

# Advanced data structures, I/O & Control Structures

David Li

# Matrix

- There are several ways to make a matrix
- To make a 2x3 (2 rows, 3 columns) matrix of 0's:

```
> mat<-matrix(0,2,3)
```

- To make the following matrix:

71	172
73	169
69	160
65	130

```
> mat2<-rbind(c(71,172),c(73,169),c(69,160),c(65,130))
```

```
> mat3<-cbind(c(71,73,69,65),c(172,169,160,130))
```

- To make the following matrix:

```
– mat4<-matrix(1:10,2,5, byrow=T)
```

1	2	3	4	5
6	7	8	9	10

# Revisit vectors: access data

- Accessing individual observations  
    `>whales[2]`
- Slicing  
    `>whales[2:5]`
- Negative indices  
    `>whales[-1]`
- Logical values  
    `>whales[whales > 100]`  
    `>which(whales > 100)`  
    `>which.max(whales)`

# Indexing of vector/matrix

- `x=1:10`

`ith` element

all *but* `ith` element

first `k` elements

specific elements.

all greater than some value

bigger than or less than some values

`x[2]` ( $i = 2$ )

`x[-2]` ( $i = 2$ )

`x[1:5]` ( $k = 5$ )

`x[c(1,3,5)]` (First, 3rd and 5th)

`x[x>3]` (the value is 3)

`x[ x< -2 | x > 2]`

□ `mat=matrix(1:24, nrow=4)`

`mat[,2]` # 2<sup>nd</sup> column

`mat[2,]` # 2<sup>nd</sup> row

`mat[c(2,4),]` # 2<sup>nd</sup> and 4<sup>th</sup> row

`mat[1:3,1]` # 1 to 3 element in column 1

`mat[-c(2,4),]` # all but row 2 and 4

# Create logical vectors by conditions

- Logical operators: `<`, `<=`, `>`, `>=`, `==`, `!=`
- Comparisons
  - Vectors: `AND &`; `OR |`
  - Longer forms `&&`, `||`: return a single value
  - `all()` and `any()`
- Examples
  - `X=1:5`
  - `X<5; X>1`
  - `X > 1 & X < 5; X > 1 | X < 5;`
  - `all(X<5); any(X>1); all(X<5) && any(X>1)`
- `%in%` operator: `x %in% c(2,4)`

# Missing values

- R codes missing values as NA
- `is.na(x)` is a logical function that assigns a T to all values that are NA and F otherwise

```
>x[is.na(x)]<-0
```

```
>mean(x, na.rm=TRUE)
```

# Reading in other sources of data

- Use R's built-in libraries and data sets
  - > range(lynx) #lynx is a built-in dataset
  - > library(MASS) # load a library
  - > data(survey) # load a dataset in the library
  - > data(survey, package="MASS") #load just data
  - > head(survey)
  - > tail(survey)
- Copy and paste by scan()
  - > whales=scan()
  - 1: 74 122 235 111 292 111 211 133 156 79
  - 11:
  - Read 10 items

# Read formatted data

- Read data from formatted data files, e.g. a file of numbers from a single file, a table of numbers separated by space, comma, tab etc, with or without header

```
> whale=scan(file="whale.txt")
```

```
"whale.txt" :
```

```
74 122 235 111 292 111 211 133 156 79
```

```
> whale=read.table(file="whale.txt", header=TRUE)
```

```
"whale.txt" :
```

```
    texas florida
```

```
1  74    89
```

```
2 122   254
```

```
3 ....   ....
```

```
> read.table(file=file.choose()) # specify the file
```

```
> read.table(file="http://statweb.stanford.edu/~rag/stat141/exs/whale.txt",head  
er=T)  # read from internet
```



# Data frame

- A “data matrix” or a “data set”
  - it looks like a matrix (rectangular grid)
  - But unlike matrix, different columns can be of different types
  - Row names have to be unique
- `> alphabet <- data.frame(index=1:26, symbol=LETTERS)`
- `read.table()` stores data in a *data frame*
- Access var in a dataset: `$`, `attach()`, `with()`
  - `> library(ISwR) #load the package that provides thuesen data`
  - `> data(thuesen)`
  - `> names(thuesen) #variable names`
  - `> blood.glucose # not visible`
  - `> length(thuesen$blood.glucose)`
  - `> with(thuesen, range(blood.glucose))`

```
> attach(thuesen)
> range(blood.glucose)
> detach(thuesen)
```

# More about data frame

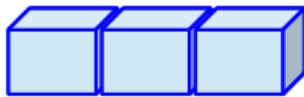
- Indexing of data frames is the same as that of vector and matrix `>energy[energy$stature== "lean",]`
- Sorting rows by `order()`
  - `>energy[order(energy$expend),]`
  - `>energy[with(energy, order(stature, expend)),]`
- Selecting subsets of data by `subset()`
  - `>subset(energy, stature=="lean" & expend>8)`
- Splitting data
  - `>split(energy$expend, energy$stature)`

# Lists

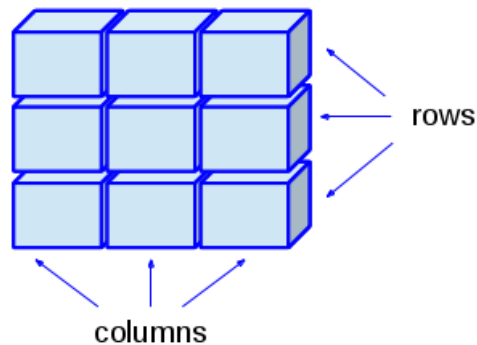
- A larger composite object for combining a collection of objects
    - Different from data frame, each object can be of different length, in addition to being of different types
- ```
>a=list(whales=c(74,122,235,111,292,111,211,133,16,79),  
simpsons=c("Homer", "Marge", "Bart", "Lisa", "Maggie"))
```
- Access by `$` or `[[ ]]`: `a$simpsons` or `a[[2]]`

# Summary of data structures

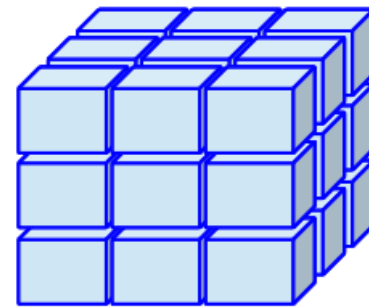
Vector



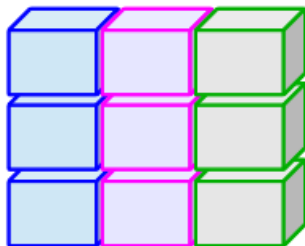
Matrix



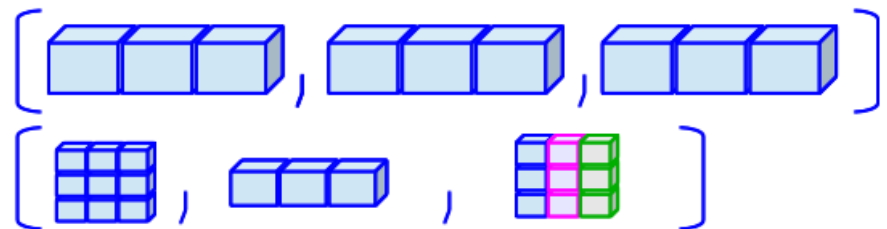
Array



Data Frame  
(Table)



Lists



# Manage the work environment

- What if there are more variables defined than can be remembered?
- `ls()` list all the objects(var, fun, etc) in a given environment
- `rm(a, b)`: delete variables a and b
  - `rm(list=ls())` will ?
- Get and set working directory
  - > `getwd()`
  - > `setwd("working/directory/path")`
- Save and load working environment
  - > `save.image(file="filename.RData")`
  - > `load(file="filename.RData")`

# scripting

- Edit your commands using your favorite text editors
- How to run

Inside R: >**source**(filename)

- Takes the input and runs them
- Do syntax-check before anything is executed
- Set echo=T to print executed commands

OR copy & paste

Outside R: **R CMD BATCH** filename

output is in \*.Rout

Or: **Rscript** filename

# How to install packages

- To install CRAN packages, execute from the R console the following command:

```
> install.packages( 'UsingR' )
```

OR download the package and install it directly

```
R CMD INSTALL aPackage_0.1.tar.gz
```

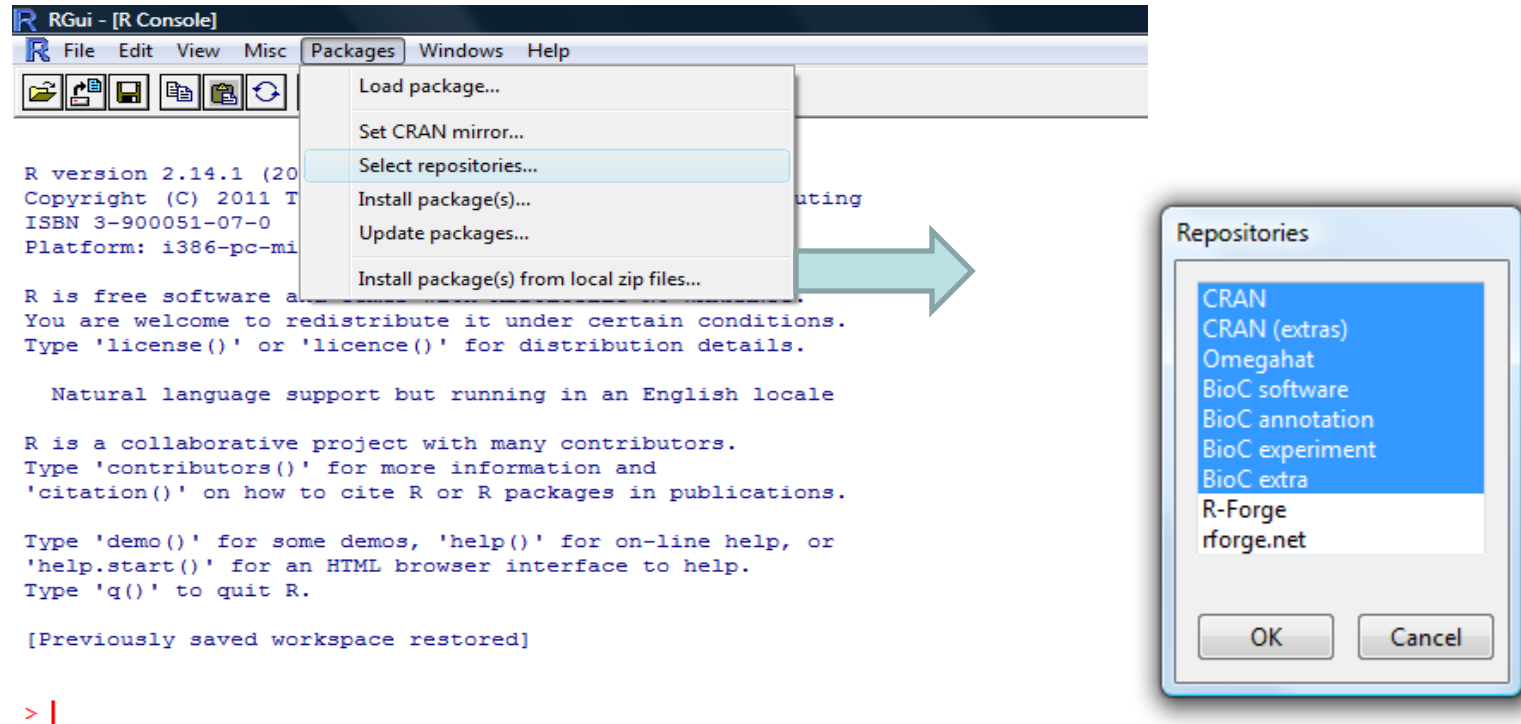
- Load a library

```
> library("UsingR")
```

