Data Objects in R

Objects, Modes and Attributes

- The entities that R operates on are known as ***objects.***
- Examples are **vectors** of numeric (real) or complex values, vectors of logical values, and vectors of character strings.
- These are **atomic** structures since their components are all of the same type, or ***mode***:
  - **numeric**
  - **complex**
  - **logical**
  - **character**
  - **others . . .**

Objects, Modes and Attributes

- ***Lists*** are ordered sequences of objects which can be of any mode.
  - Are ***recursive: lists*** can be comprised of other ***lists.***
- ***Functions*** and ***expressions*** are also recursive objects in **R**.
- By ***mode*** of an object we mean the basic type of its fundamental constituents.
- Further properties of an object are provided by **`attributes(object)`**.
- If **`z`** is a complex vector of length 100, then **`mode(z)`** will return “complex” and **`length(z)`** will return 100.

The Class of an Object

- All objects in R have a ***class***, reported by the function **`class(object)`**.
- **Vectors** are the most important type of object in **R,** but there are others that play important roles:
  - **Matrices** (more generally ***arrays***) are multi-dimensional generalizations of **vectors**.
  - **Factors** help handle categorical data.
  - **Lists** are a general form of vector in which the elements can be of different type.
  - **Data frames** are matrix-like structures in which columns can be of different types.
  - **Functions** are objects which can be stored in a project's workspace.

Data Types in R

Data Types in R

- **Double** (is numeric)
- **Integer** (is numeric)
- **Complex**
- **Logical**
- **Character**
- **Factor**
- **Dates and Times**

Double

- The **R** data type ‘double’ represents numbers.
  - Doubles represent continuous variables like the height or weight of a person.
  - You can perform calculations on numbers:

```
> x <- 8.14
> y <- 8.0
> z <- 87.0 + 12.9
> y/z
[1] 0.08008008
```

Double

- Use the function **`is.double()`** to check if an object is of type ‘double’.
- Alternatively, use the function **`typeof()`** to ask R the type of the object x.

```
> typeof(x)
[1] "double"
> is.double(8.9)
[1] TRUE
> test <- 1223.456
> is.double(test)
[1] TRUE
```

Integer

- The **R** data type ‘integer’ is a natural number.
  - They can be used to represent counting variables, for example, the number of children in a household.

```
> nchild <- 3.0
> is.integer(nchild)
[1] FALSE
> typeof(nchild)
[1] "double"
> nchild <- 3
> is.integer(nchild)
[1] FALSE
> typeof(nchild)
[1] "double"
```

Integer

- So a 3 of type ‘integer’ in **R** is something different than a 3.0 of type ‘double’.
- However, you can mix objects of type ‘double’ and ‘integer’ in one calculation without any problems.

```
> x <- as.integer(7)
> y <- 2.0
> z <- x/y
> z
[1] 3.5
> typeof(z)
[1] "double"
```

Complex

- Objects of type 'complex' represent complex numbers. Use the function **`as.complex()`** or **`complex()`** to create objects of type 'complex'.

```
> test1 <- as.complex(-25+5i)
> sqrt(test1)
[1] 0.497543+5.024694i

> test2 <- complex(5,real=2,im=6)
> test2
[1] 2+6i 2+6i 2+6i 2+6i 2+6i
> typeof(test2)
[1] "complex"
```

Logical

- An object of data type ‘logical’ can have the value of **TRUE** or **FALSE** and are used to indicate if a condition is true or false.
- Such objects are used to evaluate a logical expression:

```
> x <- 13
> y <- x > 15
> y
[1] FALSE
```

Logical

- Logical expressions are often built from logical operators:

<	smaller than
<=	smaller than or equal to
>	larger than
>=	larger than or equal to
==	is equal to
!=	is unequal to

Logical

- The logical operators **and**, **or** and **not** are given by **&**, **|** and **!**, respectively.

```
> x <- c(9,166)
> y <- (3 < x) & (x <= 10)
> x
[1]  9 166
> y
[1] TRUE FALSE
```

