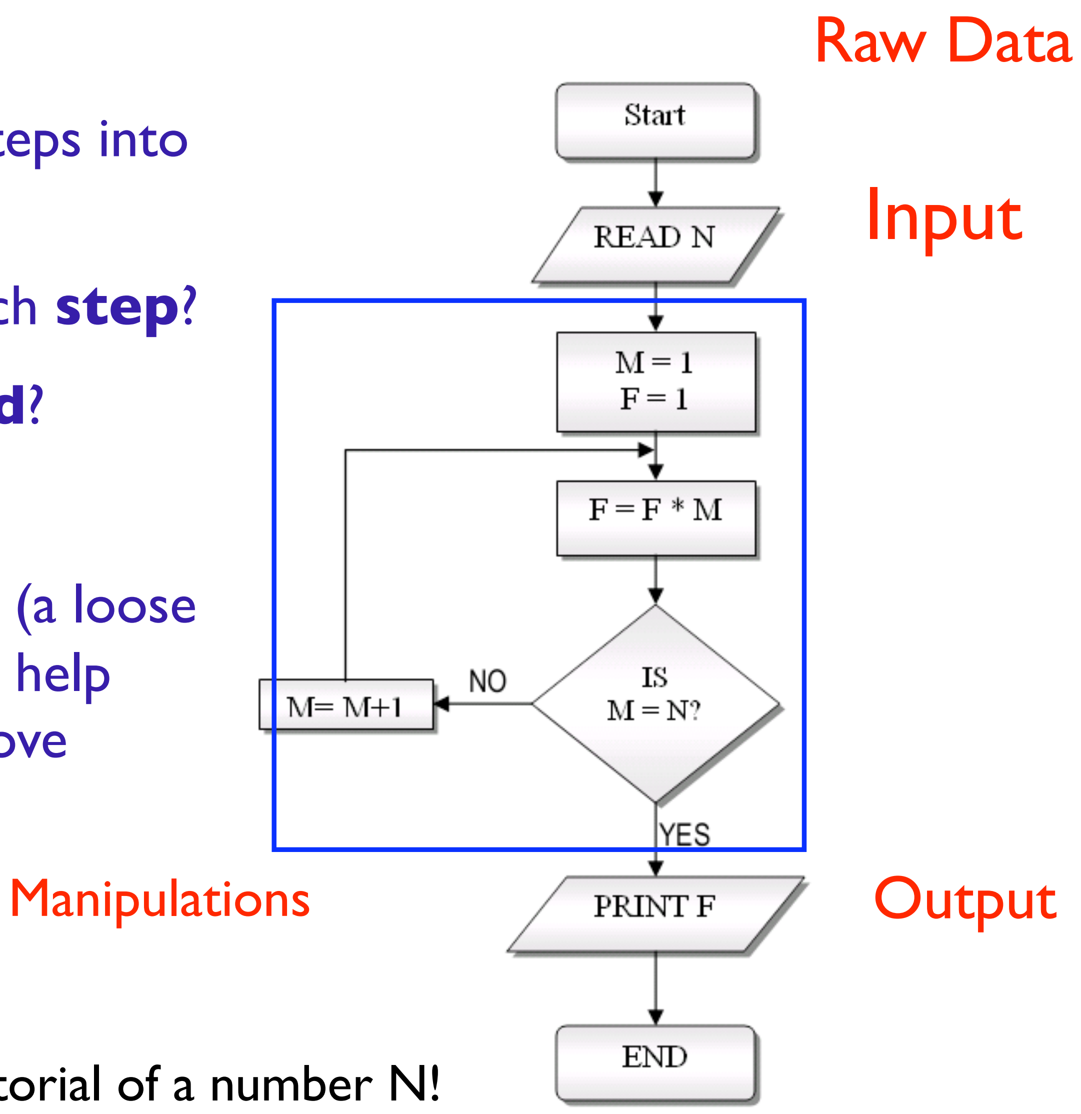
# Work Flow

**Abstract** your programming steps into distinct actions or **steps**

What has to happen **within** each **step**?

