**Example (Ross; Albert & Rizzo):** Assume that the social class of a child (lower, middle, upper) depends only on his/her parents. The class of the parents is indicated by row and the class of the child by column. Assume that class mobility is represented in terms of the following transition probability matrix:

Children

Parents

| | lower | middle | upper |
|---|---|---|---|
| lower | 0.45 | 0.48 | 0.07 |
| middle | 0.05 | 0.70 | 0.25 |
| upper | 0.01 | 0.50 | 0.49 |

# Introduction: R basics

- To create the transition probability matrix in **R:**

```
> probs=c(.45,.05,.01,.48,.70,.50,.07,.25,.49)
> P=matrix(probs,nrow=3,ncol=3)
> P
     [,1] [,2] [,3]
[1,] 0.45 0.48 0.07
[2,] 0.05 0.70 0.25
[3,] 0.01 0.50 0.49
> rownames(P) <- colnames(P) <- c(“lower", "middle", “upper")
> P
       lower middle upper
lower   0.45   0.48  0.07
middle  0.05   0.70  0.25
upper   0.01   0.50  0.49
```

# Introduction: Examples

```
> rowSums(P)
lower middle upper
1 1 1
> apply(P, MARGIN=1, FUN=sum)
lower middle upper
1 1 1
```

- The transition probabilities for two generations are given by $P = P \times P = P^2$

```
> P2 = P %*% P # Note that we didn't use P^2
> P2
        lower middle upper
lower  0.2272 0.5870 0.1858
middle 0.0600 0.6390 0.3010
upper  0.0344 0.5998 0.3658
```

# Introduction: Examples

- In 2 generations, the probability that the descendants of lower class parents can transition to upper class is:

```
> P2[1,3]
[1] 0.1858
```

- The distribution for the lower class is:

```
> P2[1,]
 lower middle  upper
0.2272 0.5870 0.1858
```

- After 4 and 8 generations we have:

```
> P4=P2%*%P2; P8=P4%*%P4
> P8
             lower    middle     upper
lower  0.06350395 0.6233444 0.3131516
middle 0.06239010 0.6234412 0.3141687
upper  0.06216410 0.6234574 0.3143785
```

# Introduction: Examples

The function `sweep` can be used to sweep out a statistic from a matrix:

```
> A=matrix(1:16,4,4,byrow=TRUE)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
> m=max(A)
> m
[1] 16
> A1=sweep(A,MARGIN=1:2,STATS=m,FUN="-")
```

# Introduction: Examples

```
> A1
     [,1] [,2] [,3] [,4]
[1,]  -15  -14  -13  -12
[2,]  -11  -10   -9   -8
[3,]   -7   -6   -5   -4
[4,]   -3   -2   -1    0
> colMeans(A)
[1]  7  8  9 10
> sweep(A,2,colMeans(A))
     [,1] [,2] [,3] [,4]
[1,]   -6   -6   -6   -6
[2,]   -2   -2   -2   -2
[3,]    2    2    2    2
[4,]    6    6    6    6
```

# Introduction: Examples

- **R** supports higher order arrays:

```
> S = array(1:24, dim=c(4,2,3))
> S
, , 1
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
, , 2
     [,1] [,2]
[1,]    9   13
[2,]   10   14
[3,]   11   15
[4,]   12   16
```

**Higher order arrays**

# Introduction: Examples

```
, , 3
     [,1] [,2]
[1,]   17   21
[2,]   18   22
[3,]   19   23
[4,]   20   24
```

**Higher order arrays**

# Introduction

- A list is similar to a vector, but it can contain different types of elements (including vectors and other lists):

```
> x = list("California", c(1,3,5,1), 92, TRUE)
> x
[[1]]
[1] "California"

[[2]]
[1] 1 3 5 1

[[3]]
[1] 92

[[4]]
[1] TRUE
```

# Introduction

- You can also create an empty list with the vector function:

```
> x = vector(“list”, length=2)
> x
[[1]]
[1] NULL
[[2]]
[1] NULL
```

- The sub-setting operator for lists is `[[]]`, and it can be used recursively with `[]`

```
> x = list("California", c(1,3,5,1), 92,
TRUE)
> x[[1]]
[1] "California”
> x[[2]][4]
[1] 1
```

# Introduction

- Be careful: If you use `[ ]` with a list you recover a list with a single element, rather than the element itself:

```
> x[1]
[[1]]
[1] "California"
```

- Unfortunately, the `[[ ]]` operator does not work like `[ ]`, e.g., multiple indexes are interpreted recursively:

```
> x[[1:2]]          #Same as x[[1]][2]
Error in x[[1:2]] : subscript out of
bounds
> x[[c(2,4)]]       #Same as x[[2]][4]
[1] 1
```

