# R and Rstudio: Part II

# Data manipulation and Flow Control


Dr. Irene Vrbik

University of British Columbia Okanagan

irene.vrbik@ubc.ca

# Logical Operators

- R has several **operators** to perform tasks
- We have already seen two:
  - assignment operators (eg. `=` and `<-`)
  - arithmetic operators (eg. `+`, `-`, `*`, `/`, `^`, `%%`)
- Other types of operators include:
  - Relational operators
  - Logical operators

# Logical Operators (not R specific)

There are three logical operators that are used to compare values.

| Operator | True if: | Examples | Output |
|---|---|---|---|
| `AND` | `both are true` | `True and True` | `True` |
| | | `False and True` | `False` |
| `OR` | `either or both are true` | `True or True` | `True` |
| | | `False or True` | `True` |
| | | `False or False` | `False` |
| `NOT` | `false` | `not True` | `False` |
| | | `not False` | `True` |

# Logical Operators (R specific)

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>!</td><td>Logical NOT</td></tr>
<tr><td>&</td><td>Element-wise logical AND</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>|</td><td>Element-wise logical OR</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
</table>

The element-wise operators produce result having length of the longer operand while && and || result in a single length logical vector and examines only the first element of the operands.

# Logical Operators

The element-wise operators produce result having length of the longer operand

```
x <- c(TRUE,FALSE,TRUE,FALSE)
y <- c(FALSE,TRUE,TRUE,FALSE)
!x
```

```
## [1] FALSE  TRUE FALSE  TRUE
```

```
x&y
```

```
## [1] FALSE FALSE  TRUE FALSE
```

```
x|y
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

# Logical Operators

```
x <- c(TRUE,FALSE,TRUE, FALSE)
z1 <- c(TRUE, TRUE, FALSE)   # recycles TRUE
z2 <- c(FALSE, TRUE, FALSE)  # recycles FALSE
x|z1
```

```
## Warning in x | z1:  longer object length is not a multiple
of shorter object length
## [1] TRUE TRUE TRUE TRUE
```

```
x|z2
```

```
## Warning in x | z2:  longer object length is not a multiple
of shorter object length
## [1]  TRUE  TRUE  TRUE FALSE
```

# Logical Operators

&& and || result in a single length logical vector and examines only the first element of the operands.

```
x <- c(TRUE,FALSE,FALSE,TRUE)
y <- c(FALSE,TRUE,FALSE,TRUE)
x&&y
## [1] FALSE
x||y
## [1] TRUE
```

# Logical Operators

If you are doing comparisons of scalars (ie length one vectors), there is not effective difference between the two:

```
TRUE & FALSE

## [1] FALSE

TRUE && FALSE

## [1] FALSE

TRUE | FALSE

## [1] TRUE

TRUE || FALSE

## [1] TRUE
```

# Logical Operators

TRUE/FALSEs are converted to 0/1 in numerical operations. Therefore, to check how many elements return `TRUE`, we can simply take the sum on the logical vector.

```
 x <- c(TRUE,FALSE,TRUE,TRUE)
sum(x) # counts the number of TRUEs
## [1] 3
sum(!x) # counts the number of FALSEs
## [1] 1
```

# Relational Operators

| Operator | Description |
|---|---|
| `<` | Less than |
| `>` | Greater than |
| `<=` | Less than or equal to |
| `>=` | Greater than or equal to |
| `==` | Equal to |
| `!=` | Not equal to |

# Relational Operators

```
x <- 4
y <- 16
x < y
## [1] TRUE
4*x >= y
## [1] TRUE
y == 16
## [1] TRUE
x != 4
## [1] FALSE
```

# Conditional Selection

- We can use these operators in some advanced indexing.
- Last lecture we saw how to extract elements from a vector/matrix from using one or several indices (eg `x[1], x[c(4,2)]`)
- In practice, you often need to extract data that satisfy a certain criteria.
- To do this in one step, we use conditional selection.