Logical

- Calculations can also be carried out on logical objects, in which case the **FALSE** is replaced by a zero and a one replaces the **TRUE**. For example, the sum function can be used to count the number of TRUE's in a vector or array:

```
> x <- 1:15
# number of elements in x larger than 9
> sum(x>9)
[1] 6
```

Character

- A ‘character’ data type or object is represented by the collection of characters in between double quotes (“”), for example: “y”, “test character” and “hungry man”.

```
> x <- c(“a”, “b”, “c”)
> x
[1] “a” “b” “c”
> mychar1 <- “This is a test”
> mychar1
[1] “This is a test”
```

Factor

- The ‘factor’ data type is used to represent categorical data, for example
  - variable **gender** with values **male** and **female**
  - variable **blood type** with values **A**, **AB** and **O**.
- Each unique value in the value range is called a ***level*** of the factor variable.
- Factor objects can be created from character objects or from numeric objects. The following creates a vector of length four of data type ‘character’:

```
> gender <- c("male","male","female","male")
```

Factor

- The object **gender** is a character object. You need to transform it to a factor using the function **factor**:

```
> gender <- factor(gender)
> gender
[1] male male female male
Levels: female male
```

Factor

- The function **`levels()`** can be used to determine the levels of a variable of type ‘factor’:

```
> levels(gender)
[1] "female" "male"
```

- Note that the result of the **`levels()`** function is of type ‘character’. One could also create the gender variable as follows

```
> gender <- c(1,1,2,1)
```

Factor

- The object **gender** is an integer variable, you need to transform it to a factor:

```
> gender <- factor(gender)
> gender
[1] 1 1 2 1
Levels: 1 2
```

- You cannot perform arithmetic operations on the **gender** variable:

```
> gender + 6
[1] NA NA NA NA
Warning message: . . + not meaningful . . .
```

Factor

- You can transform factor variables to double or integer variables using the **`as.double()`** or **`as.integer()`** functions:

```
> gender.numeric <- as.double(gender)
> gender.numeric
[1] 1 1 2 1
```

Ordered Factors

- If the order of the levels is important, you should use ordered factors using the function **`ordered()`** and specifying the order with the **levels** argument:

```
> Income <- c("High", "Low","Average","Low","Average","High")
> Income <- ordered(Income, levels=c("Low","Average","High"))
> Income
[1] High Low Average Low Average High
Levels: Low < Average < High
```

Ordered Factors

- The last line indicates the ordering of the levels with the factor variable. When you transform an ordered factor variable, the order is used to assign numbers to the levels:

```
> Income.numeric <- as.double(Income)
> Income.numeric
[1] 3 1 2 1 2 3
```

- The order of the levels is used in linear models and is important for the interpretation of regression coefficient parameter estimates.

Dates and Times

- To represent a calendar date in **R** use the function **`as.Date()`** to create an object of class **Date**.

```
> temp <- c("12-09-1973", "29-08-1974")
> z <- as.Date(temp, "%d-%m-%Y")
> z
[1] "1973-09-12" "1974-08-29"
> data.class(z)
[1] "Date"
> format(z, "%d-%m-%Y")
[1] "12-09-1973" "29-08-1974"
```

Upper case 'Y'

Dates and Times

- If you add a number to a date object, the number is interpreted as the number of days to add to the date:

```
> z + 19
[1] "1973-10-01" "1974-09-17"
```

- If you subtract one date from another, the result is an object of class **difftime**:

```
> z[1] <- "1973-09-12"
> z[2] <- "1974-09-17"
> dz <- z[2]-z[1]
> dz
Time difference of 370 days
> data.class(dz)
[1] "difftime"
```

Data Structures in R

Matrices

- Are several ways of making a **matrix**:

```
> x <- matrix(1:9,nrow=3)
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> class(x)
[1] "matrix"
> attributes(x)
$dim
[1] 3 3
```

Matrices

- Make a matrix row-wise with **byrow=T**:

```
> vector <- c(1,2,3,4,4,3,2,1)
> v <- matrix(vector, byrow=T, nrow=2)
> v
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    4    3    2    1
Another way is to provide vector two dimensions:
> dim(vector) <- c(4,2)
Can check that vector is a matrix:
> is.matrix(vector)
[1] TRUE
```

