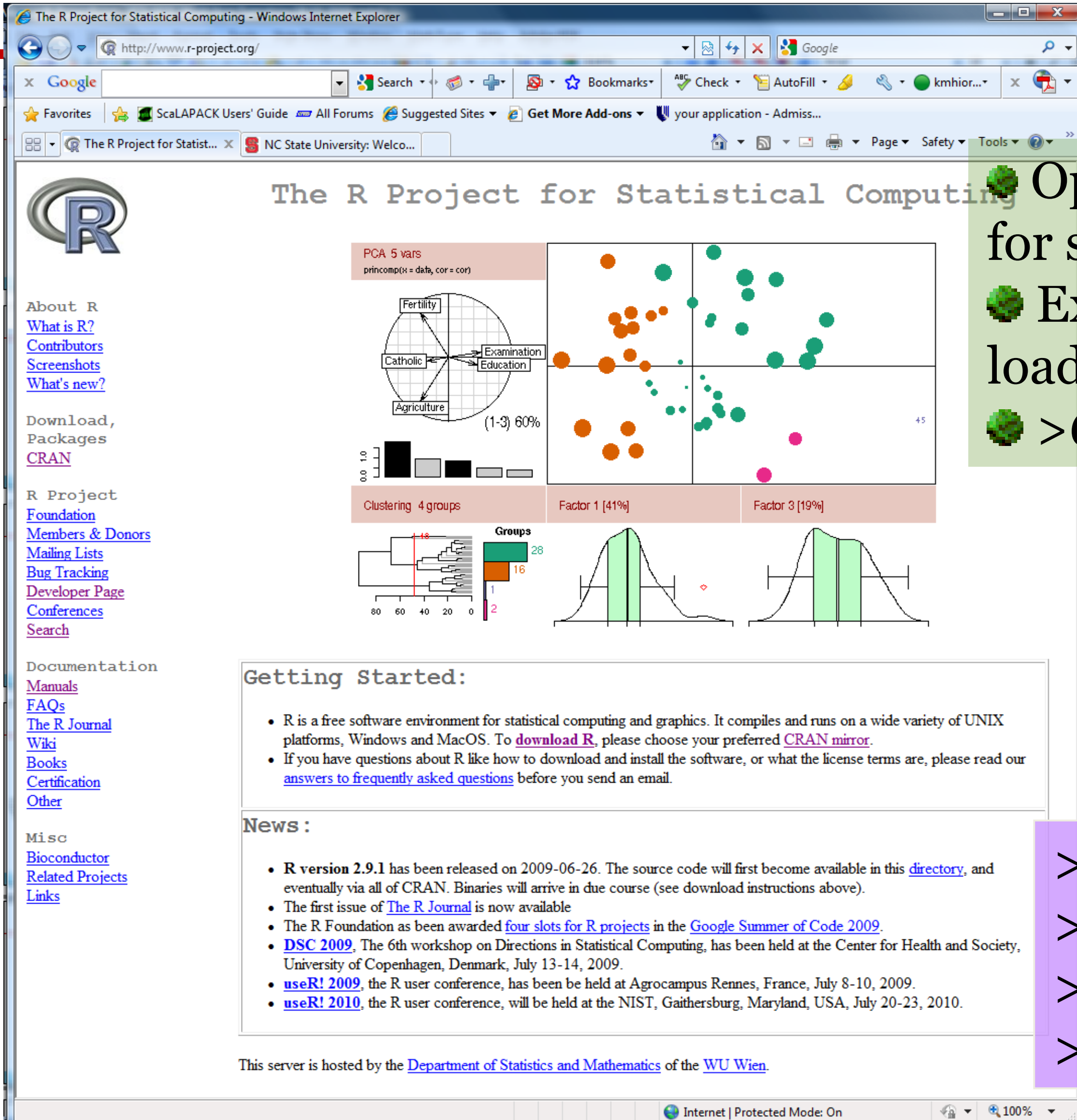
Introduction to R: Parts I-III

Nagiza F. Samatova, samatova@csc.ncsu.edu
Professor, Department of Computer Science
North Carolina State University

Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

What is R and why do we use it?



- Open source, most widely used for statistical analysis and graphics
- Extensible via dynamically loadable add-on packages
- >6,000 packages on CRAN

```
> v <- rnorm(256)
> A <- as.matrix (v,16,16)
> summary(A)
> library (fields)
> image.plot (A)
```

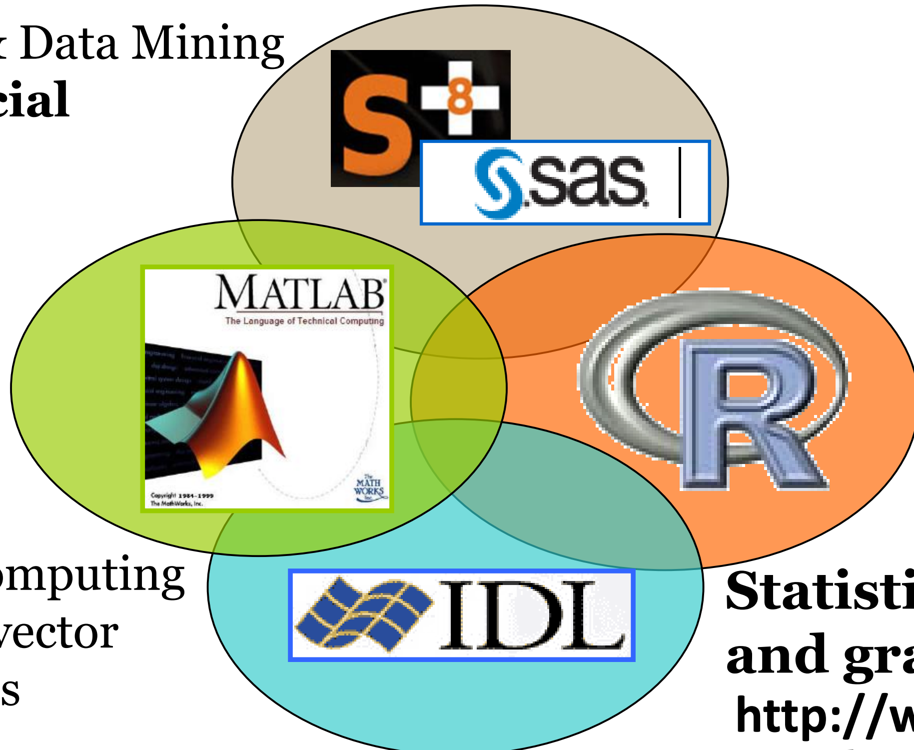
```
> …
> dyn.load( “foo.so”)
> .C( “foobar” )
> dyn.unload( “foo.so” )
```