```r
# sample twelve numbers from 1--10 with replacement
(y = sample(10, 12, replace=TRUE))

##  [1] 10  2  1  2  1  6  8  9  9  6  7  7

y[y>7] # returns any numbers(s) larger than 7

## [1] 10  8  9  9

y[y>7 & y%%2==0] # returns any even number(s) larger than 7

## [1] 10  8
```

To get the index rather than the values, use `which()`

```r
x = c("female","male","female","male","male","female")
which(x=="female")

## [1] 1 3 6
```

# SQLish functions

- There are a number of functions in R that will mimic tasks performed using SQL commands on relational databases.

  subset for filtering rows (like `WHERE`)

  subset with select for filtering columns (like `SELECT`)

  transform for updating columns (like `UPDATE`)

- **Aside:** There is another useful package called dplyr that can handle more complicated SQL-like statments (eg `GROUP BY` and `JOINS`).

# SQLish functions

- To highlight these functions, we will use the data set called `iris`.
- R has many useful built-it data sets for us to play with. To see them listed by name, type `data()`.
- Notice that the object will not appear in our Enviornment panel until we execute:

```
data("iris")
```

- To see a description of the data type: `?<nameofdataset>`

`head()` prints out the first 6 rows by default.

```
head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

As an optional second argument we can specify `n`, the number of rows we want to include:

```
head(iris, 2)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
```

To see the last `n` rows use `tail()`:

```r
tail(iris) # default is to print the last 6 lines
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```r
tail(iris, 2) # specify a certain number of lines as a 2nd arg
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

# Subsetting

- The `subset()` function allows us to filter the rows of a data set based on some criteria applied to a column (or multiple criterion on multiple columns)
- This is similar to the `WHERE` statement in SQL.
- For example, lets select all rows in `iris` that have a `Petal.Length` greater than 5.

```r
data(iris)
nrow(iris) # 150 observations
## [1] 150
longPetals = subset(iris, Petal.Length>5)
nrow(longPetals) # subset of 42 observations
## [1] 42
```

We could have done it without this function with a little work:

```r
rows_to_include = iris$Petal.Length > 5
long_Petals = iris[rows_to_include,]
nrow(long_Petals) # same subset of 42 observations
## [1] 42
```

## An example that combines multiple conditions

```
data(iris)
nrow(iris) # 150 observations

## [1] 150

# all observations having a petal length greater than 5
# or belong to the setosa family.
newdat = subset(iris, Species=="setosa" | Petal.Length>5)
nrow(newdat)

## [1] 92
```

# SELECT

- We could also select specific columns using the `select` (optional) argument in the `subset` function.
- This is similar to the `SELECT` statement in SQL.
- For example, lets select all the rows in `iris` but only the columns pertaining to length measurements.

```
lengthDat = subset(iris, select = c(Sepal.Length, Petal.Length))
print(paste(nrow(lengthDat), ncol(lengthDat)))
```

```
## [1] "150 2"
```

# SELECT

```
head(lengthDat)

##   Sepal.Length Petal.Length
## 1          5.1          1.4
## 2          4.9          1.4
## 3          4.7          1.3
## 4          4.6          1.5
## 5          5.0          1.4
## 6          5.4          1.7
```

# SELECT

To do this task without the `subset` function would require us to know which column the `Sepal.Length` and `Petal.Length` fall under. One way we could find that is using the `%in%` operator.

```
cols_to_include = colnames(iris) %in%
  c("Sepal.Length", "Petal.Length")
cols_to_include
```

```
## [1]  TRUE FALSE  TRUE FALSE FALSE
```

```
length_data = iris[,cols_to_include]
print(paste(nrow(length_data), ncol(length_data)))
```

```
## [1] "150 2"
```

# Object Equality

As a sidenote, if we want to check equality of two objects in R, we could use `all.equal`

```
all.equal(longPetals, long_Petals)

## [1] TRUE

all.equal(lengthDat, length_data)

## [1] TRUE
```

# Transforming

- The `transform()` function provides a quick and easy way to transform the data frames (just like SQL `UPDATE` function).
- For instance if we want to increase all the `Sepal.Lengths` by 0.1 we could type:

```
dim(iris)

## [1] 150   5

irisLonger = transform(iris, Sepal.Length = Sepal.Length + 0.1)
dim(irisLonger)

