

www.dilbert.com scottadams@aol.com



8/14/00 © 2000 United Feature Syndicate, Inc.



DCMB BioComputing BootCamp

Day 3, Session II:

R Control Structures and Functions

Armand Bankhead

bankhead@umich.edu

8/22/2018



Overview

1. Working Directory
2. Reading and Writing Data in R
3. Factors
4. Using Indexes
5. Merging Data Frames
6. Functions
7. Program Control Structures

Use Directory Structure to Organize Your Work

- Imagine if all of your input data, code, output data, images files were all in the same directory!
- Sounds like a mess!
- There is no perfect/standard directory structure, but here are a few suggestions:
 1. Create a unique directory for each project
 2. Break your code up into parts
 3. Write output (i.e. tables, images) to a separate sub-directory



The R Working Directory

- R executes commands from a ‘working directory’
 - scripts, input files, output files
 - absolute and relative directories may be specified
- Use `getwd()` to display current working directory

```
> getwd()
```

```
[1] "C:/Users/bankhead/Documents"
```

- Use `setwd()` to change your working directory

```
> setwd("../Desktop/armandsFolder")
```

- Use `dir()` to list files and folders in your working directory

Exercise: Create a directory/folder outside of R and use the `setwd()` function to navigate to that directory

Writing Data To Text Files

- First lets create something to write as a text file
 - > `m4 = matrix(1:300,nrow=100,ncol=3)`
 - > `colnames(m4) = c('A','B','C')`
 - > `m4 = data.frame(m4,D = c(rep('X',50),rep('Y',50)))`
 - Use `write.table()` to write data to a file
 - Many arguments! Use `?write.table` to find out more
- ```
> write.table(m4, 'myData.txt', quote=F, row.names=F,
sep="\t")
> dir()
[1] "myData.txt"
```







# Using Indexes

- Indexing is a powerful tool for filtering large data frames or matrices
- There are two central ways to index:

1. Logical vectors:

```
> m5$A < 10
```

```
[1] TRUE TRUE TRUE
```

2. Integer vectors:

```
> which(m5$A < 10)
```

```
[1] 1 2 3 4 5 6 7 8 9
```

```
> m5$A < 10
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[20] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[39] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[58] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[77] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[96] FALSE FALSE FALSE FALSE FALSE
> which(m5$A < 10)
[1] 1 2 3 4 5 6 7 8 9
```

# Using Indexes

- We can combine multiple conditions use the &, |, and parens

```
> m5$A < 10 & m5$B > 205
> m5$A < 10 & m5$B > 205 | m5$D == 'Y'
> m5$A < 10 | m5$D == 'Y' & m5$A < 55
```
- Be aware of operator precedence
- Use the sum command to count how many positive values survive
- Indexes can be used to index vectors, matrices, or data frames

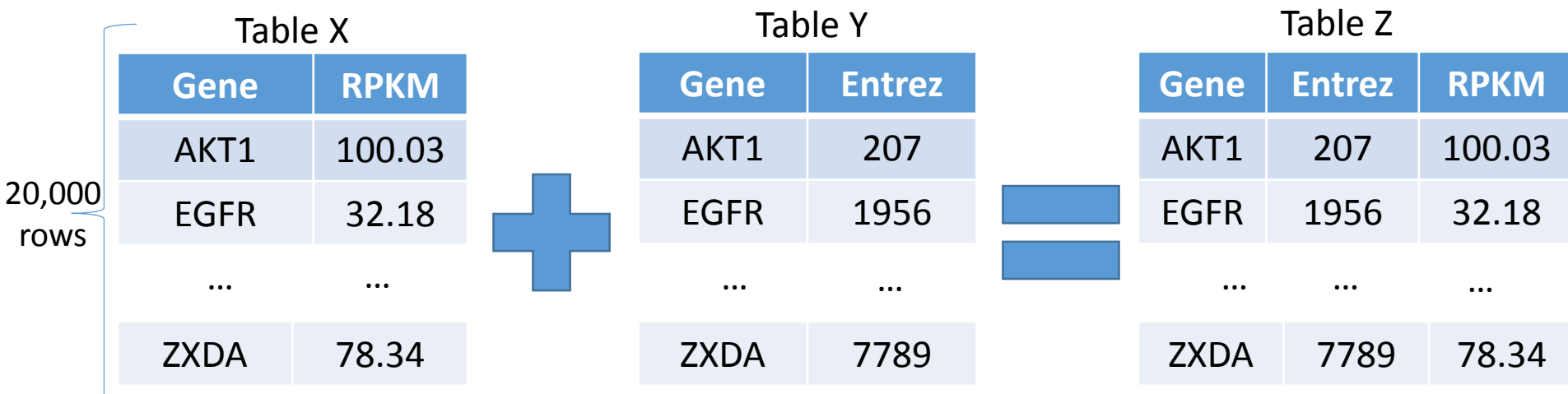
```
> idx = m5$A < 10 & m5$B > 205
> subMatrix = m5[idx,]
```