Matrices

- We will need to **transpose** vector:

```
> vector
     [,1] [,2]
[1,]    1    4
[2,]    2    3
[3,]    3    2
[4,]    4    1
We want the transpose, t, of this matrix:
> vector <- t(vector)
> vector
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    4    3    2    1
```

Naming Matrices Rows and Columns

- Here is a 4 x 5 matrix of random integers from a Poisson distribution with mean 1.5:

```
> x <- matrix(rpois(20,1.5),nrow=4)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    2    5    3
[2,]    1    1    3    1    3
[3,]    3    1    0    2    2
[4,]    1    0    2    1    0
```

We want to label the rows "Trial.1" etc.:

```
> rownames(x) <-
rownames(x,do.NULL=FALSE,prefix="Trial. ")
> x
```

(see next slide)

Naming Matrices Rows and Columns

```
> x (from previous slide)
          [,1] [,2] [,3] [,4] [,5]
Trial.1     1    0    2    5    3
Trial.2     1    1    3    1    3
Trial.3     3    1    0    2    2
Trial.4     1    0    2    1    0
```

We want drug names for the columns:

```
> colnames(x) <- c("aspirin",
"paracetamol","nurofen","hedex","placebo")
> x
```

(see next slide)

Naming Matrices Rows and Columns

```
> x (from previous slide)
          aspi..  pare.. nuro.. hede.. plac..
Trial.1      1       0      2      5      3
Trial.2      1       1      3      1      3
Trial.3      3       1      0      2      2
Trial.4      1       0      2      1      0
```

Could also use the dimnames function:

```
> dimnames(x) <-
list(NULL,paste("drug.",1:5,sep=""))
> x
```

(see next slide)

Naming Matrices Rows and Columns

```
> x (from previous slide)
      drug.1  drug.2 drug.3 drug.4 drug.5
[1,]       1       0      2      5      3
[2,]       1       1      3      1      3
[3,]       3       1      0      2      2
[4,]       1       0      2      1      0
```

Adding Rows and Columns to the Matrix

- Use **`rbind()`** and **`cbind()`** to add rows and columns:

```
> x <- rbind(x,apply(x,2,mean))
> x <- cbind(x,apply(x,1,var))
> x
     [,1] [,2] [,3] [,4] [,5]     [,6]
[1,]  1.0  0.0 2.00 5.00    3  3.70000
[2,]  1.0  1.0 3.00 1.00    3  1.20000
[3,]  3.0  1.0 0.00 2.00    2  1.30000
[4,]  1.0  0.0 2.00 1.00    0  0.70000
[5,]  1.5  0.5 1.75 2.25    2  0.45625
```

Note that number of decimal places varies across columns. Default is minimum number consistent with contents of column.

Adding Rows and Columns to the Matrix

- Label **variance** and **mean** columns and rows:

```
> colnames(x) <- c(1:5,"variance")
> rownames(x) <- c(1:4,"mean")
> x
       1    2    3    4 5 variance
1    1.0  0.0 2.00 5.00 3  3.70000
2    1.0  1.0 3.00 1.00 3  1.20000
3    3.0  1.0 0.00 2.00 2  1.30000
4    1.0  0.0 2.00 1.00 0  0.70000
mean 1.5 0.5 1.75 2.25 2 0.45625
```

Arrays

- **Arrays** are numeric objects with dimension attributes:

```
> array <- 1:25
> is.matrix(array)
[1] FALSE
> dim(array)
NULL
The vector is not a matrix and it has no
(i.e. NULL) dimensional attributes. So:
> dim(array) <- c(5,5)
> dim(array)
[1] 5 5
> is.matrix(array)
[1] TRUE
```

Arrays

```
> array
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
> is.table(array)
[1] FALSE
```

Arrays

- Here is a three-dimensional array of the first 24 lower-case letters with three matrices each of four rows and two columns:

```
> a <- letters[1:24]
> dim(a) <- c(4,2,3)
> a
```

