Computational Sociology

Data structures

Dr. Thomas Davidson

Rutgers University

January 25, 2021

Plan

- Object-oriented programming
- Basic types
- Vectors
- Lists
- Matrices
- Data frames and tibbles
- A note on style

Object-oriented programming

- A paradigm of computer programming
  - We create *objects* of different *classes* such as numbers, strings, and data frames
  - These objects have *attributes*, properties such as data
    - e.g. The numeric object we call `A` has an attribute called `value` equal to `1`
  - Objects are associated with *methods* that allow us to manipulate them
    - e.g. a numeric object might have a method called `add`, such that `A + A` will return `2`.

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.

```r
# Character (also known as called "strings")
name <- "Tom"
# Numeric ("float" in Python)
height <- 6.1
# Integer ("int" for short)
age <- 32L
# Logical
human <- TRUE
```

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)
```

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

```
class(name)
```

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

```
typeof(name)
```

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

```
length(name)
```

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

```
attributes(name)
```

```
## NULL
```

Basic types

```
print(height)
```

```
## [1] 6.1
```

```
class(height)
```

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

```
typeof(height)
```

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

```
length(height)
```

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

```
attributes(height)
```

```
## NULL
```

Basic types

We can also use the == expression to verify the content of an object. For numeric values we can also use some other expressions We will cover Boolean logic and truth statements more next lecture.

```
name == "Tom"
```

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

```
age == 33L
```

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

```
age >= 30L # is greater than
```

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

```
age != 33L # is not
```

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

Vectors

A vector is a collection of elements of the same class

```r
# We can define an empty vector with N elements of a class
x <- logical(5)
print(x)
```

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

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

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

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

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

Vectors

Let's take a closer look at numeric vectors.

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

```
## [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
```

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

Vectors

There are lots of commands for generating special types of numeric vectors. For example,

```
N <- 10

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

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

```
rev(seq(N)) # reverses a vector
```

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

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

```
##  [1]  0.08728531  0.25461834 -1.20297494  0.07799331 -0.72801390  0.
##  [7] -0.33820616 -0.61704464 -0.13819932  1.27234480
```

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

```
##  [1] 1 0 1 0 1 1 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] -0.75640140  1.12319504 -0.35351581 -0.50638240  1.26826112  0.
##  [7] -0.64259237 -2.16981539 -0.09650433 -0.16146804
```

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

```
## [1] -0.7564014
```

```
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] -2.0128660  0.5065504  1.0067731 -0.6848139  0.3764037  1.174502
```

```
tail(x)
```

```
## [1]  0.08628226 -1.57341981  0.02224987 -1.12555047 -1.92411167  0.1
```

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

```
##  [1] -2.012866032  0.506550384  1.006773139 -0.684813860  0.37640369
##  [6]  1.174502036 -1.393600812 -0.219323882 -0.608195734  0.19692315
## [11] -0.359293100 -0.433730925 -1.846887611 -0.396667026 -0.76189870
## [16]  0.006976895 -1.443551579  0.502759462  1.870320562  0.08354797
```

Vectors

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

```
x <- c(1,2,NULL)
x
```

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

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

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

```
v1.list[[1]][1] # Double brackets then single to access a specific elem
```

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

```
v1.list[1][1] # If first brackets are not double we just get the whole
```

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

Note that indexing lists can be confusing so stick to vectors if possible.

Lists

We can easily combine multiple vectors into a list.

```
v.list <- list(v1,v2) # We could store both vectors in a list
v.list[[1]][4] # We can use double brackets to get element 4 of list 1
```

```
## [1] 4
```

Lists

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

```
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

We can then use the $ to index elements of the list.

```
v$v1
```

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

```
v$v1[4] # still need to use brackets to access a specific element
```

```
## [1] 4
```

See https://cran.r-project.org/doc/manuals/R-lang.html#Indexing for more on indexing.

Lists

A list could contain a mix of different types. These can be handy data structures but can also get complex very quickly.

```
L <- list(name, numeric(5), TRUE, c("a","b","c"))
print(L)
```

```
## [[1]]
## [1] "Tom"
##
## [[2]]
## [1] 0 0 0 0 0
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] "a" "b" "c"
```

Matrices

A matrix is a two-dimensional data structure.

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

```
##      [,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` or `rbind`.

```
M1 <- cbind(v1,v2)
print(M1)
```

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

```
M2 <- rbind(v1, v2)
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.

```
i = 1 # row index
j = 1 # column index
M1[i,j] # Returns element
```

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

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

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

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

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

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`.

```
df <- iris
df$a <- 1

df.2 <- df
df.2$b <- 2

df <- "wrggrwrg"
print(df)
```

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

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`.

```
colnames(iris) # gets column names, rownames will print index of each r
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Spe
```

```
nrow(iris) # count rows
```

```
## [1] 150
```

```
ncol(iris) # count columns
```

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

```
dim(iris) # count rows and columns
```

```
## [1] 150   5
```

Data frames

We can use indexing in the same way as lists to extract elements.

```
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.

```
library(nycflights13)
head(flights, n=5)
```

```
## # A tibble: 5 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1   2013     1     1      517            515         2      830
## 2   2013     1     1      533            529         4      850
## 3   2013     1     1      542            540         2      923
## 4   2013     1     1      544            545        -1     1004
## 5   2013     1     1      554            600        -6      812
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distanc
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Tibbles

We can easily convert any `data.frame` into a `tibble` and vice versa.

```
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"
```

```
iris.df <- as.data.frame(iris.t) # convert back to data.frame
class(iris.df)
```

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

Style

A note on style

- Not only do programming languages require a specific syntax to function, but there are also stylistic conventions
- There are packages you can use to automatically style your code (`styler` and `lintr`)
- See https://style.tidyverse.org/ for more info on R style

Style

Some style tips

- Naming
  - Use informative variable names
  - Keep names short
  - Maintain a consistent naming convention
- Use appropriate spacing to make code readable
  - e.g. `a = 1` is preferable to `a=1`
- Try to avoid extremely long expressions
  - Make complex functions modular (more next lecture)
  - Tidyverse uses the `%>%` operator to help with this (more next lecture)
- Comment your code for your future self and others

Questions?