How are the **steps connected**?

**Flowcharts** or **Pseudocode** (a loose flowchart in words) can really help clarify your thinking and improve **efficiency**



A flowchart for computing the factorial of a number N!
from wikipedia http://en.wikipedia.org/wiki/Flowchart

## the R environment

The R language is an integrated suite of software facilities for:

- Data Handling and storage
- Matrix Math: Manipulating matrices, vectors, and arrays
- Statistics: A large, integrated set of tools for data analysis
- Graphics: Graphical facilities for data analysis and display
- Programming: Powerful programming language ('S')

## R works on Objects

# R object types

**Vector:** a one-dimensional array of arbitrary **length**.

**Matrix:** a two-dimensional array with an arbitrary **number of rows and columns**.

**Array:** as a matrix, but of **arbitrary dimension** (i.e., more than 2).

**Data frame:** a set of data organized similarly to a matrix. However each column of the data frame may contain its own type of data. Columns typically correspond to variables in a statistical study, while rows correspond to observations of these variables.

**Function:** a set of commands that are packaged into a unit with defined input and output (I/O is not necessary, though).

**List:** an arbitrary collection of other R objects (which may include other lists).

```
> x <- cbind(a=1:3, pi=pi) # simple
                  # matrix w/ dimnames
> x
     a       pi
[1,] 1 3.141593
[2,] 2 3.141593
[3,] 3 3.141593

> class(x)
[1] "matrix"

> attributes(x)
$dim
[1] 3 2

$dimnames
$dimnames[[1]]
NULL

$dimnames[[2]]
[1] "a"  "pi"

> attributes(x) <- NULL
> x # now just a vector of length 6

[1] 1.000000 2.000000 3.000000 3.141593
   3.141593 3.141593

> class(x)      # vector is default mode
[1] "numeric"
```

# R works on objects

**What is an object?**

- data vector, matrix, array, data frame, list
- function

**But, really, what is an object?**

*It is an abstraction:*

Something that has ATTRIBUTESs (values) and BEHAVIORs (actions) (sometimes called STATEs and METHODs). These are formally defined in the object definition.

<u>real world example: radio:</u>

**states** (on, off, current volume, current station)

**behaviors** (turn on, turn off, increase volume, decrease volume, seek, scan, and tune)

## What is an object?

•Objects are ways of bundling parts of programs into small, manageable pieces.

•Objects are simply a definition for a type of data to be stored.

•An object is a component of a program that knows how to perform certain actions and to interact with other pieces of the program.

•Functions can be described as "black boxes" that take an input and spit out an output. Objects can be thought of as "smart" black boxes. That is, objects can know how to do more than one specific task (method or behavior), and they can store their own set of data.

## Object example: Medieval Video Game

Two types of players: **Monsters** and **Heros**

Heros have to know the values of certain **attributes**:

- Health = 16
- Strength = 12
- Agility = 14
- type of weapon = “mace”
- type of armor = “leather”

Heros must also be able to (**behaviors**):

- move through the maze
- attack monsters
- pick up treasure

These **attributes** and **behaviors** completely define the Hero. Modules may be written that know how to interpret (interact with) heros.

## Benefits of Objects and OOP (or: why bother?)

**Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

**Information-hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

**Code re-use**: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

**Pluggability and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

## The R Environment

## the R environment : Session

**SESSION**: A "session" is a single use of the R language, the period between starting R up and shutting it down.

- Within a session, you may load packages, create R objects, produce graphics, or write/run scripts.

**WORKSPACE**: The collection of objects currently stored.

```
> ls()   # display the names of objects in workspace
> rm(list=ls()) # deletes all objects from workspace
```

## the R environment : Session

At end of SESSION:

```
Save workspace image? [y/n/c]:
```

If you say “y”, the objects are written to a file called “.RData” in the current directory, and the command lines used in the session are saved to a file called “.Rhistory”.

