Computational Social Science

Data structures

Dr. Thomas Davidson

Rutgers University

January 24, 2022

Plan

- Basic types
- Vectors
- Lists
- Matrices
- Data frames and tibbles

Basic types

There are four basic types we will be using throughout the class. Here I used them to record some information about one of my cats. In R, it is convention to use the <- operator to assign an object to a name.

```
# Character (also known as "strings")
name <- "Gary"
# Numeric ("float" in Python)
weight <- 13.2
# Integer ("int" for short)
age <- 4L
# Logical
human <- FALSE
```

The other two are called `complex` and `raw`. See documentation

Basic types

There are a few useful commands for inspecting objects.

```
print(name) # Prints value in console
```

```
## [1] "Gary"
```

```
class(name) # Shows class of object
```

```
## [1] "character"
```

```
typeof(name) # Shows type of object, not always equal to class
```

```
## [1] "character"
```

Basic types

```
print(weight) # Prints value in console
```

```
## [1] 13.2
```

```
class(weight) # Shows class of object
```

```
## [1] "numeric"
```

```
typeof(weight) # Shows type of object, not always equal to class
```

```
## [1] "double"
```

Basic types

We can use the == expression to verify the value of an object. We will discuss this in more detail next lecture.

```
name == "Tabitha"
```

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

```
age == 3L
```

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

```
age >= 3L # is greater than
```

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

```
age != 3L # is not
```

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

Vectors

A vector is a collection of elements of the *same* type. We can define an empty vector with N elements of a type. Empty vectors assume certain default values depending on the type.

```
N <- 5
x <- logical(N)
print(x)
```

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

```
y <- numeric(N)
print(y)
```

```
## [1] 0 0 0 0 0
```

```
z <- character(N)
print(z)
```

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

Vectors

Let's take a closer look at numeric vectors. We can use the combine function `c()` to concatenate multiple values into a vector.

```
v1 <- c(1,2,3,4,5)
v2 <- c(1,1,1,1,1)
class(v1) # check the class of v1
```

```
## [1] "numeric"
```

Vectors

We can easily perform various mathematical operations on numeric vectors

```
v1 + v2 # addition
```

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

```
v1 - v2 # subtraction
```

```
## [1] 0 1 2 3 4
```

```
v1 * v2 # multiplication
```

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

```
sum(v1) # sum over v1
```

```
## [1] 15
```

Note how the different methods return different types of outputs. The arithmetic operations return vectors while `sum` returns a numeric value.

Vectors

What happens if we try to combine objects of different types using combine?

```
t <- c("a", 1, TRUE)
typeof(t)
```

```
## [1] "character"
```

```
t
```

```
## [1] "a"    "1"    "TRUE"
```

Vectors

There are lots of commands for generating special types of numeric vectors. Note how `N` has already been defined above.

```
seq(N) # generates a sequence from 1 to N
```

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

```
rev(seq(N)) # reverses order
```

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

```
rnorm(N) # samples N times from a normal distribution
```

```
## [1] -0.1267183  0.8250173 -0.3284504  0.9216282  2.3323050
```

```
rbinom(N,1,0.5) # N observations of a single trial with a 0.5 probabili
```

```
## [1] 1 0 0 1 1
```

Vectors

We can use the help ? command to find information about each of these commands.

```
?rnorm
```

Vectors

We can use the `index` to access the specific elements of a vector. R uses square brackets for such indexing.

```
x <- rnorm(N)
print(x)
```

```
## [1] -0.1504429  1.4684573  1.1609942 -0.9365067 -0.1285665
```

```
print(x[1]) # R indexing starts at 1; Python and some others start at 0
```

```
## [1] -0.1504429
```

```
x[1] <- 9 # We can combine indexing with assignment to modify elements
print(x[1])
```

```
## [1] 9
```

Vectors

The `head` and `tail` commands are useful when we're working with larger objects. Here we draw 10,000 observations from an normal distribution.

```
x <- rnorm(10000)
length(x)
```

```
## [1] 10000
```

```
head(x)
```

```
## [1]  1.140705  1.157497 -1.035728  1.199136  2.768524  1.611434
```

```
tail(x)
```

```
## [1]  0.1519312 -1.2212472 -1.0595300  1.0854631  1.0778022 -0.167315
```

```
head(x, n=10)
```

```
##  [1]  1.14070521  1.15749718 -1.03572822  1.19913611  2.76852433  1.
##  [7]  0.49457020  0.48552104  1.49792913 -0.05427134
```

Vectors

Retrieve the final element from `x` using indexing.

```
# Delete this comment and write answer here
```

Vectors

Vectors can also contain null elements to indicate missing values, represented by the NA symbol.

```
x <- c(1,2,3,4,NA)
is.na(x) # The is.na function indicates whether each value is missing.
```

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

```
!is.na(x) # Prepending ! denotes the inverse of a logical operation
```

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

NA is a logical type but can exist within numeric and character vectors. It is an exception to the rule discussed above regarding the presence of multiple types in the same vector.

