c(2,3,4,5,6,8)
>evens <- nums[(nums%%2)==0]
>print (evens)
[1] 2 4 6 8
```

Note:

%% - modulus operator (returns remainder after division)

== - equality condition (left side value must equal the right side value)

7

Let's look at how we can dynamically access elements of a vector based on conditions.

We have a simple numeric vector called nums.

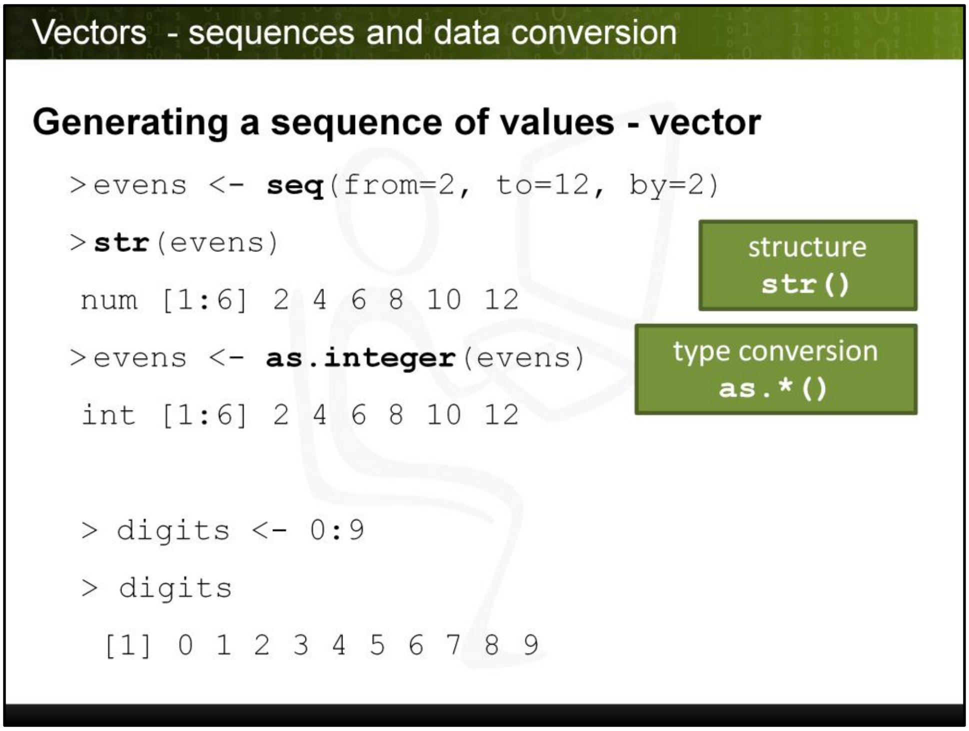
We would like to extract the even numbers from the vector and store them in a new vector.

R can handle this task using conditional access. To make this happen we will use the modulus and equality operators to help use determine if the element is even or odd.

R will apply the conditional expression to each element of the vector because the vector is referenced within the square brackets.

If the expression evaluates as true, a copy of the element will be appended to the new vector called evens.

Finally, we examine the contents of the new vector to verify that the proper values have been copied.



Vectors - sequences and data conversion

Generating a sequence of values - vector

```
>evens <- seq(from=2, to=12, by=2)
>str(evens)
 num [1:6] 2 4 6 8 10 12
>evens <- as.integer(evens)
 int [1:6] 2 4 6 8 10 12


 > digits <- 0:9
 > digits
  [1] 0 1 2 3 4 5 6 7 8 9
```

structure
str()

type conversion
as.*()

R provides a few different techniques to generate a sequence of values.

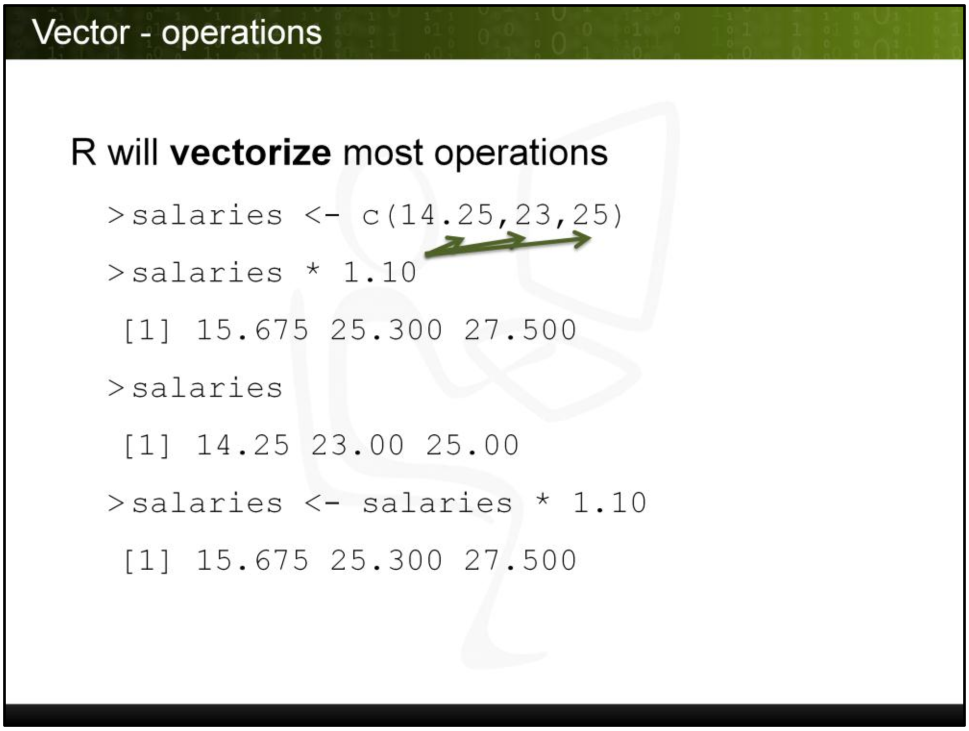
Here we create a sequence or numeric vector using the seq() or sequence function.

The function str() is used to display the structure of an object. In this case we notice that the initial structure is a vector of numbers.

We use the type conversion function called **as.integer()** to change the default double values into integers. There are various other type conversion functions available in R.

Another method of generating a simple sequence is to use the colon operator (:).

Here we are generating a vector of integers from 0 through 9.



Vector - operations

R will **vectorize** most operations