# Introduction

- If you want to recover a subset of elements of the list you can use `[]`, but remember that you will get another list!

```
> x = list("California", c(1,3,5,1), 92,
TRUE)
> x[c(1,3)]
[[1]]
[1] "California"

[[2]]
[1] 92
```

# Introduction

- Assigning names to the elements of a list:

```
> x = list(state="CA",co=c(1,3,5,1),loc=92)
> x
$state
[1] "CA"
$co
[1] 1 3 5 1
$loc
[1] 92
> x$state
[1] "CA"
> x[["state"]]
[1] "CA"
> x["state"]
$state
[1] "CA"
```

# Introduction

A compact way to describe a list is to use the function `str`:

```
> str(faithful)
'data.frame':272 obs. of  2 variables:
 $ eruptions: num 3.6 1.8 3.33 2.28 4.53 ...
 $ waiting  : num 79 54 74 62 85 55 88 85 51
85 ...
```

Many functions in **R** return lists:

```
> H=hist(faithful$waiting)
> names(H)
[1] "breaks"    "counts"    "density"   "mids"
"xname"     “equidist"
```

# Introduction

- Factors are meant to store nominal variables:

```
> z=factor(c("CA","NE","OR","CA","CA","OR",
  "OR","CA","NE"))
> z    #Categorical variable with 3 levels
[1] CA NE OR CA CA OR OR CA NE
Levels: CA NE OR
> table(z)
z
CA NE OR
 4  2  3
```

Example: Rolling a die

```
> y=c(1, 4, 3, 5, 4, 2, 4)
> possible.rolls=c(1, 2, 3, 4, 5, 6)
> labels.rolls=c("one", "two", "three", "
four", "five", "six")
> fy=factor(y, levels=possible.rolls,
labels=labels.rolls)
> fy
 [1] one four three five four two four
 Levels: one two three four five six
> table(fy)
 fy
   one two three four five six
     1   1     1    3    1   0
```

# Introduction

- Strings are denoted by quotation marks:

```
> x = "Hello"
> y = "Bye"
> z = "world"
> paste(x, z)
[1] "Hello world"
> w = c(x,y)
> paste(w, z, sep=" cruel ")
[1] "Hello cruel world" "Bye cruel world"
> paste(w, z, sep=" cruel ", collapse=" and ")
[1] "Hello cruel world and Bye cruel world"
```

# Introduction

- You can also trim a string:

```
> substr(x,2,4)
[1] "ell”
> substr(x, nchar(x)-2, nchar(x))
#Pick last 3 chars
[1] “llo”
```

# Data frames

- Data frames are special types of objects in **R** designed to store and manipulate data in a convenient manner.

- Unlike matrices, the columns of a `data.frame` object can be of different types such as numeric or character.

- Several data sets are installed with **R** as data frames; a list of these data sets can be displayed by the command `data()`.

- Data frames are somewhat similar to a spreadsheet, with variables corresponding to columns and observations to rows. Variables can be numeric or categorical (characters or factors).

# Data frames

```
> x = data.frame(var1 = 1:4, var2 = c(T, T, F, F),
var3=factor(c(1,1,2,1)))
> x
  var1  var2 var3
1    1  TRUE    1
2    2  TRUE    1
3    3 FALSE    2
4    4 FALSE    1
```

- Data frames are usually constructed when loading data from a file. They are infrequently constructed “by hand”.

# Data frames

- You can access elements of a data frame in the same way you access elements of a matrix:

```
> x = data.frame(var1 = 1:4, var2 = c(T, T, F, F),
var3=factor(c(1,1,2,1)))
> x
  var1  var2 var3
1    1  TRUE    1
2    2  TRUE    1
3    3 FALSE    2
4    4 FALSE    1
> x[2,3]
[1] 1
Levels: 1 2
> x[4,1]
[1] 4
```

# Data frames

- Sometimes it is easier to “attach” the data frame:

```
> x = data.frame(var1 = 1:4, var2 = c(T, T,
F, F), var3=factor(c(1,1,2,1)))
> var1
Error: object 'var1' not found
> attach(x)
> var1
[1] 1 2 3 4
```

- Do not forget to detach the data frame when you are done. The columns in the data frame will mask other variables in your environment that have the same name

```
> detach(x)
> var1
Error: object 'var1' not found
```

- Changes made to the “attached” values are not stored.

# Data frames

**Example (Albert & Rizzo):** The data frame `USArrest` contains records of violent crimes in the US. The statistics are given as arrests per 100,000 residents for assault, murder and rape in each of the 50 US states in 1973. The percentage of the population living in urban areas is also given.