NOTE: Generally, files that start with a “.” are hidden in the directory.

**IF** you restart R from the same directory, it will reload the default (.Rhistory, .Rdata) history and workspace files.

use save(), save.image(), savehistory() to save these files with your own names specified within the parentheses.

## the R environment : Packages

The R programming language is written in **modules** called **packages**, which are groups of related functions organized together in a bundle.

Packages are the means by which R is extended by the open-source community (user contributed packages)

Not all aspects of the R programming language are **loaded** every time you fire R up. -- you have to load optional packages with every new session.

## the R environment : Packages

**default** packages at startup are **base** & a few others.

```
> search()   # gives search path for R objects

 [1] ".GlobalEnv" "tools:RGUI" "package:methods" "package:stats"
 [5] "package:graphics" "package:grDevices" "package:utils"
"package:datasets"
 [9] "Autoloads" "package:base"
```

***When you type a command or object name in R, it "searches" through the "search path" for a match and then takes appropriate action (be it the name of a data object, function, operator, etc.).***

**NOTE:** Don't create objects with the same name as R commands! (e.g., t, T, F, c all are special characters) Results are unpredictable

## the R environment : Packages

```
> searchpaths()   # gives path to package source code on your
                   # computer's file system

 [1] ".GlobalEnv"
 [2] "tools:RGUI"                                                \
 [3] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/methods"
 [4] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/stats"
 [5] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/graphics"
 [6] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/grDevices"
 [7] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/utils"
 [8] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/datasets"
 [9] "Autoloads"
[10] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/base"
```

INSTALLING Packages: saves the package source code to the appropriate place in your computer's file directory (usually a place that users don't mess with)

LOADING Packages: attaches the package from your computer's R library to your search path (adds it to your session).

```
> library("packagename")
```

# the R environment : attach(), with()

ATTACH: Attach Set of R Objects to Search Path

The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.

```
> data.frame(xx=1:10,yy=runif(10)) -> ddata
> attach(ddata)
> search()
 [1] ".GlobalEnv"   "ddata"   "tools:RGUI"   "package:methods"
 [5] "package:stats"   "package:graphics"   "package:grDevices"
"package:utils"
 [9] "package:datasets"   "Autoloads"   "package:base"
```

Now ddata is in the search path, so R knows what  you mean when you specify "xx" or "yy" without the data.frame.

**WITH: Creates a tiny local environment for single line of code, using a dataframe. Less confusing that ATTACH**

## Programming Elements

## inside R : functions

To get the code of a function, you can just type its name -- with no brackets.

```
> summary
function (object, ...)         # It is a generic function, that uses different methods
UseMethod("summary")           # depending on the class attribute of the object
<environment: namespace:base>

> methods("summary")          # Here are all the methods that are defined for “summary”

 [1] summary.Date           summary.POSIXct        summary.POSIXlt        summary.aov
 [5] summary.aovlist        summary.connection     summary.data.frame     summary.default
 [9] summary.ecdf*          summary.factor         summary.glm            summary.infl
[13] summary.lm             summary.loess*         summary.manova         summary.matrix
[17] summary.mlm            summary.nls*           summary.packageStatus* summary.ppr*
[21] summary.prcomp*        summary.princomp*      summary.stepfun        summary.stl*
[25] summary.table          summary.tukeysmooth*


   Non-visible functions are asterisked
```

## inside R : functions

Here's how we find out what is inside summary.lm:

```
> summary.lm
function (object, correlation = FALSE, symbolic.cor = FALSE, ...)   {
    z <- object
    p <- z$rank
    if (p == 0) {
        r <- z$residuals
        n <- length(r)
        w <- z$weights
        if (is.null(w)) {
            rss <- sum(r^2)
        }
        else {
            rss <- sum(w * r^2)
            r <- sqrt(w) * r
        }
        resvar <- rss/(n - p)
        ans <- z[c("call", "terms")]
        class(ans) <- "summary.lm"
        ans$aliased <- is.na(coef(object))
        ans$residuals <- r
...
}
```