```
>salaries <- c(14.25,23,25)
>salaries * 1.10
 [1] 15.675 25.300 27.500
>salaries
 [1] 14.25 23.00 25.00
>salaries <- salaries * 1.10
 [1] 15.675 25.300 27.500
```

R is a functional language with built-in support for vector operations. Whenever possible it is best to utilize vector operations instead of iteration or loops.

R will use very efficient native libraries for many vector operations and therefore your R script will execute much faster than it would if each operation was performed using loops.

For example, let's take a vector of hourly salaries.

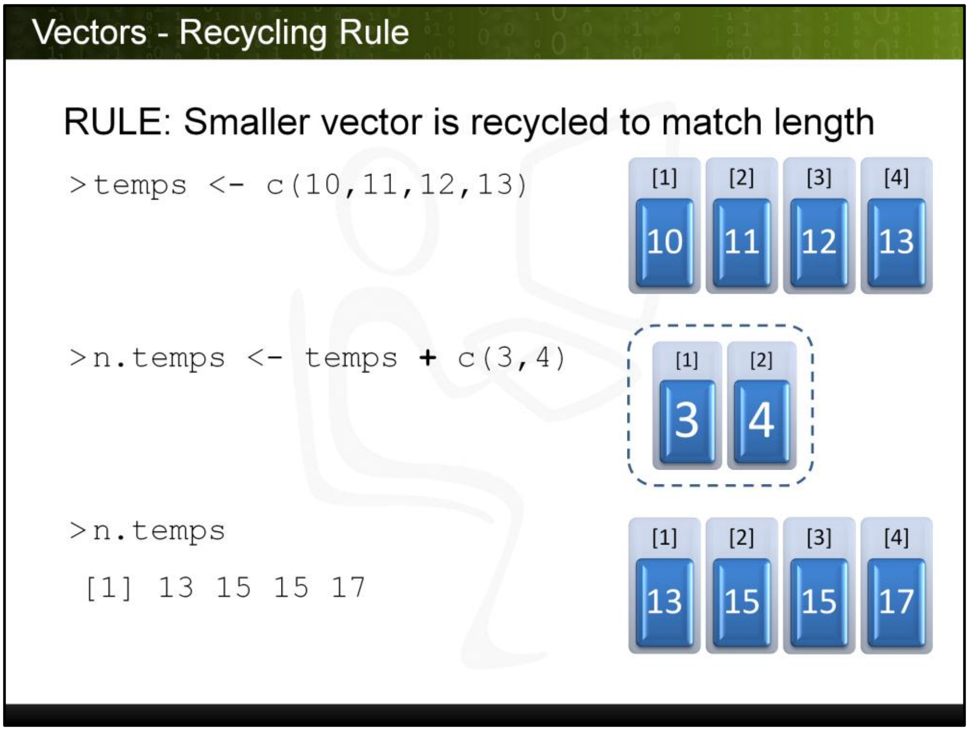
We would like to increase all of the hourly salaries by 10%.

To accomplish this task we will multiply the existing salary by 1.10 to obtain the new salary.

The multiplication operator or asterisk (*) can be used. In R, the constant value of 1.10 is multiplied with each element of the vector. This is an example of R's vector recycling rule.

Note that since we did not store the new salaries the original values remain unchanged.

To replace the existing values we must use the assignment operator.



Vectors - Recycling Rule

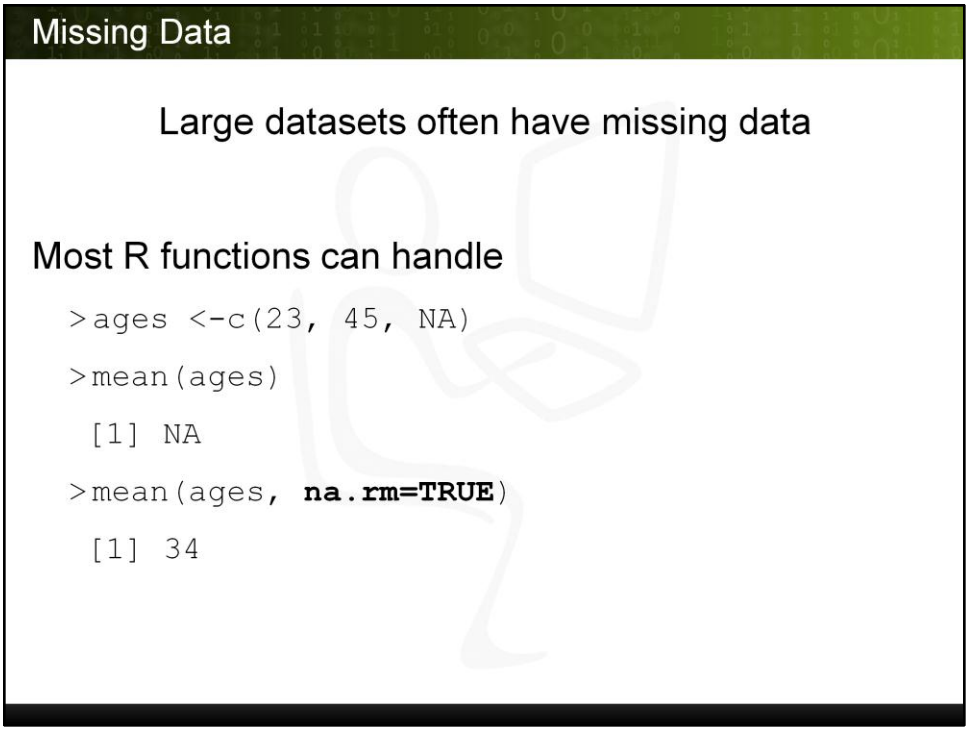
RULE: Smaller vector is recycled to match length