or

```
> library(UsingR)
```

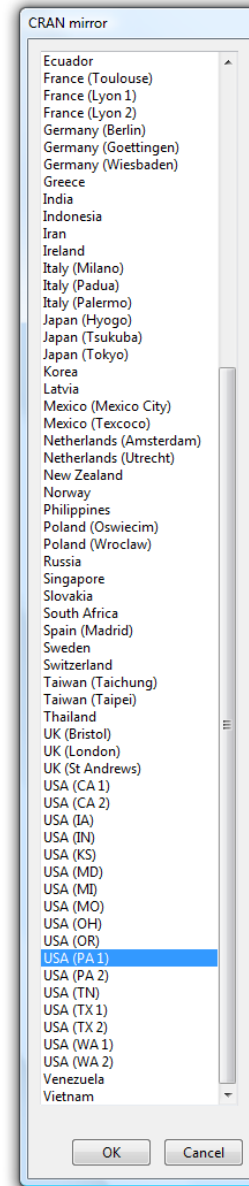
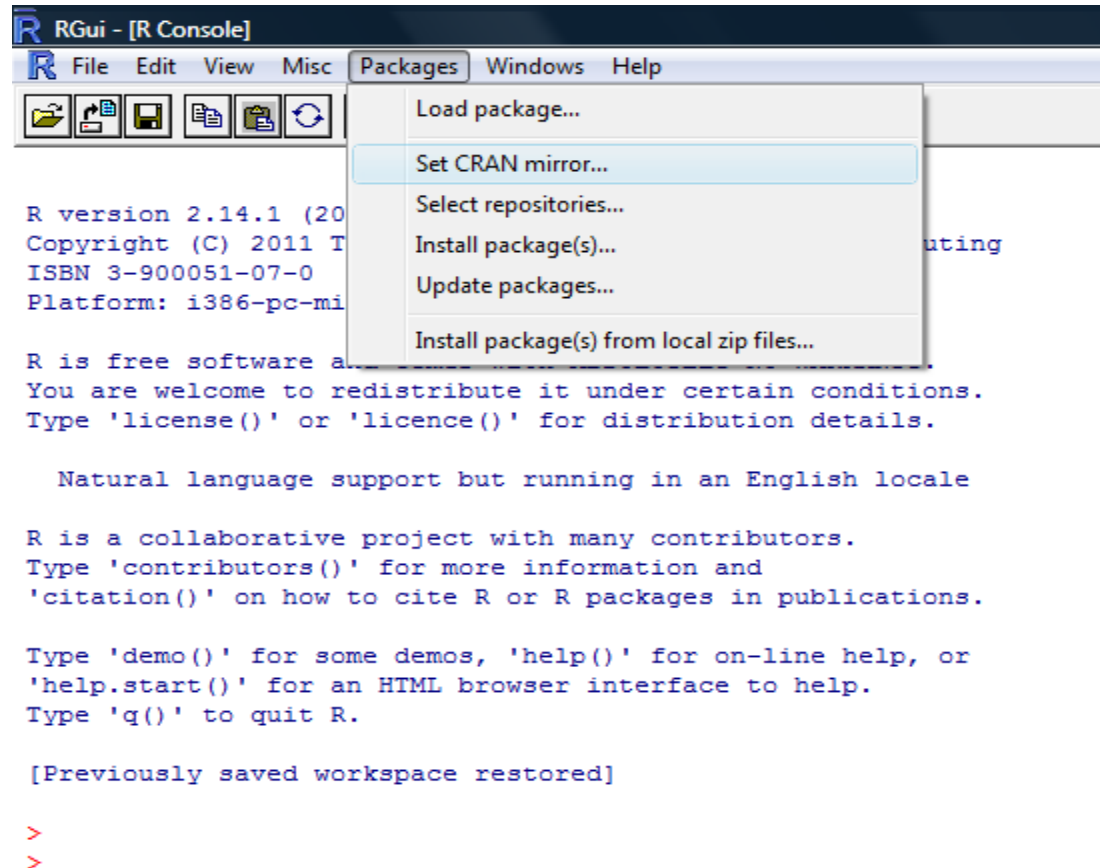
# Windows: Set repositories



- Make sure you include necessary repositories (you may simply select all of them)

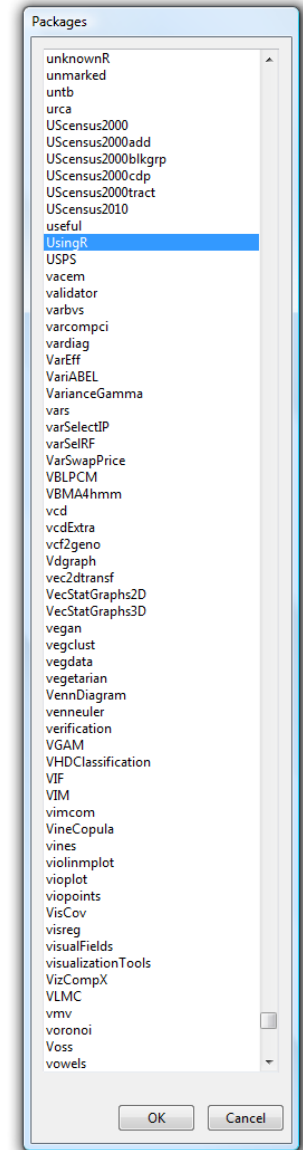
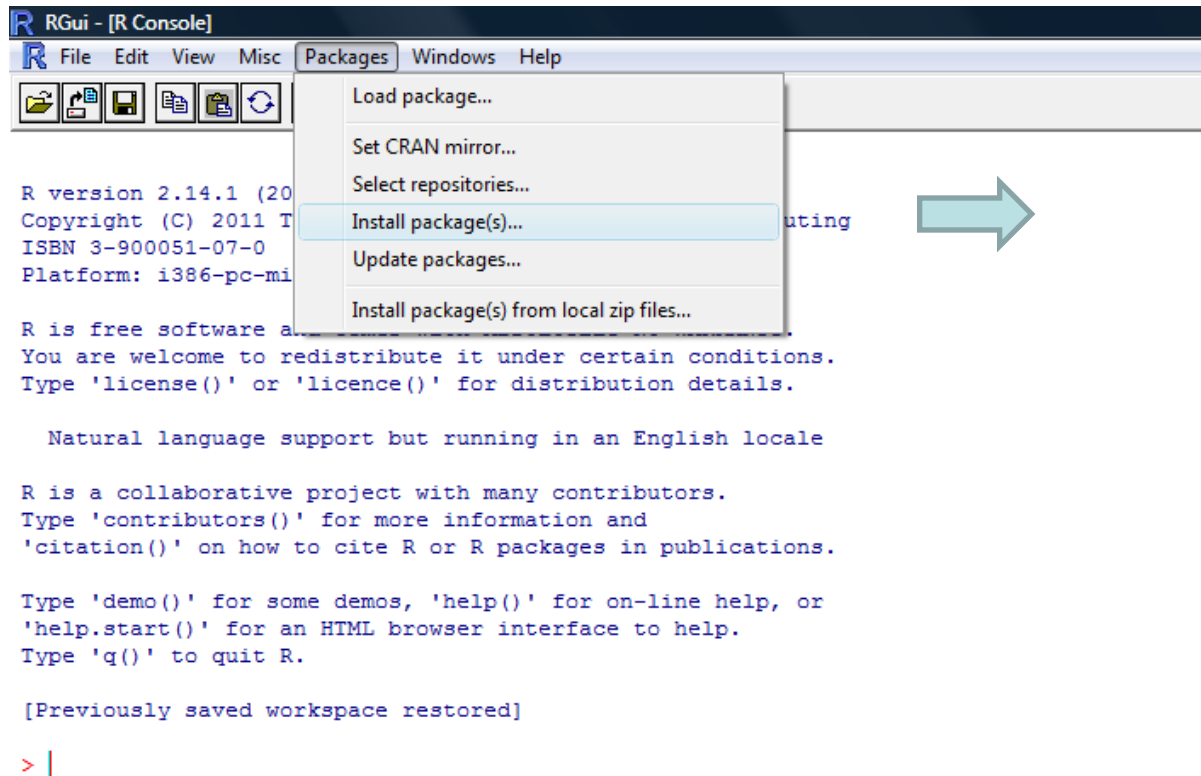


