R Basics Course

Contents

Introduction

This course is an introduction into using R for data analysis. It contains a series of demonstrations on how to use basic functions and operators in R.

Assorted Useful Functions

- `ls()` show all objects in workspace
- `args` provides arguments for a function
- `data()` lists all datasets provided in R
- `exp` and `log` exponentiate and take logs, and default to the natural
- `class` gives object type
- `str` gives details of an object (structure of dataset)
- `names` gives columns names of object
- `length` gives # of entries in a vector
- `levels` gives factors in column
- `identical` returns a logical: whether the two supplied objects are identical
- `table` returns the frequency of unique elements in a vector
- A useful note: using [[“”]] rather than $ returns a data.frame rather than a vector

Vectors

Types of Vectors

Vectors can hold different elements including numbers, strings, or named numbers.

```
codes <- c(380,124, 818)
country <- c("italy", "canada", "egypt")
name_codes <- c(italy = 380, canada = 124, egypt = 818)
```

The function names() can be used to assign names to entries in a vector.

```
names(codes) = country
codes
```

```
##  italy canada  egypt
##    380    124    818
```

Vector Coercion

- Creating a vector with a string in it will lead R to coerce everything else to a string
  - one can force coercion with `as.numeric()`, `as.character()`, and `as.factor()`
  - if coercion fails, R returns an `NA`

The Integer Class

- A distinct class from numeric is the integer, can be created by adding "L" after a number
  - useful, because it takes up less space than a number

Sorting and Finding Elements Vectors can be sorted in various ways.

- `sort` orders the column
- `order` produces an index which, when applied to the vector, would sort it
  - e.g. writing `index = order(x)` followed by`x[index]` returns x in ascending order
  - one column can be ordered by the order of another using its index
- `max` returns the largest element
- `which.max` returns which element is the max
- `which.mix` is the same as above for the min
- `rank` returns the rank of each number in a vector (from lowest to highest, -x gives highest to lowest)

Data Frames

Data frames are a more versatile type of matrix that can hold multiple types of elements.

```
sample_dat <- data.frame(NameOfColumn1 = "VectorObject1", NameofColumn2 = "VectorObject2")
```

Some notes: * arithmetic operations occur element-wise * indexing refers to ordering based on a certain criterion * a useful way to create a new column is to create a var from manipulation of columns + e.g. col1/col2 = col3; df$col3 = col3 * `stringsAsFactors` within the `data.frame` function can be used to prevent it from turning characters into factors

Indexing Functions

- one can create an index by defining a var e.g. "index" as a logical
  - one can thaen subset based on the logical with data[index]
- `sum` coerces TRUE to numeric i.e. 1, so sum() of logical gives total TRUE
- To satisfy multiple conditions, create a variable for each, and then use the and operator "&" to create an index that contains each
- `which` gives entries for which something is true
- `match` give it a vector of names and a column, and it returns the row # where they appear
- `%in%` checks to see if the elements of a first vector are in a second vector

An exmple using `which()` and `%in%` in concert to identify which components of one vector are present in another:

```
abbs <- c("MA", "ME", "MI", "MO", "MU")
ind <- which(!abbs%in%murders$abb)
abbs[ind]
```

```
## [1] "MU"
```

Dplyr Package

The `dplyr` package contains useful packages for manipulating datasets.

```
install.packages("dplyr")
library(dplyr)
```

Mutate

Create new columns with `mutate`.

```
murders <- mutate(murders, rate= total/population*100000)
```

```
## Warning: package 'bindrcpp' was built under R version 3.2.5
```

Note: total and population are not defined; mutate knows to look to column names

Filter

Select rows based on specified criteria with `filter`.

```
filter(murders, rate < 0.71) %>% head()
```

```
##           state abb        region population total      rate
## 1        Hawaii  HI          West    1360301     7 0.5145920
## 2          Iowa  IA North Central    3046355    21 0.6893484
## 3 New Hampshire  NH     Northeast    1316470     5 0.3798036
## 4  North Dakota  ND North Central     672591     4 0.5947151
## 5       Vermont  VT     Northeast     625741     2 0.3196211
```

Select

Subset the columns of a data frame with `select`.

```
new_table <- select(murders, state, region, rate)
```

The Pipe

The pipe, part of the tidyverse, is a useful way to apply multiple operations to the same dataset without creating intermediate variables.