```
>temps <- c(10,11,12,13)

>n.temps <- temps + c(3,4)

>n.temps
 [1] 13 15 15 17
```

As we saw in the previous example, when vector operations are performed using vectors of different sizes the smaller vector will be recycled or reused as the operation is completed.

In this example we are performing vector addition, but temps has 4 elements and it is being summed with a 2 element vector called n.temps.

Following the operation we notice has the elements 3 and 4 are applied a second time for the last 2 elements of the vector temps. This may or may not be the result that you expected so be careful when using these built-in functions with vectors.



Missing Data

Large datasets often have missing data

Most R functions can handle

```
>ages <-c(23, 45, NA)
>mean(ages)
 [1] NA
>mean(ages, na.rm=TRUE)
 [1] 34
```

When you are working with large datasets it is common to have some missing data values.

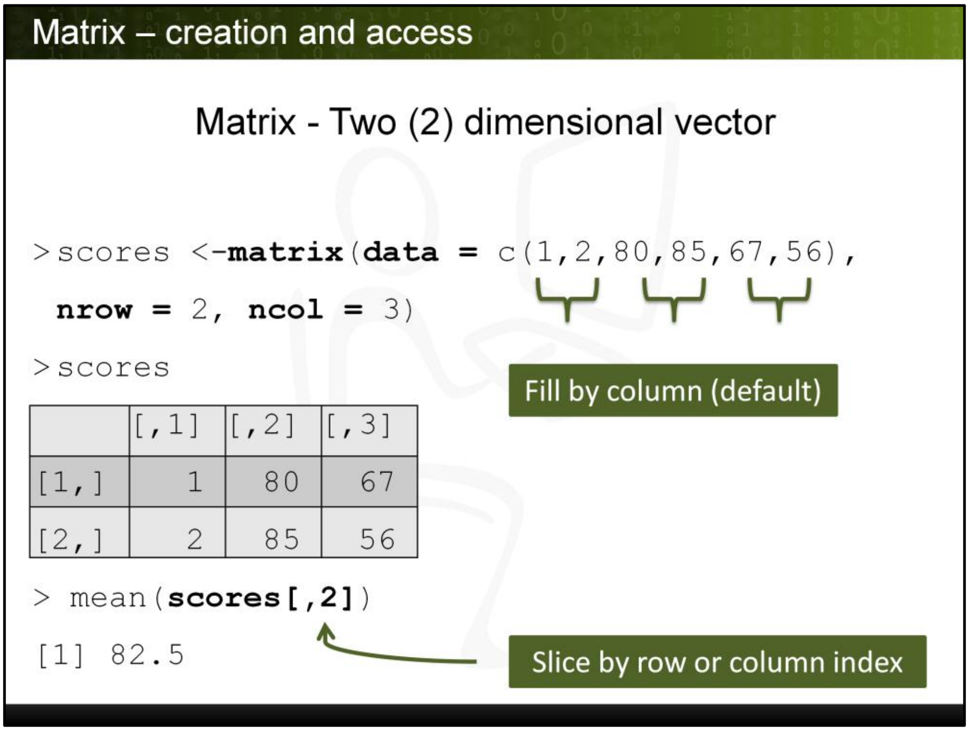
R has a special value that is used to represent missing data.

Missing data can be represented with the value of NA. NA simply means "Not Available".

By default, R will recognize NA values and in this example the mean() cannot be computed when there are missing values.

If you would like to ignore the missing data you can pass the optional argument of na.rm=TRUE.

There are other special values in R, but we will not discuss them in this lesson.



Matrix – creation and access

Matrix - Two (2) dimensional vector

