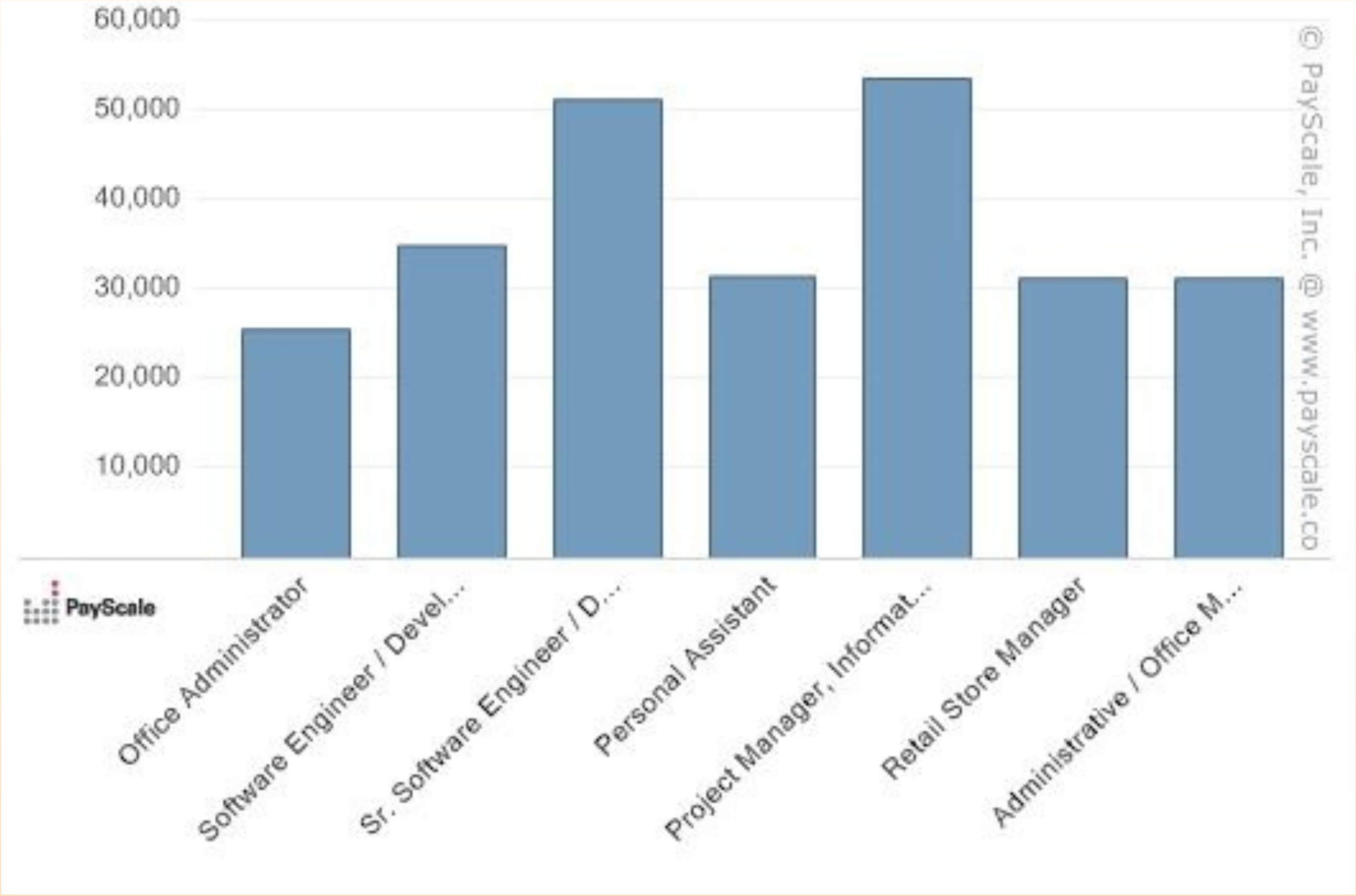
Session 2: Data

Median Salary by Job: Ireland



http://www.payscale.com/research/IE/Country=Ireland/Salary

Mean versus Median

- Lee (2006) describes Stephon Marbury, a 29 year-old from Coney Island, who plays professional basketball for the New York Knicks. Earns $20,000,000 a year.

Mean versus Median

- Lee (2006) describes Stephon Marbury, a 29 year-old from Coney Island, who plays professional basketball for the New York Knicks. Earns $20,000,000 a year.
- “Imagine Stephon Marbury went back last year to meet with 9 teammates on the 10th anniversary of their winning a high school basketball championship.”

Mean versus Median

- Lee (2006) describes Stephon Marbury, a 29 year-old from Coney Island, who plays professional basketball for the New York Knicks. Earns $20,000,000 a year.
- "Imagine Stephon Marbury went back last year to meet with 9 teammates on the 10th anniversary of their winning a high school basketball championship."
- 50% of high school leavers do not go on to college.