Note: **Internal** functions are "hidden" inside the namespace of a package -- the programmer has chosen to not make it available to the global environment. To find these, use `getAnywhere("functionname")`

## Create your own functions - much simpler!

A function is defined as follows.

```
> quadx <- function(x) {        # function definition, input variables
    x^2 + x + 1               # R statements, what the function does
  }
```

The return value is the last value computed -- but you can also use the "return" function.

```
> quadx <- function(x) {
  return( x^2 + x + 1 )
}
> fxy <- function(x, y=3) { x+y^2 } # Arguments can have default values.
```

When you call a function and list the argument **names**, you don’t have to worry about the order you list them in (this is very useful for functions that expect many arguments -- in particular arguments with default values).

```
> f(y=4, x=3.14)
> f(4, 3.14)       #  without argument names, it will assign x=4, y=3.14
```

# Create your own functions - much simpler!

After the arguments, in the definition of a function, you can put three dots represented the arguments that have not been specified and that can passed through another function (very often, the "plot" function).

```
f <- function(x, ...) {          # ... you could put here col="red", pch=19, or any
  plot(x, ...)                   # other valid argument
}
```

But you can also use this to write functions that take an arbitrary number of arguments:

```
f <- function (...) {
  query <- paste(...)  # Concatenate all the arguments to form a string
  con <- dbConnect(dDriver("SQLite"))
  dbGetQuery(con, query)
  dbDisconnect(con)
}

f <- function (...) {
  l <- list(...)       # Put the arguments in a (named) list
  for (i in seq(along=l)) {
    cat("Argument name:", names(l)[i], "Value:", l[[i]], "\n")
  }
}
```

Functions have NO SIDE EFFECTS: all the modifications are local. In particular, you cannot write a function that modifies a global variable.

# Control structures

## Loops, conditionals, recursion, etc.

Conditional statements:

```
if(x>0.001) {     # if condition is true, then execute statements
  1/x          # inside the parentheses
} else {       # if condition is false, then do these other statements
 print(“Sorry! must be greater than zero”);
 break;        # “break” is the emergency exit -- break out of
}              # function
```

Conditionals may be used inside other constructions.

```
x <- if(...) 3.14 else 2.71   # if True, x=3.14, else x=2.71
```

You can also construct vectors from conditional expressions, with the "ifelse" function.

```
x <- rnorm(100)
y <- ifelse(x>0, 1, -1)   # if x>0, then y=1, else y=-1
z <- ifelse(x>0, 1, ifelse(x<0, -1, 0))
x > 0       # this is also a conditional, produces a logical vector
```

Switch is also available, but not really necessary usually.

## Loops are slow in R

## (but sometimes necessary)

## Loops - Repeat, repeat

For loop (we loop over the elements of a vector or list): This means that we count the number of times we repeat -- according to the number of elements in the list:

```
for (i in 1:10) {   # loop through 10 times, from i=1 to i=10
  dat <- read.table()
  ...               # some statements to execute
  if(length(dat)<2) { next }

  if(is.na(dat)) { break }   ...

}
```

```
# next halts the processing of the current iteration and advances t he
looping index (increases i but doesn't execute any statements -- like a
"bad apple -- pass on this one and move on to the next"

# stops execution, "breaks" out of for loop, while, or repeat loop;
control is transferred to the first statement outside of the inner-most
loop.
```

*Both break and next apply only to the innermost of nested loops.*

# Loops

While loop:

```
while(money<100000) {  # while (condition) is true, allows execution
  print("Please make more money")
  money = workhard(stockmarket)   # the value inside the condition
                                 # must be able to change -- make sure
                                 # you can meet the condition or you
                                 # will have an infinte loop
}
```

Repeat loop:

```
repeat {            # keeps repeating until it reaches the break
  ...
  if(...) { break }
  ...
}
```

In general, try to avoid while and repeat loops -- they can go on forever if you're not careful.