```
>scores <-matrix(data = c(1,2,80,85,67,56),
 nrow = 2, ncol = 3)
>scores
```

Fill by column (default)

	[,1]	[,2]	[,3]
[1,]	1	80	67
[2,]	2	85	56

```
> mean(scores[,2])
[1] 82.5
```

Slice by row or column index

A matrix is simply a vector with 2 dimensions. We still have the restriction that all of the elements must be of the same datatype.

In this example we create a matrix using the matrix() function to represent the marks for 2 different students. The dimensions of the matrix should be provided when it is created. Here we would like to create a 2 by 3 matrix as we have 2 students and 3 different attributes.

The first column is used as a unique student identifier and the 2 elements across each row represent the results of 2 different tests.

When we use the matrix() function the data is provided as a single dimension vector, but we then also describe the number of rows and columns that R should use to represent the data.

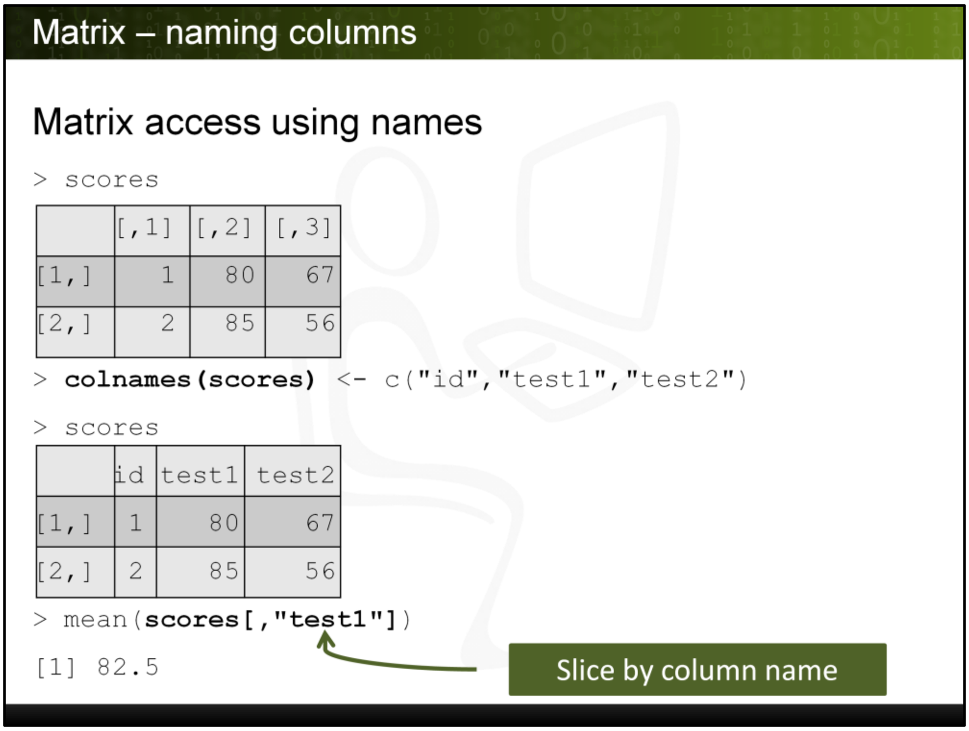
By default the data in the vector fills the data structure by columns.

In our example here we have test scores of 80 and 67 for student 1 and scores of 85 and 56 for student 2.

Data within the matrix can be analyzed as a single element or a range of elements using the comma symbol.

Here we calculating the mean or average of the first test for all students.

It is important to remember that the first index value used represents the row in the matrix and the second value represents the column index.



Matrix – naming columns

Matrix access using names