Exercise #1: R Packages

1. Open RStudio and answer the following questions:
   - What R packages are installed on your system?
   - What R packages are loaded into R environment?
   - What is the difference between installed and loaded package?
2. Go to CRAN web-site: http://www.r-project.org/
   - How many Contributed Packages are in R?
   - How are the contributed packages organized?
     - Do you see the value of each type of package organization?
   - How comprehensive Task Views?
   - Are all the contributed packages included in one/more Task Views?
   - What packages exist for analysis of social networks?
   - What packages exist for analysis of text data?
3. From RStudio's environment:
   - Click on Help and type ***sna*** to find out what sna package is for?
   - Is sna package installed in your environment?
   - Install sna package from RStudio?
   - Type ***sna*** in Help again. What do you see?
   - Is ***sna*** package loaded into your environment? Can you load it?
   - How do you start with using capabilities offered by sna package?

Why R?

- Statistics & Data Mining
- **Commercial**



- Technical computing
- Matrix and vector formulations

- Data Visualization and analysis platform
- Image processing, vector computing

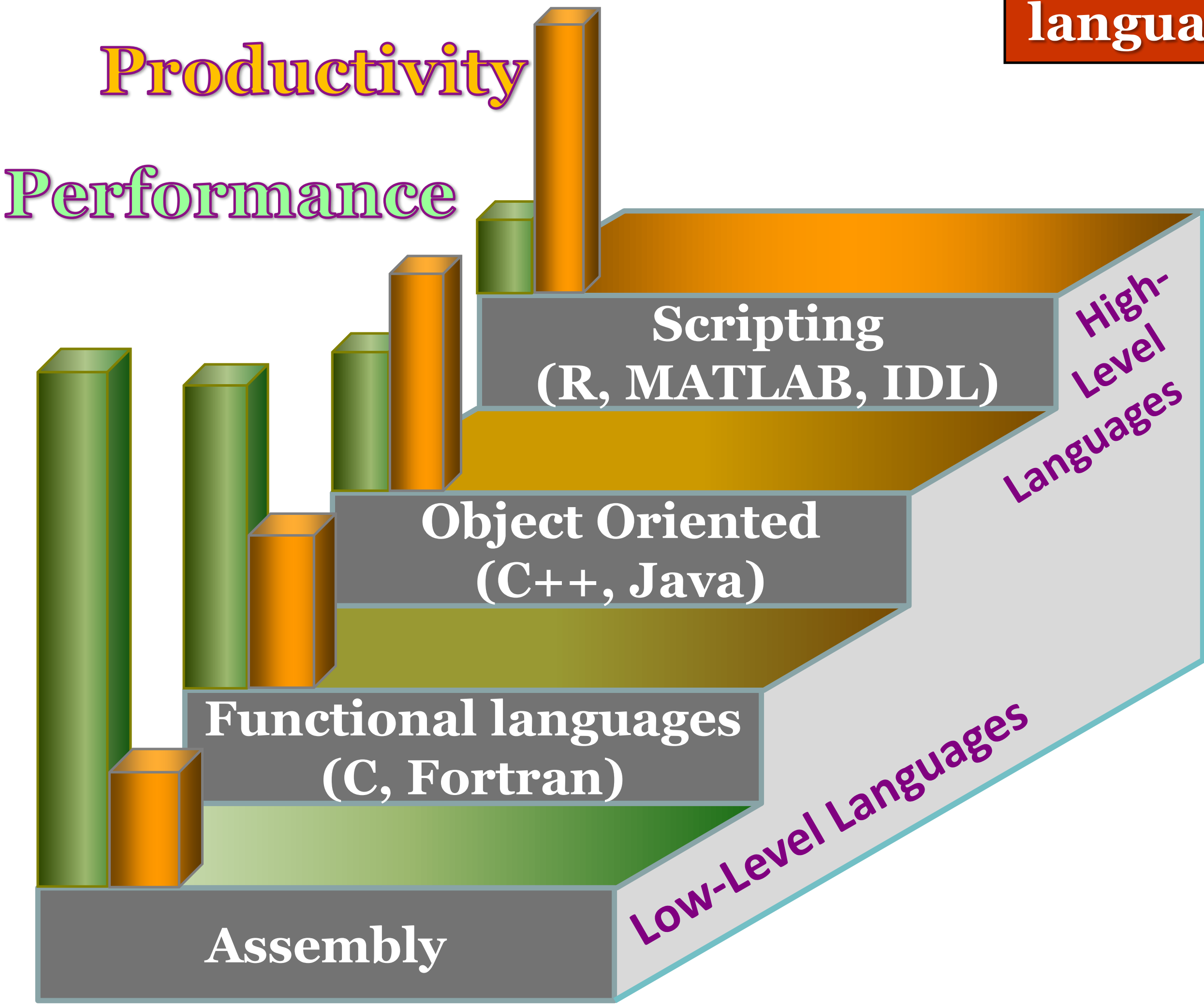
Statistical computing and graphics

http://www.r-project.org

- Developed by **R**. Gentleman & **R**. Ihaka
- Expanded by community as **open source**
- **Rich in stat and machine learning** capabilities

The Programmer's Dilemma

What programming language to use & why?