Lists

A list is an object that can contain different types of elements, including basic types and vectors.

```
print(v1)
```

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

```
v1.l <- list(v1) # We can easily convert the vector v1 into a list.
print(v1.l)
```

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

Lists

Lists have a slightly complex form of indexing. This can be one of the most confusing aspects of R!

```
v1.l[1] # The first element of the list contains the vector
```

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

```
v1.l[[1]] # Double brackets allows us to access the vector itself
```

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

```
class(v1.l[1]) # first element is a list
```

```
## [1] "list"
```

```
class(v1.l[[1]]) # double indexing gives us the contents
```

```
## [1] "numeric"
```

Lists

We can access specific elements of a list by using standard indexing.

```
v1.l[[1]][1] # Followed by single brackets to access a specific element
```

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

```
v1.l[1][1] # If we're not careful, we will just get the entire sublist
```

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

Lists

We can easily combine multiple vectors into a list.

```
v.list <- list(v1,v2) # We could store both vectors in a list
print(v.list)
```

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

```
v.list[[2]][4] # We can use double brackets to get element 4 of list 1
```

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

Lists

We index sublists using double brackets, then specific elements with single brackets.

```
v.list[[2]][4] # We can use double brackets to get element 4 of list 1
```

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

Lists

We can make indexing easier if we start with an empty list and then add elements using a named index via the $ operator.

```
v <- list() # initialize empty list
v$v1 <- v1 # the $ sign is used for named indexing
v$v2 <- v2
print(v)
```

```
## $v1
## [1] 1 2 3 4 5
##
## $v2
## [1] 1 1 1 1 1
```

Lists

Combine $ and square bracket indexing to extract the 5th element of `v1` from `v`.

```
# Add code here
```

Lists

We can define lists more concisely by providing sublists as named arguments.

```
cats <- list(names = c("Gary", "Tabitha"), ages = c(4,2))
print(cats)
```

```
## $names
## [1] "Gary"    "Tabitha"
##
## $ages
## [1] 4 2
```

See Chapter 20 of *R4DS* for more on lists and vectors.

Matrices

A matrix is a two-dimensional data structure. Like vectors, matrices hold objects of a single type. Here we're defining a matrix using two arguments, the number of rows and columns.

```
matrix(nrow=5,ncol=5) # Here there is no content so the matrix is empty
```

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

Matrices

We can also pass an argument to define the initial contents of a matrix.

```
M <- matrix(0L, nrow=5, ncol=5) # 5x5 matrix of zeros
M
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    0
```

Matrices

We can create a matrix by combining vectors using `cbind`.

```
M1 <- cbind(v1,v2) # Treat vectors a columns
print(M1)
```

```
##      v1 v2
## [1,]  1  1
## [2,]  2  1
## [3,]  3  1
## [4,]  4  1
## [5,]  5  1
```

Matrices

If we want to treat the vectors as rows, we alternatively use `rbind`. We could also get the same result by *transposing* `M1`.

```
M2 <- rbind(v1, v2) # Vectors as rows
print(M2)
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2    1    1    1    1    1
```

```
print(t(M1)) # t() is the transpose function
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2    1    1    1    1    1
```

Matrices

The `dim` function provides us with information about the dimensions of a given matrix. It returns the number of rows and columns.

```
dim(M1) # Shows the dimensions of the matrix
```

```
## [1] 5 2
```

```
dim(M2)
```

```
## [1] 2 5
```

Matrices

We can get particular values using two-dimensional indexing. By convention `i` denotes the row and `j` the column.

```
i <- 1 # row index
j <- 2 # column index
M1[i,j] # Returns element i,j
```

```
## v2
##  1
```

```
M1[i,] # Returns row i
```

```
## v1 v2
##  1  1
```

```
M1[,j] # Returns column i
```

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

Matrices

Like lists, we can also name rows and columns to help make indexing easier. The `colnames` and `rownames` functions show the names of each column and row.

```
colnames(M1)
```

```
## [1] "v1" "v2"
```

```
rownames(M1)
```

```
## NULL
```

Matrices

We can use these functions to assign new names.

```
colnames(M1) <- c("X", "Y")
rownames(M1) <- seq(1, nrow(M1))
print(M1)
```

```
##   X Y
## 1 1 1
## 2 2 1
## 3 3 1
## 4 4 1
## 5 5 1
```

Data frames

Like its component vectors, a matrix contains data of the same type. If we have a mix of data types we can use a `data.frame`. Note how the printed version shows the type of each column.

```
df <- as.data.frame(M1) # convert matrix to data frame
class(df)
```

```
## [1] "data.frame"
```

```
df$Z <- c("a","b", "c", "d", "e") # assign new column
print(df)
```

```
##   X Y Z
## 1 1 1 a
## 2 2 1 b
## 3 3 1 c
## 4 4 1 d
## 5 5 1 e
```

Data frames

We can use indexing in the same way as lists to extract elements. I recommend always using $ indexing where possible.

```
data(iris) # The `data` function loads a built in dataset
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
```

```
iris$Sepal.Length[1] # explicitly call column name
```

```
## [1] 5.1
```

```
iris[[1]][1] # reference column using index
```

```
## [1] 5.1
```

Tibbles

A `tibble` is the `tidyverse` take on a data.frame. We can easily convert any `data.frame` into a `tibble`.

```
library(tidyverse) # the library is required to use the as_tibble funct
iris.t <- as_tibble(iris) # convert to tibble
class(iris.t)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Tibbles

Tibbles only show the first ten rows when printing (both look the same in RMarkdown, so we have to use the console to compare.) Tibbles also provide information on the type of each variable.

```
#print(iris)
print(iris.t)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            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           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
```

Tibbles

Tibbles also tend to provide more warnings when potential issues arise, so they should be less prone to errors than data frames.

```
iris$year # data.frame shows null
```

```
## NULL
```

```
iris.t$year # tibble provides a warning
```

```
## Warning: Unknown or uninitialised column: `year`.
```

```
## NULL
```

Questions?