```
> scores
      [,1] [,2] [,3]
[1,]     1   80   67
[2,]     2   85   56
> colnames(scores) <- c("id","test1","test2")
> scores
     id test1 test2
[1,]  1    80    67
[2,]  2    85    56
> mean(scores[,"test1"])
[1] 82.5
```

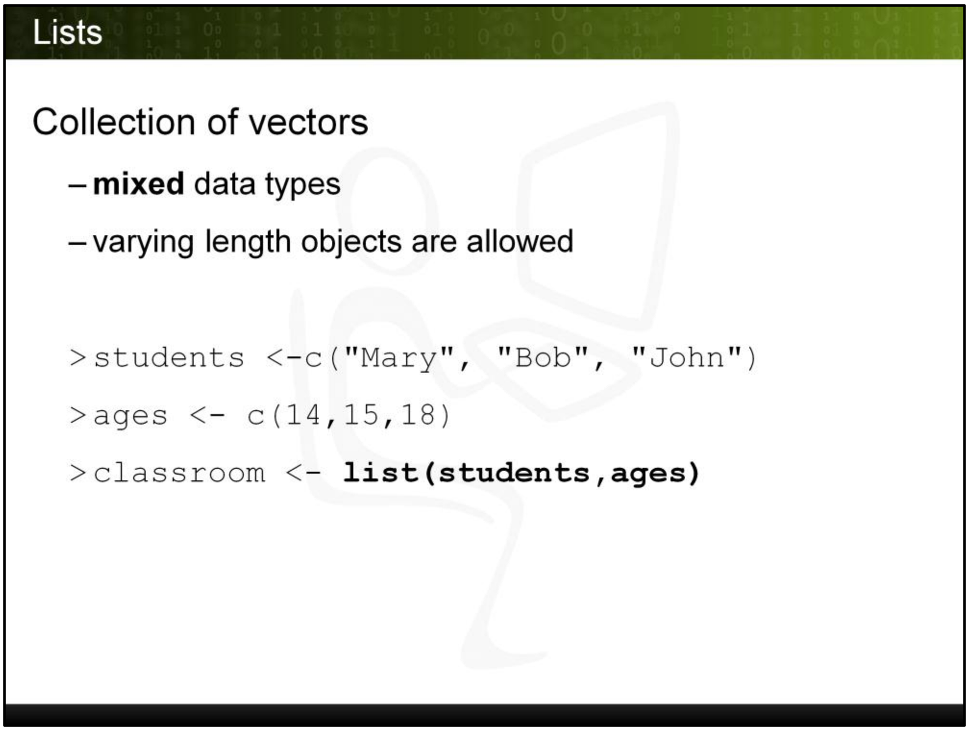
Slice by column name

We previously mentioned that the data stored in an R matrix must be of the same data type, but the rows and columns can be given names instead of simply using numeric index values. In this example, we want to clarify that the columns really represent the student id, test1, and test2 elements.

We accomplish this using the **colnames()** function. This function accepts a vector with the terms to be associated with each column in the matrix.

Now, we can perform the same computation across the scores matrix using the column name reference of "test1".

Using column names provides additional flexibility as you no longer need to worry about the index value changing over time.



Lists

Collection of vectors

- **mixed** data types
- varying length objects are allowed

```
>students <-c("Mary", "Bob", "John")
>ages <- c(14,15,18)
>classroom <- list(students,ages)
```

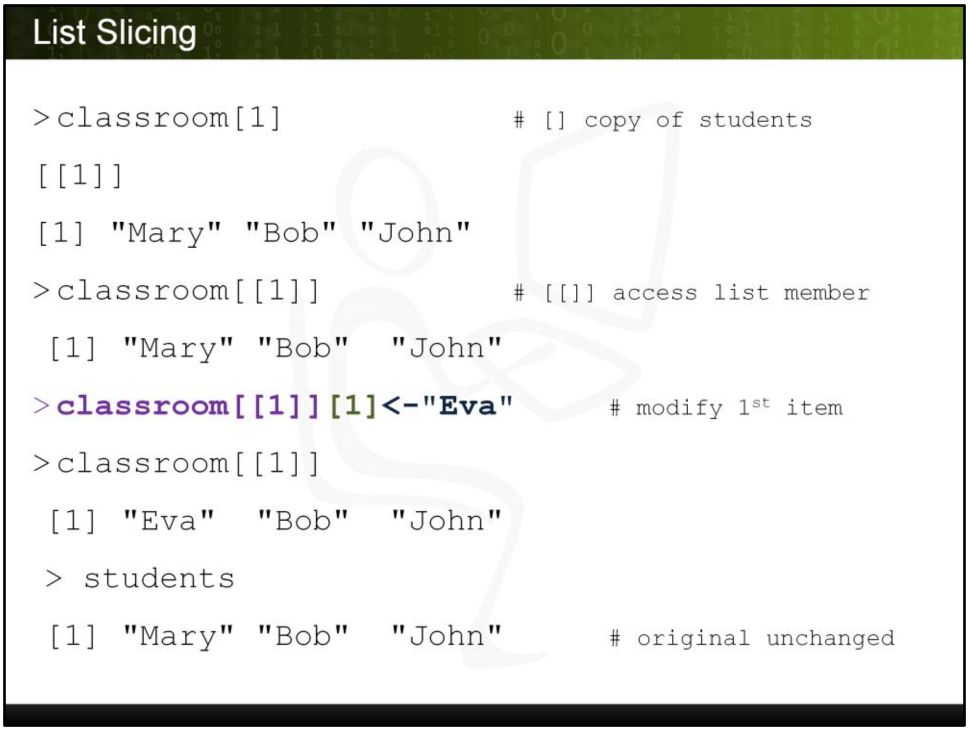
A list is an ordered collection of objects.

Unlike vectors, the objects can be of mixed data types and they can also be of different lengths.

In this example, we start with 2 independent vectors.

A character vector of student names and a numeric vector of ages.

A list structure called classroom is created using the list() function. The list consists of copies of the students and ages vectors.



List Slicing