- To display the first lines of data:

```
> head(USArrests,3)
        Murder Assault UrbanPop Rape
Alabama   13.2     236       58 21.2
Alaska    10.0     263       48 44.5
Arizona    8.1     294       80 31.0
```

# Data frames

- Sample size and dimension:

```
> nrow(USArrests)
[1] 50
> dim(USArrests)
[1] 50 4
```

- Names of variables:

```
> names(USArrests)
[1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

- Structure of the data frame:

```
> str(USArrests)
'data.frame': 50 obs. of  4 variables:
 $ Murder  : num  13.2 10 8.1 8.8 ...
 $ Assault : int  236 263 294 190 ...
 $ UrbanPop: int  58 48 80 50 ...
 $ Rape    : num  21.2 44.5 31 19.5 ...
```

# Data frames

- When data frames contain only numbers they can be converted into matrices but all variables must be of the same type so **R** will convert integers to numeric:

```
> arrests=as.matrix(USArrests)
> str(arrests)
 num [1:50, 1:4] 13.2 10 8.1 8.8 9 7.9 3.3 5.9
15.4 17.4 ...
 - attr(*, "dimnames")=List of 2
  ...$ : chr [1:50] "Alabama" "Alaska" "Arizona"
"Arkansas" ...
  ...$ : chr [1:4] "Murder" "Assault" "UrbanPop"
"Rape"
```

Note: Row labels were preserved and the variable names were converted to column labels (now use `rownames`, `colnames`, `dimnames`, etc)

# Data frames

- Computing summary statistics:

```
> summary(USArrests)
     Murder          Assault         UrbanPop          Rape
 Min.   : 0.800   Min.   : 45.0   Min.   :32.00   Min.   : 7.30
 1st Qu.: 4.075   1st Qu.:109.0   1st Qu.:54.50   1st Qu.:15.07
 Median : 7.250   Median :159.0   Median :66.00   Median :20.10
 Mean   : 7.788   Mean   :170.8   Mean   :65.54   Mean   :21.23
 3rd Qu.:11.250   3rd Qu.:249.0   3rd Qu.:77.75   3rd Qu.:26.18
 Max.   :17.400   Max.   :337.0   Max.   :91.00   Max.   :46.00
```

- Extract data:

```
> USArrests["California", "Murder"]
[1] 9
> USArrests["California", ]
           Murder Assault UrbanPop Rape
California      9     276       91 40.6
```

# Importing data

- **<u>Entering data manually</u>**

Example: Car mileage on 4 models of Japanese cars:

| Model | | | |
|---|---|---|---|
| A | B | C | D |
| 22 | 28 | 29 | 23 |
| 26 | 24 | 32 | 24 |
| | 29 | 28 | |

```
> y1 = c(22, 26)
> y2 = c(28, 24, 29)
> y3 = c(29, 32, 28)
> y4 = c(23, 24)
> y = c(y1, y2, y3, y4)
> Model = c(rep("A", 2),rep("B", 3),rep("C", 3),
rep("D", 2))
> milages=data.frame(y, Model)
```

# Importing data

- **Importing data from a text file**

Example:

- Go to https://dasl.datadescription.com/datafiles/ and download the file a-rod-2016.txt
- Read it as:

```
>arod=read.table("a-rod-2016.txt",
header=TRUE)
```

Note that in this example the data are delimited by space characters. Data in spreadsheet format are typically delimited by tab characters or commas: use the argument `sep`

For spreadsheet data, the simplest is to save it in .csv format and use `read.csv`

# Importing data

- **Data available on the internet**

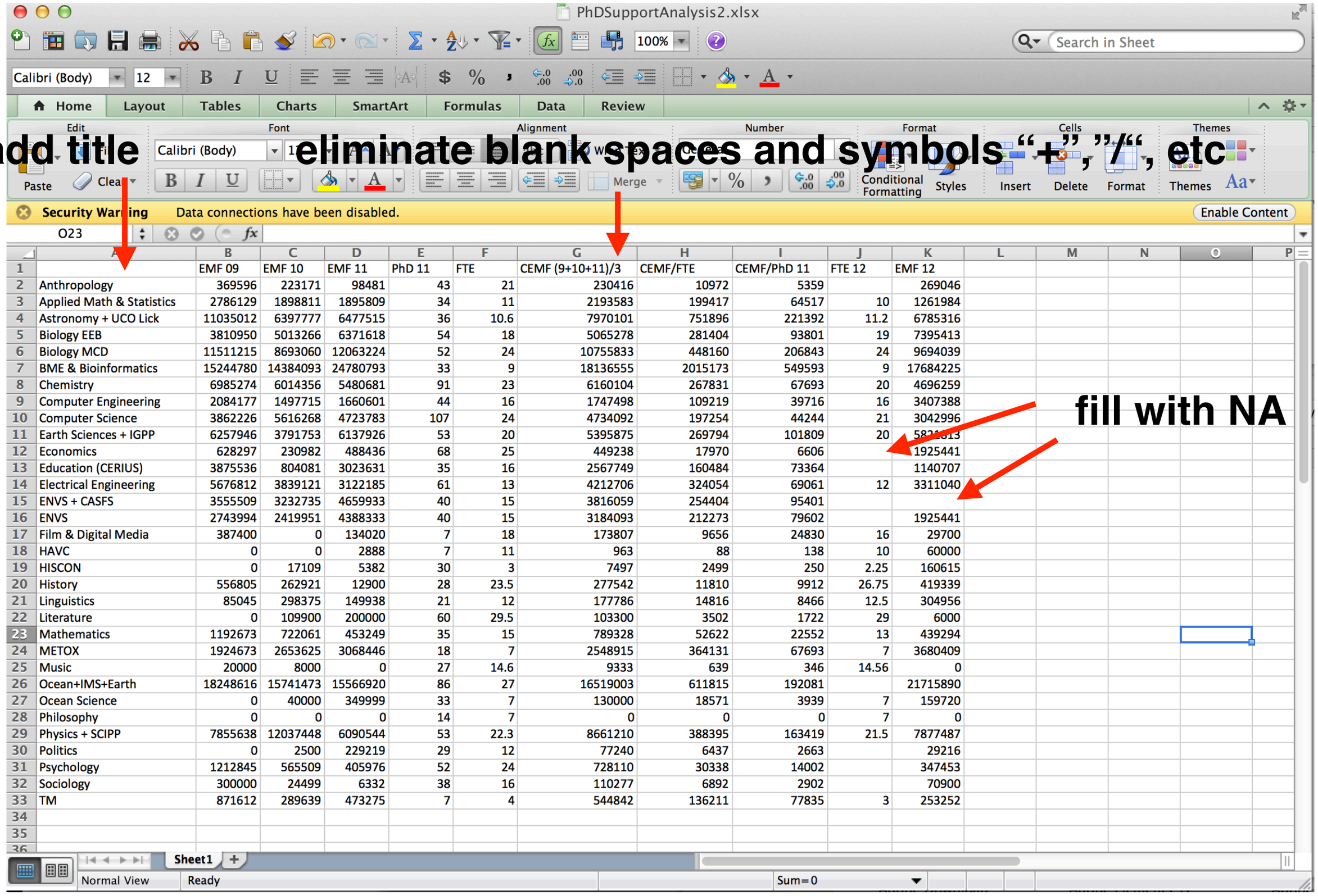
Example:

```
>pidigits = read.table("http://www.itl.nist.gov/div898/
strd/univ/data/PiDigits.dat",skip=60)
>head(pidigits)
  V1
1  3
2  1
3  4
4  1
5  5
6  9
```

# Importing data: Preparing your files in R

- Make sure you have the same number of columns (sometimes called “fields”) on each row:
  - Add column names as appropriate
  - If not a tab- or space-delimited file, empty cells (representing missing data) should be filled with letters NA
- Eliminate any special characters (e.g., accents, punctuations, quotation marks, etc)
- For column headers, eliminate blank spaces or replace them with some other symbol (avoid “.” and “_”)
  - White spaces are OK in other places, as long as you use .csv files

# Importing data: Preparing your files in R




| | | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | EMF 09 | EMF 10 | EMF 11 | PhD 11 | FTE | CEMF (9+10+11)/3 | CEMF/FTE | CEMF/PhD 11 | FTE 12 | EMF 12 |
| 2 | Anthropology | 369596 | 223171 | 98481 | 43 | 21 | 230416 | 10972 | 5359 | | 269046 |
| 3 | Applied Math & Statistics | 2786129 | 1898811 | 1895809 | 34 | 11 | 2193583 | 199417 | 64517 | 10 | 1261984 |
| 4 | Astronomy + UCO Lick | 11035012 | 6397777 | 6477515 | 36 | 10.6 | 7970101 | 751896 | 221392 | 11.2 | 6785316 |
| 5 | Biology EEB | 3810950 | 5013266 | 6371618 | 54 | 18 | 5065278 | 281404 | 93801 | 19 | 7395413 |
| 6 | Biology MCD | 11511215 | 8693060 | 12063224 | 52 | 24 | 10755833 | 448160 | 206843 | 24 | 9694039 |
| 7 | BME & Bioinformatics | 15244780 | 14384093 | 24780793 | 33 | 9 | 18136555 | 2015173 | 549593 | 9 | 17684225 |
| 8 | Chemistry | 6985274 | 6014356 | 5480681 | 91 | 23 | 6160104 | 267831 | 67693 | 20 | 4696259 |
| 9 | Computer Engineering | 2084177 | 1497715 | 1660601 | 44 | 16 | 1747498 | 109219 | 39716 | 16 | 3407388 |
| 10 | Computer Science | 3862226 | 5616268 | 4723783 | 107 | 24 | 4734092 | 197254 | 44244 | 21 | 3042996 |
| 11 | Earth Sciences + IGPP | 6257946 | 3791753 | 6137926 | 53 | 20 | 5395875 | 269794 | 101809 | 20 | 5821513 |
| 12 | Economics | 628297 | 230982 | 488436 | 68 | 25 | 449238 | 17970 | 6606 | | 1925441 |
| 13 | Education (CERIUS) | 3875536 | 804081 | 3023631 | 35 | 16 | 2567749 | 160484 | 73364 | | 1140707 |
| 14 | Electrical Engineering | 5676812 | 3839121 | 3122185 | 61 | 13 | 4212706 | 324054 | 69061 | 12 | 3311040 |
| 15 | ENVS + CASFS | 3555509 | 3232735 | 4659933 | 40 | 15 | 3816059 | 254404 | 95401 | | |
| 16 | ENVS | 2743994 | 2419951 | 4388333 | 40 | 15 | 3184093 | 212273 | 79602 | | 1925441 |
| 17 | Film & Digital Media | 387400 | 0 | 134020 | 7 | 18 | 173807 | 9656 | 24830 | 16 | 29700 |
| 18 | HAVC | 0 | 0 | 2888 | 7 | 11 | 963 | 88 | 138 | 10 | 60000 |
| 19 | HISCON | 0 | 17109 | 5382 | 30 | 3 | 7497 | 2499 | 250 | 2.25 | 160615 |
| 20 | History | 556805 | 262921 | 12900 | 28 | 23.5 | 277542 | 11810 | 9912 | 26.75 | 419339 |
| 21 | Linguistics | 85045 | 298375 | 149938 | 21 | 12 | 177786 | 14816 | 8466 | 12.5 | 304956 |
| 22 | Literature | 0 | 109900 | 200000 | 60 | 29.5 | 103300 | 3502 | 1722 | 29 | 6000 |
| 23 | Mathematics | 1192673 | 722061 | 453249 | 35 | 15 | 789328 | 52622 | 22552 | 13 | 439294 |
| 24 | METOX | 1924673 | 2653625 | 3068446 | 18 | 7 | 2548915 | 364131 | 67693 | 7 | 3680409 |
| 25 | Music | 20000 | 8000 | 0 | 27 | 14.6 | 9333 | 639 | 346 | 14.56 | 0 |
| 26 | Ocean+IMS+Earth | 18248616 | 15741473 | 15566920 | 86 | 27 | 16519003 | 611815 | 192081 | | 21715890 |
| 27 | Ocean Science | 0 | 40000 | 349999 | 33 | 7 | 130000 | 18571 | 3939 | 7 | 159720 |
| 28 | Philosophy | 0 | 0 | 0 | 14 | 7 | 0 | 0 | 0 | 7 | 0 |
| 29 | Physics + SCIPP | 7855638 | 12037448 | 6090544 | 53 | 22.3 | 8661210 | 388395 | 163419 | 21.5 | 7877487 |
| 30 | Politics | 0 | 2500 | 229219 | 29 | 12 | 77240 | 6437 | 2663 | | 29216 |
| 31 | Psychology | 1212845 | 565509 | 405976 | 52 | 24 | 728110 | 30338 | 14002 | | 347453 |
| 32 | Sociology | 300000 | 24499 | 6332 | 38 | 16 | 110277 | 6892 | 2902 | | 70900 |
| 33 | TM | 871612 | 289639 | 473275 | 7 | 4 | 544842 | 136211 | 77835 | 3 | 253252 |

# Importing data: Preparing your files in R

- You can then save the file as a `.csv` file
- Sometimes you need to edit the file manually to add quotation marks, delete additional characters etc
- You can then read your file in **R** or **RStudio**

# Reformatting your data

When categorical predictors are present there are multiple ways in which they can be stored:

- As a series of vectors, one for each combination of categorical predictors.
- As a series vectors, one containing all the stacked data and the rest containing factors corresponding to each of the predictors.
- As matrices or higher dimensional arrays, with each dimension corresponding to one categorical predictor.

The second format is the most useful for working with **R**

# Reformatting your data

You can move between the 2nd and 3rd formats relatively easily:

```
> x = data.frame(matrix(seq(1, 120), ncol=2))
> x
   X1  X2
1   1  61
2   2  62
3   3  63
....
> z = stack(x)
> z
    values ind
1        1  X1
2        2  X1
3        3  X1
....
> w = unstack(x)
> w
   X1  X2
1   1  61
2   2  62
3   3  63
....
```

# Pseudo-random number generation

**R** provides functions to compute the pdf/pfm, cdf, inverse cdf of many standard distributions (these functions are vectorized and values are recycled) and sample from those distributions:

```
> dnorm(1, mean=0, sd=1)  #Density at 1
[1] 0.2419707
> pnorm(1, mean=0, sd=1)  #Pr(X ≤ 1)
[1] 0.8413447
> qnorm(0.5, mean=0, sd=1)#x s.t. Pr(X ≤ x)=0.5
[1] 0
> rnorm(4, mean=3, sd=1)
[1] 4.024884 5.293606 2.172033 2.425855
> rbinom(10, size=12, prob=0.2)
[1] 3 2 2 1 3 2 5 3 0 1
```

# Pseudo-random number generation

- Samples from a discrete distribution

```
> x = c("CA","NE","OR","WA","UT")
> sample(x, 3, replace=TRUE)
[1] "WA" "NE" "NE"
> sample(x, 3, replace=FALSE)
[1] "WA" "UT" "NE"
```

- Sometimes it is useful to get the results in the log-scale:

```
> exp(pnorm(1, mean=0, sd=1, log.p=T))
[1] 0.8413447
> pnorm(1, mean=0, sd=1)
[1] 0.8413447
```

- You can set the seed of the underlying uniform random variate generator, this is useful to reproduce your results (e.g., reproducibility and debugging purposes)

```
> set.seed(13)
```

# Defining functions

- You can define your own functions in **R**

```
> logit = function(x){
+   z = log(x/(1-x))
+   return(z)}
> logit(runif(3, 0, 1))
[1] -4.3336057 -2.7145211 -0.6638038
> my_norm = function(x){sqrt(x%*%x)}
> my_norm(1:4)
[1] 5.477226
```

- Functions with more than one argument:

```
> f = function(x, a=1, b=0){a*x^2+b}
> f(2)      #Alternatively, f(x=2)
[1] 4
> f(2,2)    #This is equivalent to f(x=2, a=2)
[1] 8
> f(2,2,2)
[1] 10
```

# Defining functions

- Values can be returned invisibly:

```
> f1 = function(x){x}
> f2 = function(x){invisible(x)}
> f1(1)      #Normal behavior
[1] 1
> f2(1)      #Return “invisible” value
> a = f2(2) #The value is assignable!
> a
[1] 2
```

- To see internal calculations in a function you need to print:

```
> f1 = function(x){
+    paste(”Attempt 1 to show value of x =”, x)
+    print(paste(”Attempt 2 to show value of x =”, x))
+    return(invisible(0))}
> f1(3)
[1] “Attempt 2 to show value of x = 3”
```

# Defining functions

- Function arguments can be <u>matched positionally</u> or by <u>name</u>:

```
> f = function(x, y){
>   print(paste("x =", x, "y =", y))
>   return(invisible(0))}
> f(2,3)
[1] "x = 2 y = 3"
> f(x=2, y=3)
[1] "x = 2 y = 3"
> f(y=3, x=2)
[1] "x = 2 y = 3"
> f(x=2,3)
[1] "x = 2 y = 3"
> f(y=3,2)
[1] "x = 2 y = 3"
```

You can mix positional matching with matching by name

# Defining functions

- **R** doesn't check for formal parameters in the function definition if not used by the function (lazy evaluation):

```
> f = function(a, b){
>   return(a)}
> f(3)      # No error reported
[1] 3
> f(3,2)
[1] 3
```

- Functions can be passed as arguments to other functions:

```
> a = c(1,2,-1,4,0)
> summarizex = function(x, ff){
>   z = ff(x)
>   return(z)}
> summarizex(a, mean)
[1] 1.2
> summarizex(a, median)
[1] 1
```

# Defining functions

- **Scoping: R** allows for free variables in functions (variables that are not formal arguments or defined inside the function):

```
> b = 4
> f = function(x, y){
>   z = x + y/b     # Avoid free variables!
>   return(z)}
> f(3,2)
[1] 3.5
```

- <u>Warning:</u> **R** does lexical scoping, i.e., values of free variables are searched for in the environment in which the function was defined. Free variables might be handy for some purposes but avoid them unless you know exactly what you are doing.

# Defining functions

**Note**:

- If the value of a symbol is not found in the environment in which a function was defined, the search is continued in the parent environment.
- After the top-level environment, the search continues down the search list until the empty environment is hit.
- If a value for a given symbol cannot be found once the empty environment is arrived at, an error is thrown.
- The `search()` function tells us what the search list is.

# Defining functions

