2 - Variables, Data Types, Data Structures

Variables

A variable is a name for a value. We can create a new variable by assigning a value to it using <-.

```
width <- 5
```

RStudio provides the Alt+- keyboard shortcut to type the <- assignment operator.

```
width = 5
```

is also valid, but there are reasons to use <- instead of =.

Examples of valid variables names are hello, subject_id, subject.ID, x42. Spaces are not allowed in variable names. Dots (.) are ok in R, unlike in many other languages. Numbers are ok, except as the first character. Punctuation is not allowed, with two exceptions: _ and .

Arithmetic with variables

```
# Area of a square (this is a comment)
width * width
## [1] 25

# Save area in a new "area" variable (also a comment)
area <- width * width
```

We can change a variable's value by assigning it a new value:

```
width <- 10
width
## [1] 10
area
## [1] 25
```

Notice that the value of area we calculated earlier hasn't been updated. Assigning a new value to one variable does not change the values of other variables. This is different from a spreadsheet, but the usual behavior for programming languages.

Saving code in an R script

Once we've created a few variables, we may want to record how they were calculated so we can reproduce them later. The usual workflow is to save your code in an R script (".R file"). In RStudio, you can go to "File/New File/R Script" to create a new R script. Code in your R script can be sent to the console by selecting it or placing the cursor on the correct line, and then pressing Control-Enter (Command-Enter on a Mac).

With this, we can write the "hello, world" program in hello.R:

```
# hello.R: first program in R

myString <- "Hello, World!" # assign a string to a variable

# familiar to Python programmers
print(myString)
```

Try creating and running the above hello.R script in RStudio.

Data Types

R has 6 basic data types:

1. character: "a", "swc"
    - also called "strings"
2. numeric (real or decimal)
    - 2, 15.5
3. integer

        - 2L (the L tells R to store this as an integer)
    4. logical
        - TRUE, FALSE
        - a.k.a. boolean
    5. complex
        - 1+4i (complex numbers with real and imaginary parts)
    6. raw (not discussed)

In addition, everything in R is an object. This is characteristic of many modern scripting languages, like Python and Ruby. R provides functions to examine features of objects, for example:

- class() - what kind of object is it?
- typeof() - what is the object's data type?
- length() - how long is it, i.e., how many elements?
- attributes() - does it have any metadata?

R also provides *is.** functions (functions that start with 'is.') that allow you to query whether an object is of a certain kind. In the R console, type

```
is.
```

followed by a couple of Tabs to see what functions are available.

Data Structures

R has many data structure objects. These include:

- atomic vector (a.k.a. vector)
    - for one-dimensional collections of the same data type (thus, "atomic")
- matrix
    - two-dimensional atomic vectors with dimensions, i.e., the number of rows and columns
    - As with atomic vectors, the elements of a matrix must be of the same data type.
- list
    - for one-dimensional collections of different data types
- array
    - While matrices are confined to two dimensions, arrays can be of any number of dimensions.
- data frame
    - for multi-dimensional collections of different data types
- factors
    - categories

Vectors

A vector is the most common and basic data structure in R and considered the "workhorse" of R. A vector is a collection of one data type ("atomic"). In R, it is usually a collection of numbers. We call the individual numbers "elements" of the vector.

We can make vectors with the c( ) function, for example c(1,2,3). c means "combine". In R **even single numbers are vectors of length=1. Many things that can be done with a single number can also be done with a vector.** For example, arithmetic can be done on vectors as it can be on single numbers.

(Try the following in the R console.)

```
myvec <- c(10,20,30,40,50)
myvec
## [1] 10 20 30 40 50

myvec + 1
## [1] 11 21 31 41 51

# note that myvec is unchanged after the +1 operation above
myvec
## [1] 10 20 30 40 50

newvec <- myvec + 1
newvec
## [1] 11 21 31 41 51

myvec + myvec
## [1]  20  40  60  80 100

# note use of the length function
length(myvec)
## [1] 5

c(60, myvec)
## [1] 60 10 20 30 40 50

c(myvec, myvec)
##  [1] 10 20 30 40 50 10 20 30 40 50

# create a 20-element vector of numbers containing only 0s
v <- vector(mode="numeric", length=20)
```

We will also encounter vectors of character strings, for example "hello" or c('hello','world'). Also we will encounter logical (boolean) vectors, which contain TRUE and FALSE values.

Indexing vectors

Access elements of a vector with [ ], for example myvec[1] (NOT myvec[0]) to get the first element. You can also assign to a specific element of a vector.