```
>classroom[1]                # [] copy of students
[[1]]
[1] "Mary" "Bob" "John"
>classroom[[1]]              # [[]] access list member
 [1] "Mary" "Bob"  "John"
>classroom[[1]][1]<-"Eva"       # modify 1st item
>classroom[[1]]
 [1] "Eva"  "Bob"  "John"
 > students
 [1] "Mary" "Bob"  "John"       # original unchanged
```

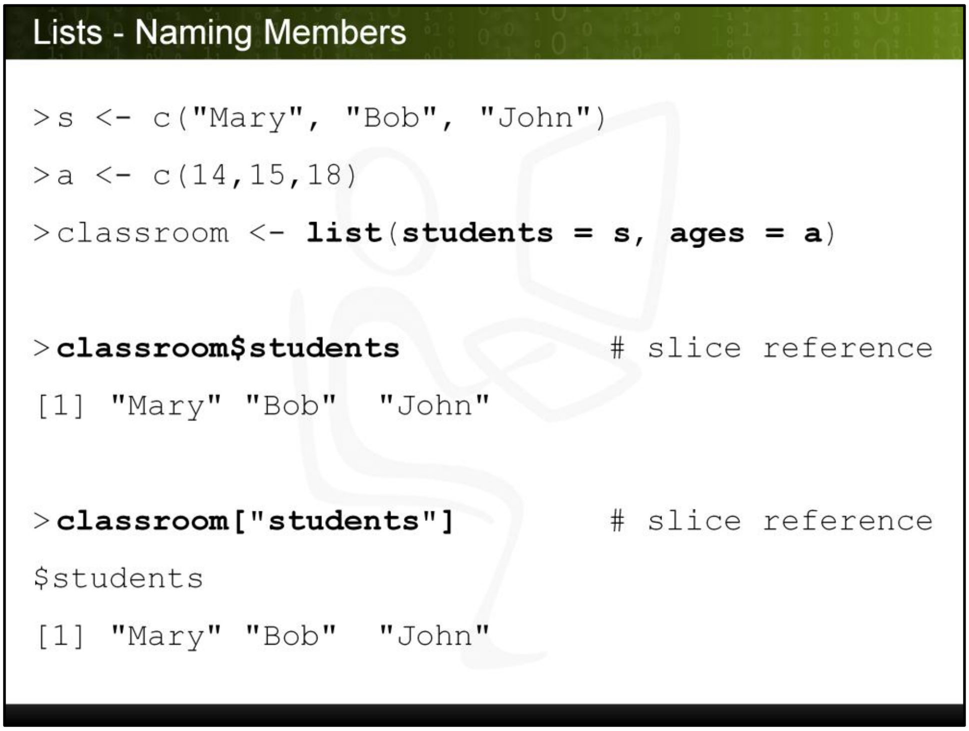
A single set of square brackets [] is used to retrieve a copy of the data contained in a list. The data is always returned in the form of a list data structure.

The student names can be retrieved using a set of single or double square brackets.

If we want to modify a value stored in the classroom list we can use the double bracket indexing method.

Here we are replacing the first student, "Mary" with a new student "Eva", in the classroom list structure.

The initial vector called students that was used to create the initial classroom data structure is left unchanged.



Lists - Naming Members

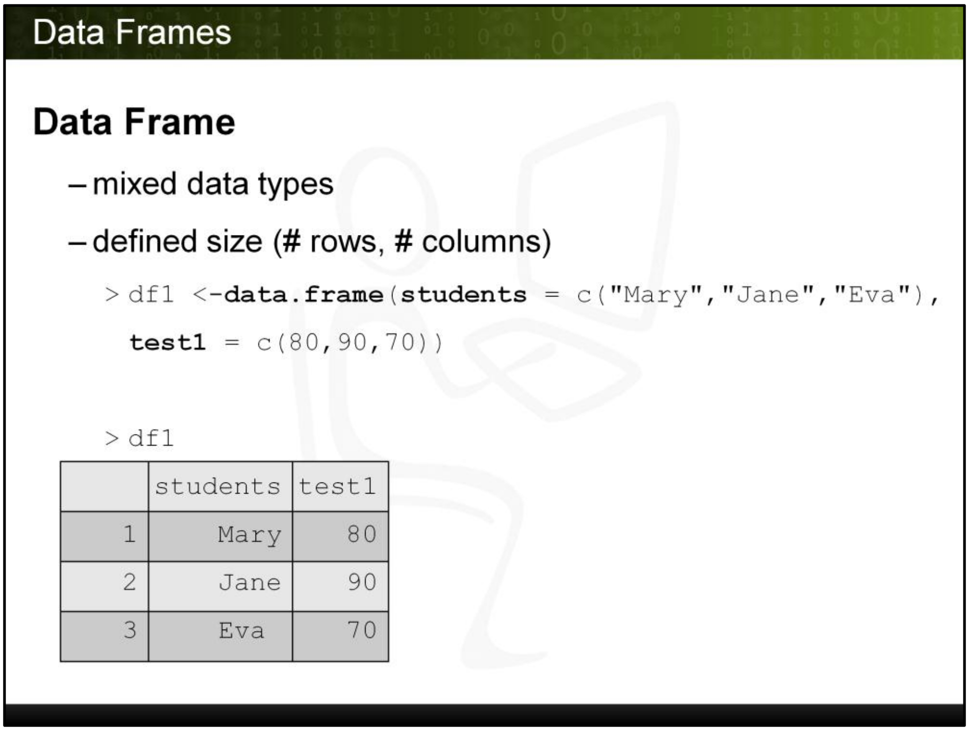
```
>s <- c("Mary", "Bob", "John")
>a <- c(14,15,18)
>classroom <- list(students = s, ages = a)