```
> b = 4
> g = function(x, y){
>   b = 3
>   f = function(x, y){
>     z = x + y/b
>     return(z)}
>   w = f(x,y)
>   return(w)}
> g(3,2)
[1] 3.666667
```

Example of dynamic scoping

Moving up on search space

```
> b = 4
> g = function(x, y){
>   f = function(x, y){
>     z = x + y/b
>     return(z)}
>   w = g(x,y)
>   return(w)}
> g(3,2)
[1] 3.5
```

# Defining functions

- The … argument indicates a variable number of arguments that are usually passed on to other functions. **Example:** `plot(x,y,…)`

- Arguments that appear after … on the argument list must be named explicitly and cannot be partially matched:

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
> paste(“a”, “b”, “ Hello ”)
[1] "a b  Hello "
> paste(“a”, “b”, sep=“ Hello ”)
[1] "a Hello b"
```

# Defining functions

- If you need to define functions that you regularly use, it is useful to put them in a file and “source” the file:

```
> source(file="myfunctions.R")
```

- Make sure the file is on your working directory, or add the path for the location. Useful function: `setwd`

# Loops

- Loops can be easily constructed:

```
> for(i in 1:3){
+   print(2*i)}
[1] 2
[1] 4
[1] 6
```

- Index values can be a vector of any type:

```
> counties = (“CA”,”NE”,”OR”)
> for(i in counties){
+   print(i)}
[1] "CA"
[1] "NE"
[1] “OR"
```

# Loops

- **R** does not like zero-length loops:

```
> n = 1
> for(i in 1:n){print(i)}
[1] 1
> n = 0
> for(i in 1:n){print(i)}
[1] 1
[1] 0
```

Common source of problems

- `While` loops:

```
> x = 1
> while(x<10){
+   x = x^2 + 1
+   print(x)}
[1] 2
[1] 5
[1] 26
```

# Loops

- There are multiple ways to fill a vector using a loop:

```
> x = numeric(0)
> for(i in 1:10){x[i] = i}     #Bad (slow)!
>
> x = numeric(0)
> for(i in 1:10){x = c(x,i)}   #Even worse(slower)!
>
> x = numeric(10)
> for(i in 1:10){x[i] = i}     #Good!
```

Memory allocation is very expensive!

# Loops

- For loops in **R** are **very** slow:

```
> x = rexp(1000000,rate=1)
> logown = function(x){
+   z = rep(0, length(x))
+   for(i in 1:length(x)){
+     z[i] = log(x[i])}
+   return(z)}
> system.time(log(x))
   user  system elapsed
  0.011   0.001   0.011
> system.time(logown(x))
   user  system elapsed
  1.024   0.008   1.014
```

Slightly better to store length(x) in a variable than to recalculate it multiple times!

# Loops

- The `apply` function:

```
> x = matrix(rnorm(60000), nrow=10000, ncol=6)
> apply(x, 2, mean) #Easy to read
[1] -0.004258705 -0.000315459  0.002038574
-0.004258705 -0.000315459  0.002038574
> z = numeric(6)     #"C style", more convoluted
> for(i in 1:6){
+   z[i] = mean(x[,i])}
> z
[1] -0.004258705 -0.000315459  0.002038574
-0.004258705 -0.000315459  0.002038574
```

- A number of "relatives": `lapply, tapply, sapply,` etc…

# Loops

- `apply` can be used with functions that require multiple arguments:

```
> x = matrix(rnorm(6000), ncol=3)
> apply(x, 2, quantile, c(0.025,0.975))
          [,1]      [,2]      [,3]
2.5%  -1.949164 -1.963706 -1.853970
97.5%  1.982152  1.913266  1.925459
```

- First argument must correspond to the first argument of the function to be applied (FUN). Other arguments can be appended in the same order that they appear in the definition of FUN.
- `apply` is cleaner but not necessarily faster than a for loop and it might not work with multiple arguments…

# Loops

```
> x = matrix(rexp(6000000), ncol=6000)
> loopsum=function(x){
+   n = dim(x)[2]
+   z=numeric(n)
+   for(i in 1:n){
+     z[i] = mean(x[,i])}
+   return(z)}
> system.time(loopsum(x))
   user  system elapsed
  0.108   0.008   0.104
> system.time(apply(x, 2, sum))  #About the same!
   user  system elapsed
  0.104   0.015   0.108
```