Rather than,

```
new_table <- select(murders,state, region,rate)
filter(new_table, rate < 0.71) %>% head()
```

```
##           state        region      rate
## 1        Hawaii          West 0.5145920
## 2          Iowa North Central 0.6893484
## 3 New Hampshire     Northeast 0.3798036
## 4  North Dakota North Central 0.5947151
## 5       Vermont     Northeast 0.3196211
```

We can write:

```
murders %>% select(state, region, rate) %>% filter(rate > 0.71) %>% head()
```

```
##        state region     rate
## 1    Alabama  South 2.824424
## 2     Alaska   West 2.675186
```

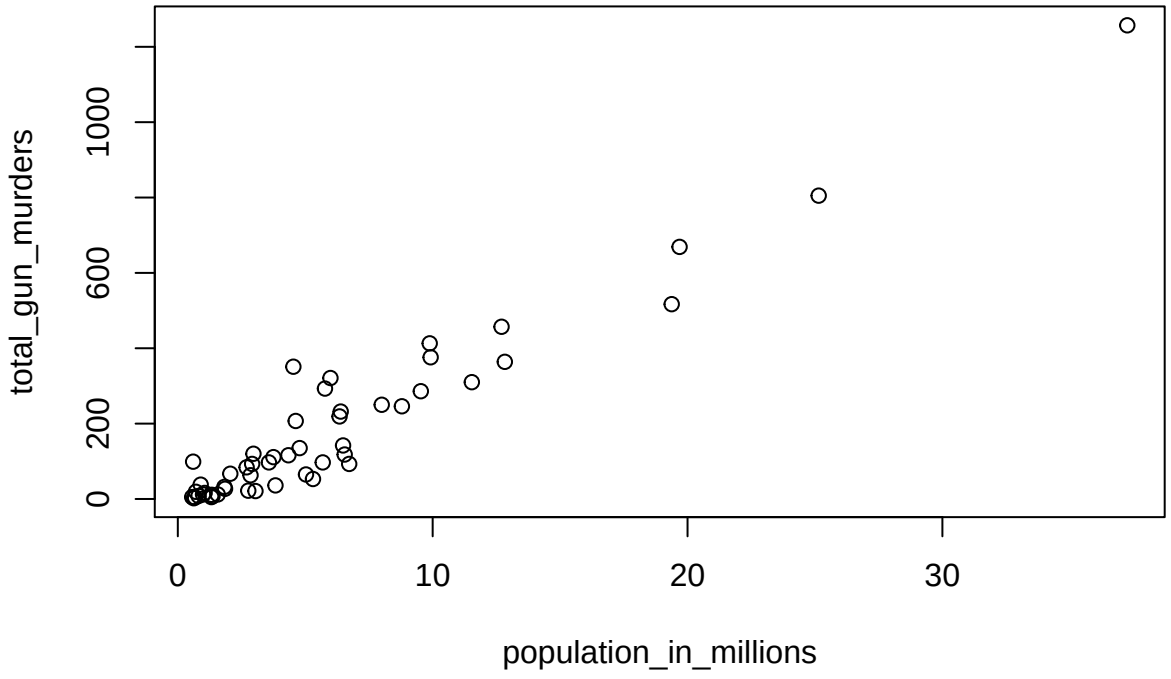
```
## 3    Arizona   West 3.629527
## 4   Arkansas  South 3.189390
## 5 California   West 3.374138
## 6   Colorado   West 1.292453
```

