R Cheat Sheet: Brief Introduction to Language Elements and Control Structures

Comments

```
# from the hash to the end of the line
```

Basic (underlying) data-types

1) logical – Boolean TRUE/FALSE
2) integer – 32 bit signed integer number
3) double – double precision real number
4) character – text in quotes – strings
5) complex – complex numbers (3+2i)

Note: integer and double of mode numeric

Common R objects

1) atomic vector – 1-N, all of only one basic data type, can be named. R does not have a single value object. Single values are held in a length=1 vector.
2) list – 1-N of any R object (including lists), list elements can have different types, list elements can be named
3) factor – 1-N of ordinal (ordered) or categorical (unordered) data (typically character to integer coding)
4) data.frame 1-M rows by 1-N cols, cols is a named list, the data for each column is a vector/factor, rows can be named
5) matrix – numeric vector with 2 dimensions, 1-M rows by 1-N cols, rows and cols can be named
6) array – essentially a matrix with (typically) 3 or more dimensions

Note: While these are the most common objects used for analysis, most things in R are objects that can be manipulated.
Note: Some objects only contain certain types (eg. matrix), or everything in the object is of the same type (eg. vector)

Indexing objects

Because objects contain multiple values, understanding indexing is critical to R:

1) x[i], x[r, c] – can select multiple
2) x[[i]], x[[r, c]] – select single
3) xi, x"i" – select single by name

a) by number: x[5]; x[1:10]; x[length(x)]
b) by logic: x[T,F,T,F]; x[!is.na(x)]
c) by name: x['me']; x$me; x[c('a', 'b')]

Note: 2-dimension indexes are x[row, col]
Trap: x[i] and x[[i]] can return very different results from the same object

Classes

R has class mechanisms for creating more complex data objects. Common classes include Date, ts (time series data), lm (the results of a regression linear model). These are often used like other objects.

Objects and variables

Objects can be assigned to variables: <-
Note: objects have mode/type, not variables
Note: if an object has a rule your code will be quietly coerced to meet the rule:

```
x <- c(1, "2"); cat(x) # -> "1", "2"
```

Determine the nature of an object

1) typeof(x) – the R type of x
2) mode(x) – the data mode of x
3) storage.mode(x) – the storage mode of x
4) class(x) – the class of x
5) attributes(x) – the attributes of x (common attributes: 'class' and 'dim')
6) str(x) – print a summary structure of x
7) dput(x) – print full text R code for x

NULL v NA

1) NULL is an object, typically used to mean the variable contains no object.
2) NA is a value that means: missing data item here

```
x <- NULL; is.null(x); y <- NA; is.na(y)
length(NULL); length(NA) # -> 0, 1
```

Trap: can have a list of NULLs but not a vector of NULLs. Can have a vector of NAs.

Other non-number numbers (NA the first)

```
1) Inf       # positive infinity
2) -Inf      # negative infinity
3) NaN       # not a number
   1/0; 0/0  # -> Inf, NaN
```

Operators

```
+, -, *, /   # addition, subtraction,
             # multiplication, division
^ or  **     # exponentiation
%%           # modulus
%/%          # integer division
%in%         # membership
:            # sequence generation
<, <=, ==, >=, >, != # Boolean comparative
|, ||                # (vectorised/not vec)
&, &&                # (vectorised/not vec)
```

Note: with few exceptions (&&, || and :) operators take vectors and return vectors.

Flow control structures

1) if (cond) expr
2) if (cond) expr1 else expr2
3) for (var in seq) expr
4) while (cond) expr
5) repeat expr

Note: break exits a loop, next moves flow to the start of the loop with the next var
Note: expressions typically enclosed in {}
But single expressions do not need the {}
Multiple expression on a line ; separated

Flow control functions

1) the vectorised if statement:
   result <- ifelse(cond, expr1, expr2)
2) the switch statement (not vectorised):

```
switch( expr.string,
    case1 = expr1,
    case2 = expr2,
    default = expr 3 # default optional
)
```

expr.string evaluates to a char string
Note: cases not enclosed in quotes.

R Cheat Sheet: Basic List of Useful Functions

Built-in constants
```
LETTERS; letters; month.abb; month.name; pi
```

Object creation
```
sz <- 20; x <- 4; t <- 'c' #length 1 vector
a <- letters[ceiling(runif(sz,0.00001,26))]
names(a) <- LETTERS[1:sz] # a named vector
i <- 1:sz; j <- i + rnorm(sz, 0, 2)
z <- exp((0+1i)*pi) + 1+0i #complex numbers

d <- as.Date('2012-01-01') + seq(1, sz)
f <-factor(rep(1:x, sz/x), levels=x:1)
df <- data.frame(a=a, d=d, f=f, i=i, j=j)
m <- matrix(rnorm(x^2), nrow=x, ncol=x)
l <- list(df, m, a, z)
```

Object inspection
```
mode(i); class(df); typeof(j); dput(a)
str(l); summary(df); head(df); tail(df)
attributes(l); is.null(i)

names(a); dimnames(df); colnames(df)
rownames(df); dim(df); nrow(m); ncol(df)
is.list(l); is.factor(f); is.complex(z)
is.character(a); is.matrix(m); is.real(z)
is.numeric(z); is.integer(i); is.vector(i)
is.data.frame(df); is.ordered(f)

isTRUE(all.equal(1:10, as.numeric(1:10)))
identical(1:10, as.numeric(1:10)) # FALSE
```

Utility functions
```
assign('variable.name', 5) # like: <-, ->
c(i, j); rep(NA, 20) # concatenate; repeat
append(l, list(c(1, 2, 3))); seq_along(a)
seq(from=5,to=100, by=5); seq_len(nrow(df))

sort(a); order(a); rank(a); rev(a)
any(i %in% c(1,3,5)); all(i %in% c(1,3,5))
which(a %in% letters[21:26]); match('c', a)
max <- ifelse(i > j, i, j) # vectorised if
switch(txt, a={print('a')}, b={print('b')},
      {print('default')} ) #not vectorised

df <- transform(df, k=j+i)
df <- within(df, s <- i/j) #merge(df1, df2)
x <- with(df, j+5); df <- cbind(df, z)
row.df <- head(df, 1); rbind(df, row.df)
df$f <- reorder(df$f, df$j, mean)

# many similar style conversion functions:
as.integer(f); as.data.frame(m) # etc.
```

Maths
```
is.na(i); is.nan(j); is.null(d)
is.finite(j); is.infinite(j)
Re(z); Im(z) # real and im coefficients

abs(j); sqrt(i); log(i); log10(j); exp(j)
ceiling(j); floor(j); round(j, digits=2)
trunc(j); sin(j); cos(j); tan(j); asin(0.5)
acos(0.5); atan(1) # atan2(y, x);
sum(j) prod(j); cumsum(j); cumprod(j)
```

Stats
```
length(j); sum(j); min(j); max(j); range(j)
cut(i, 5); mean(j); median(j); sd(j)
var(i); cov(i, j); cor(i, j)
diff(j, lag=1, diff=1) # difference data
rnorm(n=10, mean=0, sd=1)   # normal dist
runif(n=10, min=1, max=100) # uniform
# Also poisson and binomial random numbers

r <- lm(j ~ i, data=df); summary(r)
anova(r); residuals(r); coef(r);
plot(r); plot(r$fitted); plot(r$resid)
# Also glm() gam() lme() lmer() nls() etc.
```