(Try the following in the R console.)

```
myvec <- c(10,20,30,40,50)

myvec[1]
```

```
## [1] 10

myvec[2]
## [1] 20

myvec[2] <- 5
myvec
## [1] 10  5 30 40 50
```

Can use a vector to index another vector in order to get a subset (or slice) of a vector.

```
myind <- c(4,3,2)
myvec[myind]
## [1] 40 30  5
```

Equivalently:

```
myvec[c(4,3,2)]
## [1] 40 30  5
```

Sequences

Another way to create a vector is with the : (sequence or range) operator. The vector that is created will be of integer class.

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
```

This can be useful when combined with indexing:

```
items <- c("spam", "eggs", "beans", "bacon", "sausage")
items[1:4]
## [1] "spam"  "eggs"  "beans" "bacon"
```

Sequences are useful for other things, such as a starting point for calculations:

```
x <- 1:10
x*x
##  [1]   1   4   9  16  25  36  49  64  81 100
plot(x, x*x)
```

Useful example 1 - vectors, functions, plotting

Generate 10,000 numbers in a random distribution, organize the numbers based on their frequency of occurrence, and plot the frequencies in a bar chart.

```
# generates vector of 10,000 random numbers in a normal distribution with mean=500,
# standard deviation=100; store in vector called data;
# floor function takes each number from rnorm and removes the decimal point
data <- floor(rnorm(10000, 500, 100))
```

```
# the table function counts frequency of numbers in data; stores in integer table
# called occurrences; also see ?table
occurrences <- table(data)

# generate bar plot from occurrences
barplot(occurrences)
```

Since the data are normally distributed (via the rnorm function), the frequencies of the numbers are highest around the mean (500), as expected.

Do it again, but with one line of code. Add bar chart axis labels this time.

```
barplot(table(floor(rnorm(10000, 500, 100))), xlab="Numbers", ylab="Frequencies")
```

Matrices

In R, matrices are an extension of the numeric or character vectors. They are not a separate type of object but simply an atomic vector with dimensions, i.e., the number of rows and columns. As with atomic vectors, the elements of a matrix must be of the same data type.

A matrix can be created using the matrix() function. The dimensions of the matrix can be defined by specifying appropriate values for the *nrow* and *ncol* arguments when using the matrix() function.

Providing a value for both dimension is not necessary. If one of the dimension is provided, the other is inferred from the length of the data.

```
m <- matrix(1:9, nrow = 3, ncol = 3)
m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

# same result is obtained by providing only one dimension
m <- matrix(1:9, nrow = 3)
m
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

# dimensions of matrix m
dim(m)
[1] 3 3

# matrices are vectors with a class attribute of matrix
class(m)
[1] "matrix" "array"

typeof(m)
[1] "integer"
```

We can see that the matrix is filled column-wise. This can be reversed to row-wise filling by passing TRUE to the argument *byrow*.

```
m <- matrix(1:9, nrow=3, byrow=TRUE)    # fill matrix row-wise
m
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

You can also create a matrix from a vector by setting its dimension using dim().

```
x <- c(1,2,3,4,5,6)
x
[1] 1 2 3 4 5 6

class(x)
[1] "numeric"

dim(x) <- c(2,3)
x
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

class(x)
[1] "matrix" "array"
```

Accessing matrix elements

Elements of a matrix can be referenced by specifying the index along each dimension, i.e., "row" and "column") in single square brackets. Again, note that the row and column index start at 1, not 0.

```
x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

x[1,1]
[1] 1
x[1, 3]
[1] 5

# access entire row or entire column
r1 <- x[1,]  # assign entire row 1 of x to vector r1
c2 <- x[,2]  # assign entire column 2 of x to vector c2

# modify elements of a matrix

x[1,1] <- 0
```

```
x
     [,1] [,2] [,3]
[1,]    0    3    5
[2,]    2    4    6

# something interesting: modify all elements less than 6 (conditional assignment)
x[x < 6] <- 0
x
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    6
```

Lists

In R, lists act as containers. Unlike atomic vectors, the contents of a list are not restricted to a single type and can encompass any mixture of data types.

Lists are sometimes called generic vectors, because the elements of a list can by of any type of R object, even lists containing further lists.

