LECTURE 3: DATA STRUCTURES IN R (contd)

STAT598z: Intro. to computing for statistics

Vinayak Rao

Department of Statistics, Purdue University

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 [2]:
```
sum(rnorm(10) > 0)
```

2

Lists (generic vectors) in R

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

Lists are created using list() :

In [3]:
```
car <- list("Ford", "Mustang", 1999, TRUE); length(car)
```

4

In [ ]:
```

```

Can have nested lists:

In [45]:
```
# 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 [5]:
```
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 [ ]:

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 [43]:
```
str(car)
```

```
List of 4
 $ Manufacturer: chr "Ford"
 $ Make        : chr "Mustang"
 $ Year        : num 1999
 $ Petrol      : logi TRUE
```

In [7]:
```
people <- c("Alice", "Bob", "Carol")
str(people)
```

```
 chr [1:3] "Alice" "Bob" "Carol"
```

Indexing elements of a list

Use brackets [ ] and double brackets [[ ]]

Brackets [ ] return a sublist of indexed elements

In [44]:
```
car[1]
```

$Manufacturer = 'Ford'

In [9]:
```
typeof(car[1])
```

'list'

Double brackets [[ ]] return element of list

In [10]:
```
car[[1]]
```

'Ford'

In [11]:
```
typeof(car[[1]])
```

'character'

Vector in double brackets recursively indexes list

In [12]:
```
possessions[[1]][[1]]
```

'Ford'

In [13]:
```
possessions[[c(1,1)]]
```

'Ford'

Named lists

Can assign names to elements of a list

In [14]:
```
names(car) <- c("Manufacturer", "Make", "Year", "Gasoline")
```

In [15]:
```
car
```

$Manufacturer
'Ford'
$Make
'Mustang'
$Year
1999
$Gasoline
TRUE

Equivalently, on definition

In [16]:
```
car <- list("Manufacturer" = "Ford", "Make" = "Mustang",
            "Year" = 1999, "Gasoline" = TRUE )
```

See also `setNames()`

Accessing elements using names

In [17]:
```
car[c("Manufacturer", "Make")]  # A two-element sublist
```

$Manufacturer
'Ford'
$Make
'Mustang'

In [18]:
```
car[["Year"]]   # A length-one vector
```

1999

In [19]:
```
car$Year   # Shorthand notation
```

1999

In [20]:
```
car$year   # R is case-sensitive!
```

NULL

Names

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

In [21]:
```
names(car)   # Returns a character vector
```

'Manufacturer' 'Make' 'Year' 'Gasoline'

In [22]:
```
names(car)[4] <- "Petrol"; names(car)
```

'Manufacturer' 'Make' 'Year' 'Petrol'

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 [23]:
```
attributes(car)
```

$names = 'Manufacturer' 'Make' 'Year' 'Petrol'

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 [24]:
```
my_mat <- 1 : 6  # vector
```

In [25]:
```
dim(my_mat) <- c(2,3) # 2 rows and 3 columns
print(my_mat)
```

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

Equivalently

In [12]:
```
my_mat <- matrix(c(1,2,3) , nrow = 2, ncol = 3) # ncol is redundant
print(my_mat)
```

```
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    1    3
```

Arrays work similarly

In [27]:
```
my_arr <- array(1 : 8, c(2,2,2)); print(my_arr)
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

Matrices and arrays

Useful functions include

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

In [5]:
```
dimnames(my_mat) <- list(c("r1", "r2"), c("c1", "c2", "c3"))
print(my_mat);my_mat['r1','c2']
```

```
   c1 c2 c3
r1  1  3  2
r2  2  1  3
```

3

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

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

FALSE

Use `drop()` to eliminate empty dimensions

In [10]:
```
my_mat <- array(1 : 6, c(2,3,1)) # dim(my_mat) is (2,3,1)
print(my_mat)
```

```
, , 1

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

In [9]:
```
my_mat <- drop(my_mat) # dim is now (2,3)
print(my_mat)
```

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

Indexing matrices and arrays

In [15]:
```
print(my_mat); my_mat[2,3]   # Again, use square brackets
```

```
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    1    3
```

3

Excluding an index returns the entire dimension

In [21]:
```
print(my_mat[-3,-3])
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    1
```

In [22]:
```
my_arr[1,,1] # slice along dim 2, with dims 1, 3 equal to 1
```

```
Error in eval(expr, envir, enclos): object 'my_arr' not found
Traceback:
```

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 [38]:
```
my_mat <- matrix(1 : 6, nrow = 3, ncol = 2); print(t(my_mat))
```

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

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

In [33]:
```
print(my_mat[1,2])
```

```
[1] 4
```

Recycling

Column-major order explains recycling to fill larger matrices

In [35]:
```
ones <- matrix(1, nrow = 3, ncol = 3)
```

In [39]:
```
my_seq <- matrix(c(1,2,3), nrow = 3, ncol = 3); print(my_seq)
```

```
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
```

In [46]:
```
print(t(t(my_seq) + c(.1,.2,.3)))
```

```
     [,1] [,2] [,3]
[1,]  1.1  1.2  1.3
[2,]  2.1  2.2  2.3
[3,]  3.1  3.2  3.3
```

In [ ]: