Stat 260, Lecture 11, Vectors

Brad McNeney

Load packages and datasets

```r
library(tidyverse)
```

Reading

- Vectors: Chapter 16 of the printed text or Chapter 20 of the online text.
- Optional: Wickham (2014), Advanced R, Chapter 2, [http://adv-r.had.co.nz/Data-structures.html/]

R Data Structures

- There are four main data structures in R: atomic vectors, lists, matrices and data frames.
- Atomic vectors and lists are 1d objects called vectors, while matrices and data frames are 2d objects
- R has no true scalars; e.g., in `x<-1`, `x` is a vector of length one.
- Use `str()` to see the structure of an object
- Our focus today is on vectors.

Vector properties

- All vectors have a type and length, which you can determine with the `typeof()` and `length()` functions, respectively.
- Vectors can have other "attributes".
  - For example, a factor is an integer vector with a levels attribute.
  - The text calls such vectors "augmented".

Types of objects

- Common types we will encounter are "logical", "integer", "double", "character" and "list".
  - Find the type of an object with `typeof()`.

```
x <- 6 # stores as double by default
typeof(x)
```

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

```
y <- 6L # The "L" suffix forces storage as integer
typeof(y)
```

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

Type *versus* Mode

- In addition to the type of an object, there is its “mode”.
- The mode of an object is generally the same as its type, but the modes are coarser.
  - Notably, integer and double types are both of mode “numeric”.

```
mode(x)
```

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

```
mode(y)
```

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

More on numeric variables

- Note that doubles are floating point (finite-precision, base-2, with floating decimal place) approximations of real numbers.

```
sqrt(2)^2 - 2
```

```
## [1] 4.440892e-16
```

- doubles inlude NaN, Inf, and -Inf for division by zero:

```
c(-1,0,1)/0
```

```
## [1] -Inf  NaN  Inf
```

- What does NA/0 return? Why does this make sense?

Creating Vectors

- Vectors can be either atomic or list
    - The elements of an atomic vector must be the **same** type.
    - Lists can be comprised of **multiple** data types
- Empty vectors can be created by the `vector()` function:

```
# help("vector")
avec <- vector(mode="numeric",length=4)
lvec <- vector(mode="list",length=4)
```

Creating vectors with `c()` and `list()`

- Data vectors can be created with `c()` or `list()`:

```
avec <- c(52,200,77)
lvec <- list(52,200,77,c("grey","thin"))
```

Combining vectors

- Use c() to combine vectors

```
c(avec,c(100,101))
```

```
## [1]  52 200  77 100 101
```

```
c(lvec,TRUE)
```

```
## [[1]]
## [1] 52
##
## [[2]]
## [1] 200
##
## [[3]]
## [1] 77
##
## [[4]]
## [1] "grey" "thin"
##
## [[5]]
## [1] TRUE
```

Examples of vector type and length

```r
typeof(avec)
```

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

```r
length(avec)
```

```
## [1] 3
```

```r
str(avec)
```

```
##  num [1:3] 52 200 77
```

```r
typeof(lvec)
```

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

```r
length(lvec)
```

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

Named vectors

- Vector elements can have names.
- Names can be assigned after the vector has been created, or in the process of creating the vector.

```
names(lvec) = c("age","weight","height","hair")
str(lvec)
```

```
## List of 4
##  $ age   : num 52
##  $ weight: num 200
##  $ height: num 77
##  $ hair  : chr [1:2] "grey" "thin"
```

```
lvec <- list(age=52,weight=200,height=77,hair=c("grey","thin"))
```

NULL

- The absence of a vector is indicated by `NULL`.
- `NULL` is its own type, and is of length 0.

```r
typeof(NULL)
```

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

```r
length(NULL)
```

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

Exercise

- Write a function `append1()` that takes an argument `n`. The function body should (i) initialize an object `x` to `NULL`, (ii) loop from i in 1 to `n` and at each iteration use `c(x,i)` to extend `x` by one element, and (ii) return `x`. Use the `system.time()` function to time `append1()`. In particular, compare the following:

```r
system.time({x <- append1(10000)})
system.time({x <- 1:10000})
```

Subsetting vectors

- Subset with `[` or by name.
- Index values indicate the subset.
- Negative values drop elements.
- Subsetting with a logical vector keeps all elements where there is a `TRUE` in the logical.:

```
lvec[c(1,3)]
```

```
## $age
## [1] 52
##
## $height
## [1] 77
```

```
#lvec[c("age","height")]
#lvec[-2]
#lvec[c(TRUE,FALSE,TRUE)]
```

Extracting vector elements

- Extract individual elements with `[[`, or `$` for named objects:

```
avec[[2]]
```

```
## [1] 200
```

```
lvec[[4]]
```

```
## [1] "grey" "thin"
```

```
lvec$hair
```

```
## [1] "grey" "thin"
```

  - **Exercise** How would you extract 200 from `lvec`? How would you extract the sub-list containing weight and height data from `lvec`? How would you extract `grey` from `lvec`?