interpretable

```
> ...
> dyn.load( “foo.so”)
> .C( “foobar” )
> dyn.unload( “foo.so” )
```

compiled

R is a programming language that is…

- **Object-oriented language**
  - every item in R is an object and has a type definition called ***class***: check with **class()** command
  - the same function may behave differently based on the class type of the object: e.g., apply print() or summary() command to different objects
- **Functional language**
  - functions are first class objects in R
- **Vectorized**
  - many R operations work on every element of a vector
  - vector operations are preferred over explicit loops: e.g., apply() family
- **Generic language**
  - type signatures for objects do not have to be declared explicitly
  - although functions use them to determine the object class
- **Dynamic/run-time**
  - only object values have types but the variables they are assigned to do not have types

R is an object-oriented language

Building Blocks in R

R Objects

- **Data Structures**
- **Functions**
- **Expressions**

To see the names of objects in the environment:

```
> ls()
> objects()
```

To remove object from the env:

```
> rm(object)
> rm(list = ls())
```

Oak Ridge National Laboratory

NC STATE UNIVERSITY
Department of Computer Science

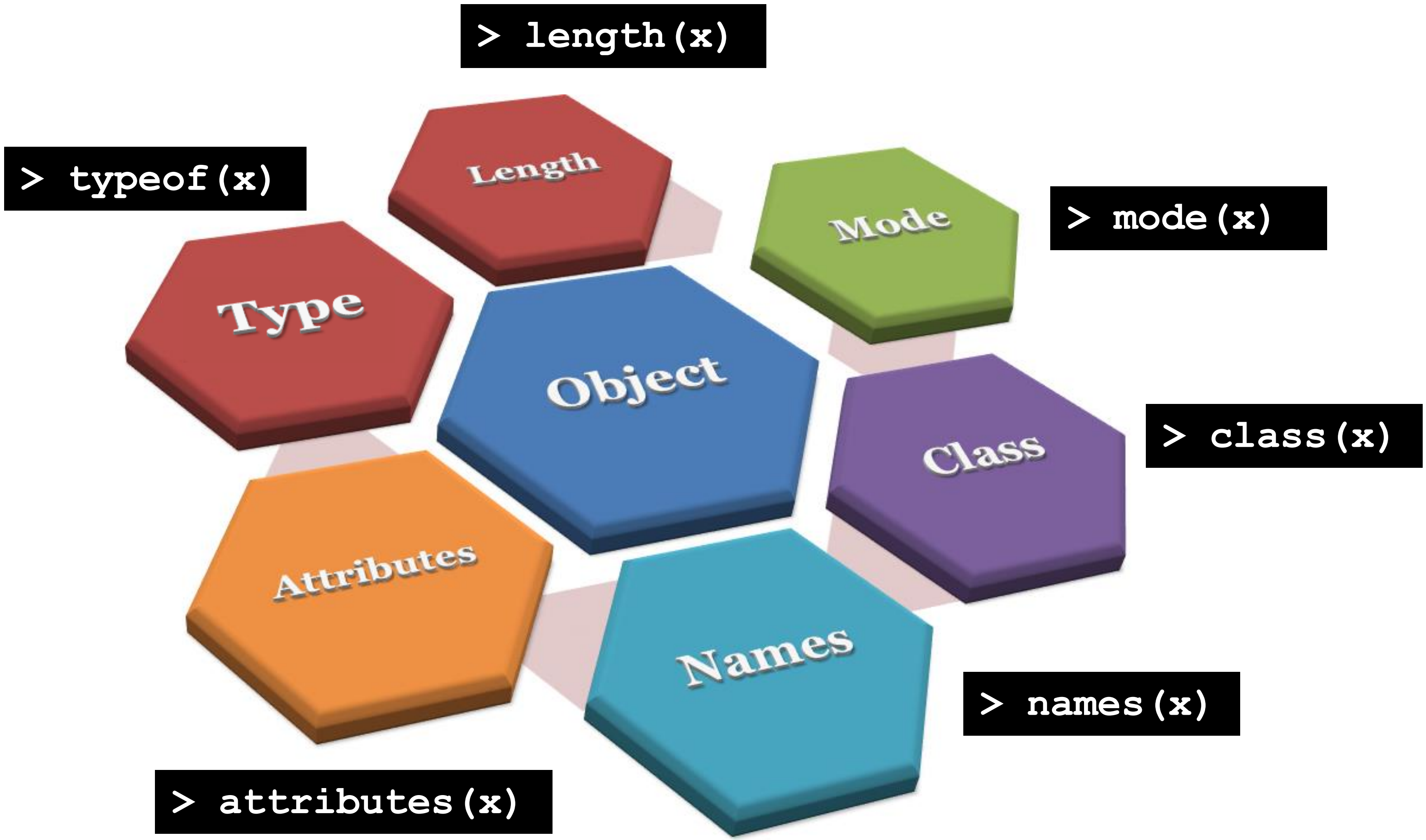
Exercise #2: R Objects

1. Open RStudio:
   - Type ***ls()*** or ***objects()*** command. What do you see?
   - Type ***demo()*** command from the Console window
   - Type ***demo(package = .packages(all.available = TRUE))***
     - What packages on your system provide demos?
   - Type ***demo(graphics)*** command from the Console window
     - As you are going through the demo by Hitting Return, what values and or functions appear in your environment?
     - What types of values do you see in Global Environment?
     - Type ***ls()*** or ***objects()*** command. What do you see now?
2. Do you have a general idea of what different functions in the graphics package have to offer?
   - If you need a particular view graphs presented in the demo, will you be able to follow the R code that accompany this graph in the demo?

Punchline from Exercises #1 and #2

- While R has >6,000 packages on http://CRAN.r-project.org, finding the ones suitable for your task at hand is a challenge:
  - Table of packages sorted by date is only partially useful
  - Table of packages sorted by package name: package names do not often reflect what the package is about (too cryptic)
  - CRAN Task Views organizes packages by topics but many packages from the list of 6,000 are not mentioned in any of these views
- Even once installed, the package may contain dozens of functions that make it very difficult to figure out where even to start (e.g., sna package)
- Good practice is to provide demo(package_name) function that illustrates the capabilities of the package and provides examples of how to use its core functions (e.g., demo(graphics))

Elements of R Object



Exercise #3: Object Elements

1. Open RStudio:
   - Type ***ls()*** or ***objects()*** command after you ran ***demo(graphics)***.
   - Type each of the commands related to object's elements (e.g., mode, class, type, length, attributes, names) for the following variable: ***pie.sales***
     - What is the class of pie.sales?
     - What are its names?
   - Delete a specific object from your environment, e.g., type ***rm(x)*** in the Console?
     - Type ***ls()*** or ***objects()*** command. What do you see now?
   - Remove all the objects produced by the demo(graphics) command:
     - ***rm(list = ls())***
     - How did the display in the Environment tab change?
     - Type ***ls()*** or ***objects()*** command. What do you see now?