Basic Plots

Basic Plotting

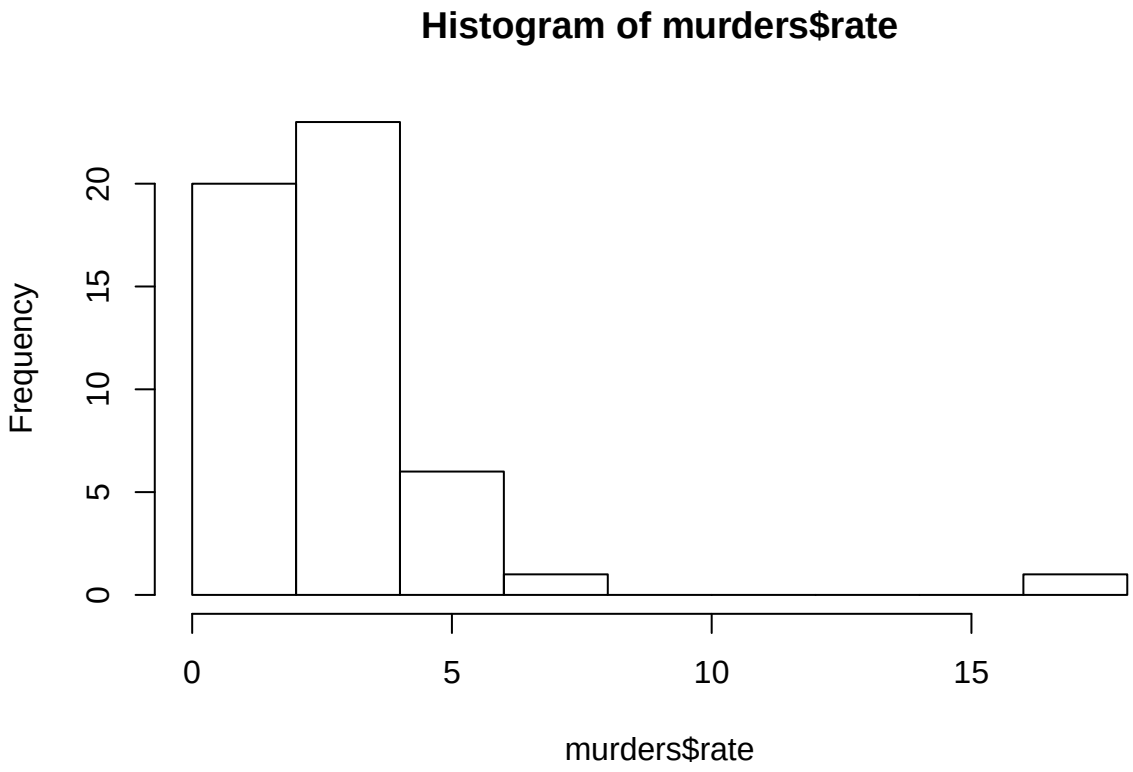
Base R provides the `plot` function for rendering scatterplots,

```
population_in_millions <- murders$population/10^6
total_gun_murders <- murders$total
plot(population_in_millions, total_gun_murders)
```



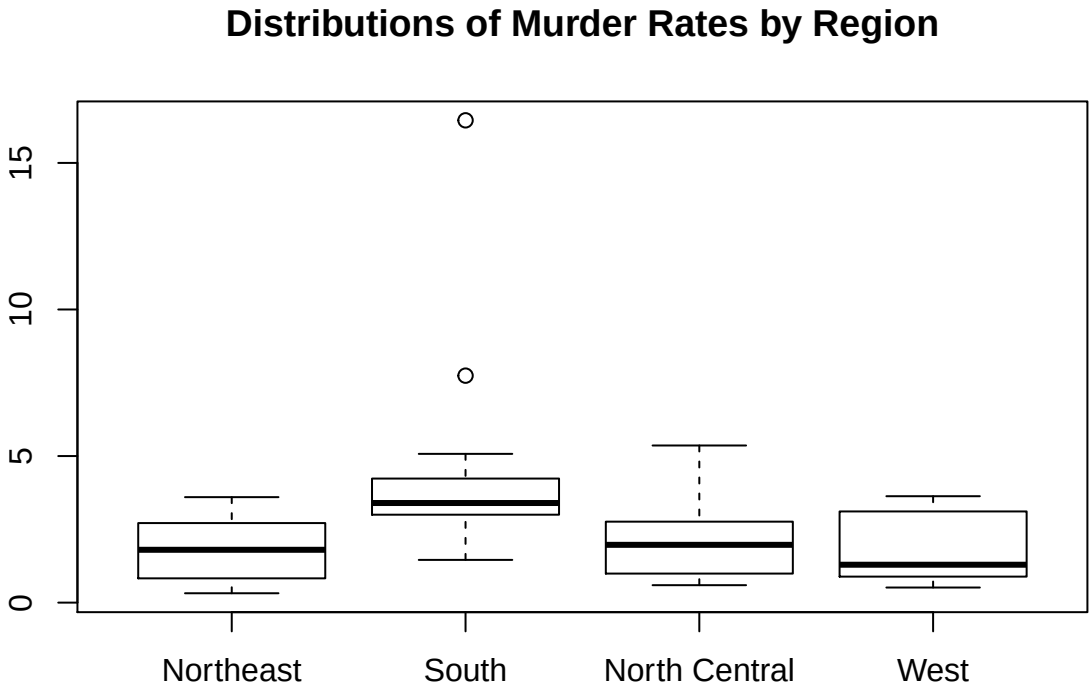
as well as the `hist` function for creating histograms:

```
hist(murders$rate)
```



Boxplots are generated with `boxplot`:

```
boxplot(rate~region, data = murders, main = "Distributions of Murder Rates by Region")
```