>classroom$students            # slice reference
[1] "Mary" "Bob"  "John"

>classroom["students"]         # slice reference
$students
[1] "Mary" "Bob"  "John"
```

Just like how we were able to give columns a name with matrix data structures, we can name each component of our list.

Here we decided to label or name the list components of our classroom using the terms "students" and "ages".

Now we can reference the list elements using the dollar sign ($) symbol or using the name of the list component.

Since lists allow us the ability to group different types of data they are used frequently in data analysis tasks.



Data Frames

Data Frame

- mixed data types
- defined size (# rows, # columns)

```
> df1 <-data.frame(students = c("Mary","Jane","Eva"),
  test1 = c(80,90,70))

> df1
```

	students	test1
1	Mary	80
2	Jane	90
3	Eva	70

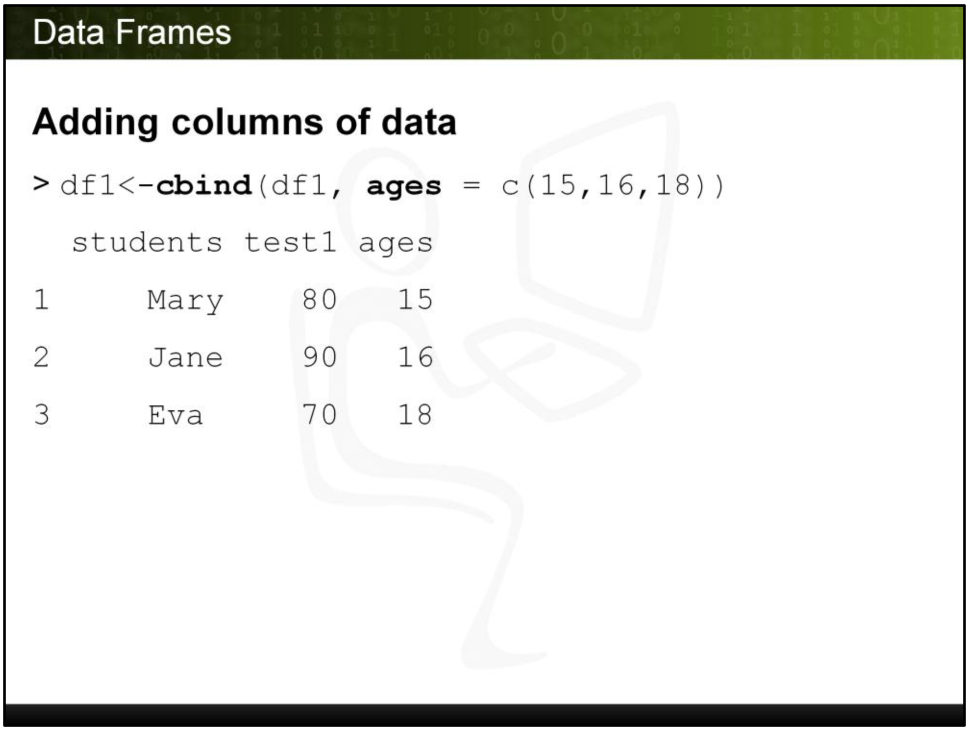
Data frames are useful data structures to represent tabular data.

Like lists, a data frame can consist of different types of data.

Unlike lists, data frames have a defined size – or number of rows and columns.

Here we have created a data frame called df1 using the data.frame() function.

The data frame represents our three students and their marks on a single test.



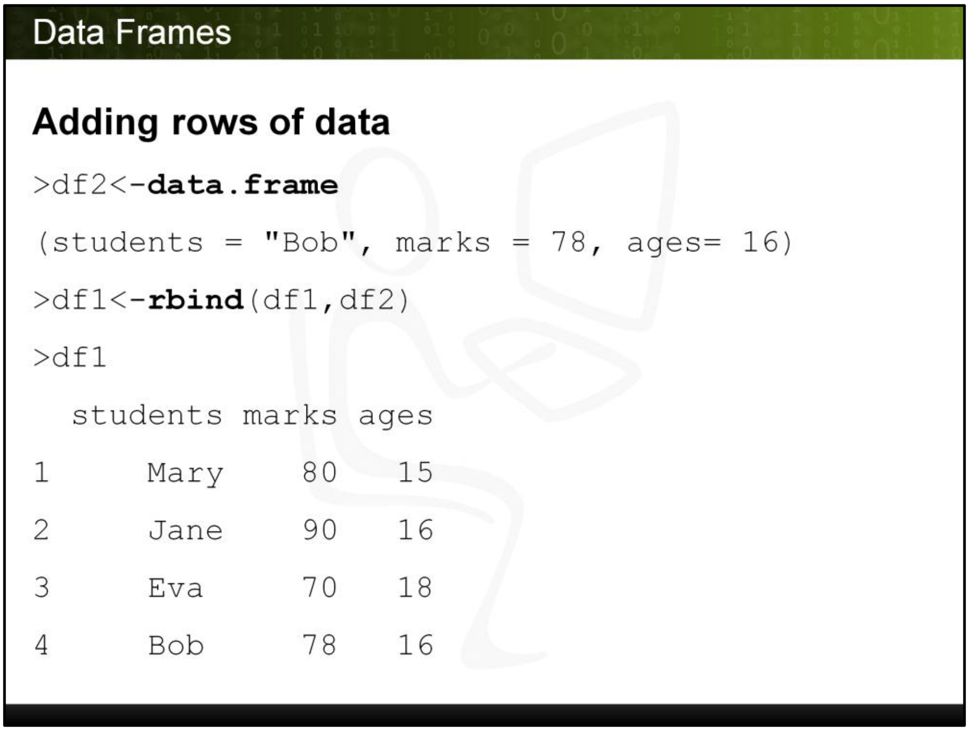
Data Frames

Adding columns of data