Character strings
```
as.character(j); toString(l)
nchar(a); B <- toupper(a); b <- tolower(B)
s <- "the cow jumped over the moon."
sub('the', 'a', s) #  -> a cow … the moon
gsub('the', 'a', s) # -> a cow … a moon
substr(s, 5, 7); substr(s, 5, 7) <- "dog"
substr(s, 5, 7) <- "monkey" # FAIL!
paste('a', 'b', 'c', sep='; ') # 'a; b; c'
strsplit(s, ' ') # -> list; regex pattern
grep('the', s) # see also: grepl and agrep
make.unique(a) # -> change dups in vector
format(j, digits=2); sprintf("%d: %s", i,a)
format(d, format="%A %Y-%b-%d")
```

Dates
```
# Date: a double; days since 1970-01-01
x<- as.Date('03-06-1930',format='%d-%m-%Y')
Sys.Date(); # today: date only
days.apart <- d-x; weekdays(d); months(d)
dp <- as.POSIXlt(d); dp$year <- dp$year - 1
d <- as.Date(dp); names(unclass(dp))
# Time: fraction of a date in a double
Sys.time(); date() # today's date and time
# Really useful: zoo and lubridate packages
```

I/O and the file system
```
cat(i); print(j) # cat: tighter, no newline
getwd(); setwd('~/Desktop'); list.files()
list.dirs(); dir(); Sys.glob() # wild card
save(z, file='z.bin'); load('z.bin')
unlink('z.bin'); con <- file('f.txt', 'rt')
y <- readLines(con, 1) # read lines of text
# writeLines(text, con=c, sep="\n") # etc.
write.csv(df, file='fileName.csv')
```

Script and package management
```
source('program.R') # incorporate R code
install.packages('ggplot2') # install pack
library('ggplot2'); require('ggplot2')
```

In the workspace
```
ls(); rm(z); help('help') # q() to quit()
help.search('help') # look for help
```

Useful debugging functions
```
# browser(); debug(); trace()
stopifnot(i[1] == 1) # assert!
warning('message'); stop('message')
```


R Cheat Sheet: Atomic Vectors (often just called "vectors" in R)

Atomic vectors:
- An object with contiguous, indexed values
- Indexed from 1 to length(vector)
- All values of the same basic atomic type
- Vectors do not have a dimension attribute
- Has a fixed length once created

Six basic atomic types:

Class	Example
logical	TRUE, FALSE, NA
integer	1:5, 2L, 4L, 6L
numeric	2, 0.77 (double precision)
complex	3.7+4.2i, 0+1i
character	"string", 'another string'
raw	(byte data from 0-255)

No scalars
In R, these basic types are always in a vector. Scalars are just length=1 vectors.

Creation (length determined at creation)
Default value vectors of length=4
```
u <- vector(mode='logical', length=4)
print(u) # -> FALSE, FALSE, FALSE, FALSE
v <- vector(mode='integer', length=4)
Also: numeric(4); character(4); raw(4)
```
Using the sequence operator
```
i <- 1:5 # produces an integer sequence
j <- 1.4:6.4 # a numeric sequence
k <- seq(from=0, to=1, by=0.1) # numeric
```
Using the c() function
```
l <- c(TRUE, FALSE)     # logical vector
n <- c(1.3, 7, 7/20)    # numeric vector
z <- c(1+2i, 2, -3+4i) # complex vector
c <- c('pink', 'blue') # character vector
```
Other things
```
v1 <- c(a=1, b=2, c=3) # a named vector
v2 <- rep(NA, 3) # 3 repeated NAs
v3 <- c(v1, v2) # concatenate and flatten
v4 <- append(origV, insertV, position)
```

Conversion
```
as.vector(v); as.logical(v); as.integer(v)
as.numeric(v); as.character(v) # etc. etc.
unlist(l) # convert list to atomic vector
```
Trap: unlist() wont unlist non-atomic items
```
unlist(list(as.name('fred')))      # FAILS
```

Basic information about atomic vectors

Function	Returns
dim(v)	NULL
is.atomic(v)	TRUE
is.vector(v)	TRUE
is.list(v)	FALSE
is.factor(v)	FALSE
is.recursive(v)	FALSE
length(v)	Non-negative number
names(v)	NULL or char vector

```
mode(v); class(v); typeof(v); attributes(v)
is.numeric(v); is.character(v); # etc. etc.
```
Trap: lists are vectors (but not atomic)
Trap: array/matrix are atomic (not vectors)
Tip: use (is.vector(v) && is.atomic(v))

The contents of a vector
```
cat(v); print(v) # print vector contents
str(v); dput(v); # print vector structure
head(v); tail(v) # first/last items in v
```

Indexing: [and [[(but not $)
- [x] selects a vector for the cell/range x
- [[x]] selects a length=1 vector for the single cell index x (*rarely used*)
- $ operator invalid for atomic vectors

Index by positive numbers: *these ones*
```
v[c(1,1,4)] # get 1st one twice then 4th
v[m:n] # get elements from indexes m to n
v[[7]] <- 6 # set seventh element to 6
v[which(v == 'M')] # which() yields nums
```
Index by negative numbers: *not these*
```
v[-1] # get all but the first element
v[-length(v)] # get all but the last one
v[-c(1,3,5,7,9)] # get all but ...
```
Index by logical atomic vector: *in/out*
```
v[c(TRUE, FALSE, TRUE)] # get 1st and 3rd
v[v > 2] # get all where v is g.t. two
v[v > 2 & v < 9] # get where v>2 and v<9
v[v == 'M'] # get where v equals char 'M'
v[v %in% c('me', 'andMe', 'meToo')] # get
```
Indexed by name (*only with named vectors*)
```
v[['alpha']] # get single by name
v[['beta']] <- 'b' # set single by name
v[c('alpha', 'beta')] # get multiple
v[!(names(v) %in% c('a', 'b'))] # exclude
names(v)['z'] <- 'omega' # change name
```

Most functions/operators are vectorised
```
c(1,3,5) + c(5,3,1) # -> 6, 6, 6
c(1,3,5) * c(5,3,1) # -> 5, 9, 5
```

Sorting
```
upSorted <- sort(v) # also: v[order(v)]
d <- sort(v, decreasing=TRUE) # rev(sort(v))
```

Raw vectors (byte sequences)
```
s <- charToRaw('raw') # string input
r <- as.raw(c(114, 97, 119)) # decimal in
print(r) # -> 72 61 77 (hex output)
```

Traps
Recycling vectors in math operations
```
c(1,2,3,4,5) + 1 # -> 2, 3, 4, 5, 6
c(1,2,3,4,5) * c(1,0) # -> 1, 0, 3, 0, 5
```
Automatic type coercion (often hidden)
```
x <- c(5, 'a') # c() converts 5 to '5'
x <- 1:3; x[3] <- 'a' # x now '1' '2' 'a'
typeof(1:2) == typeof(c(1,2)) # -> FALSE
```
For-loops on empty vectors
```
for(i in 1:length(c())) print(i) # loopx2
for(i in seq_len(x)) # empty vector safe
```
Also: for(j in seq_along(x))

Some Boolean ops not vectorised
```
c(T,F,T) && c(T,F,F) # TRUE (!vectorised)
c(T,F,T) & c(T,F,F) # TRUE, FALSE, FALSE
```
Similarly: || is not vectorised; | is

Factor indexes are treated as integers
Tip: decode with v[as.character(f)].

R Cheat Sheet: Lists

Context: R has two types of vector

Atomic vectors contain *values*

These values are all of the same type. They are arranged contiguously. Atomic vectors cannot contain objects. There are six types of atomic vector: raw, logical, integer, numeric, complex and character.