Intrinsic Attributes of R Object

How the object is stored:

mode attribute: the basic type of object's fundamental constituents

- e.g.: ***numeric, character, logic, list,…***

Tells functions (e.g. plot() or summary()) how to handle objs:

class attribute: to allow for object-oriented style of programming

- e.g., **lm, data.frame, matrix**

Modes of R Object

mode **attribute:** the basic type of obj's fundamental constituents

- automic structures: ***numeric, complex, character, logical, raw***
- recursive structure: ***list***
- advanced structures: ***function***

To **see** the mode of object ***x***:

```
> mode(x)
```

To **query** the mode of object ***x***:

```
> is.numeric(x)
> is.list(x)
> is.vector(x)
> ...
```

Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Exercise #4: Confirm the modes

R object	Its Mode, mode(z)
`a <- c("1","2")`	character
`b <- c(1,2)`	numeric
`c <- data.frame(1,2)`	list
`d <- plot`	function
`e <- 1`	numeric
`f <- TRUE`	logical

Coercion: Change of Mode

Coercion: the change of the object's mode

- not every mode can be coerced to any other mode (***incompatibility***)
- may result in distorted values (***loss of precision***)
- not always reversible (***irreversible***)
- may result in NA (***missing values***)
- ***explicit*** vs. ***implicit*** coercion

```
> as.numeric(x)
> as.list(x)
> as.vector(x)
> ...
```

NAs introduced by coercion:

```
> z <- c("1","2","CISCO")
> mode(z)
> y <- as.numeric(z)
> y
> mode(y)
```

Incompatible coercion:

```
> df<-data.frame(x=1:3, y=4:6)
> mode(df)
> num <- as.numeric(df)
> mode(num)
```

Fundamental Concept:

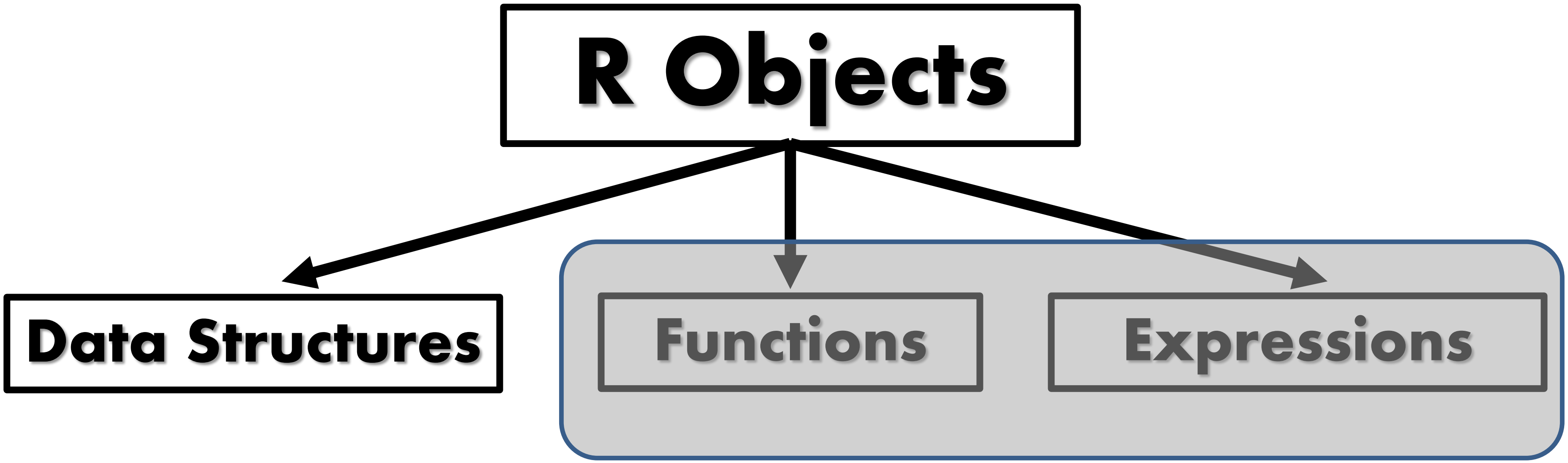
Mode coercion can be dangerous!
Watch for *implicit* coercion!

Exercise #5: mode vs. class

```
> x <- c(1:5)
> y <- c(6:10)
> fm <- lm(y ~ x)
> class(fm)
> mode(fm)
> mode(lm)
> class(lm)
> plot(fm)
```

- What object does the ***lm()*** function return?
  - Is it common for R functions to return ***list*** objects?
- Does ***plot()*** function behave the same way as in the demo?
- Is ***lm*** an object? What is its mode and class?

Building Blocks in R



Advanced Reading: http://adv-r.had.co.nz/Data-structures.html

Taxonomy of Data Structures

Data Structure

Constituents:
all character
all numeric
etc.

Homogeneous Modes

Heterogeneous Modes

Constituents:
mix of numeric, character, etc.

Vector [ ]

Matrix [,]

Array [, ,]

Data Frame [,]

List [[ ]]

Vectors
Arrays
Lists
Data Frame
Matrices

ordered collection of objects

Fundamental Concept:

Using *heterogeneous* modes in a vector, matrix, or array will result in *implicit coercion*!

Exercise #6: help…

help.search("matrix")

This lists all functions whose help pages have a title or alias in which the text string "matrix" appears

help(matrix)

This lists the description of function matrix()

```
> help.search("matrix")
> help.search(matrix)

> help("matrix")
> help(matrix)
> example(matrix)
```

Summary: R objects

- ***R objects***: data structures, functions, and expressions
- ***Object attributes:*** mode, class, type, length, names, dimensions
- ***Data structures:*** homogeneous and heterogeneous based on mode of their constituent elements
- ***Homogeneous data structures:*** vectors, matrices, arrays
- ***Heterogeneous data structures:*** data frames and lists
- ***Coercion:*** change the mode of the object
  - dangerous: missing values, irreversible, loosing precision
  - *explicit* coercion: caused by your action (as.*numeric*(), as.*matrix*())
  - *implicit* coercion: caused by internal actions inside of R functions

R is a programming language that is…

- Object-oriented language
- Functional language
- Vectorized
- Generic language
- Dynamic

R is a functional language

“To understand computations in R, two slogans are helpful:

Everything that exists is an object.
Everything that happens is a function call."

— John Chambers

Advanced Reading: http://adv-r.had.co.nz/Functions.html

What is this in R?

```
> boo()
```

boo ( )

a function!

how do you know it?

( )

```
> data()
> demo()
> ksvm(x,y)
> prcomp(foo)
```


Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

What is inside (???)

(???)

formula

an object of
class "formula"

```
> lm(y~x)
> class (y~x)
```

argument(s)

objects

name = object

named argument

```
> ksvm(x,y)
> lm(data=dummy)
```

... argument

arguments to pass
to another function

How to write a function?

name keyword model formula input arguments '...' argument

```
fname <- function ( formula=, arg1=default_value1, arg2, . . . )
{
    [function body omitted ]
  optional keyword
    return (list_of_output_values)  ←  return/output list of objects
}
```

optional parameters to the function

Example:

```
> help(lm)
```

lm (**formula**, **data**, subset, weights, na.action, **method = "qr"**,
model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
singular.ok = TRUE, contrasts = NULL, offset,
...)

Exercise #7: hist() function

```
> help(hist)
```

```
hist(x, breaks = "Sturges",
     freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, warn.unused = TRUE, ...)
```

```
> hist(x=rnorm(50),breaks=data.frame(-3,1,3),freq=“yes”)
```

Will this work? And why? How will you fix it?

Oak Ridge National Laboratory

NC STATE UNIVERSITY
Department of Computer Science

<u>Fundamental Concept:</u>

Argument object's class in f() call must match!!!

Watch for each object's class…

```
> hist(x=rnorm(50),breaks=data.frame(-3,0,3),freq="yes")
```

`x` a vector of values for which the histogram is desired. class(rnorm(50))?

`breaks` one of:

- a vector giving the breakpoints between histogram cells,
- a function to compute the vector of breakpoints,
- a single number giving the number of cells for the histogram,
- a character string naming an algorithm to compute the number of cells (see 'Details'),
- a function to compute the number of cells.

In the last three cases the number is a suggestion only; the breakpoints will be set to pretty values. If `breaks` is a function, the `x` vector is supplied to it as the only argument.

`freq` logical; if `TRUE`, the histogram graphic is a representation of frequencies, the `counts` component of the result; if `FALSE`, probability densities, component `density`, are plotted (so that the histogram has a total area of one). Defaults to `TRUE` *if and only if* `breaks` are equidistant (and `probability` is not specified).

How to call a function? – by signature…

Function Signature:

```
fname ( formula=, arg1=, arg2=, . . . )
```

To check function's signature and usage:

```
> help(fname)
> args(fname)
> example(fname)
```

How to call a function:

```
> x <- rnorm(10)
> noise <- rnorm(10,mean=0, sd=0.1)
> y<- 2*x + noise
> my_data <- data.frame(x,y)
> help(lm)
> args(lm)
> lm(formula=y ~ x, data=my_data)
> z <- lm(y ~ x, my_data)
> summary(lm(y ~ x, data=my_data))
```

Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Function: Code vs. Description

Function Description:

```
help(function_name)

> help(lm)
> help(plot)
> example(plot)
```

- Signature
- Input arguments
- Default values for inputs
- Output values
- Examples

R Code for Function:

```
function_name

> mysum <- function (x1, x2)
{
    return (x1+x2)
}
> mysum
> plot
> lm
```

Cheat sheet for writing R functions

```
fname <- function ( formula=, arg1=default_value1, arg2, . . . )
{
    [function body omitted]
    return (list_of_output_values)
}
```

Annotated:

name keyword model formula input arguments '...' argument

```
fname <- function ( formula=, arg1=default_value1, arg2, . . . )
{
```

optional parameters to the function

```
    [function body omitted ]
```

keyword

```
    return (list_of_output_values)
```

return/output list of objects

```
}
```

Exercise: Step-by-step example

Step 1: From RStudio console: check your working directory with ***getwd()*** command
Step 2: Create/open a file "***functions.R***" in that working directory
Step 3: Write R code for a function with the following specs:

- name: **myprod**
- arguments: ***x*** and ***y***
- output value: ***product*** of x and y

Step 3: Load the function into environment from RStudio with ***source()*** command
Step 4: RStudio: Check that function is in the environment with ***ls()*** or ***objects()***
Step 5: Call ***myprod()*** function by passing values x=3 and y=5 as arguments
Step 6: See the code for ***myprod()*** function by typing ***myprod***

RStudio

```
> getwd()
> source("functions.R")
> ls()
> myprod(3,5)
> # see f() code
> myprod
```

functions.R

```
myprod <- function (x, y)
{
   product <- x * y
   return (product)
}
```


OAK RIDGE National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Exercise: Returning a *list* of *named* objs

Step 1: Open a file "***functions.R***" in the working directory (***getwd()***)
Step 3: Add another R code for a function with the following specs:

- name: **mymath**
- arguments: ***x*** and ***y***
- output values: ***product*** of x and y as well as ***sum*** of x and y as **named objs**

Step 3: Reload functions into environment from RStudio with ***source()*** command
Step 4: RStudio: Check that function is in the environment with ***ls()*** or ***objects()***
Step 5: Call ***mymath()*** function by passing values x=2 and y=6 as arguments and assign the return value to variable ***result***
Step 6: Check that function returns ***list*** object with ***class(result)***
Step 7: Output ***result***

functions.R

```
mymath <- function (x, y)
{
   multiplication <- x * y
   addition <- x + y
   output <- list(sum=addition,
          product = multiplication)
   return (output)
}
```

