R Basics

1.1 Data Types

Key Points

- The function “class” helps us determine the type of an object.
- Data frames can be thought of as tables with rows representing observations and columns representing different variables.
- To access data from columns of a data frame, we use the dollar sign symbol, which is called the accessor.
- A vector is an object consisting of several entries and can be a numeric vector, a character vector, or a logical vector.
- We use quotes to distinguish between variable names and character strings.
- Factors are useful for storing categorical data, and are more memory efficient than storing characters.

```
# loading the dslabs package and the murders dataset
library(dslabs)
data(murders)

# determining that the murders dataset is of the "data frame"
class
class(murders)
# finding out more about the structure of the object
str(murders)
# showing the first 6 lines of the dataset
head(murders)

# using the accessor operator to obtain the population column
murders$population
# displaying the variable names in the murders dataset
names(murders)
# determining how many entries are in a vector
pop <- murders$population
```

```
length(pop)
# vectors can be of class numeric and character
class(pop)
class(murders$state)

# logical vectors are either TRUE or FALSE
z <- 3 == 2
z
class(z)

# factors are another type of class
class(murders$region)
# obtaining the levels of a factor
levels(murders$region)
```

1.2 Vectors

Key Points

- The function `c()`, which stands for concatenate, is useful for creating vectors.
- Another useful function for creating vectors is the `seq()` function, which generates sequences.
- Subsetting lets us access specific parts of a vector by using square brackets to access elements of a vector.

Code

```
# We may create vectors of class numeric or character with the
concatenate function

codes <- c(380, 124, 818)

country <- c("italy", "canada", "egypt")


# We can also name the elements of a numeric vector

# Note that the two lines of code below have the same result

codes <- c(italy = 380, canada = 124, egypt = 818)

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


# We can also name the elements of a numeric vector using the
names() function

codes <- c(380, 124, 818)

country <- c("italy","canada","egypt")

names(codes) <- country


# Using square brackets is useful for subsetting to access
specific elements of a vector
```

```
codes[2]

codes[c(1,3)]

codes[1:2]


# If the entries of a vector are named, they may be accessed
by referring to their name

codes["canada"]

codes[c("egypt","italy")]
```

1.3 Sorting

Key Points

- The function `sort()` sorts a vector in increasing order.
- The function `order()` produces the indices needed to obtain the sorted vector, e.g. a result of 2 3 1 5 4 means the sorted vector will be produced by listing the 2nd, 3rd, 1st, 5th, and then 4th item of the original vector.
- The function `rank()` gives us the ranks of the items in the original vector.
- The function `max()` returns the largest value while `which.max()` returns the index of the largest value. The functions `min()` and `which.min()`work similarly for minimum values.
    - The function `as.character()` turns numbers into characters.
    - The function `as.numeric()` turns characters into numbers.

1.4 Vector Arithmetic

Key Points

- In R, arithmetic operations on vectors occur element-wise.

Code

```
# The name of the state with the maximum population is found
by doing the following

murders$state[which.max(murders$population)]


# how to obtain the murder rate

murder_rate <- murders$total / murders$population * 100000


# ordering the states by murder rate, in decreasing order

murders$state[order(murder_rate, decreasing=TRUE)]
```

1.5 Indexing

Key Points

- We can use logicals to index vectors.
- Using the function `>sum()` on a logical vector returns the number of entries that are true.
- The logical operator “&” makes two logicals true only when they are both true.

```
# defining murder rate as before

murder_rate <- murders$total / murders$population * 100000

# creating a logical vector that specifies if the murder rate
in that state is less than or equal to 0.71

index <- murder_rate <= 0.71

# determining which states have murder rates less than or
equal to 0.71

murders$state[index]

# calculating how many states have a murder rate less than or
equal to 0.71

sum(index)
```

```
# creating the two logical vectors representing our conditions

west <- murders$region == "West"

safe <- murder_rate <= 1

# defining an index and identifying states with both
conditions true

index <- safe & west

murders$state[index]
```

1.6 Indexing Functions

Key Points

- The function `which()` gives us the entries of a logical vector that are true.
- The function `match()` looks for entries in a vector and returns the index needed to access them.
- We use the function `%in%` if we want to know whether or not each element of a first vector is in a second vector.

Code

```
# to determine the murder rate in Massachusetts we may do the
following

ind <- which(murders$state == "Massachusetts")

murder_rate[ind]



# to obtain the indices and subsequent murder rates of New
York, Florida, Texas, we do:

ind <- match(c("New York", "Florida", "Texas"), murders$state)

ind

murder_rate[ind]



# to see if Boston, Dakota, and Washington are states

c("Boston", "Dakota", "Washington") %in% murders$state
```