Recursive vectors contain *objects*

R has two types of recursive vector:

Class	Example
list	list(a=1, b=2, c=3:10)
expression	expression(a + b)

Lists are an oft-used workhorse in R.

Lists

- At top level: 1-dimension indexed object that contains objects (not values)
- Indexed from 1 to length(list)
- Contents can be of different types
- Lists can contain the NULL object
- Deeply nested lists of lists possible
- Can be arbitrarily extended (not fixed)

List creation: usually using list()

```
l1 <- list('cat', 5, 1:10, FALSE) # unnamed
l2 <- list(x='dog', y=5+2i, z=3:8) # named
l3 <- c(l1, l2) # one list partially named
l4 <- list(l1, l2) # a list of 2 lists
l5 <- as.list( c(1, 2, 3) ) # conversion
l6 <- append(origL, insertVorL, position)
```

Basic information about lists

Function	Returns
dim(l)	NULL
is.list(l)	TRUE
is.vector(l)	TRUE
is.recursive(l)	TRUE
is.atomic(l)	FALSE
is.factor(l)	FALSE
length(l)	Non-negative number
names(l)	NULL or char vector

mode(l); class(l); typeof(l); attributes(l)

The contents of a list

```
print(l) # print vector contents
str(l); dput(l); # print list structure
head(l); tail(l) # first/last items in l
```

Trap: cat(x) does not work with lists

Indexing: [versus [[versus $

- use [to get/set multiple items at once
  Note: [always returns a list
- use [[and $ to get/set a specific item
- $ only works with named list items
  all same: $name $"name" $'name' $`name`
- indexed by positive numbers: *these ones*
- indexed by negative numbers: *not these*
- indexed by logical atomic vector: *in/out*
- an empty index l[] returns the list

Tip: When using lists, most of the time you want to index with [[or $; and avoid [

Indexing examples: one-dimension get

```
j <- list(a='cat', b=5, c=FALSE)
x <- j$a # puts 1-item char vec 'cat' in x
x <- j[['a']] # much the same as above
x <- j['a'] # puts 1-item list 'cat' in x
x <- j[[1]] # 1-item char vec 'cat' in x
x <- j[1] # puts 1-item list 'cat' in x
```

Indexing examples: set operations

- start with example data
  ```
  l <- list(x='a', y='b', z='c', t='d')
  ```
- next: use [[$ because specific selection
  ```
  l[[6]] <- 'new' # also l[[5]] set to NULL
  l$w <- 'new-w' # becomes l[[7]] named 'w'
  l[['w']] <- 'dog' # l[[7]] set to 'dog'
  ```
- change named values: (note order ignored)
  ```
  l[names(l) %in% c('t', 'x')] <- c(1, 2)
  # in previous: l$x set to 1 and l$t to 2
  ```

Indexing example: multi-dimension get

- Indexing evaluated from left to right
- Let's start with some example data ...
  ```
  i <- c('aa', 'bb', 'cc') #
  j <- list(a='cat', b=5, c=FALSE)
  k <- list(i, j) #list of things
  ```
- Let's play with this data ...
  ```
  k[[1]] -> x # puts the vector from i in x
  k[[2]] -> y # puts the list from j in y
  k[1] -> x   # puts vec from i into a list
              # and puts that list into x
  x <- k[[1]][[1]] #puts the 'aa' vec in x
  x <- k[1][1] # same as k[1] - SILLY
  x <- k[1][1][1][1][1][1] # same as above
  x <- k[[1]][[2]] # puts the 'bb' vec in x
  x <- k[1][2] # WRONG: k[1] is 1-item list
  x <- k[1][[2]] # WRONG same as above
  x <- k[[2]][1] # put list of 'cat' in x
  x <- k[[2]][[1]] # put vector 'cat' in x
  ```

List manipulation

1 Arithmetic operators cannot be applied to lists (as content types can vary)

2 Use the apply() functions to apply a function to each element in a list:
```
x <- list(a=1, b=month.abb, c=letters)
lapply(x, FUN=length) # (list) 1 12 26
sapply(x, FUN=length) #(vector) 1 12 26
# Next eg: passing args to apply fn
y <- list(a=1, b=2, c=3, d=4)
sapply(y, FUN=function(x,p) x^p, p=2)
# -> (vector) 1 4 9 16
sapply(y, FUN=function(x,p) x^p, p=2:3)
# -> (matrix 2x4) 1 4 9 16 / 1 8 27 64
```

3 Use unlist to convert list to vector
```
unlist(x) # -> "1", "Jan", ... "z"
```
Trap: unlist() wont unlist non-atomic
```
unlist(list(expression(a + b))) # FAILS
```

4 Remove NULL objects from a list
```
z <- (a=1:9, b=letters, c=NULL)
zNoNull <- Filter(Negate(is.null), z)
```

5 Use named lists to return multiple values

6 Trap: factor indexes treated as integer
Tip: decode with v[as.character(f)] etc.

R Cheat Sheet: Data Frames (tabular data in rows and columns)

Create

- The R way of doing spreadsheets
- Internally, a data.frame is a list of equal length vectors or factors.
- Observations in rows; Variables in cols

```
empty <- data.frame()# empty data frame
c1 <- 1:10           # vector of integers
c2 <- letters[1:10]  # vector of strings
df <- data.frame(col1=c1,col2=c2)
```

Import from and export to file

```
d2 <- read.csv('fileName.csv', header=TRUE)
library(gdata); d3 <- read.xls('file.xls')
write.csv(df, file='fileName.csv') # export
print(xtable(df), type = "html") # to HTML
```

Basic information about the data frame

Function	Returns
is.data.frame(df)	TRUE
class(df)	"data.frame"
nrow(df); ncol(df)	Row and Col counts
colnames(df);	NULL or char vector
rownames(df)	NULL or char vector

```
# Also: head(df); tail(df); summary(df)
```

Referencing cells [row, col] [[r, c]]

```
# [[ for single cell selection; [ for multi
vec <- df[[5, 2]] # get cell by row/col num
newDF <- df[1:5, 1:2] # get multi in new df
df[[2, 'col1']] <- 12 # set single cell
df[3:5, c('col1', 'col2')] <- 9 # set multi
```

Referencing rows [r,]

```
# returns a data frame (and not a vector!)
row.1 <- df[1, ]; row.n <- df[nrow(df),]
# to get a row as a vector, use following
vrow <- as.numeric(as.vector(df[row,]))
vrow <- as.character(as.vector(df[row,]))
```

Referencing columns [,c] [c] [[c]] $col

```
# most column references return a vector
col.vec <- df$cats # returns a vector
col.vec <- df[, 'horses'] # returns vector
col.vec <- df[, a] # a is int or string
col.vec <- df[['frogs']] # returns a vector
frogs.df <- df['frogs'] # returns 1 col df
first.df <- df[1] # returns 1 col df
first.col <- df[, 1] # returns a vector
last.col <- df[, ncol(df)] # returns vector
```

Adding rows

```
# The right way ... (both args are DFs)
df <- rbind(df, data.frame(col1='d',
    col2=3, col3='A'))
```

Adding columns

```
df$newCol <- rep(NA, nrow(df)) # NA column
df[, 'copyOfCol'] <- df$col # copy a col
df$y.percent.of.x <- df$y / sum(df$x) * 100
df <- cbind(col, df); df <- cbind(df, col)
df$c3 <- with(df, c1 + c2) # no quotes
transform(df, col3 = col1 * col2)
df <- within(df, colC <- colA + colB)
```