RStudio

```
getwd()
source("functions.R")
result<-mymath(2,6)
class(result)
result
```

Exercise: Accessing objs in the list

Step 1: Call ***mymath()*** function by passing values x=2 and y=6 as arguments and assign the return value to variable ***result***
Step 2: Output ***result***
Step 3: Output ***result$sum*** and ***result$product***
Step 4: Output ***result[1]*** and ***result[2]***
Step 5: Output ***result[[1]]*** and ***result[[2]]***
Step 6: Output ***result["sum"]*** and ***result["product"]***
Step 7: Output ***result['sum']*** and ***result['product']***
Step 8: Check the ***summary(result)***

- Have you observed many ways to access individual elements of the list object?
- Do you see how you can access objects by index and by its name in the list?
- Do you see the power of creating named objects: **name = object**?

```
list(sum=addition, product = multiplication)
```

name = obj_value

RStudio

```
result<-mymath(2,6)
result
result$sum
result$product
result[1]
result[2]
result[[1]]
result[[2]]
result["sum"]
result["product"]
result['sum']
result['product']
summary(result)
```

Exercise: *Named* arguments with *default* values

Step 1: Open a file "***functions.R***" in the working directory (***getwd()***)

Step 3: Add another R code for a function with the following specs:

- name: **mydiv**
- arguments: ***nominator*** and ***denominator*** w/ ***default values*** of 1
- output values: ***division*** of x and y as well as ***status***

Step 3: Make sure you understand the commands in RStudio

RStudio

```
# reload R f()'s
source("utils.R")
ls()
mydiv(8,2)
# swap named args
mydiv(denominator=2,
nominator=8)
# check division by 0
mydiv(5,0)
# default arg values
mydiv(denominator=2)
mydiv(nominator=5)
```

functions.R

```
mydiv <- function(nominator=1, denominator=1)
{
   div <- NA
   error <- "OK"
   if (denominator == 0)
      { error <- "Division by zero" }
   else { div <- nominator/denominator }
   output <- list(division=div, status = error)
   return (output)
}
```

Summary: R functions

- You call a function by its ***signature*** (note the indicators: (and) after f() name)
- When you call a function and pass object as input argument to the function:
  - class (object) = class (argument) ← types must match / be compatible
  - always check what types of objects function expects: help(func_name)
- ***Implicit coercion*** from one type to another may lead to undesirable results
- ***Explicit coercion*** should be used to change the mode of the object (as.*type*()):
  - watch for possible side effects
  - ***compatible coercion*** is often applied to objs before passing them to f()
- Because f() can only return ONE object, create a ***list of multiple objects*** as an output object to ***return*** from the function call (see *mymath()* example)
- Check the ***cheat sheet*** of how to write a function
- For functions with ***named arguments*** (name=default_value), the order of the objects passed to the function during the function call is not important as long as they are used with their names (see *mydiv()* example)
- Function with ***default value*** arguments can be called with fewer objects passed as inputs:
  - in this case, the default values will be used for ***missing input args***

R is a programming language that is…

- Object-oriented language
- Functional language
- Vectorized
- Generic language
- Dynamic

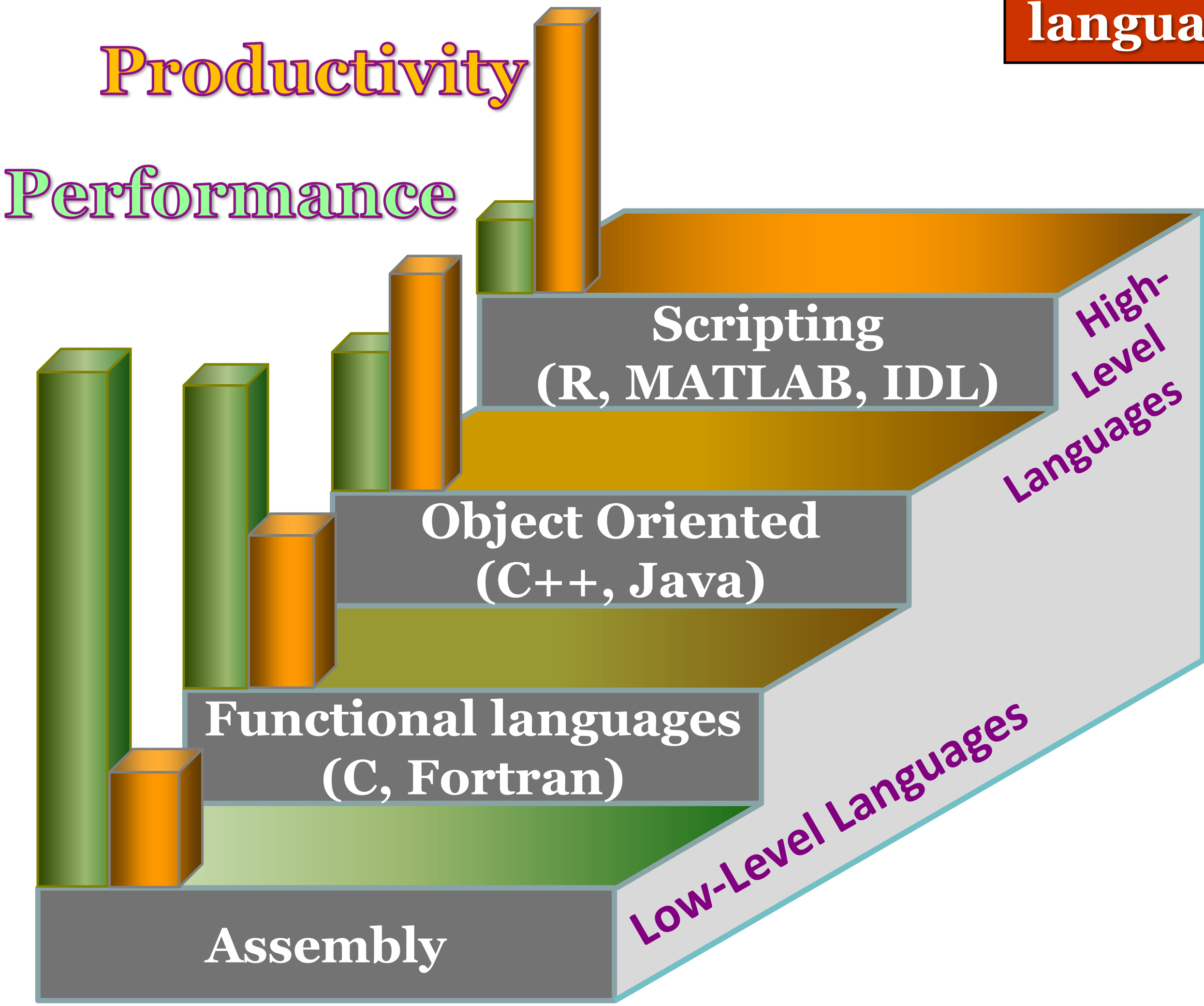
Read a nice blog (and some cited references at the end of the blog):
http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html