How do these expressions evaluate?:

```
> a[,,1:2]
> a[,,3]
> a[3,,]
```

Arrays

```
> a[,,1:2]
,,1
      [,1] [,2]
[1,]  "a"  "e"
[2,]  "b"  "f"
[3,]  "c"  "g"
[4,]  "d"  "h"
,,2
      [,1] [,2]
[1,]  "i"  "m"
[2,]  "j"  "n"
[3,]  "k"  "o"
[4,]  "l"  "p"
```

Arrays

```
> a[,,3]
     [,1] [,2]
[1,] "q"  "u"
[2,] "r"  "v"
[3,] "s"  "w"
[4,] "t"  "x"

> a[3,,]
     [,1] [,2] [,3]
[1,] "c"  "k"  "s"
[2,] "g"  "o"  "w"
```

Why is the shape of the resulting matrix altered in the last example?

Character Strings

- In **R**, **character strings** are defined by double quotation marks:

```
> a <- "abc"
> b <- "123"
> as.numeric(a)
[1] NA
Warning message:
NAs introduced by coercion

> as.numeric(b)
[1] 123
```

Character Strings

```
> pets <- c("cat", "dog", "gerbil", "terrapin")
```

Here, pets is a vector comprising four character strings:

```
> length(pets)
[1] 4
```

And the individual character strings have 3, 3, 6 and 8 characters, respectively:

```
> nchar(pets)
[1] 3 3 6 8
```

When first defined, character strings are not factors:

```
> class(pets)
[1] "character"
> is.factor(pets)
[1] FALSE
```

Character Strings

R has built-in vectors that contain 26 letters of the alphabet in lower case (letters) and in upper case (LETTERS):

```
> letters
[1] "a" "b" "c" "d" .. etc
> LETTERS
[1] "A" "B" "C" "D" .. etc
```

Can match numbers with letters using `which()`:

```
> which(letters=="n")
[1] 14
> letters[14]
[1] "n"
```

Character Strings

Function `noquote()` *suppresses printed quote marks:*

```
> noquote(letters)
[1] a b c d e f g h i j k l m o p .. etc
```

Can amalgamate strings into character vectors:

```
> a <- "abc"
> b <- "123"
> c(a,b)
[1] "abc" "123"
> paste(a,b,sep="")
[1] "abc123"
```

No separator above; the default in one blank:

```
> paste(a,b)
[1] "abc 123"
```

Character Strings

```
> paste(a,b, "A longer phrase with blanks",sep="")
[1] "abc123A longer phrase with blanks"
When pasting a vector:
> d <- c(a,b, "new")
> e <- paste(d,"A longer phrase")
> e
[1] "abc A longer phrase" "123 A longer phrase"
+   "new A longer phrase" (all on same line)
```

Data Structures: Lists

- In these examples, we have looked at data structures based on a ***single underlying data type***.
- In **R**, one can construct more complicated structures comprised of ***multiple data types***.
- The ‘built in’ data type for mixing objects of different types are ***lists***.
  - Lists can contain a ***heterogeneous selection of objects***.
  - One can ***name each component*** in a list.
  - Items in a list may be referred to by either ***location*** or ***name***.

Data Structures: List Components

- Here is a **list** with two named components:

```
> # a list containing a number and a string
> e <- list(thing="hat",size="8.25")
> e
$thing
[1] "hat"
$size
[1] "8.25"
```

Data Structures: Accessing Lists

- **List** items may be accessed in multiple ways:

```
> e$thing
[1] "hat"

> e[1]
$thing
[1] "hat"

> e[[1]]
[1] "hat"
```

Data Structures: Lists Within Lists

- A **list** can even contain other lists:

```
> g <- list(" this list references another list",e)
> g
[[1]]
[1] "this list references another list"
[[2]]
[[2]]$thing
[1] "hat"
[[2]]$size
[1] "8.25"
```

Data Frames in R

Data Structures:
Data Frames

- A **data frame** is a list that contains multiple named vectors that are the same length.
- Let's construct a data frame with the win/loss results in the National League (NL) East in 2008:

```
> teams <- c("PHI","NYM","FLA","ATL","WSN")
> w <- c(92, 89, 94, 72, 59)
> l <- c(70, 73, 77, 90, 102)
> nleast <- data.frame(teams,w,l)
> nleast
  teams  w   l
1   PHI 92  70
2   NYM 89  73
3   FLA 94  77
4   ATL 72  90
5   WSN 59 102
```