Set column names # same for rownames()

```
colnames(df) <- c('date', 'alpha', 'beta')
colnames(df)[1] <- 'new.name.for.col.1'
colnames(df)[colnames(df) %in% c('a', 'b')]
    <- c('x', 'y') # order of sub from cols
```

Selecting multiple rows

```
firstTenRows <- df[1:10, ] # head(df, 10)
everythingButRowTwo <- df[-2, ]
sub <- df[ (df$x > 5 & df$y < 5), ]
sub <- subset(df, x > 5 & y < 5)
# Note: vector Boolean (&, |) in above
notLastRow <- head(df, -1) # df[-nrow(df),]
```

Selecting multiple columns

```
df <- df[, c(1, 2, 3)] # keep cols 1 2 3
df <- df[, c('col1', 'col3')] # by name
# drop columns ...
df <- df[, -1] # keep all but first column
df <- df[, -c(1, 3)] # drop cols 1 and 3
df <- df[, !(colnames(df) %in%
    c('notThis', 'norThis'))]# drop by name
```

Replace column elements by row selection

```
df[df$col3 == 'A', 'col2'] <-
    c('j', 'a', 'a', 'a', 'j')
```

Manipulation

```
sorted <- df[order(df$col2), ]
backwards <- df[rev(order(df$col2)), ]
transposed <- as.data.frame(t(df))
merged <- merge(df1,df2,by='col',all=TRUE)
molten <- melt( df, id=c('year', 'month'),
    measure=c('col5', 'col10', col'15) )
# Note: melt comes from the reshape package
rownames(df)<- seq_len(nrow(df))#renum rows
summary <- ddply( df, ~col1, summarise,
    N=length(col3). mean=mean(col3) )
summary <- ddply(df, .(col1, col2),
    summarise, sd=sd(col3) mean=mean(col3))
# Note: ddply comes from the plyr package
```

Missing data (NA)

```
any(is.na(df)) # detect anywhere in df
any(is.na(df$col)) # anywhere in col
# delete selected missing data rows
df <- df[!is.na(df$col), ]
# replace NAs with something else
df[is.na(df)] <- 0 # works on whole df
df$col[is.na(df$col)] <- newValue
df$col <- ifelse(is.na(df$col), 0, df$col)
df <- orig[!is.na(orig$series),
    c('Date', series)] # selecting on r & c
```

Traps

1. for loops on possibly empty df's, use: for(i in seq_len(nrow(df))
2. columns coerced to factors, avoid with the argument stringsAsFactors=FALSE
3. confusing row numbers and rows with numbered names (hint: avoid row names)
4. although rbind() accepts vectors and lists; this can fail with factor cols

R Cheat Sheet: Matrices and Arrays

Context

Matrices and arrays are an extension on R's atomic vectors. Quick recap: atomic vectors contain *values* (not objects). They hold a contiguous set of values, all of which are of the same basic type. There are six types of atomic vector: logical, integer, numeric, complex, character and raw. Importantly: atomic vectors have no dimension attribute. Matrices and arrays are effectively vectors with a dimension attribute. Matrices are two-dimensional (tabular) objects, containing values all of the same type (unlike data frames). Arrays are multi-dimensional objects (typically with three plus dimensions), with values all of the same type.

Matrix versus data.frame

In a matrix, every column, and every cell is of the same basic atomic type. In a data.frame each column can be of a different type (eg. numeric, character, factor). Data frames are best with messy data, and for variables of mixed modes.

Matrix creation

```
# generalCase <- matrix(data=NA, nrow=1,
#       ncol=1, byrow=FALSE, dimnames=NULL)
M <- matrix(
       c(2, -1, 5, -1, 2, -1, 9, -3, 4),
       nrow=3, ncol=3, byrow=TRUE)
# which yields the following 3x3 matrix:
#          [,1] [,2] [,3]
#    [1,]    2   -1    5
#    [2,]   -1    2   -1
#    [3,]    9   -3    4
# Trap: R vectors are not matrix column
# vectors; however, the matrix class
# produces 1-column vectors by default
b <- matrix(c(0, -1, 4)) # column vector
I <- diag(3) # create a 3x3 identity matrix
D <- diag(c(1,2,3)) # 3x3 with speced diag
d <- diag(M) # R vector with the diag of M
MDF <- as.matrix(df) # data.frame to matrix
```

Basic information about a matrix

Function	Returns
dim(M)	NROW NCOL (2 numbers)
class(M)	"matrix"
is.matrix(M)	TRUE
is.array(M)	TRUE
is.atomic(M)	TRUE
is.vector(M)	FALSE
is.list(M)	FALSE
is.factor(M)	FALSE
is.recursive(M)	FALSE
nrow(M); ncol(M)	Row and Col counts
length(M)	NROW*NCOL (1 number)
rownames(M)	NULL or char vector
colnames(M)	NULL or char vector

Matrix manipulation

```
newM <- cbind(M, N, ...) # horizontal join
newM <- rbind(M, N, ...) # vertical join
# M and N either matrices or atomic vectors
v <- c(M) # convert matrix back to a vector
df <- data.frame(M) # convert to data frame
```

Matrix multiplication

```
InnerProduct <- A %*% B # matrix multiply
OuterProduct <- A %o% B
CrossProduct <- crossprod(A, B)
Trap: A * B -> element wise multiplication
```

Matrix maths

```
rowMeans(M) # R vector of row means
colMeans(M) # R vector of column means
rowSums(M) # R vector of row sums
colSums(M) # R vector of column sums
t <- t(M) # transpose the M matrix
inverse <- solve(M) # get the inverse of M
# solve the system of equations Mx = b
x <- solve(M, b) # simultaneous equation
e <- eigen(M) # -> list with values/vectors
d <- det(M) # determinant of square matrix
```

Matrix indexing [row, col] [[row, col]]

```
# [[ for single cell selection; [ for multi
# indexed by positive numbers: these ones
# indexed by negative numbers: not these
# indexed by logical atomic vector: in/out
# named rows/cols can be indexed by name
# M[i] or M[[i]] is vector-like indexing
# $ operator is invalid for atomic vectors
# M[r,] # get/set selected row(s)
# M[,c] # get/set selected col(s)
```

Arrays

```
A <- array(1:8, dim=c(2,2,2))
# A three dimensional example
#  , , 1
#  [,1] [,2]
#  [1,]    1    3
#  [2,]    2    4
#  , , 2
#  [,1] [,2]
#  [1,]    5    7
#  [2,]    6    8
# Could have created in two steps:
A <- 1:8; dim(A) <- c(2,2,2)
# A matrix is a special case of array ...
M <- array(1:9, dim=c(3,3)) # a matrix
# Matrices are arrays with two dimensions
```


R Cheat Sheet: Factors

Factors
- A one-dimensional array of categorical (unordered) or ordinal (ordered) data.
- Indexed from 1 to N. Not fixed length.
- Named factors are possible (*but rare*)

Trap: the hidden/unexpected coercion of an object to a factor is a key source of bugs

Why use factors
1 Specifying a non-alphabetical order
2 Some statistical functions treat cat/ord data differently from continuous data.
3 Deep ggplot2 code depends on it