OAK RIDGE National Laboratory

NC STATE UNIVERSITY Department of Computer Science

The Programmer’s Dilemma

What programming language to use & why?



interpretable

```
> …
> dyn.load( “foo.so”)
> .C( “foobar” )
> dyn.unload( “foo.so” )
```

compiled

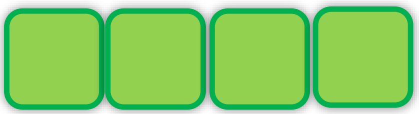
Key Points: Vectorization

- **R is an *interpretable* language:**
  - it interprets a lot of things at ***run-time*** rather than ***compile-time*** (e.g., type of a variable)
- **Vectors are *typed*; all their elements are of the <u>same</u> type**
  - once R figures out the type of the first vector element, it knows the type of all the other elements → ***vector operations*** require much less run-time work in terms of checking the type of each element
- **At the very very low-level, every object in R is represented in memory as a *vector* (even a scalar)**
  - applying vector operations will take advantage of such representation
- ***Pre-allocated* vs. re-allocated memory**
  - dynamic, run-time memory reallocation is expensive:
    - In RStudio: `obj <- c(); length(obj); obj[5] <- 7;  length(obj)`
  - pre-allocate memory for objects:
    - In RStudio: `obj <- rep(NA, 5);  length(obj); obj[5] <- 7;  length(obj)`
- Use ***apply()*** family rather than ***for-loops*** if possible

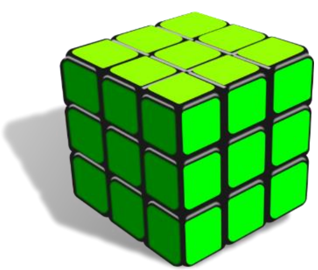
Goal: Repeat applying the same FUN() to multiple elements

Multi-element objects in R

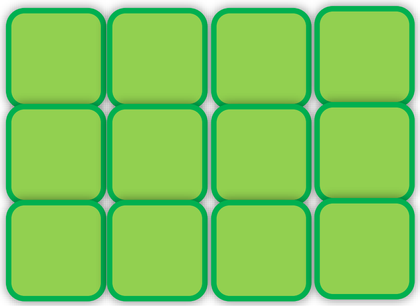
Vector [ ]



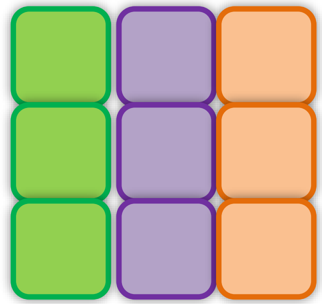
Array [, ,]



Matrix [,]



Data Frame [,]



List [[ ]]

- Vectors
- Arrays
- Lists
- Data Frame
- Matrices

ordered collection of objects

Example FUN():

- sum()
- mean()
- nchar()
- …

Example: Repeat applying the same FUN()=nchar() to each *vector* element

Vector [ ]

I	work	at	CISCO

What if *vv* is a vector of 1000's of words in a large document, such as book?

RStudio

```
vv <- c("I", "work", "at", "CISCO")
vv
# Option 1:
result <- c(nchar(vv[1]), nchar(vv[2]),
            nchar(vv[3]), nchar(vv[4]))

result

# Option 2: Use sapply()
sapply(vv, nchar)

# Option 3: Use vectorized function
nchar(vv)
```

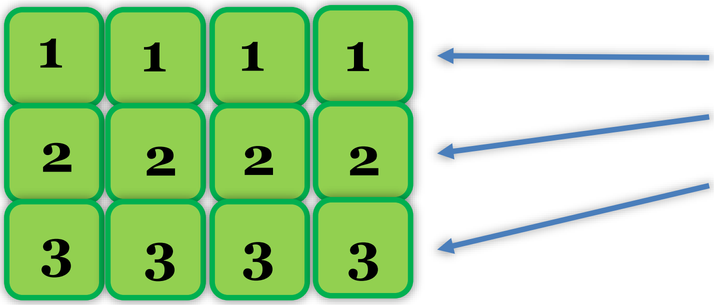
Which option is the easiest to use?

Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Example: Repeat applying the same FUN()=sum() to each *matrix* row

Matrix [,]



What if *AA* has 1000's of rows and 100'ss of columns?

Which option is better?