Operator Precedence in R

| Operator                                      | Description              | Associativity |
|-----------------------------------------------|--------------------------|---------------|
| <code>^</code>                                | Exponent                 | Right to Left |
| <code>-x, +x</code>                           | Unary minus, Unary plus  | Left to Right |
| <code>%%</code>                               | Modulus                  | Left to Right |
| <code>*, /</code>                             | Multiplication, Division | Left to Right |
| <code>+, -</code>                             | Addition, Subtraction    | Left to Right |
| <code>&lt;, &gt;, &lt;=, &gt;=, ==, !=</code> | Comparisons              | Left to Right |
| <code>!</code>                                | Logical NOT              | Left to Right |
| <code>&amp;, &amp;&amp;</code>                | Logical AND              | Left to Right |
| <code> ,   </code>                            | Logical OR               | Left to Right |
| <code>-&gt;, -&gt;&gt;</code>                 | Rightward assignment     | Left to Right |
| <code>&lt;-, &lt;&lt;-</code>                 | Leftward assignment      | Right to Left |
| <code>=</code>                                | Leftward assignment      | Right to Left |

# Merging Data Frames

- A common programming task in bioinformatics is to “join” two tables:



- Joins are performed in R using the `merge()` function
  - Requires a common column between tables (e.g. gene) be specified using the “by” parameter
  - Multiple types of joins (e.g. inner, outer) are possible, use `?merge` to find out more

# Merging Data Frames

- Use `merge()` to join together data from two different matrices or data frames
- **Important: both matrices must contain unique row identifiers to join on!**

```
> df1 = data.frame(gene = c('AKT1','ERBB2','EGFR'),
log2rpkm = c(5,.5,10))
```

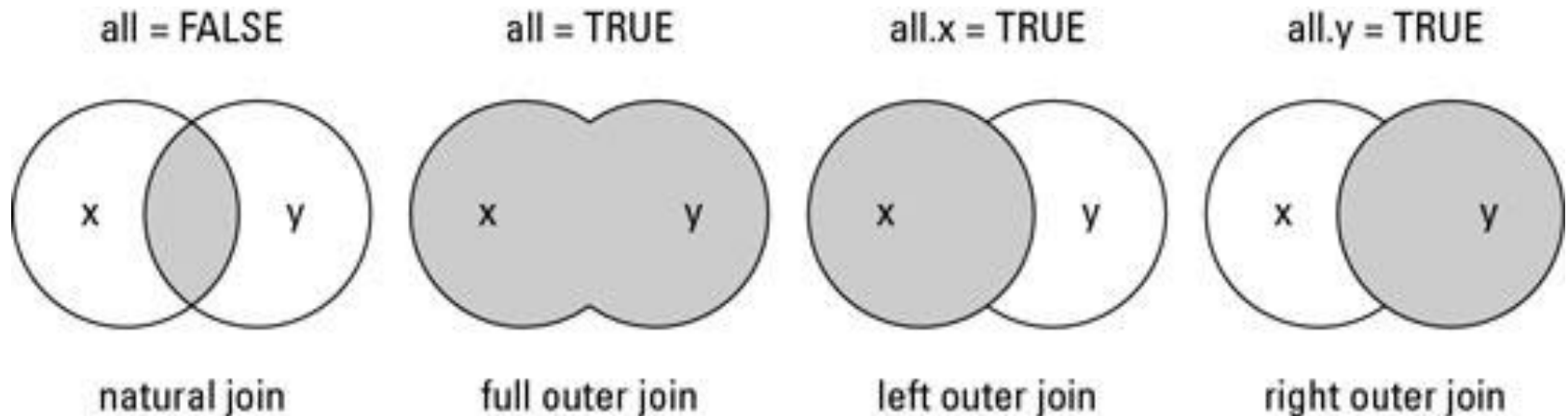
```
> df2 = data.frame(gene = c('AKT1','ERBB2','EGFR'), entrez
= c(207,2064,1956))
```

```
> combined = merge(df1,df2,by='gene')
```