Create
Example 1 - unordered
```
sex.v <- c('M', 'F', 'F', 'M', 'M', 'F')
sex.f <- factor(sex.v) # unordered
sex.w <- as.character(sex.f) # restore
```
Eg 2 - ordered (small, medium, large)
```
size.v <- c('S', 'L', 'M', 'L', 'S', 'M')
size1.f <- factor(size.v, ordered=TRUE)
# ordered L < M < S from underlying type
```
Eg 3 - ordered, where we set the order
```
size.lvls <- c('S', 'M', 'L') # set order
sz2.f <- factor(size.v, levels=size.lvls)
# above: ordered (low to high) by levels
```
Eg 4 - ordered with levels and labels
```
levels <- c(1, 2, 3, 99) # from codesheet
labels <- c('Love','Neutral','Hate',NA)
data.v <- c(1, 2, 3, 99, 1, 2, 1, 2, 99)
data.f <- factor(data.v, levels=levels,
        labels=labels)
# levels: input - how factor() reads in
# labels: output - how factor() puts out
# Note: if specified, labels become
# the internal reference and coding frame
```
Eg 5 – using the cut function to group
```
i <- 1:50 + rnorm(50,0,5); k <- cut(i, 5)
```

Basic information about a factor

Function	Returns
dim(f)	NULL
is.factor(f)	TRUE
is.atomic(f)	TRUE
is.vector(f)	FALSE
is.list(f)	FALSE
is.recursive(f)	FALSE
length(f)	Non-negative number
names(f)	NULL or char vector
mode(f)	"numeric"
class(f)	"factor"
typeof(f)	"integer"
is.ordered(f)	TRUE or FALSE

```
unclass(f) # -> R's internal coding
cat(f); print(f); str(f); dput(f); head(f)
```

Indexing: much like atomic vectors
- [x] selects a factor for the cell/range x
- [[x]] selects a length=1 factor for the single cell index x (*rarely used*)
- The $ operator is invalid with factors

Factor arithmetic & Boolean comparisons
- factors cannot be added, multiplied, etc.
- same-type factors are equality testable
```
z <- sex.f[1] == sex.f[2]    # OKAY
z <- sex.f[1] == size.f[2]   # WRONG
```
- ordered factors can be order compared
```
z <- size1.f[1] < size1.f[2] # OKAY
z <- sex.f[1] < sex.f[2]     # WRONG
```

Managing the enumeration (levels)
```
f <- factor(letters[1:3]) # example data
levels(f)    # -> get all levels
levels(f)[1] # -> get a specific level
```
test existence of a level
```
any(levels(f) %in% c('a', 'b')) # -> TRUE
```
add new levels:
```
levels(f)[length(levels(f))+1] <- 'ZZ'
levels(f) <- c(levels(f), 'AA')
```
reorder levels
```
levels(f) # -> 'a' 'b' 'c' 'ZZ' 'AA'
f <- factor(f, levels(f)[c(4,1:3,5)])
```
change/rename levels
```
levels(f)[1] <- 'XX' # rename a level
levels(f)[levels(f) %in% 'AA']<- 'BB'
```
delete (or drop) unused levels
```
f <- f[drop=TRUE]
```

Adding an element to a factor
```
f <- factor(letters[1:10]) # example data
f[length(f) + 1] <- 'a'    # add at end
```
Trap: above only adds an existing level
Tip: decode/recode for general add below
```
f <- factor(c(as.character(f), 'zz'))
```

Merging/combining factors
```
a <- factor(1:10); b <- factor(letters[a])
union <- factor(c(as.character(a),
         as.character(b)))      # union
cross <- interaction(a, b)      # a.b
# both merges produced unordered factors
# Levels: union 20; cross 100
# Items:  union 20; cross 10.
```

Using factors within data frames
```
# df$x <- reorder(df$f, df$X, F, order=T)
#    yields factor ordered by function F
#    applied to col X grouped by col f
# by(df$x, df$f, F) – apply F by factor f
```

Traps
1 Strings loaded from a file converted to factors (Hint: in read.table or read.csv use: stringsAsFactors=FALSE)
2 Numbers from a file factorised. Revert: as.numeric(levels(f))[as.integer(f)]
3 One factor (enumeration) cannot be meaningfully compared with another.
4 NA's (missing data) in factors and levels can cause problems (Hint:avoid)
5 Adding a row to a data frame, which adds a new level to a column factor. (Hint: make the new row a data frame with a factor column then use rbind).

R Cheat Sheet: tRips and tRaps for new players

General
Trap: R error messages are not helpful
Tip: use `traceback()` to understand errors

Object coercion
Trap: R objects are often silently coerced to another class/type as/when needed.
Examples: `c(1, TRUE) # -> 1 1`
`c(1, TRUE, 'cat') # -> "1" "TRUE" "cat"`
`30 < '8'` # yields `TRUE`; 30 became "30"
Tip: inspect objects with `str(x)` `mode(x)` `class(x)` `typeof(x)` `dput(x)` or `attributes(x)`

Factors (special case of coercion)
Trap: Factors cause more bug-hunting grief than just about anything else in R (especially when string and integer vectors and data.frame cols are coerced to factors)
Tip: Learn about factors and using them.
Tip: explicitly test with `is.factor(df$col)`
Tip: use `stringsAsFactors=FALSE` argument when you create a data frame from file
Trap: maths doesn't work on numeric factors and they are tricky to convert back.
Tip: try `as.numeric(as.character(factor))`
Trap: appending rows to a data frame with factor columns is tricky. Tip: make sure the row to be appended is a presented to `rbind()` as a data.frame, and not as a vector or a list (which works sometimes))
Trap: the combine function `c()` will let you combine different factors into a vector of integer codes (probably garbage).
Tip: convert factors to strings or integers (as appropriate) before combining.

Garbage in the workspace
Trap: R saves your workspace at the end of each session and reloads the saved workspace at the start of the next session. Before you know it, you can have heaps of variables lurking in your workspace that are impacting on your calculations.
Tip: use `ls()` to check on lurking variables
Tip: clean up with `rm(list = ls(all=TRUE))`
Tip: `library()` to check on loaded packages
Tip: avoid saving workspaces, start R with the `--no-save --no-restore` arguments

The 1:0 sequence in for-loops
Trap: `for(x in 1:length(y))` fails on the zero length vector. It will loop twice: first setting x to 1, then to 0.
Tip: use `for(x in seq_len(y))`
    not `for(x in 1:length(y))`
Tip: `for(x in seq_along(y))` not `for(x in y)`

Space out your code and use brackets
Trap: `x<-5` # parses as `x <- 5` not `x < -5`
Trap: `1:n-1` # -> `(1:n)-1` not `1:(n-1)`
Trap: `2^2:9` # -> `(2^2):9` not `2^(2:9)`

Vectors and vector recycling
Trap: most objects in R are vectors. R does not have scalars (just length=1 vectors). Many Fns work on entire vectors at once.
Tip: In R, for-loops are often the inefficient and inelegant solution. Take the time to learn the various "apply" family of functions. Hadley Wickham's plyr package is also worth learning and using.
Trap: Math with different length vectors will work with the shorter vector recycled
Eg: `c(1, 2, 3) + c(10, 20) # -> 11, 22, 13`
Trap: `is.vector(list(1, 2, 3)) # -> TRUE`

Vectors need the c() operator
Wrong: `mean(1, 2, 3, 4, 5, 6) # -> 1`
Correct: `mean(c(1, 2, 3, 4, 5, 6)) # -> 3.5`

Use the correct Boolean operator
Tip: `|` and `&` are vectorised - use `ifelse()`
(`|` and `&` also used with indexes to subset)
Tip: `||` and `&&` are not vectorised - use `if`
Trap: `||` `&&` lazy evaluation; `|` `&` full eval
Trap: `==` (Boolean equality) `=` (assignment)