## [1] 150   5
```

```r
head(iris,4)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
```

```r
head(irisLonger, 5)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.2         3.5          1.4         0.2  setosa
## 2          5.0         3.0          1.4         0.2  setosa
## 3          4.8         3.2          1.3         0.2  setosa
## 4          4.7         3.1          1.5         0.2  setosa
## 5          5.1         3.6          1.4         0.2  setosa
```

# Transforming

- We could also use the `transform()` function to create new variables in our data frame (like the `ALTER` SQL command)
- For instance if we want to add a new column which holds the log values of the petal lengths we could type:

```
dim(iris)

## [1] 150   5

irisMore = transform(iris, logPL = log(Petal.Length))
dim(irisMore)

## [1] 150   6
```

# Transforming

```
head(irisMore)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species     logPL
## 1          5.1         3.5          1.4         0.2  setosa 0.3364722
## 2          4.9         3.0          1.4         0.2  setosa 0.3364722
## 3          4.7         3.2          1.3         0.2  setosa 0.2623643
## 4          4.6         3.1          1.5         0.2  setosa 0.4054651
## 5          5.0         3.6          1.4         0.2  setosa 0.3364722
## 6          5.4         3.9          1.7         0.4  setosa 0.5306283
```

# Splitting

- Another handy function is `split`.
- `split()` generates a list of vectors according to a grouping

```
iSpecies = split(iris, iris$Species)
names(iSpecies)

## [1] "setosa"     "versicolor" "virginica"

# iSpecies$setosa is the same thing as
setosa = subset(iris, Species=="setosa")
all.equal(iSpecies$setosa,setosa)

## [1] TRUE
```

# Sorting and Order

- The `sort()` function is a operator that we saw in a very simple context:

```
x <- c(4,0,-7,9,8,3)
x

## [1]  4  0 -7  9  8  3

sortx <- sort(x)
sortx

## [1] -7  0  3  4  8  9
```

# Sorting and Order

- A related function is `order()` which order provides the indexing of `x` which provides the sorted vector `sortx`.

```
(o <- order(x))

## [1] 3 2 6 1 5 4

x[o]

## [1] -7  0  3  4  8  9
```

- We can use order to rearange the rows of data set to agree with a sorting of a particular column, for instance.

Example: rerrange the rows of `iris` so that the `Petal.length` is sorted from smallest to largest:

```
o = order(iris$Petal.Length)
head(o)

## [1] 23 14 15 36  3 17

irisSorted = iris[o,]
head(irisSorted)

##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 23          4.6         3.6          1.0         0.2  setosa
## 14          4.3         3.0          1.1         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
## 36          5.0         3.2          1.2         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 17          5.4         3.9          1.3         0.4  setosa
```

# Sorting and Order

- Note that `order()` can also take multiple sorting arugments
- For instance, we `order(gender, age)` in the example of the following slide will give a main division into men and women, and within each group, they will be ordered by age.

# Sorting and Order

```
gender = c("female","male","female","male","male","female")
age = c(36, 24, 25, 40, 22, 23)
df = data.frame(gender=gender, age=age)
o = order(df$gender, df$age)
df[o,]
```

```
##   gender age
## 6 female  23
## 3 female  25
## 1 female  36
## 5   male  22
## 2   male  24
## 4   male  40
```

# Missing Data

- In R, missing values are represented as `NA` (**N**ot **A**vailable).
- `NaN` (**N**ot **a** **N**umber) is usually the product of some arithmetic operation and represents impossible values (e.g., dividing by zero).
- We can check for these using `is.na()`, `is.nan()`

```
y = -1:3  # fills elements 1--5
y[7] = 7  # element 6 is missing
y
## [1] -1  0  1  2  3 NA  7
is.na(y)
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
(sy = sqrt(y))  # take the square roots
## Warning in sqrt(y):  NaNs produced
## [1]      NaN 0.000000 1.000000 1.414214 1.732051       NA 2.6457
is.nan(sy)
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

# Missing Values

- An easy way to remove rows from a data set having missing values is:

```
newdata <- na.omit(mydata)
```

- Some functions may having built in arguments to remove missing values from the calculation:

```
mean(y)

## [1] NA

mean(y, na.rm = TRUE)

## [1] 2
```

