LECTURE 3: DATA STRUCTURES IN R (contd)

STAT598z: Intro. to computing for statistics

Vinayak Rao

Department of Statistics, Purdue University

In [ ]:
```
options(repr.plot.width=3, repr.plot.height=3)
```

SOME USEFUL R FUNCTIONS

```
seq(), seq_len(), min(), max(), length(), range(),
any(), all()
```

Comparison operators:

```
<, <=, >, >=, ==, !=
```

Logical operators:

```
&& , ||, !, & , |, xor()
```

More on coercion:

```
is.logical(),is.integer(),is.double(),is.character()
as.logical(),as.integer(),as.double(),as.character()
```

Coercion often happens implicitly in function calls:

In [ ]:
```
sum(rnorm(10) > 0)
```

Lists (generic vectors) in R

Elements of a list can be any R object (including other lists)

Lists are created using list() :

In [ ]:
```
car <- list("Ford", "Mustang", 1999L, TRUE); length(car)
```

In [ ]:
```
is.list(car)
```

Can have nested lists:

In [ ]:
```
# car, house, cat and sofa are other lists
house <- "Apartment";
cat   <- list("Calico", "Flopsy", 3L);
sofa  <- "Red"
possessions <- list(car, house, cat, sofa, "3000USD")
```

Or lists containing functions:

In [ ]:
```
mean_list <- list(mean, "Calculates mean of input");
```

Lists in R

Elements of a list can be anything (including other lists)

Lists are vectors (but not "atomic vectors")

See: `is.vector(), is.list(), is.atomic()`

In [ ]:
```
car
```

What does concatenating lists do? E.g. `c(car, house)`

What does concatenating a list with a vector do?

What does `unlist()` do?

The str() function

Just as with vectors, can apply `typeof()` and `class()`

Another very useful function is `str()`

Provides a summary of the R object

In [ ]:
```
str(car)
```

In [ ]:
```
people <- c("Alice", "Bob", "Carol")
str(people)
```

Indexing elements of a list

Use brackets `[]` and double brackets `[[]]`

Brackets `[]` return a sublist of indexed elements

In [ ]:
```
car[1]
```

In [ ]:
```
typeof(car[1])
```

Double brackets `[[]]` return element of list

In [ ]:
```
car[[1]]
```

In [ ]:
```
typeof(car[[1]])
```

Vector in double brackets recursively indexes list

In [ ]:
```
possessions[[1]][[1]]
```

In [ ]:
```
possessions[[c(1,1)]]
```

Named lists

Can assign names to elements of a list

In [ ]:
```
names(car) <- c("Manufacturer", "Make", "Year", "Gasoline")
```

In [ ]:
```
car
```

Equivalently, on definition

In [ ]:
```
car <- list("Manufacturer" = "Ford", "Make" = "Mustang",
            "Year" = 1999, "Gasoline" = TRUE )
```

See also `setNames()`

Accessing elements using names

In [ ]:
```
car[c("Manufacturer", "Make")]  # A two-element sublist
```

In [ ]:
```
car[["Year"]]   # A length-one vector
```

In [ ]:
```
car$Year   # Shorthand notation
```

In [ ]:
```
car$year   # R is case-sensitive!
```

Names

The names() function can get/set names of elements of a list

In [ ]:
```
names(car)   # Returns a character vector
```

In [ ]:
```
names(car)[4] <- "Gasoline"; names(car)
```

Names need not be unique or complete

Can remove names using `unname()`

Can also assign names to atomic vectors

Object attributes

`names()` is an instance of an object attribute

These store useful information about the object

Get/set attributes using `attributes()`

In [ ]:
```
attributes(car)
```

Get/set individual attributes using `attr()`

Object attributes

Other common attributes: `class`, `dim` and `dimnames`

Many have specific accessor functions e.g. `class()` or `dim()`

You can create your own

- Warning: careful about the effect of functions on attributes

Matrices and arrays

Are two- and higher-dimensional collections of objects

These have an appropriate `dim` attribute

In [ ]:
```
my_mat <- 1:6  # vector
my_mat
```

In [ ]:
```
dim(my_mat) <- c(2,3) # 2 rows and 3 columns
print(my_mat)
```

Equivalently

In [ ]:
```
my_mat <- matrix(0 , nrow = 2, ncol=3) # ncol is redundant
my_mat
```

Arrays work similarly

In [ ]:
```
my_arr <- array(1 : 8, c(2,2,2)); print(my_arr)
```

Matrices and arrays

Useful functions include

- `typeof(), class(), str()`
- `dim(), nrow(), ncol()`
- `is.matrix(), as.matrix(), ...`
- `dimnames(), rownames(), colnames()`

In [ ]:
```
dimnames(my_mat) <- list(c("r1", "r2"), c("c1", "c2", "c3"))
print(my_mat);my_mat['r1','c2']
```

A vector/list is NOT an 1-d matrix (no dim attribute)

In [ ]:
```
is.matrix(1 : 6)
```

Use `drop()` to eliminate empty dimensions

In [ ]:
```
my_mat <- array(1 : 6, c(2,3,1)) # dim(my_mat) is (2,3,1)
print(my_mat)
```

In [ ]:
```
my_mat <- drop(my_mat) # dim is now (2,3)
print(my_mat)
```

Indexing matrices and arrays

In [ ]:
```
print(my_mat); my_mat[2,3]   # Again, use square brackets
```

Excluding an index returns the entire dimension

In [ ]:
```
my_mat[2,]
```

In [ ]:
```
my_arr[1,,1] # slice along dim 2, with dims 1, 3 equal to 1
```

Usual ideas from indexing vectors still apply

In [ ]:
```
print(my_mat[,c(2,3)])
```

Column-major order

We saw how to create a matrix from an array

In [ ]:
```
my_mat <- (matrix(1 : 6, nrow = 3, ncol = 2)); print((my_mat)); print(t(my_mat)
)
```

In R matrices are stored in column-major order (like Fortran , and unlike C and Python)

In [ ]:
```
print(my_mat[1,2])
```

Recycling

Column-major order explains recycling to fill larger matrices

In [ ]:
```
ones <- matrix(1, nrow = 3, ncol = 3); print(ones)
```

In [ ]:
```
my_seq <- matrix(c(1,2,3), nrow = 3, ncol = 3); print((my_seq))
```

In [ ]:
```
print(t(t(my_seq) + c(.1,.2,.3)))
```