Equality testing with numbers
Trap: `==` and `!=` test for near in/equality
Eg: `as.double(8) == as.integer(8)` is `TRUE`
`isTRUE(all.equal(x, y))` tests near equality
Tip: `identical(x, y)` is more fussy

Think hard about NA, NaN and NULL
Trap: `NA` and `NaN` are valid values.
Eg: `c(1, 2) == c(1, NA) # -> TRUE, NA`
Trap: many Fns fail by default on `NA` input
Tip: many functions take: `na.rm=TRUE`
Tip: vector test for NA: `any(is.na(y))`
Trap: `x == NA` is not the same as `is.na(x)`
Trap: `x == NULL` not the same as `is.null(x)`
Trap: `is.numeric(NaN)` returns `TRUE`

Indexing ([], [[]], $)
Tip: Objects are indexed from 1 to N.
Trap: many subtle differences in indexing for vectors, lists, matrices, arrays and data.frames. Return types vary depending on object being indexed and indexation method.
Tip: take the time to learn the differences
Trap: the zero-index fails silently
Eg: `c(1, 2, 3)[c(0,1,2,0,2,3)] # -> 1,2,2,3`
Trap: negative indexes return all but those
Eg: `c(1, 2, 3, 4)[-c(1, 3)]] # -> 2, 4`
Trap: `NA` is a valid Boolean index
Eg: `c(1, 2)[c(TRUE, NA)] # -> 1, NA`
Trap: mismatched Boolean indexes work
Eg: `c(1, 2, 3)[c(T,F,T,F,T)] # -> 1, 3, NA`

Coding practice
Tip: liberally use `stopifnot()` on function entry to verify argument validity (ie. enforce programming by contract)
Tip: `<-` for assignment; `=` for list names

R Cheat Sheet: Writing Functions

Functions in R are called closures.

```
# Don't be deceived by the curly brackets:
# R is much more like Lisp than C or Java.
# Defining problems in terms of function
# calls and their lazy, delayed evaluation
# (variable resolution) is R's big feature.
```

Standard form (for named functions)

```
plus <- function(x, y) { x + y }
plus(5, 6) # -> 11
# return() not needed – last value returned
# Optional curly brackets with 1-line fns:
x.to.y <- function(x, y) return(x ^ y)
```

Returning values

```
# return() – can use to aid readability and
#   for exit part way through a function
# invisible() - return values that do not
#   print if not assigned.
# Traps: return() is a function, not a
#   statement. The brackets are needed.
```

Anonymous functions

```
# Often used in arguments to functions:
v <- 1:9; cube <- sapply(v,function(x) x^3)
```

Arguments are passed by value

```
# Effectively arguments are copied, and any
# changes made to the argument within the
# function do not affect the caller's copy.
# Trap: arguments are not typed and your
#   function could be passed anything!
#   Upfront argument checking advised!
```

Arguments passed by position or name

```
b <- function(cat, dog, cow) cat+ dog+ cow
b(1, 2, 3) # cat=1, dog=2, cow=3
b(cow=3, cat=1, dog=2) # order no problem
b(co=3, d=2, ca=1) # unique abbreviations
# Trap: not all arguments need be passed
f <- function(x) missing(x); f(); f('here')
# match.arg() – argument partial matching
```

Default arguments

```
# Default arguments can be specified. Eg.
x2y.1 <- function(x, y = 2) { x ^ y }
x2y.2 <- function(x, y = x) { x ^ y }
x2y.2(3); x2y.2(2, 3) # -> 27 8
```

The dots argument (...) is a catch-all

```
f <- function (...) {
    # simple way to access dots arguments
    dots <- list(...) # return list
}
x <- f(5); dput(x) # -> 5 (in a list)
g <- function (...) {
    dots <- substitute(list(...))[-1]
    dots.names <- sapply(dots, deparse)
}
x <- g(a, b, c); dput(x)# -> c("a","b","c")
# dots can be passed to another function:
h <- function(x, ...) g(...)
x <- h(a, b, c); dput(x) # -> c("b", "c")
```

Function environment

```
# When a function is called a new
#   environment (frame) is created for it.
# These frames are found in the call stack
# First frame is the global environment
# Next fn reaches back into the call stack

called.by <- function() { # returns string
    # technically: who is my grandparent?
    if(length(sys.parents()) <= 2)
        return('.GlobalEnv')
    deparse(sys.call(sys.parent(2)))
} # Note: designed to be called from a fn
g <- function(...) { called.by() }
f <- function(...) g(...); f(a, 2)
```

Variable scope and unbound variables

```
# Within a function, variables are
# resolved in the local frame first,
# then in terms of super-functions (when a
# function is defined inside a function),
# then in terms of the global environment.
h <- function(x) { x + a } # a undefined
a <- 5 # a defined in global environment
h(5) # -> returns 10
k <- function(x) { a <- 100; h(x) }
k(10) # -> returns 15
# Note: local a in k() not seen in h()
# variables not defined by the call stack!
# [See my cheat sheet on R Environments]
```

Super assignment <<-

```
# x <<- y ignores the local x, and looks up
# the super-environments for a x to replace
accumulator <- function() {
    a <- 0 # super assignment finds this a
    function (x) {
        a <<- a + x # the super assignment
        a # alone: this a will be printed
    } # NOTE: anonymous function returned
}     # when accumulator() is called !!!
acc <- accumulator() # create accumulator
acc(1); acc(5); acc(2) # prints: 1, 6, 8
```

Operator and replacement functions

```
`+`(4, 5) # -> 9 - operators are just fns
`%plus%` <- function(a, b) { a + b }
3 %plus% 2 # -> 5 # new defined functions
# "FUN(x) <- v is parsed as: x <- FUN(x, v)
"cap<-" <- function(x, value)   # must use
    ifelse(x > value, value, x) # 'value'
x <- c(1,10,100); cap(x) <- 9 # x -> 1,9,9
```

Exceptions

```
tryCatch(print('pass'), error=function(e)
  print('bad'), finally=print('done'))
tryCatch(stop('fail'), error=function(e)
  print('bad'), finally=print('done'))
```

Useful language reflection functions

```
# exists(); get(); assign() – for variables
# substitute(); bquote(); eval(); do.call()
# parse(); deparse(); quote(); enquote()
```

R Cheat Sheet: Avoiding For-Loops

What is wrong with using for-loops?
Nothing! R's (for-while-repeat) loops are intuitive, and easy to code and maintain. Some tasks are best managed within loops.

So why discourage the use of for-loops?
1) Side effects and detritus from inline code. Replacing a loop with a function call means that what happened in the function stayed in the function. 2) In some cases increased speed (especially so with nested loops and from poor loop-coding practice).