## Vectorized Calculations are *fast* in R (but what is that?)

**Operate on the entire vector, matrix, array, dataframe, or list, *all at once!***

# Vectorised Calculations

<u>Simple matrix math:</u>

```
x <- matrix( c(1:4), nrow=2)
x + 1
```

<u>Apply functions:</u>

```
apply(X, MARGIN, FUN, ...)
```

if you want to apply more than one function, then you have to write your own function and embed it

# Vectorised Calculations

apply operates on successive sections of an array

apply(array or vector, MARGIN(which dimension?), function, optional arguments to function)

```
iris[1:4,]                    # the data
apply(iris[,-5],2,mean)# iris[,-5] takes the iris dataset, but drops
                              #     the 5th column
                         # the "2" is the row index and function= "mean"
                         # so we're taking the mean over rows
```

flavors

tapply operates on a "ragged" array (i.e., groups of different sizes, for instance, one vector subscripted by a factor)

tapply(array or vector, indexing vector, function)

```
spnames <- iris[,5]
tapply(iris[,4], spnames, mean) # take 4th column of data
...                            # iris[,5] contains species names
                                # so we're taking the mean by species
for(i in 1:4) {print(tapply(iris[,i], iris[,5],
mean)) }                              # why the print statement?
```

# Vectorised Calculations

```
for(i in 1:4) {print(tapply(iris[,i], iris[,5],
mean)) }             # the print statement is needed because we're inside
...                 # of a loop -- any calculations will go on, but nothing will
                    # display on the screen unless we "print" to screen
```

these all operate on components of a list or vector
**apply** # apply functions over a margin of an array (row, column, etc.)
**lapply** # returns a list
**sapply** # returns friendly vector or array (will return a matrix or vector if
# appropriate
**mapply** # works on multiple arguments
**aggregate** # compute summary statistics over subsets of the data

*Why bother with the apply family when I know/understand looping???*

# Vectorized versus loops

Example: a silly program that subtracts one if less than 5, and divides by itself if greater than 5.

traditional program:

```
x <- 1:10; z <- NULL     # set up
for (i in 1:length(x)) # looping over the length of your vector is
{                      # very convenient -- you don't have to manually
                       # figure out how long your loop should be
  if (x[i]<5) {z <- c(z,x[i]-1)} # conditionally modifies elements of x
  else { z <- c(z,x[i]/x[i])}
}
```

Same thing, but in two-steps: "test" function to decide which method, then apply to all elements of x vector

```
x <- 1:10; z <- NULL
test <- function(x) {if (x<5) { x-1 } else { x/x }}
apply(as.matrix(x), 1, test)
```

simplest apply version:

```
apply(as.matrix(x), 1, test <- function(x) { if (x<5) {x-1}
```

# Vectorized versus loops

Example: a silly program that subtracts one if less than 5, and divides by itself if greater than 5.

Finally, shortest apply version: define the function inside the call to apply

```
apply(as.matrix(x), 1, test <- function(x) { if (x<5) {x-1}
else {x/x} })
```

# function is embedded, so it doesn't need a name because we won't ever need to refer to it by name. just used once and then it "disappears"

# Timing: which is more efficient?

**system.time** returns a numeric vector of length 5: user cpu, system cpu, elapsed, subproc1, subproc2 times (latter are usually zero).

```
> system.time(for (i in 1:length(x)) {if (x[i]<5) {z <-
c(z,x[i]-1)} else { z <- c(z,x[i]/x[i])}})[[3]] #elapsed
```