# Missing Values

- It may happen that we would like to replace values that meet a certain condition with NA

```
y = sample(c(0,1,2,3,4), 10, replace= TRUE)
# replace 0s with NAs
y[y==0] = NA
```

- On the flip side, we could easily replace NAs by some value

```
# replace NAs with 0s
y[is.na(y)] = 0
```

# Loops

- **Looping**, (AKA cycling or iterating) provides a way of replicating a set of statements multiple times until some condition is satisfied.
  - Each time a loop is executed is called an iteration.
- A **for loop** repeats statements a number of times. It will iterate based on the number of group/collection elements.
- A **while loop** repeats statements while a condition is TRUE

# `while` loops

The most basic looping structure is the while loop which continually executes a set of statements while a condition is true. Syntax:

```
while loops

while (condition){
  statement 1
  ...
  statement n
}
```

While `(condition)` is `TRUE` execute the set of statements `statement 1`, ...`statement n`

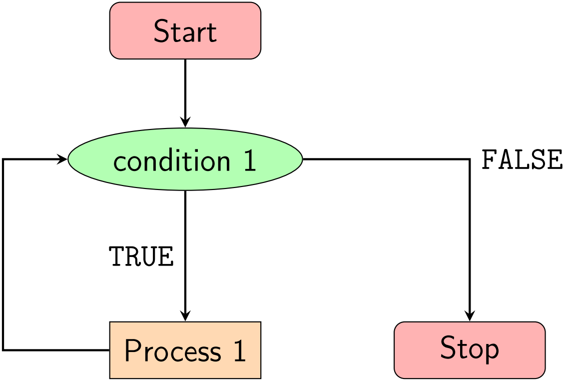
# `while` loops Example

```
n = 1
while (n<=3){
  print(n)
  n = n + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

*Recall that blocks are contained within curly braces and indentation is not necessary (though recommended for readability).*

# `while` loops



# repeat

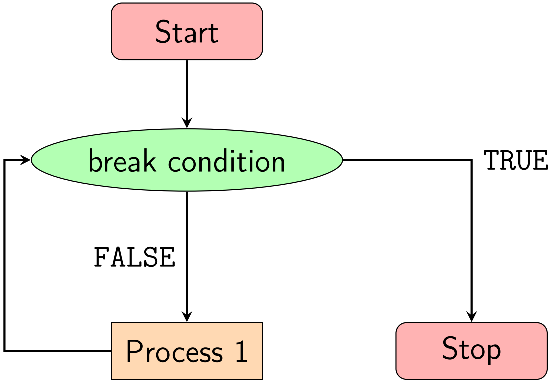
An alternative way of coding this up would be to use `repeat`

```
n=1
repeat {
  print(n)
  n = n + 1
  if (n>5) break
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

- `break` is a reserved word (you are not allowed to use reserved words as variable names) in R that allows you to break out of a `for`, `while`, or `repeat`
- Notice that the condition we place for breaking this loop will be the opposite condition that we had for our `while` statement.
  - Remember: if the `while` condition is `TRUE`, we continue to the next iteration
  - Remember: if the `break` condition is `FALSE`, we continue to the next iteration

# repeat loops



# The `for` loop

- A `for` loop repeats statements a given number of times.

  Syntax:

  for loops

```
for (i in seq){
  statement 1
  ...
  statement n
}
```

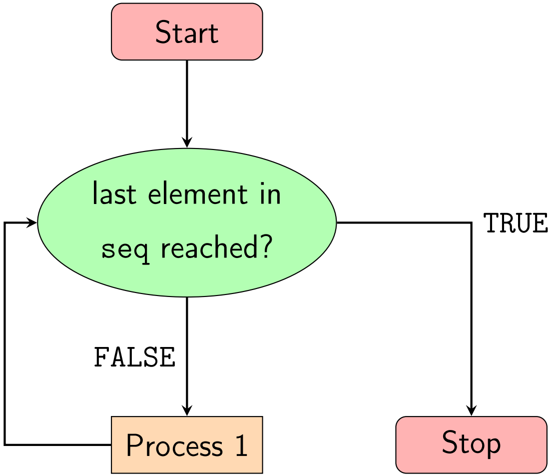
# The `for` loop

This simple example prints the numbers 1–6. *Don't forget that the end number is inclusive in* `R`*!*

```
for (i in 1:6){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

# for loops



# The `for` loop

- Notice that we must wrap the condition (in both `for` loop and `while` loops) in parenthesis `()`, otherwise, we get an error