How to make the paradigm shift?
1) Use R's vectorisation features. 2) See if object indexing and subset assignment can replace the for-loop. 3) If not, find an "apply" function that slices your object the way you need. 4) Find (or write) a function to do what you would have done in the body of the for-loop. Anonymous functions can be very useful for this task. 5) if all else fails: move as much code as possible outside of the loop body

Play data (for the examples following)
```
require('zoo'); require('plyr'); n <- 100;
u <- 1:n; v <- rnorm(n, 10, 10) + 1:n
w <- round(runif(n, 0.6, 9.4)) #min=1 max=9
df <- data.frame(month=u, x=u, y=v, z=w)
l <- list(x=u, y=v, z=w, yz=v*w, xyz=u*v*w)
trivial.add <- function(a, b) { a + b }
```

Use R's vectorisation features
```
tot <- sum(log(u))   # replaces the C-like:
# tot <- 0;                            # YUK
# for(i in seq_along(u))               # YUK
#     tot <- tot + log(u[i])           # YUK
```

Clever indexing and subset assignment
```
df[df$z == 5, 'y'] <- -1        # replaces:
# for(row in seq_len(nrow(df)))        # YUK
#    if(df[row, 'z'] == 5)             # YUK
#        df[row, 'y'] <- -1            # YUK
df[is.na(df)] <- 0 # remove NAs from the df
```

The base apply family of functions
```
# apply(X, MARGIN, FUN, ...)
# lapply(X, FUN, ...)
# sapply(X, FUN, ...) # has more options
# vapply(X, FUN, FUN.VALUE, ...) # ditto
# tapply(X, INDEX, FUN = NULL, ...) # "
# mapply(FUN, ..., MoreArgs = NULL) # "
# eapply(env, FUN, ...) # has more options
# replicate(n, expr, simplify = "array")
# by(data, INDICES, FUN, ...) # more opts
# aggregate(x, by, FUN, ...) # for a df
# rapply() # see help for options!?
```

lapply (on vector or list, return list)
```
lapply(l, mean) # returns a list of means
unlist( lapply(u, trivial.add, 5) )
# Last case: vapply() or sapply() better
```

sapply (a simplified lapply on v or l)
```
# Object: v, l; Returns: usually a vector
sapply(l, mean) # returns a vector
sapply(u, function(a) a*a) # vec of squares
sapply(u, trivial.add, -1) # function above
```

tapply (group v/l by factor & apply fn)
```
count.table <- tapply(v, w, length)
min.1 <- with(df, tapply(y, z, min))
```

by (on l or v, returns "by" objects)
```
min.2 <- by(df$y, df$z, min) # like above
min.3 <- by(df[, c('x', 'y')], df$z, min)
# last one: finds min from two columns
```

aggregate
```
ag <- aggregate(df, by=list(df$z), mean)
aggregate(df, by=list(w, 1+(u%%12)), mean)
# Trap: variables must be in a list
```

apply (by row/column on two+ dim object)
```
# Object: m, t, df, a (has 2+ dimensions)
# Returns: v, l, m (depends on input & fn)
column.mean <- apply(df, 2, mean)
row.product <- apply(df, 1, prod)
# Traps: apply coerces a df to a matrix to
#        do its magic. Col names are lost.
```

rollapply – from the zoo package
```
# A 5-term, centred, rolling average
v.ma5 <- rollapply(v, 5, mean, fill=NA)
# Sum 3 months data for a quarterly total
v.qtrly <- rollapply(v, 3, sum, fill=NA,
           align='right') # align window
# Note: zoo has rollmean(), rollmax() and
# rollmedian() functions
```

Inside a data.frame
```
# Use transform() or within() to apply a
# function to a column in a data.frame. Eg:
df <- within(df, v.qtrly <- rollapply(v,
       3, sum, fill=NA, align='right'))
# use with() to simplify column access
```

The plyr package
Plyr is a fantastic family of apply like functions with a common naming system for the input-to and output-from split-apply-combine procedures. I use ddply() the most.
```
# ddply(.data, .var, .fun=NULL, ...)
ddply( df, .(z), summarise, min = min(y),
    max = max(y) )
ddply( df, .(z), transform, span = x - y )
```

Other packages worth looking at
```
# foreach – a set of apply-like fns
# snow - parallelised apply-like functions
# snowfall – a usability wrapper for snow
```

Abbreviations
v=vector, l=list, m=matrix, df=data.frame, a=array, t=table, f=factor, d=dates

R Cheat Sheet: OOP and S3 Classes

What is object oriented programming?

While definitions for OOP abound without clear agreement, OOP languages typically focus programmers on the actors/objects (nouns) of a problem rather than the actions/procedures (verbs), by using a common set of language features, including:

1) Encapsulation of data and code – the data and the code that manages that data are kept together by the language (in classes, modules or clusters, etc.) Implicitly, this includes the notion of class definitions and class instances.
2) Information hiding – an exposed API with a hidden implementation of code and data; encourages programming by contract
3) Abstraction and inheritance – so that similarities and differences in the underlying model/data/code/logic for related objects can be grouped & reused
4) Dynamic dispatch – more than one method with the same name – where the method used is selected at compile or run-time by the class of the object and also the class of the method parameter types and their arity (argument number).

Note: R is a functional programing language (FPL). Typically FPLs are a better approach than OOP for the scientific analysis of large data sets. Nonetheless, over time, some OOP features have been added to R.

Four R mechanisms with some OOP features

1) Lexical scoping – simple – encapsulation – mutability - information hiding – BUT not real classes – no inheritance.
2) S3 classes – multiple dispatch on class only - inheritance – BUT just a naming convention – no encapsulation – no information hiding – no control over use – no consistency checks – easy to abuse.
3) S4 formal classes – multiple inheritance – multiple dispatch – inheritance – type checking – BUT no information hiding – verbose and complex to code – lots of new terms – immutable classes only.
4) R5 reference classes – built on S4 - mutable (more like Java, C++) – type checking – multiple inheritance – BUT no information hiding – inconsistent with R's functional programming heritage

Note: None of R's OOP systems are as full featured or as robust as (say) Java or C++. (See table at the bottom of this sheet).

What are S3 classes

```
# An S3 class is any R object to which a
# class attribute has been attached.
```

S3 classes – key functions

```
class(x); class(x) <- 'name' #get/set class
methods('method')          # list S3 methods
UseMethod('method', x)   # generic dispatch
NextMethod()     # inheritance sub-dispatch
```

Class code example

```
c.list <- list(hrs=12, mins=0, diem='am')
class(c.list) <- 'clock' # set the class
class(c.list) # -> "clock"
# Also can be constructed with structure()
# (not recommended), and attr() functions
clock <- structure(list(hrs = 12, mins = 0,
         diem = "am"), .Names = c("hrs",
         "mins", "diem"), class = "clock")
c.list <- unclass(c.list) # remove class
attr(c.list, 'class') <- 'clock' # and back
```

Dynamic dispatch – UseMethod()

```
# the UseMethod for print already exists:
# print <- function(x) UseMethod('print',x)
# So we just need to add a generic method:
print.clock <- function(x) {
    cat(x$hrs);  cat(':');
    cat(sprintf('%02d', x$mins));
    cat(' '); cat(x$diem);  cat('\n')
}
print(c.list) # prints "12:00 am"
# you can find the many S3 print methods:
methods('print') # -> a very long list ...
```

Inheritance dispatch - NextMethod()

```
# S3 classes allow for a limited form of
# class inheritance for the purposes of
# method dispatch. Try the following code:
sound <- function(x) UseMethod('sound', x)
sound.animal  <- function(x) NextMethod()
sound.human   <- function(x) 'conversation'
sound.cat     <- function(x) 'meow'
sound.default <- function(x) 'grunt'
Cathy <- list(legs=4)
class(Cathy) <- c('animal', 'cat')
Harry <- list(legs=2)
class(Harry) <- c('animal', 'human')
Leroy <- list(legs=4)
class(Leroy) <- c('animal', 'llama')
sound(Cathy); sound(Harry); sound(Leroy)
```

Should I use S3 or S4 or R5?