1.7 Basic Data Wrangling

Key Points

- To change a data table by adding a new column, or changing an existing one, we use the `mutate` function.
- To filter the data by subsetting rows, we use the function `filter`.
- To subset the data by selecting specific columns, we use the `select` function.
- We can perform a series of operations by sending the results of one function to another function using what is called the pipe operator, `%>%`.

```
# installing and loading the dplyr package

install.packages("dplyr")

library(dplyr)



# adding a column with mutate

library(dslabs)

data("murders")

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



# subsetting with filter

filter(murders, rate <= 0.71)



# selecting columns with select

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



# using the pipe

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

1.8 Create Data Frames

Key Points

- We can use the `data.frame()` function to create data frames.
- By default, the `data.frame()` function turns characters into factors. To avoid this, we utilize the `stringsAsFactors` argument and set it equal to false.

Code

```
# creating a data frame with stringAsFactors = FALSE

grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),

                     exam_1 = c(95, 80, 90, 85),

                     exam_2 = c(90, 85, 85, 90),

                     stringsAsFactors = FALSE)
```

1.9 Basic Plots

Key Points

- We can create a simple scatterplot using the function `plot()`.
- Histograms are graphical summaries that give you a general overview of the types of values you have. In R, they can be produced using the `hist()` function.
- Boxplots provide a more compact summary of a distribution than a histogram and are more useful for comparing distributions. They can be produced using the `boxplot()` function.

Code

```
# a simple scatterplot of total murders versus population
x <- murders$population / 10^6

y <- murders$total
```

```
plot(x, y)

# a histogram of murder rates

hist(murders$rate)

# boxplots of murder rates by region
boxplot(rate~region, data = murders)
```

1.10 Basic Conditionals

Key Points

- The most common conditional expression in programming is an if-else statement, which has the form "if [condition], perform [expression], else perform [alternative expression]".
- The ifelse() function works similarly to an if-else statement, but it is particularly useful since it works on vectors by examining each element of the vector and returning a corresponding answer accordingly.
- The any() function takes a vector of logicals and returns true if any of the entries are true.
- The all() function takes a vector of logicals and returns true if all of the entries are true.

Code

```
# an example showing the general structure of an if-else
statement
a <- 0

if(a!=0){

  print(1/a)

} else{

  print("No reciprocal for 0.")

}
```

```r
# an example that tells us which states, if any, have a murder
rate less than 0.5
library(dslabs)

data(murders)

murder_rate <- murders$total / murders$population*100000
ind <- which.min(murder_rate)

if(murder_rate[ind] < 0.5){

  print(murders$state[ind])

} else{

  print("No state has murder rate that low")

}

# changing the condition to < 0.25 changes the result
if(murder_rate[ind] < 0.25){

  print(murders$state[ind])

} else{

  print("No state has a murder rate that low.")

}

# the ifelse() function works similarly to an if-else conditional
a <- 0

ifelse(a > 0, 1/a, NA)

# the ifelse() function is particularly useful on vectors
a <- c(0,1,2,-4,5)

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

# the ifelse() function is also helpful for replacing missing
values
data(na_example)

no_nas <- ifelse(is.na(na_example), 0, na_example)
```

```
sum(is.na(no_nas))

# the any() and all() functions evaluate logical vectors
z <- c(TRUE, TRUE, FALSE)

any(z)

all(z)
```

1.11 Basic Functions

Key points

- The R function, called function() tells R you are about to define a new function.
- Functions are objects, so must be assigned a variable name with the arrow operator.
- The general way to define functions is: (1) decide the function name, which will be an object, (2) type `function()` with your function's arguments in parentheses, (3) write all the operations inside brackets.
- Variables defined inside a function are not saved in the workspace.

Code

```
# example of defining a function to compute the average of a
vector x
avg <- function(x){

  s <- sum(x)

  n <- length(x)

  s/n

}

# we see that the above function and the pre-built R mean()
function are identical
x <- 1:100

identical(mean(x), avg(x))
```

```
# variables inside a function are not defined in the workspace
s <- 3

avg(1:10)

s

# the general form of a function
my_function <- function(VARIABLE_NAME){

  perform operations on VARIABLE_NAME and calculate VALUE

  VALUE

}

# functions can have multiple arguments as well as default
values
avg <- function(x, arithmetic = TRUE){

  n <- length(x)

  ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))

}
```

1.12 For Loops

Key points

- For-loops perform the same task over and over while changing the variable. They let us define the range that our variable takes, and then changes the value with each loop and evaluates the expression every time inside the loop.
- The general form of a for-loop is: "For i in [some range], do operations". This i changes across the range of values and the operations assume i is a value you're interested in computing on.
- At the end of the loop, the value of i is the last value of the range.

Code

```
# creating a function that computes the sum of integers 1
through n
compute_s_n <- function(n){

  x <- 1:n

  sum(x)

}

# a very simple for-loop
for(i in 1:5){

  print(i)

# a for-loop for our summation
m <- 25

s_n <- vector(length = m)  # create an empty vector

for(n in 1:m){

  s_n[n] <- compute_s_n(n)

}

# creating a plot for our summation function
n <- 1:m

plot(n, s_n)

# a table of values comparing our function to the summation
formula
head(data.frame(s_n = s_n, formula = n*(n+1)/2))

# overlaying our function with the summation formula
plot(n, s_n)

lines(n, n*(n+1)/2)
```