- For a few row/column operations there is a fast alternative (loop built in low-level C)

# Loops

```
> x = matrix(rexp(6000000), ncol=6000)
> system.time(apply(x, 2, sum))
   user  system elapsed
  0.104   0.015   0.108
> system.time(colSums(x))
    user  system elapsed
   0.007   0.000   0.008    #Much better!
```

- Another common problem is standardization of matrices, i.e., applying the same operation to each column/row but using a different argument for each:

```
> x = matrix(rexp(6000000), ncol=6000)
> system.time(for(i in 1:6000){x[,i] = x[,i]/
sum(x[,i])})
    user  system elapsed
   0.231   0.031   0.249
```

# Loops

- There is a built-in function that can help:

```
> system.time(scale(x, center=FALSE, scale=colSums(x)))
   user  system elapsed
  0.157   0.006   0.152
```

- Recycling can be used to vectorize complex operations:

```
> S = matrix(1, 2, 2)
> x = c(2,4)
> S/x
     [,1] [,2]
[1,] 0.50 0.50
[2,] 0.25 0.25
```

However, it can make your code hard to read!

Note: by default **R** fills matrices by column

# Loops

- Consider evaluating a function over a 2-dimensional grid of arguments:

```
> x = seq(-2,2,length=50)
> y = seq(-2,2,length=50)
> f = function(x,y) x^2 + y^2
> f(x,y)
[1] 8.000000000 7.360266556 ... #50 values
```

- Standard vectorization evaluates the function only on diagonal elements of the grid, to get what you want:

```
> z = expand.grid(x,y)
> f(z[,1], z[,2])
[1] 8.000000000 7.680133278 ... #2500 values

> system.time(f(z[,1], z[,2]))
   user  system elapsed
      0       0       0  #So small, it is rounded down!
```

# Loops

```
> fgrid = function(x,y){
>   n1 = length(x)
>   n2 = length(y)
>   M  = matrix(NA,n1,n2)
>   for(i in 1:n1){
>     for(j in 1:n2){
>       M[i,j] = f(x[i],y[j])}}
>   return(M)}
> system.time(fgrid(x,y))
   user  system elapsed
  0.028   0.001   0.023
```

The vectorized version is much faster!

- Alternative way to vectorize:

```
> system.time(outer(x,y,f))
   user  system elapsed
      0       0       0
```

# Conditional statements

- Example:

```
> x = 7
> if(x <= 10){
>   print("Less or equal to 10!")}
> else{
>   print("Greater than 10!")}
[1] "Less or equal to 10!"
```

- The `and` operator in **R** is & and the `or` operator is |:

```
> x = 7
> if(x <= 10 & x >5){
>   print("Between 5 and 10!")}
[1] "Between 5 and 10!"
```

- If/then/else statements are not vectorized, but you can use `ifelse` to vectorize some conditional statements…

# Conditional statements

```
> x = c(1,3,-1)
> if(x < 2){
>   print("Hello")}
> else{
>   print("Bye")}
[1] "Hello"
Warning message:
In if (x < 2) { :the condition has length > 1 and only
the first element will be used
```

Not vectorized

Vectorized

```
> ifelse(x < 2,"Hello","Bye")
[1] "Hello" "Bye"   "Hello"
```

- `ifelse` can be nested:

```
> ifelse(x < 2, ifelse(x < 0, "Hello", ""), "Bye")
[1] ""      "Bye"   "Hello"
```

# Saving your results

- To save the whole session (workspace):

```
>save.image("myworkspace.Rdata")
```

- To save specific objects (as a binary object):

```
>x = seq(1,8)
>y = c("CA","NE")
>save(x,y,file="myobjects.R")
```

- To save a matrix as a text file:

```
>x = matrix(1:10, ncol=5, nrow=2)
>write(t(x), file="myfile.txt",ncolumns=5)
```

# Merging dataframes

- The `merge` function allows you to combine two dataframes without having to write a complex loop:

```
>authors <- data.frame(surname = c(“Tukey",
"Venables", "Tierney", "Ripley", "McNeil"),
nationality = c("US", "Australia", "US", “UK",
"Australia"),deceased = c("yes", rep("no", 4)))

>books <- data.frame(name = c("Tukey", "Venables",
“Tierney","Ripley", "Ripley", "McNeil", "R Core"),
title = c("Exploratory Data Analysis",
"Modern Applied Statistics ...","LISP-STAT","Spatial
Statistics", "Stochastic Simulation”,"Interactive
Data Analysis",”An Introduction to R”), other.author
= c(NA, "Ripley", NA, NA, NA, NA,"Venables & Smith"))

>m1 <- merge(authors, books, by.x = "surname", by.y =
"name")
```