**Exercise: What happens if gene is not unique?**

**Add a duplicate gene name and find out.**

# Merging Data Frames: Types of Joins



- natural join: intersection of common rows
- full outer join: union of rows
- left outer join: all x rows represented
- right outer join: all y rows represented

# Functions

- Functions allow us to break our R scripts up into modular pieces
- Modular program design has already been discussed in unix (day1) and python (day2)
- Benefits to our code include:
  - Program design
  - Readability
  - Re-use
  - Trouble-shooting
- Functions are specified using the 'function' key word

```
myFunction = function(arg1,arg2,...) {
 statements
}
```

- When multiple arguments are specified R will match first by name, prefix matching arguments, then by position

# Functions

```
> sq1 = function(x) return(x * x)
> sq1 = function(x) x * x

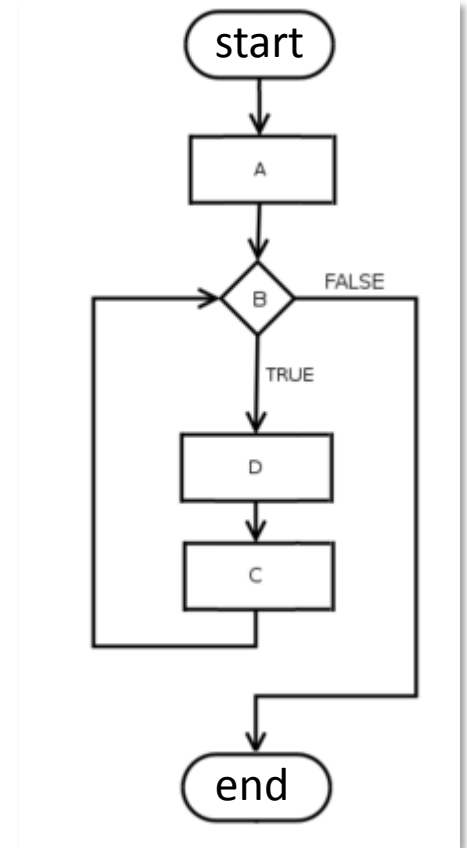
> randomValues = rnorm(1000)
> randomValuesSquared = sq1(randomValues)

> myAnalysis = function(randomValues, randomValuesSquared) {
 vector1 = randomValues
 vector2 = randomValuesSquared
 result = cor(vector1,vector2)
}
```

**Exercise:** Create an R script contains the myAnalysis function  
What correlation value is generated?

# Program Control Structures

- R program flow is not always a linear sequence of operations
- Besides functions, program flow may be modified using control structures
  1. apply
  2. if/else/else if
  3. for
  4. while
- Other important R commands that can alter program flow:
  1. break – exit loop
  2. next – skip to the next iteration



program flowchart



# Program Control Structures: apply

- Use the apply function to iterate through a data.frame, matrix, or arrays
  - Use lapply() to iterate through lists or vectors
- apply function takes at least 3 values
  1. data frame/matrix
  2. 1 or 2 indicating rows or columns respectively
  3. function to 'apply' to each value (standard or custom)

```
> m4 = matrix(1:300,nrow=100,ncol=3)
```

```
> rowMeans = apply(m4,1,mean)
```

```
> columnMeans = apply(m4,2,mean)
```

Exercise: Run the code above.