```
> df1<-cbind(df1, ages = c(15,16,18))
  students test1 ages
1     Mary    80   15
2     Jane    90   16
3     Eva     70   18
```

The **cbind()** or column bind function can be used to append another vector as a new **column** to our data.frame.

In this case the new vector represents the a set of ages of the 3 students.



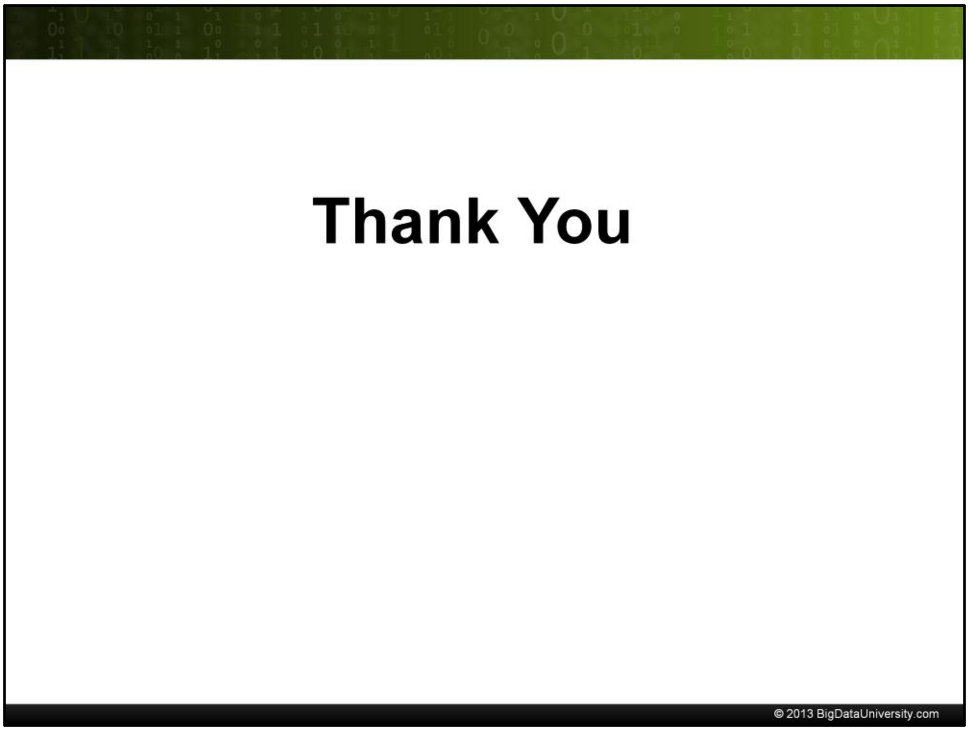
Data Frames

Adding rows of data

```
>df2<-data.frame
(students = "Bob", marks = 78, ages= 16)
>df1<-rbind(df1,df2)
>df1
  students marks ages
1     Mary    80   15
2     Jane    90   16
3     Eva     70   18
4     Bob     78   16
```

If a new row of data is to be appended to the data frame the **rbind()** or row bind function can be used.

In this example we are appending a new student to our data frame.



Thank You

Thank you for completing this lesson.

Now it is time to practice using R data structures.