S3: for small/medium projects; S4 for larger; R5 if mutability is necessary

The various OOP features available in R

	Type checking	Mutable classes	Encapsulation	Info hiding	Data abstraction	Inheritance	Dynamic dispatch
LS	No	Yes	Yes	Yes	No	No	No
S3	No	No	No	No	No	Yes, clunky	Yes/limited
S4	Yes	No	Yes	No	Yes	Yes	Yes
R5	Yes	Yes	Yes	No	Yes	Yes	Yes

R Cheat Sheet: Environments, Frames and the Call Stack

Environments

1) R uses environments to store the name-object pairing between variable name and the R object assigned to that variable (assign creates pair: <-, <<-, assign())
2) They are implemented with hash tables.
3) Like functions, environments are "first class objects" in R: They can be created, passed as parameters and manipulated like any other R object.
4) Environments are hierarchically organised (each env. has a parent).
5) When a function is called, R creates a new environment and the function operates in that new environment. All local variables to the function are found in that environment (aka frame).

Code example:

```
dictionary <- function() {
    # private ... effectively hidden
    e <- new.env(parent=emptyenv())
    # use emptyenv() to stop chained lookup
    keyCheck <- function(key) # sanity chk
        stopifnot(is.character(key) &&
            length(key) == 1)
    # public ... made public by list below
    hasKey <-function(key) {
        keyCheck(key)
        exists(key, where=e,
            inherits=FALSE)
    }
    rmKey <- function(key) {
        stopifnot( !missing(key) )
        keyCheck(key)
        rm(list=key, pos=e)
    }
    putObj <- function(key, obj=key) {
        stopifnot( !missing(key) )
        keyCheck(key)
        if(is.null(obj)) return(rmObj(key))
        assign(key, obj, envir=e)
    }
    getObj <- function(key) {
        stopifnot( !missing(key) )
        keyCheck(key)
        if(!hasKey(key)) return(NULL)
        e[[key]] # also $ indexing possible
    }
    allKeys <- function()
        ls(e, all.names=TRUE)
    allObjs <- function()
        eapply(e, getObj, all.names=TRUE)
    list(hasKey=hasKey, allKeys=allKeys,
        rmKey=rmKey, getObj=getObj,
        putObj=putObj, allObjs= allObjs)
}
d <- dictionary();                # create
sapply(LETTERS, d$putObj)         # populate
d$hasKey('A'); d$allKeys()        # inspect
d$allObjs()                       # inspect
d$getObj('A')                     # retrieve
d$rmKey('A'); d$hasKey('A')       # remove
```

Code example explained

The above dictionary function returns the list at the end of the function. That list and the listed callable functions exist in the environment created when the dictionary function was called. This use of functions and lexical scoping is a poor man's OOP-class-like mechanism. The function also creates an environment (e), which it uses for its hash table properties to save and retrieve key-value pairs.

Lexical and dynamic scoping

R is a lexically scoped language. Variables are resolved in terms of the function in which they were written, then the function in which that function was written, all they way back to the top-level global/ package environment where the program was written. Variables are not resolved in terms of the functions that called them when the program is running (dynamic scoping). Interrogating the function call stack allows R to simulate dynamic scoping.

Frames and environments

A frame is an environment plus a system reference to a calling frame. R creates each frame to operate within (starting with the global environment, then a new frame with each function call). All frames have associated environments, but you can create environments that are not associated with the call stack (like we did with e above).

The call stack

As a function calls a new function, a stack of calling frames is built up. This call stack can be interrogated dynamically.

```
# some call stack functions ...
sys.frame()     # the current frame
parent.frame()  # get the frame for the
                # calling function (an env)
parent.frame(1) # same as above
parent.frame(2) # get the grandparent
                # function's frame
# parent.frame(n) is the same as ...
#   sys.frame(sys.parent(n))
sys.nframe()    # the current frame number
                # (global environment = 0)
                # on the call stack
sys.call()      # returns the call (which
                # is language expression)
sys.call(-1)    # parent function's call
sys.call(1)     # the first function call
                # on the call stack down
                # from the global env.
deparse(sys.call())[[1]] # string name
                # of this function
# potential confusions ...
parent.env(sys.frame()) # lexical scoping
Sys.getenv() #Operating System environment
Sys.setenv() #as above – not an R env.
```

R Cheat Sheet: R5 Reference Classes

Summary of some key class mechanisms

1) create/get object-generator:
```
gen <- setRefClass('name', fields = ,
  contains = , methods =, where =, ...)
gen <- getRefClass('name') – generator
gen$lock('fieldName') – lock a field
  (better to lock with accessor methods)
gen$help(topic) – get help on the class
gen$methods(...) – add methods to class
gen$methods() – get a list of methods
gen$fields() – get a list of fields
gen$accessors(...) – create get/set fns
```

2) generator object used to get instance:
```
inst <- gen$new(...) – instantiation
  parameters passed to initialize(...)
inst$copy(shallow=F) – copy instance
inst$show() – called by print
inst$field(name, value) – set
inst$field(name) – get
is(inst 'envRefClass') – is R5 test
[envRefClass is the super class for R5]
```

3) code from within your methods
```
initialize(...) – instance initializer
finalize() – called by garbage collector
.self – reference to the self instance
.refClassDef – the class definition
methods::show() – call the show function
callSuper(...) – call the same method in
  the super class
.self$classVariable <- localVariable
classVariable <<- localVariable
globalVariable <<- localVariable
.self$classVariable <- localVariable
.self$field(classVar, localVar)   # set
localVar <- .self$field(classVar) # get
```
Trap: very easy to confuse <- and <<-
Trap: if x is not a class field; x <<- var assigns to x in global environment

Field list – code sample
```
A <- setRefClass('A',
    fields = list(
        # 1. typed, instance field:
        exampleVar1 = 'character',
        # Note: for untyped use 'ANY'
        # 2. instance field with accessor:
        ev2.private = 'character',
        exampleVar2 = function(x) {
            if (!missing(x))
                  ev2.private <<- x
            ev2.private
        }
    ),
    methods = list(
        initialize=function (c='default') {
            exampleVar1 <<- c
            exampleVar2 <<- c
        }
    )
)
instA <- A$new('instance of A'); str(instA)
```

Inheritance code sample
```
Animal <- setRefClass('Animal',
    # virtual super class
    contains = list('VIRTUAL'),
    fields = list(
        i.am = 'character',
        noiseMakes = 'character'
    ),
    methods = list(
        initialize=function(i.am='unknown',
            noiseMakes = 'unknown') {
            .self$i.am <- i.am
            .self$noiseMakes <- noiseMakes
        },
        show = function() {
            cat('I am a '); cat(i.am)
            cat('. I make this noise: ')
            cat(noiseMakes); cat('.\n')
        }
    )
)

Cat <- setRefClass('Cat',
    contains = list('Animal'),
    methods = list(
        initialize = function()
            callSuper('cat', 'meow'),
        finalize = function()
            cat('Another cat passes.\n')
     )
)
Dog <- setRefClass('Dog',
    contains = list('Animal'),
    methods = list(
        initialize = function()
            callSuper('dog', 'woof'),
        show = function() {
            callSuper()
            cat('I like to chew shoes.\n')
        }
    )
)

mongrel <- Animal$new() # FAILS!
fido = Dog$new(); felix = Cat$new()
print(fido); print(felix)
felix <- NULL; gc() # felicide!
```

What's neither C++ nor Java
1) No information hiding. Everything is public and modifiable. (But the R package mechanism helps here).
2) No static class fields.
3) Not as developed or robust OOP space.

Tips (safer coding practices) and traps
1) use named field list to type variables
2) use accessor methods in the field list to maintain class type & state validity
3) *Trap*: methodName <- function() in methods list. Use = (it's a named list!)
4) *Trap*: cant use enclosing environments within R5 classes (as they are in one).