Create lists using the list() function or convert other objects to lists using the as.list() function. An empty list of the required length can be created using the vector() function

```
x <- list(1, "a", TRUE, 1+4i)

# the elements of a list are designated using double square brackets [[]]
x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i

x <- vector("list", length = 5) # empty list created using vector()
length(x)
[1] 5
```

The content of elements of a list can be retrieved by using double square brackets.

```
x <- list(1, "a", TRUE, 1+4i)
x[[2]]
[1] "a"
```

Vectors can be "coerced" into lists as follows:

```
# create integer vector x
x <- 1:10
class(x)
[1] "integer"

# make a list from x and then assign it back to x
x <- as.list(x)
class(x)
[1] "list"
```

Elements of a list can be named, i.e, lists can have the names attribute. This makes lists act like associative arrays (or "dictionaries" or "hashes").

```
x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)
x
$a
[1] 2.5

$b
[1] TRUE

$c
[1] 1 2 3

# a handy function is str() to see a "summary" of a data structure
str(x)
List of 3
$ a: num 2.5
$ b: logi TRUE
$ c: int [1:3] 1 2 3

# a, b and c are called "tags" which can be used to reference the components of the list x
```

Accessing list components

Lists can be accessed in similar fashion to vectors. Integer, logical or character vectors can be used for indexing.

```
x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)

# index using integer vector
x[c(1:2)]
$a
[1] 2.5

$b
[1] TRUE

# can use negative integer to exclude second component
x[-2]
$a
[1] 2.5
```

```
$c
[1] 1 2 3

# index using a logical vector (where T = TRUE, F = FALSE)
x[c(T,F,F)]
$a
[1] 2.5

# index using a character (string) vector
x[c("a","b")]
$a
[1] 2.5

$b
[1] TRUE
```

Indexing a list with single brackets [] returns a sublist, not the actual component of the list. To return the actual component, double brackets must be used.

```
x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)

a <- x[1]
class(a)
[1] "list"
a
$a
[1] 2.5
str(a)
List of 1
 $ a: num 2.5
# a is not 2.5, but instead, the sublist ["a" = 2.5]

b <- x[[1]]
b
[1] 2.5
class(b)
[1] "numeric"
# b is the numeric 2.5 because we used double x[[1]];
# we could have also used the tag "a", i.e., x[["a"]] or x$a
```

Modify a list

We can change components of a list through reassignment. We can choose any of the component accessing techniques discussed above to modify it. Note that modification may cause reordering of components in a list.

```
x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)

x[["b"]] <- FALSE

> x
```

```
$a
[1] 2.5

$b
[1] FALSE

$c
[1] 1 2 3
```

Add and delete list components

To add components, we simply assign values using new tags. We can delete a component by assigning the value NULL to it.

```
x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)

x[["d"]] <- 1000 # or x$d <- 1000
x[[4]]
[1] 1000

# remove that last component
x[["d"]] <- NULL # or use x$d

str(x)
```

Alternative $ operator to access list contents

The **$** operator is used often to access the contents of a list, as an alternative to **[[]]**.

```
x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)

x[['d']] <- 1000    # standard [[]] operator
x[['d']]
[1] 1000

x$d <- 2000         # alternative $ operator
x$d
[1] 2000
```

Data Frames

The data frame is a very important data type in R. It is used most for tabular data.

A data frame is a special type of *list* where every component of the *list* has the same length. A data frame is like a "rectangular" list.

We can create a data frame using the data.frame() function.

```
db <- data.frame("ID" = 1:2, "Age" = c(21,20), "Name" = c("John C. Student", "Dora T. Explorer"))
# db is a special list in which each component is a vector of two elements.
# The data is tabularized by default. We have a table of two rows and three columns.
db
  ID Age             Name
1  1  21  John C. Student
2  2  20 Dora T. Explorer

# db is a special case of a list, which is a data.frame object.
> typeof(db)
[1] "list"
> class(db)
[1] "data.frame"
```

Access data frame components

Components of data frame can be accessed like a list or like a matrix.