Basic Programming in R

If Else Statemenets

A sample of an if-else statement:

```
a <- 2
if(a!=0){
  print(1/a)
}else{
  print("No reciprocal for 0.")
}
```

```
## [1] 0.5
```

The general form:

```
if(boolean_condition{
 expressions
}else{
 expressions
})
```

An alternative:

```
ifelse()
  a <- 0
ifelse(a>0, 1/a, NA)
```

Some useful functions:

- any() takes a logical and returns true if any are true
- all() takes a logical and returns true if all are true

Basic Functions

Here's an example of a simple function:

```
avg <- function(x){
  s<- sum(x)
  n <- length(x)
  s/n
}
```

Note: variables defined inside a function are not defined outside, so + I can define `s` within `avg` and then define `s <- 3` outside avg without conflict

Here's the general form:

```
my_function <- function(x){
  operations_on_x
 value_of_final_line_is_returned
}
```

With arguments:

```
my_function <- function(x, arithmetic = TRUE){
 n <-length(x)
 ifelse(arithmetic = TRUE, sum(x)/n, prod(x)^(1/n))
}
```

Note: returns arithmetic mean if TRUE and geometric if false

Using For Loops

An example:

```r
compute_s_n <- function(n){
    x <- 1:n
    sum(x)
}
  compute_s_n(4)
```

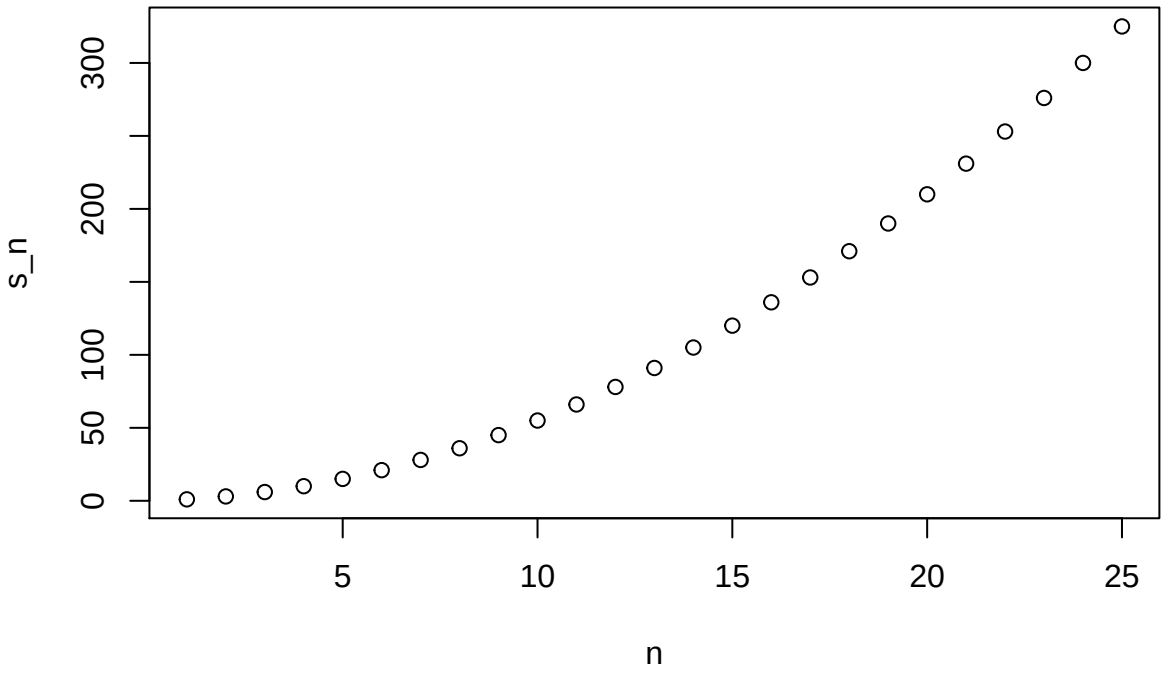
```
## [1] 10
```

General Form of Loops

```
for (i in range_of_values) {
  operations_that_use_i_which_is_changing_across_values
}
```

One example:

```r
m <- 25
s_n <- vector(length = m) #create an empty vector (necessary for a for loop)
for(n in 1:m){ #create for loop to find product for 1:25
  s_n[n] <- compute_s_n(n)
}
n <- 1:25
plot(n, s_n)
```



Conditionals

More function examples:

```r
sum_n <- function(n){ #creates a function called sum(n) that creates a series from 1 to        the inp
  x <- 1:n
  y <- sum(x)
  y
}
```