

Social Data Science

Data structures

Dr. Thomas Davidson

Rutgers University

September 8, 2021

Plan

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

Basic types

There are four basic types we will be using throughout the class. We 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:

<https://cran.r-project.org/doc/manuals/R-lang.html>

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

```
name == "Gary"
```

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

```
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* class

We can define an empty vector with N elements of a class

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

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

```
y <- numeric(5)
print(y)
```

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

```
z <- character(5)
print(z)
```

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

Vectors

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

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

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

```
v1 + v2 # addition
```

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

```
v1 * v2 # multiplication
```

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

```
v1 - v2 # subtraction
```

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

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


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. For example,

```
N <- 10 # Value to be used in arguments below
```

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

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

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

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

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

```
## [1] -1.65152535 0.49540575 1.27699530 -0.67431480 0.09245084 0.
```

```
## [7] 0.69071313 0.31415135 -2.79077117 -1.37216415
```

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

```
## [1] 0 0 0 0 0 0 1 0 0 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] 0.2637474 -1.2964717 -0.5641101 -1.6795762 -0.1985738 0.66292
## [7] 0.4114438 0.3644258 0.5985457 0.1640658
```

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

```
## [1] 0.2637474
```

```
x[1] <- 9 # We can also use indexing to modify elements
print(x[1])
```

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

Vectors

The head and tail commands are useful when we're working with larger objects.

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

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

```
head(x)
```

```
## [1] 0.96988897 0.77403781 1.18693153 -0.09818676 0.56164299 -0.0
```

```
tail(x)
```

```
## [1] -0.08370761 0.82108422 0.03274637 -1.38500515 -0.60731452 0.2
```

```
head(x, n=20)
```

```
## [1] 0.96988897 0.77403781 1.18693153 -0.09818676 0.56164299 -0.
```

```
## [7] 0.01402258 0.51048648 0.89031908 -1.37052853 1.18564925 0.
```

```
## [13] 0.02767447 1.19471623 1.09821233 1.07229614 -0.53386609 -0.
```

```
## [19] -1.32975456 1.71575083
```

Vectors

Retrieve the final element from `x` using indexing.

Vectors

Vectors can also contain null elements to indicate missing values, represented by NA logical value.

```
x <- c(1,2,3,4,NA)
```

```
x
```

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

```
is.na(x) # The is.na function indicates whether each value is missing.
```

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

Lists

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

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


Lists

Lists have a more complex form of indexing.

```
v1.list[1] # The entire vector is considered the first element of the l
```

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

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

```
v1.list[[1]] # We can access this element by using double brackets
```

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

```
v1.list[[1]][1] # Followed by single brackets to access a specific elem
```

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

```
v1.list[1][1] # Otherwise we get the entire vector
```

```
## [[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 can make indexing easier if we start with an empty list and then add elements using a named index with the \$ operator.

```
v <- list() # initialize empty list  
v$v1 <- v1 # the $ sign allows 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 bracket indexing to get the fourth element of `v`.

Lists

A list could contain a mix of different types. This type of structure forms the backbone of the dataframes we will be using throughout the class.

```
cats <- list(c("Gary", "Tabitha"), c(4,1))  
print(cats)
```

```
## [[1]]  
## [1] "Gary"      "Tabitha"  
##  
## [[2]]  
## [1] 4 1
```

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

A matrix is a two-dimensional data structure.

```
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` and `rbind`.

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

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

```
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
```


Matrices

We can get particular values using two-dimensional indexing.

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

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

```
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 generally want to use a `data.frame`. Note how the printed version shows the type of each column.

```
df <- as.data.frame(M1)
class(df)
```

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

```
df$Z <- c("a","b", "c", "d", "e")
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.

```
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. It is more “opinionated,” which helps to maintain the integrity of your data. It also has some other updated features. 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.)

```
print(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
## 7	4.6	3.4	1.4	0.3	setosa
## 8	5.0	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
## 11	5.4	3.7	1.5	0.2	setosa
## 12	4.8	3.4	1.6	0.2	setosa
## 13	4.8	3.0	1.4	0.1	setosa
## 14	4.2	2.0	1.1	0.1	setosa

Tibbles

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

```
iris$year
```

```
## NULL
```

```
iris.t$year
```

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

```
## NULL
```


Questions?