Access data frame like a list:

```
db <- data.frame("ID" = 1:2, "Age" = c(21,20), "Name" = c("John C. Student", "Dora T. Explorer"))

# Using single [] returns a sub data.frame for the "Age" component.
db['Age']
  Age
1  21
2  20

# Using double[[]] returns a vector containing the values of the "Age" component.
db[['Age']]
[1] 21 20

# Using $ is the same as using [[]], only simpler to type.
db$Age
[1] 21 20
```

Access data frame like a matrix by providing index for row and column:

```
db <- data.frame("ID" = 1:2, "Age" = c(21,20), "Name" = c("John C. Student", "Dora T. Explorer"))

# Select 2nd and 3rd columns only. The result is a sub data.frame.
db[2:3]      # also works: db[,2:3]
  Age             Name
1  21  John C. Student
2  20 Dora T. Explorer

# Select 1st through 2nd rows. Note use of the comma. The result is a sub data.frame.
db[1:2,]
  ID Age             Name
1  1  21  John C. Student
2  2  20 Dora T. Explorer
```

```
# Conditional selection: select all rows with 'Age' > 20. (note use of comma ,)
# This access mixes list and matrix notation. The result is a sub data.frame.
db[db$Age > 20,]
  ID Age            Name
1  1  21 John C. Student
```

Useful data frame functions

- head() - shows first 6 rows
- tail() - shows last 6 rows
- dim() - returns the dimensions of data frame (i.e. number of rows and number of columns)
- nrow() - number of rows
- ncol() - number of columns
- str() - structure of data frame - name, type and preview of data in each column
- names() or colnames() - both show the names attribute for a data frame
- sapply(dataframe, class) - shows the class of each column in the data frame

The data frames (and other datasets) already baked into R

R comes with several datasets ready to use for practice. These datasets can be listed using

```
library(help = 'datasets')

or

data()
```

The datasets are already loaded into the R console in the form of data frames, for example, the the **trees** data frame.

```
?trees       # or help(trees) to get detailed information about this dataset

# dim(trees) and str(trees) will tell us that we have a data frame with 31 rows and 3 columns.
dim(trees)
[1] 31  3
str(trees)
'data.frame':31 obs. of  3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
 $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
 $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 …

# select rows with Height greater than 82
trees[trees$Height > 82,]
   Girth Height Volume
6   10.8     83   19.7
17  12.9     85   33.8
18  13.3     86   27.4
31  20.6     87   77.0
```

Useful example 2 - data frames, functions, plotting

In this example, we'll use the built-in R dataset called ***iris***.

```
# see what we're working with
?iris        # or help(iris) to get detailed information about this dataset

# see all the data or portions of the data
iris # or head(iris) or tail(iris)

# see what we're working with in more detail
sapply(iris, class)

?plot # or help(plot): get help for the plot() function

plot(iris$Species)                           # categorical variable (factor)
plot(iris$Petal.Length)                      # quantitative variable
plot(iris$Species, iris$Petal.Width)         # cat x quant; get a boxplot
plot(iris$Petal.Length, iris$Petal.Width)    # quant pair
plot(iris)                                   # plot entire data frame

# scatter plot with additional options
plot(iris$Petal.Length, iris$Petal.Width,
  col = "#cc0000",                                  # hex code for color red
  pch = 19,                                         # plotting character: solid circle
  main = "Iris: Petal Length vs. Petal Width",      # plot title
  xlab = "Petal Length",
  ylab = "Petal Width")

# get summary statistics for whole data frame
summary(iris)

# get summary and descriptive statistics for particular Species
virginica <- iris[iris$Species == 'virginica',]     # creates a virginica-only data frame

summary(virginica)
mean(virginica$Petal.Width)
median(virginica$Petal.Width)
quantile(virginica$Petal.Width)
range(virginica$Petal.Width)     # also can use max() and min() functions
```

Useful example 3 - data frames, functions, plotting

```
# create data frame from two built-in vectors: state.abb and state.area
st_area <- data.frame('ST' = state.abb, 'Area' = state.area)

# use attach() to add data frame to R search path to shorten variable specifiers
# https://www.statmethods.net/management/sorting.html
```

```
attach(st_area)

# sort the data frame (descending order) by the specified variable(s)
# https://www.statmethods.net/management/sorting.html
sorted_st_area <- st_area[order(Area, decreasing=TRUE),]

# without calling the attach(st_area) function above, we would have needed to create
# sorted_st_area like this instead:
# sorted_st_area <- st_area[order(st_area[["Area"]], decreasing=TRUE),]
# or
# sorted_st_area <- st_area[order(st_area$Area, decreasing=TRUE),]

# disable scientific notation for barplot y axis
# https://statisticsglobe.com/disable-exponential-scientific-notation-in-r
options(scipen = 999)

# barplot of sorted_st_area with sideways x axis labels (las=2)
# https://www.researchgate.net/post/How-cloud-I-have-all-X-label-in-my-box-plot
barplot(sorted_st_area$Area, names.arg = sorted_st_area$ST, las=2, main='State Areas in Square
Miles')
```