# Windows: Set CRAN mirror



- You can choose anyone but physically close ones are preferred

# Windows: install packages



# Additional references

- Beginners should print out the [R Reference Card](http://cran.r-project.org/doc/contrib/Short-refcard.pdf)  
<http://cran.r-project.org/doc/contrib/Short-refcard.pdf>
- [The R-FAQ](http://cran.r-project.org/doc/FAQ/R-FAQ.html) (Frequently Asked Questions on R)  
<http://cran.r-project.org/doc/FAQ/R-FAQ.html>
- A rather terse [introduction to R](http://cran.r-project.org/doc/manuals/R-intro.html) online  
<http://cran.r-project.org/doc/manuals/R-intro.html>
- [Bioconductor](http://www.bioconductor.org/)  
<http://www.bioconductor.org/>
- A useful online manual for R & Bioconductor  
[http://manuals.bioinformatics.ucr.edu/home/R\\_BioCondManual](http://manuals.bioinformatics.ucr.edu/home/R_BioCondManual)

# Acknowledgments

- Jeff Solka: for some of the slides adapted or modified from his lecture slides at George Mason University
- Brian Healy: for some of the slides adapted or modified from his lecture slides at Harvard University

# Getting Data In and Out of R

# Principal functions reading data

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (inverse of `dump`)
- `dget`, for reading in R code files (inverse of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

# Principal functions writing data

- `write.table`, for writing tabular data to text files (i.e. CSV) or connections
- `writeLines`, for writing character data line-by-line to a file or connection
- `dump`, for dumping a textual representation of multiple R objects
- `dput`, for outputting a textual representation of an R object
- `save`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- `serialize`, for converting an R object into a binary format for outputting to a connection (or file).

# Video

- [https://youtu.be/Z\\_dc\\_FADyi4](https://youtu.be/Z_dc_FADyi4)

- ?read.table

```
read.table(file="http://statweb.stanford.edu/~rag/stat141/exs/whale.txt"),header=T) # read from internet
```

- ?readLines

```
readLines("http://statweb.stanford.edu/~rag/stat141/exs/whale.txt")
```



# Using dput() and dump()

- dput()/dget()
  - `y <- data.frame(a = 1, b = "a")`
  - `dput(y)`
  - `dput(y, file = "y.R")`
- dump()/source()
  - `x <- "foo"; y <- data.frame(a = 1L, b = "a")`
  - `dump(c("x", "y"), file = "data.R")`
  - `rm(x, y)`
  - `source("data.R")`
  - `str(y)`

# Difference between dput() and dump()

- dump can be used to output multiple objects
- dump adds the object name and can be source()'d

# Binary Formats `save()/load()`

- `a <- data.frame(x = rnorm(100), y = runif(100))`
- `b <- c(3, 4.4, 1 / 3)`
- *## Save 'a' and 'b' to a file*
- `save(a, b, file = "mydata.rda")`
- *## Load 'a' and 'b' into your workspace*
- `load("mydata.rda")`
- *## Save everything to a file*
- `save.image(file = "mydata.RData")`
- *## load all objects in this file*
- `load("mydata.RData")`

# Reading in Larger Datasets with read.table

- Video <https://youtu.be/BJYYIJO3UFI>
- A tip
  - > initial <- read.table("datatable.txt", nrows = 100)
  - > classes <- **sapply**(initial, **class**)
  - > tabAll <- read.table("datatable.txt", colClasses = classes)