RStudio

```
v1 <- rep(1,4)
v2 <- rep(2,4)
v3 <- rep(3,4)
v3
AA <- rbind(v1,v2,v3)
help(rbind)
AA
AA[1,]

# Option 1:
result <- c(sum(AA[1,]),
       sum(AA[2,]), sum(AA[3,]))
result

# Option 2:
apply(AA, 1, sum) # by row: 1
apply(AA, 2, sum) # by column: 2
```

Oak Ridge National Laboratory

Example: Closer look into *apply()*

Matrix [,]

1	1	1	1	4
2	2	2	2	8
3	3	3	3	12
6	6	6	6	

RStudio

```
apply(AA, 1, sum) # 1: by row
apply(AA, 2, sum) # 2: by column
```

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

Example: Passing f() with multiple arguments to *apply()*

AA: Matrix [,]

1	1	1	1
2	2	2	2
3	3	3	3

b: Vector [ ]

4
5
6
7

functions.R

```
inner_product <- function(rowvec, colvec)
{
   product <- rowvec * colvec
   inner <- sum(product)
   return (inner)
}
```

RStudio

```
b <- c(4,5,6,7)
apply(AA, 1, inner_product, b) # 1: by row
```

Summary of some apply()-type functions

Function	Action	Output
apply(A, MARGIN, FUN, ...)	Apply function FUN() to each dimension in MARGIN of **array** or **matrix** A; args for FUN() are passed as part of **... argument**	**vector, array**, or **list** of values returned by FUN()
lapply(L, FUN, ...)	Apply function FUN() to each element of **vector** or **list** L; args for FUN() are passed as part of **... argument**	**list** of values returned by FUN()
sapply(L, FUN, ...)	= **lapply**()	**vector, matrix,** or **array** of values returned by FUN()

OAK RIDGE National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Exercise: apply() function

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

functions.R

```
matrixRowSums <- function(data)
{
   if (is.matrix(data) == TRUE) {
        margin = 1 # summing by rows
        sums <- apply (data, margin, sum)
        return (sums)
   }
   else {
        print("ERROR: argument must be matrix")
        return (NA)
   }
}
```

1. Create function matrixRowSums() in functions.R

RStudio

```
# reload R f()'s
source("functions.R")
ls()
mm <- matrix(c(1:8),nrow=2)
mm
matrixRowSums(mm)
vv <- c(1:8)
matrixRowSums(vv)
```

2. Create 2 by 4 matrix *mm* and call matrixRowSums(*mm*) from RStudio

OAK RIDGE National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Exercise: apply() function

Driver code

mrowsumDriver.R

```
source("functions.R")
df <- data.frame(c(1:4),c(5:8))
class(df)
mode(df)
mm <- as.matrix(df) # coerce to matrix
class(mm)
mode(mm)
dim(mm)
mm
matrixRowSums(mm)
```

1. Create data.frame of numeric columns
2. Coerce it to matrix of size 4 by 2
3. Get sums by its rows

Exercise: Matrix column means using apply() and test driver code

Ex: Create a function matrixColumnMeans() in functions.R and a driver code mcolmeansDriver.R that demonstrates how to find the means of matrix *columns* using apply().

Exercise: lapply() function

lapply(L, FUN, ...)

RStudio

```
vv <- c(1:10)
beta <- exp(-3:3)
ll <- c(TRUE, FALSE, FALSE)
x <- list(v=vv, b=beta, l=ll)
x
class(x)
lapply(x, mean)
```

1. Create a list of 3 objs
2. Find mean for each obj in the list

<u>Fundamental Concept:</u>

Avoid loops;
consider apply() family of functions!!!

Exercise: Avoid for-loops in lieu of apply()

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

functions.R ← GREAT!

```
matrixRowSums <- function(data)
{
   if (is.matrix(data) == TRUE) {
        margin = 1 # summing by rows
        sums <- apply (data, margin, sum)
        return (sums)
   }
   else return (NULL)
}
```

BAD

bad.R

for (name in expr_1) { expr_2 }

```
avoidLoops <- function(data)
{
   if (is.matrix(data) == TRUE) {
     sums <- c()
     for ( rindex in 1:nrow(data) ) {
        rowSum <- sum(data[rindex, ])
         sums <- c(sums, rowSum)
     }
     return (sums)
   }
   else return (NULL)
}
```

although you get the same answer, avoid loops!!

Quiz Question

The lapply function takes the same arguments as the apply function.

This function must be passed a matrix object, an indicator for row or column, and a function that will be applied.

This function works like the apply function but operate on each elements of a list.

The sapply function returns a

The apply functions could be just as efficiently performed via loops in R.

This package contains other a superset of apply functions for many different combinations of input objects and returned objects (though perhaps a little slower than the apply functions).

TRUE

FALSE

apply()

lapply() and sapply()

vector

plyr()

R as a vectorized language: Key Concepts Summary

Avoid loops, use vectorised operations!

```
vec = vec + 1
```

vs.

```
for (i in 1:length(vec)) vec[i] = vec[i] + 1
```

Use apply() family!

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

Pre-allocate memory!

```
obj <- rep(NA, 5)
length(obj)
```

Place expensive code into compiled!

```
> ...
> dyn.load( “foo.so”)
> .C( “foobar” )
> dyn.unload( “foo.so” )
```

Fundamental Concepts: Summary I

- **R is an *object-oriented* language:**
  - data structures, functions and expressions are objects; objects have many attributes; change the object mode--coercion--can be dangerous; watch for implicit coercion; homogeneous mode data structures: vector, matrix, array; heterogeneous mode data structure: data frame, list; using different types of data in vector, matrix, or array will result in implicit coercion.
- **R is a *functional* language:**
  - class type of argument objects must match between the caller and the callee--implicit coercion may be dangerous; named arguments can be passed in any order; arguments with default values might be handy (e.g., plot()); only one object can be returned--use list objects to return more; always check against f() documentation for expected types of function arguments, returned values, default argument values (help(fname)).
- **R is a *vectorized* language:**
  - avoid loops; consider apply() family; pre-allocate memory; place compute-intensive code into compiled code.

Fundamental Concepts: Summary II

- **R is an *interpretable* language**:
  - minimize what R has to figure out at run-time by calling functions rather than writing lots of R script, by placing expensive code into third-party compiled code, and by explicitly using vector operations.
- **R is an *open source, community* effort:**
  - packages could be of different quality, maintainability; finding the right package for the task at hand is time-consuming; create demo() functions for packages you release.
- **R is a *memory-intensive* programming environment:**
  - all objects are stored in memory (check object.size(obj) for memory usage)
  - manage environment wisely (check with ls() and remove unused objects: rm(obj))
  - matrix is stored in a column-major way → accessing elements by rows is inefficient due to lots of memory hops