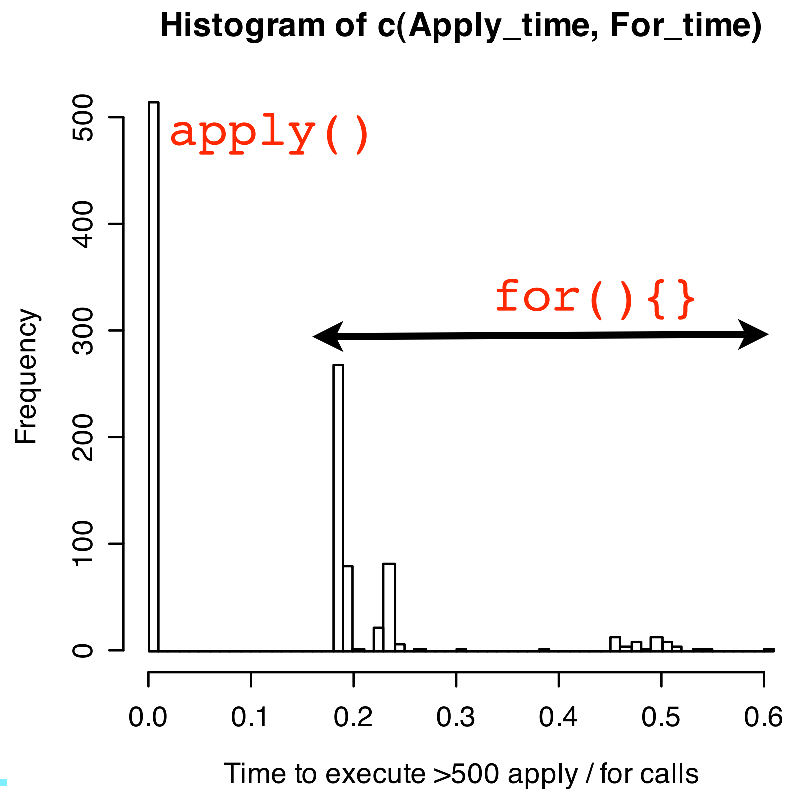
```
          [1] 0.195
```

```
>
system.time(apply(as.matrix(x)
, 1, test <- function(x) { if
(x<5) { x-1 } else { x/x } }))
[[3]]
```

```
          [1] 0.001
```

# nearly 200x faster!

# is this consistently true? ->



## Sampling w/, w/o replacement

sample() can generate a sample with or without replacement, powerful

**sample(x, size, replace = FALSE, prob = NULL)**

**sample(replace=F)** # i.e. default

```
> sample(1:12)
```
# a permutation of vector 1:12

**sample(x,replace=TRUE) #** bootstrap sampling

# only if length(x) > 1 !

```
> sample(1:12,6,T)
```
# try it out, 6 random samples w/replace

```
> sample(c(0,1), 100, replace = TRUE)
```
# 100 Bernoulli trials

# see help(sample) for cautionary tale

```
> rbinom(100,1,.5)
```
# same thing

MORE on random sampling, bootstrap, jackknife, Mantel tests & Monte-Carlo methods next semester...

# References

- ☑ "An Introduction to R: Software for StatisticalModelling & Computing" by Petra Kuhnert and Bill Venables http://cran.r-project.org/doc/contrib/Kuhnert+Venables-R_Course_Notes.zip
- ☑ Programming in R ( a great web page! ) by Vincent Zoonekynd http://zoonek2.free.fr/UNIX/48_R/02.html
- ☑ Programming in R (with R & BioConductor) by Thomas Girke http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/R_Programming.html
- ☐ An Introduction to R http://cran.r-project.org/doc/manuals/R-intro.pdf
- ☐ (Generally, see the R manuals at http://cran.r-project.org/manuals.html )

- ☐ Online C++ tutorial: What is an object? http://www.intap.net/~drw/cpp/cpp06_01.htm
- ☐ The JAVA tutorials: What is an object? http://java.sun.com/docs/books/tutorial/java/concepts/object.html