# Summary

- `write.csv()` and `write.table()` are used when you want to exchange data in tabular text format.
- `dput()` saves single data object in R code
- `dump()` saves multiple data objects and their metadata in R code
- `save()` is similar to `dump()` but saves in binary format or ASCII
- `save.image()` saves workspace in binary format

# Calculating Memory Requirements for R Objects

- An example: a data frame with 1,500,000 rows and 120 columns, all of which are numeric data.
  - $1,500,000 \times 120 \times 8$  bytes/numeric

# Control Structures

# Commonly used control structures

- **if** and **else**: testing a condition and acting on it
- **for**: execute a loop a fixed number of times
- **while**: execute a loop *while* a condition is true
- **repeat**: execute an infinite loop (must break out of it to stop)
- **break**: break the execution of a loop
- **next**: skip an iteration of a loop



# if-else

- **if**(<condition>) {  
    *## do something*  
}
- ## Continue with rest of code*
- **if**(<condition>) {  
    *## do something*  
}
- else** {  
        *## do something else*  
    }

# if-else {if-else}

```
if(<condition1>) {  
    ## do something  
} else if(<condition2>) {  
    ## do something different  
} else {  
    ## do something different  
}  
#-----  
if(<condition1>) {  
}  
if(<condition2>) {  
}
```

# Example

```
x <- runif(1, 0, 10)
```

- **if**(x > 3) {  
    y <- 10  
} **else** {  
    y <- 0  
}
- y <- **if**(x > 3) {  
    10  
} **else** {  
    0  
}
- y <- ifelse(x>3, 10, 0)

# ifelse()

- `x <- c(6:-4)`
- `sqrt(x)` #- gives warning
- `sqrt(ifelse(x >= 0, x, NA))` # no warning
- ## Note: the following also gives the warning !
- `ifelse(x >= 0, sqrt(x), NA)`

- ## example of different return modes:

```
yes <- 1:3
```

```
no <- pi^(0:3)
```

```
typeof(ifelse(NA, yes, no)) # logical
```

```
typeof(ifelse(TRUE, yes, no)) # integer
```

```
typeof(ifelse(FALSE, yes, no)) # double
```

# for Loops

- **for**(i in 1:10) {  
  **print**(i)  
}
- x <- **c**("a", "b", "c", "d")
- **for**(i in 1:4) {  
  *## Print out each element of 'x'*  
  **print**(x[i])  
}

# for Loops (cont' d)

- `seq_along()` function is commonly used in conjunction with for loops

```
for(i in seq_along(x)) {  
    print(x[i])  
}
```

- It is not necessary to use an index-type variable

```
for(letter in x) {  
    print(letter)  
}
```

- One line loops (curly braces are not required)

```
for(i in 1:4) print(x[i])
```

# Nested for loops

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))) {
    print(x[i, j])
  }
}
```

# while Loops

```
while (<condition>) {  
    ## do something  
}
```

Example:

```
count <- 0  
while(count < 10) {  
    print(count)  
    count <- count + 1  
}
```

While loops can potentially result in infinite loops if not written properly. Use with care!



# repeat

```
x0 <- 1
```

```
tol <- 1e-8
```

```
repeat {
```

```
  x1 <- computeEstimate()
```

```
  if(abs(x1 - x0) < tol) { ## Close enough?
```

```
    break
```

```
  } else {
```

```
    x0 <- x1
```

```
  }
```

```
}
```

# next, break

- **next** is used to skip an iteration of a loop.

```
for(i in 1:100){  
  if(i <= 20) {  
    ## Skip the first 20 iterations  
    next  
  }  
  ## Do something here  
}
```

- **break** is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100){  
  print(i)  
  if(i > 20) {  
    ## Stop loop after 20 iterations  
    break  
  }  
}
```

# Summary

- Control structures like if, while, and for allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if (you believe) they are theoretically correct.
- Control structures mentioned here are primarily useful for writing programs; for commandline interactive work, the “apply” functions are more useful (will cover later)
- It is more efficient to use built-in functions or vectorization rather than control structures whenever possible.