```
for i in 1:6{
  print(i)
}
```

```
## Error:  <text>:1:5:  unexpected symbol
## 1:  for i
##         ^
```

# The `for` loop

- In the example on slide 47 the `seq` was a sequence of numbers (the numbers 1 to 6 inclusive).
- More generally, `seq` can be a set of objects stored in a vector, list or data frame.

```
fruit = c("apples", "banana", "oranges")
for (i in fruit){
  print(i)
}
```

```
## [1] "apples"
## [1] "banana"
## [1] "oranges"
```

We may use `seq_along(along.with)` to create a sequence of number 1 to `length(along.with)`

```
x <- c("Hello", "World")
for (i in seq_along(x)){
  print(paste("i=",i,"x[i]=",x[i]))
}
## [1] "i= 1 x[i]= Hello"
## [1] "i= 2 x[i]= World"
```

Same as:

```
for (i in 1:length(x)){print(paste("i=",i,"x[i]=",x[i]))}
## [1] "i= 1 x[i]= Hello"
## [1] "i= 2 x[i]= World"
```

# for loops

It is worth pointing out that the sequence which we iterate over (eg. `1:6`, and `seq_along(x)`) need not start from 1, nor does it have to appearing in an increasing order. For example:

```
x <- c("apple","ball","cat","dog")
for (i in c(3,1,4,2)){
  print(paste("i=",i,"x[i]=",x[i]))
}
```

```
## [1] "i= 3 x[i]= cat"
## [1] "i= 1 x[i]= apple"
## [1] "i= 4 x[i]= dog"
## [1] "i= 2 x[i]= ball"
```

# `next`

- Another reserve word is `next`
- Like `break`, `next` does not return a value, it merely transfers control within the loop.
- A `next` statement is useful when we want to skip the current iteration of a loop without terminating it.
- On encountering next, the R proceeds to next iteration of the loop (without executing any remaing statements the current iteration).

```r
x <- c("apple", "ball","cat","dog","elephant","fish")
for (i in seq_along(x)){
  print(paste("iteration", i))
  if (i%%2==0)
    next
  print(x[i])} # wont be printed for even indices
```

```
## [1] "iteration 1"
## [1] "apple"
## [1] "iteration 2"
## [1] "iteration 3"
## [1] "cat"
## [1] "iteration 4"
## [1] "iteration 5"
## [1] "elephant"
## [1] "iteration 6"
```

# Common problems – Infinite Loops

**Infinite loops** are caused by an incorrect loop condition or not updating values within the loop so that the loop condition will eventually be false.

infinite loops

```
n = 1
while (n <= 5){
  print(n)
}
```

Here we forgot to increase $n$. Hence we get an infinite loop (i.e. the code will print 1,2,3, $\ldots$, $\infty$)

# Apply Functions

- Last lecture we saw we could perform vector operations

```r
y <- 1:6
# will add 2 to each element in y
y + 2
## [1] 3 4 5 6 7 8
```

- A similar concept for mutli-dimensional vectors can be done using the `apply` family of functions in R.

# Apply Functions

`apply` returns a vector or array or list of values obtained by applying a function to margins of an array or matrix. There are a number of variants:

lapply list apply

sapply simple apply

mapply map apply

The best choice will depend on the data structure and desired output.

# Apply Functions

The general syntax is