Data Structures:
Data Frames

- You can refer to the ***components*** of a data frame (or items in a list) by name using the **$ operator**:

```
> nleast$w
[1] 92 89 94 72 59
```

- Let's say you wanted to find the number of losses by the Florida Marlins (FLA). You can select a member of an array by using a vector of Boolean values to specify which item to return from a list:

```
> nleast$teams=="FLA"
[1] FALSE FALSE  TRUE FALSE FALSE
```

- Then you can use this vector to refer to the right element in the losses vector:

```
> nleast$l[nleast$teams=="FLA"]
[1] 77
```

Objects and Classes

- R is an ***object oriented language***.
- Every ***object*** in **R** has a ***type***.
- Every ***object*** in **R** is a ***member of a class***.
- You can use the `class` function to determine the class of an object:

```
> class(teams)
[1] "character"
> class(w)
[1] "numeric"
> class(nleast)
[1] "data.frame"
> class(class)
[1] "function"
```

Objects and Classes

- Some functions (**methods**) are associated with a specific class.
- In **R**, **methods** for different classes can share the same name. They are called ***generic functions***:
  - For example, **+** is a generic function for adding objects. You add numbers together with the **+ operator**:

```
> 17 + 6
[1] 23
```

  - For example, you can also use the **+ operator** with a date object (a different class) and a number:

```
> as.Date("2010-09-08") + 7
[1] "2010-09-15"
```

Dataframes

- A **Dataframe** is an object with ***rows*** and ***columns***.
- Whereas *values in the body of a matrix can only be numbers*, they **can be different data types** in different columns of a dataframe (e.g. text, dates, boolean, names of factor levels for categorical variables).
- **Response variables must all be in same column** !

control	preheated	prechilled			
6.1	6.3	7.1			
5.9	6.2	8.2			
5.8	5.8	7.3			
5.4	6.3	6.9			

Dataframes

Response	Treatment			
6.1	control			
5.9	control			
5.8	control			
5.4	control			
6.3	preheated			
6.2	preheated			
5.8	preheated			
6.3	preheated			
7.1	prechilled			
8.2	prechilled			
7.3	prechilled			
6.9	prechilled			

Data entered correctly, as a dataframe.

Dataframes

- Create **Dataframe** in excel.
- Save as ***tab-delimited text file*** (.txt extension) or as a ***comma-delimited file*** (.csv extension)
- Can read file directly into **R** using **`read.table()`** or **`read.csv()`** functions.

```
> worms <- read.table("c://temp/worms.txt",header=T)
> attach(worms)
```

Attach ***makes the variables accessible by name in the R session with referencing the file, e.g.*** **`worms$Slope`.**

```
> names(worms)
[1] "Field.Name"  "Area"  "Slope"     "Vegetation"
[5] "Soil.pH"    "Damp"  "Worm.density"
```

Dataframes

- Other important **R** functions to explore data frames are **`summary()`** and **`by()`**.

  `> summary(worms)`

  `Summary` ***`lists maximum, minimum, mean, median, 25 and 75 percentiles for continuous worms.df variables; lists field names for categorical variables; counts levels.`***

  `> by(worms, Vegetation, mean)`

  `By` ***`provides means (or other numeric summaries) of the numeric variables for each vegetation type.`***

Subscripts and Indices

- Key to working effectively with dataframes is to be at ease with using **subscripts** (‘**indices**’ to some).

```
> worms[3,5]
[1] 4.3
```

is the value of Soil.pH (the variable in column 5) in row 3.

```
> worms[14:19,7]
[1] 0 6 8 4 5 1
> worms[1:5,2:3]
   Area   Slope
1    3.6     11
2    5.1      2
3    2.8      3
4    2.4      5
5    3.8      0
```