What data structures are rowMeans and columnMeans?

Are rows are average larger or columns?

# Program Control Structures: if/else/else if

- 'if' statements allow us to condition our program flow
- basic syntax:

```
if(condition) {
 statement1
}
else if(condition){
 statement2
}
else {
 statement3
}
```

- conditions must be TRUE or FALSE
- statements are a series of R commands

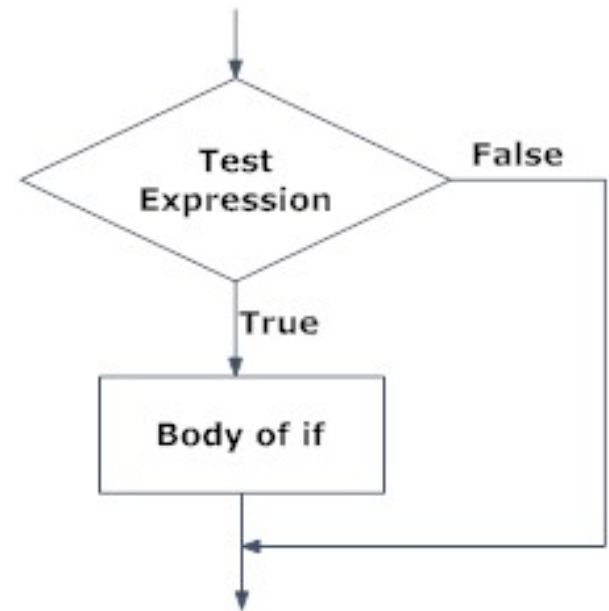


Fig: Operation of if statement

# Program Control Structures: if/else/else if

- Example if statement

```
m5 = read.delim('myData.txt')
if(ncol(m5) == 4 && is.factor(m5$D)) {
 print('factors!')
}
else {
 print('no factors!')
}
```

- Multiple conditions can be combined using:
  - || OR
  - && AND
  - ! NOT
  - () parens

# Program Control Structures: for

- ‘for’ loops allow us to iterate our code
- Basic syntax:

```
for(counter in vector) {
 statements
}
```

- ‘vector’ can represent a list of numbers (e.g. 1:10) or arbitrary data types (e.g. c(‘mon’, ‘tues’, ‘wed’, ...))

# Program Control Structures: for

- Example #1:

```
for(i in 1:5) {
 print(i)
}
```

- Example #2:

```
m5 = read.delim('myData.txt')
for(column in 1:ncol(m5)) {
 print(mean(m5[,column]))
}
```

- 'break' can be used to exit loop structure
- We could have used apply!

# Program Control Structures:

## while

- while() loops allow iteration until a condition is met
- Basic syntax:

```
while(condition) {
 statements
}
```

To exit the loop structure

- 'break' can be used to exit loop structure
- set condition to be false

# Exercises

1. Write a function that converts Fahrenheit to Celsius
  - input: temperature in Fahrenheit
  - output: temperature in Celsius
2. Write a program that builds a data frame containing degrees Fahrenheit to Celsius for -30 to 130 degrees Fahrenheit
3. Write the table from #2 to a tab-delimited text file

# Bonus Exercise (if time!)

- The `unique()` function can be used to get the unique values in a vector
- The CO2 data set contains 84 measurements from an experiment comparing the CO2 uptake of *Echinochloa crus-galli* sourced from Quebec and Mississippi. Plants were measured chilled and nonchilled.
- Using the CO2 data set determine if the average expression of chilled plants from Quebec is higher than plants from Mississippi.



# Closing Remarks/Advice

- Comment your code
- Short programs are better
- Plan!
- Be prepared to iterate
  - Make a change
  - Run
  - Make another change
  - Run
  - ...

# References

- Gentleman, Robert. R Programming for Bioinformatics. CRC Press, 2009.
- Slides Partially Sourced from Barry Grant and Hui Jiang