Subsetting and assignment

- You can combine subsetting and assignment to change the value of vectors

```
avec
```

```
## [1]  52 200  77
```

```
avec[1:2] <- c(53,215)
avec
```

```
## [1]  53 215  77
```

Assignment and lists

- To assign to a vector element, use `[[` rather than `[`.
  - This is particularly important with assignments to lists.
  - Assignment with `[` requires that the replacement element be of length 1; `[[` does not have this restriction

```
lvec[3:4] <- c("Hi","there")
lvec[3:4]
```

```
## $height
## [1] "Hi"
##
## $hair
## [1] "there"
```

```r
lvec[4] <- c("All","of","this")
lvec[4] # Only used first element of replacement vector
```

```
## $hair
## [1] "All"
```

```r
lvec[[4]] <- c("All","of","this")
lvec[3:4]
```

```
## $height
## [1] "Hi"
##
## $hair
## [1] "All"  "of"   "this"
```

Coercion: atomic vectors to lists

- Atomic vectors can be coerced to lists with `as.list()`:

```
avec = c(age=52,weight=200,height=77)
avec
```

```
##    age weight height
##     52    200     77
```

```
as.list(avec)
```

```
## $age
## [1] 52
##
## $weight
## [1] 200
##
## $height
## [1] 77
```

- **Exercise** The function `as.vector()` coerces objects to vectors. Why doesn't `as.vector(lvec)` appear to do anything?

Coercion: lists to atomic vectors

- Lists can be "flattened" into atomic vectors with `unlist()`:

```
unlist(lvec)
```

```
##    age weight height  hair1  hair2  hair3
##   "52"  "200"   "Hi"  "All"   "of" "this"
```

- Notice how the numeric values are coerced to the more flexible character type.
- The order of flexibility, from least to most, is logical, integer, numeric, character.

Test functions

- Function outputs may depend on the type of an input object.
- The test functions `as.*`, or their tidyverse equivalents `as_*` can be used to test object type.
- Useful functions are `is_logical()`, `is_numeric()`, `is_character()`, `is_list()` and `is_vector()`.

Recycling

- Arithmetic between a longer and shorter object leads to recyling of the shorter object.

```
x <- rep(100,10)
y <- 1:3
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length

##  [1] 101 102 103 101 102 103 101 102 103 101
```

- This is a handy way to add a scalar to a vector, but is dangerous for most anything else.

Generic functions

- Generic functions behave differently depending on the class of input.
- One of the most important generic functions is `print()`.

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x7f920a9d1e10>
## <environment: namespace:base>
```

- `UseMethod("print")` means that this is a generic function that will call different functions (methods) for objects of different classes.

Methods for `print()`

- There are many! Here we just print the first 10.
- In addition, there is a default that prints any object without a defined method.

```r
methods("print")[1:10]
```

```
##  [1] "print.acf"         "print.AES"         "print.all_vars"
##  [4] "print.anova"       "print.anova.lme"   "print.ansi_string"
##  [7] "print.ansi_style"  "print.any_vars"    "print.aov"
## [10] "print.aovlist"
```

Seeing methods with getS3method()

```r
getS3method("print","default")
```

```
## function (x, digits = NULL, quote = TRUE, na.print = NULL, print.gap = NULL,
##     right = FALSE, max = NULL, useSource = TRUE, ...)
## {
##     args <- pairlist(digits = digits, quote = quote, na.print = na.print,
##         print.gap = print.gap, right = right, max = max, useSource = useSour
##         ...)
##     missings <- c(missing(digits), missing(quote), missing(na.print),
##         missing(print.gap), missing(right), missing(max), missing(useSource)
##     .Internal(print.default(x, args, missings))
## }
## <bytecode: 0x7f920b67f528>
## <environment: namespace:base>
```

Defining your own class

- You can create your own `class` for an object and define methods for it.

```
class(lvec) <- "prof" # print(lvec)
print.prof <- function(p){
  cat("The prof is",p$age,"years old, and weighs",p$weight,"pounds\n")
}
print(lvec)
```

```
## The prof is 52 years old, and weighs 200 pounds
```

Exercise

- Create a list of information on this class. The list should have named elements to hold the following information:

class day	Tuesday
class start	12:30pm
class end	2:20pm
final exam	2019/12/07 12:00pm
text book	R for Data Science

- Use dates or date-times for the times and date-times in the above. Assign class `SFUcourse` to the list. Write a function `diff.SFUcourse()` that takes an object of class `SFUcourse` as input and returns the duration of the lecture. (Here "duration" is as discussed in lecture 9 on dates and times.)

Augmented vectors

- Vector attributes and classes can be used to make useful data structures out of vectors.
- Examples include factors, dates, date-times and data frames/tibbles.
- For example, a factor is an integer vector with a `levels` attribute that maps the integer values to the factor levels.

```
ff <- factor(c("a","b","c"))
typeof(ff)
```

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

```
attributes(ff)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

- See the text for a description of dates and date-times as augmented vectors.

Data frames and tibbles

- A tibble is an "improved" data frame.
- Data frames and tibbles are implemented as lists with attributed `names`, `row.names`.
- All elements of a tibble or data frame must be the same length.

```
x <- tibble(a=1:3,b=6:8)
attributes(x)
```

```
## $names
## [1] "a" "b"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
```

- The `tbl_df` and `tbl` aspects of the class are specific to tibbles. Many methods such as print are different for tibbles than data frames, but any method not defined is inherited from the data frame class.