Subscripts and Indices

- To select all the entries in one *row* the syntax is ‘number comma blank’:

```
> worms[3,]
          Field.Name Area Slope Vegetation  . . etc.
[1] Nursery.Field  2.8     3  Grassland  . . etc.
```

- To select all of the *values* (or *rows*) *in column number* 3:

```
> worms[,3]
[1] 11 2 3 5 0 2 3 0 0 4 10 1 2 6 0 0 8  . . etc.
```

Note that the objects are of different classes:

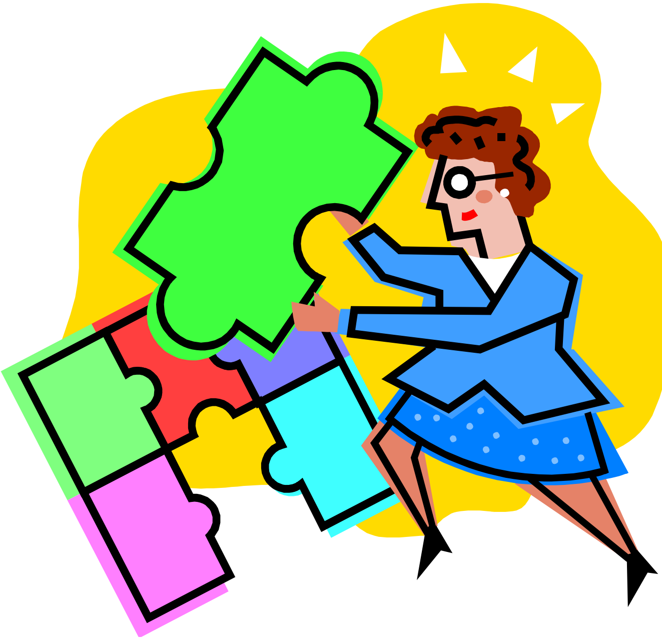
```
> class(worms[3,])
[1] "data.frame"
> class(worms[,3])
[1] "integer"
```

Subscripts and Indices

- You can create **sets** of *rows* or *columns*:

```
> worms[,c(1,5)]
        Field.Name Soil.pH
1      Nash.Fields     4.1
2   Silwood.Bottom     5.2
3    Nursery.Field     4.3
4      Rush.Meadow     4.9
5 Gunness.Thicket     4.2
6         Oak.Mead     3.9
7     Church.Field     4.2
8          Ashurst     4.8
9      . . . . . .
```

Unstack Data Exercise



Note: **Additional exercises in exercise folder**

Wrestling with Rabbits

- The **Rabbit** **`data frame`** in the **MASS** library contains 60 observations of blood pressure change measurements on five rabbits (labeled as R1, R2, … R5) under various control and treatment conditions. View the first few records of the file using the **`head(Rabbit)`** command. Use the **`unstack ()`** function (three times) to convert **Rabbit** to the data frame template that you see on the next slide. Read the unstack help file for more information.
- **Hint #1:** Use the default **`form`** argument in your unstack statements (see help file) with Animal on the right side of the tilde (e.g. ….~Animal).
- **Hint #2:** Use column subscripts (e.g. [,1]) to coerce the first two unstack statements into the proper columnar form.

Converted Rabbit Data Frame

	Treatment	Dose	R1	R2	R3	R4	R5
1	Control	6.25	0.50	1.00	0.75	1.25	1.5
2	Control	12.50	4.50	1.25	3.00	1.50	1.5
3	Control	25.00	10.00	4.00	3.00	6.00	5.0
4	Control	50.00	26.00	12.00	14.00	19.00	16.0
5	Control	100.00	37.00	27.00	22.00	33.00	20.0
6	Control	200.00	32.00	29.00	24.00	33.00	18.0
7	MDL	6.25	1.25	1.40	0.75	2.60	2.4
8	MDL	12.50	0.75	1.70	2.30	1.20	2.5
9	MDL	25.00	4.00	1.00	3.00	2.00	1.5
10	MDL	50.00	9.00	2.00	5.00	3.00	2.0
11	MDL	100.00	25.00	15.00	26.00	11.00	9.0
12	MDL	200.00	37.00	28.00	25.00	22.00	19.0