```
apply(X, MARGIN, FUN, ...)
```

- `X` an array, including a matrix.
- `MARGIN` a vector giving the subscripts which the function will be applied over. (eg. for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns.)
- `FUN` the function to be applied. In the case of functions like +, etc., the function name must be backquoted or quoted.

# Apply functions

The called function could be:

- An aggregating function which returns a single number (eg. `mean`, or the `sum`
- A vectorized functions, which may returns a list, vector, matrix or array.
- transforming or subsetting functions (see slide 18, 27)

# Apply functions

- The `apply()` function allows us to perform operations with very few lines of code.
- Apart from producing more concise code, using an `apply()` function in place of a loop can often save a lot of time (computational time that is, often they will take you longer to think about and code up)

# Apply

- Let's consider the case where `X` is a matrix.
- While there is already a built-in function to do this (`rowSums()`), lets compute the sum of each row of an $n \times p$ matrix.
- Since there are $n$ rows, the result should be an $n$-length vector.

# Apply Functions

```
m <- matrix(1:12, nrow=3, ncol=4)
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
mRowSums = apply(m, 1, sum) # index = 1 for rows
mRowSums
```

```
## [1] 22 26 30
```

# Apply Functions

To acheive the same result using a loop:

```
m <- matrix(1:12, nrow=3, ncol=4)
mRowSums = NULL
for (i in 1:nrow(m)){
  mRowSums[i] = sum(m[i,])
}
mRowSums
```

```
## [1] 22 26 30
```

# Apply Functions

Similarly we could have found the column sums:

```
m <- matrix(1:12, nrow=3, ncol=4)
m

##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

#notice how the margin changes to 2 below
mColSums = apply(m, 2, sum) # index 2 for columns
mColSums

## [1]  6 15 24 33
```

# Practical Exercise

## Exercise

Find the average Sepal length and width across for the `setosa` species within the `iris` data.

## Solution

```
setosa = subset(iris, Species=="setosa",
                select= c("Sepal.Length", "Sepal.Width"))
apply(setosa, 2, mean)
## Sepal.Length  Sepal.Width
##        5.006        3.428
```

## Question

How many of the are following statements are true?

- The longer logical operators (eg. `&&` and `||`) perform operations element-wise.
- The `subset()` function can be used to select specific columns from a data frame.
- The `sort(x)` function returns a vector consisting of the ordered *indices* of the elements in `x`.
- The `head()` function returns the last 6 rows of a data frame

A) 0 B) 1 C) 2 D) 3 E) 4

## Question

How many of the are following statements are true?

- The longer logical operators (eg. `&&` and `||`) perform operations element-wise.
- The `subset()` function can be used to select specific columns from a data frame.
- The `sort(x)` function returns a vector consisting of the ordered *indices* of the elements in `x`.
- The `head()` function returns the last 6 rows of a data frame

A) 0 B) 1 C) 2 D) 3 E) 4

## Question

How many of the are following statements are true?

- The longer logical operators (eg. `&&` and `||`) perform operations element-wise. ✗
- The `subset()` function can be used to select specific columns from a data frame.
- The `sort(x)` function returns a vector consisting of the ordered *indices* of the elements in `x`.
- The `head()` function returns the last 6 rows of a data frame

A) 0    B) 1    C) 2    D) 3    E) 4

## Question

How many of the are following statements are true?

- The longer logical operators (eg. `&&` and `||`) perform operations element-wise. ✗
- The `subset()` function can be used to select specific columns from a data frame. ✓
- The `sort(x)` function returns a vector consisting of the ordered *indices* of the elements in `x`.
- The `head()` function returns the last 6 rows of a data frame

A) 0 B) 1 C) 2 D) 3 E) 4

## Question

How many of the are following statements are true?

- The longer logical operators (eg. `&&` and `||`) perform operations element-wise. ✗
- The `subset()` function can be used to select specific columns from a data frame. ✓
- The `sort(x)` function returns a vector consisting of the ordered *indices* of the elements in `x`. ✗
- The `head()` function returns the last 6 rows of a data frame

A) 0 B) 1 C) 2 D) 3 E) 4

## Question

How many of the are following statements are true?

- The longer logical operators (eg. `&&` and `||`) perform operations element-wise. ✗
- The `subset()` function can be used to select specific columns from a data frame. ✓
- The `sort(x)` function returns a vector consisting of the ordered *indices* of the elements in `x`. ✗
- The `head()` function returns the last 6 rows of a data frame ✗

A) 0 B) *1* C) 2 D) 3 E) 4