Mean versus Median

- Lee (2006) describes Stephon Marbury, a 29 year-old from Coney Island, who plays professional basketball for the New York Knicks. Earns $20,000,000 a year.
- "Imagine Stephon Marbury went back last year to meet with 9 teammates on the 10th anniversary of their winning a high school basketball championship."
- 50% of high school leavers do not go on to college.
- Lee (2006) creates 9 imaginary former class mates: janitor ($13/hr), truck driver ($14/hr), store assistant ($18/hr), mechanic ($20/hr), fire fighter ($24hr), nurse ($25/hr), department store buyer ($28/hr), car salesman ($32/hr), IT project manager ($41/hour).

Mean versus Median

- 13+14+18+20+24+25+28+32+41 = 215

  mean wage = 215 / 9 = $23.90/hour

Mean versus Median

- 13+14+18+20+24+25+28+32+41 = 215

  mean wage = 215 / 9 = $23.90/hour

- The median is

  13 14 18 20 24 25 28 32 41

  $24/hour, a summary value of the data chosen so that half the sample lies above it and half the sample lies below it.

Mean versus Median

- 13+14+18+20+24+25+28+32+41 = 215

  mean wage = 215 / 9 = $23.90/hour
- The median is

  13 14 18 20 24 25 28 32 41

  $24/hour, a summary value of the data chosen so that half the sample lies above it and half the sample lies below it.
- The Mean and Median are both “measures of centre”.

Mean versus Median

- 13+14+18+20+24+25+28+32+41 = 215

  mean wage = 215 / 9 = $23.90/hour
- The median is

  13 14 18 20 24 25 28 32 41

  $24/hour, a summary value of the data chosen so that half the sample lies above it and half the sample lies below it.
- The Mean and Median are both “measures of centre”.
- Marbury played an average of 36.5 minutes over the course of 60 games in 2005. For this, Marbury earned about $20 million, or about $550,000/hour.

Mean versus Median

- Including Marbury‘s wage,the high school team’s mean wage is now $55,000/hour.

Mean versus Median

- Including Marbury's wage,the high school team's mean wage is now \$55,000/hour.
- By contrast, the median is

  13 14 18 20 24 25 28 32 41 55,000

  ↑median = \$24.5/hour,

  half way between the fire fighter and the nurse.

Mean versus Median

- Including Marbury's wage,the high school team's mean wage is now \$55,000/hour.
- By contrast, the median is

  13 14 18 20 24 25 28 32 41 55,000

  ↑median = \$24.5/hour,

  half way between the fire fighter and the nurse.
- In this case the median is a more appropriate "measure of centre".

Mean versus Median

- Including Marbury's wage,the high school team's mean wage is now \$55,000/hour.
- By contrast, the median is

  13 14 18 20 24 25 28 32 41 55,000

  ↑median = \$24.5/hour,

  half way between the fire fighter and the nurse.
- In this case the median is a more appropriate "measure of centre".
- Exercise: redo these calculations crediting Marbury with "working" a 30 hour / week? A 40 hour / week??

Mean versus Median

- http://www.bls.gov/oes/2008/may/oes_nat.htm#b00-0000 May 2008 National Occupational Employment and Wage Estimates United States of America
- Employment 135,185,230
  Median Hourly $15.57
  Mean Hourly $20.32

Mean versus Median

- http://www.bls.gov/oes/2008/may/oes_nat.htm#b00-0000 May 2008 National Occupational Employment and Wage Estimates United States of America
- Employment 135,185,230
  Median Hourly $15.57
  Mean Hourly $20.32
- Verzani example 1.1 quotes George W. Bush "Under this plan, 92 million Americans receive an average tax cut of $1,083" but points out that although the mean tax cut equals $1,083 whereas the median tax cut is closer to $100.

Opinion Polls

- Statistics is a science based on sampling.
- The validity of the statistical inferences arrived at in a study depend on the integrity of sampling process used.

Opinion Polls

- Statistics is a science based on sampling.
- The validity of the statistical inferences arrived at in a study depend on the integrity of sampling process used.
- Potential for inaccuracy arise from:
  - nonresponse bias;
  - response bias;
  - wording of the question;
  - coverage bias.

Opinion Polls

- Statistics is a science based on sampling.
- The validity of the statistical inferences arrived at in a study depend on the integrity of sampling process used.
- Potential for inaccuracy arise from:
  - nonresponse bias;
  - response bias;
  - wording of the question;
  - coverage bias.
- UK General Election 1992.

Opinion Polls

- Statistics is a science based on sampling.
- The validity of the statistical inferences arrived at in a study depend on the integrity of sampling process used.
- Potential for inaccuracy arise from:
  - nonresponse bias;
  - response bias;
  - wording of the question;
  - coverage bias.
- UK General Election 1992.
- Huff (1956) “How to lie with Statistics”

Margin of Error in Opinion Polls

- Which do believe? Small samples or large samples?

Margin of Error in Opinion Polls

- Which do believe? Small samples or large samples?
- Sampling 10,000 people costs more time / money than sampling 1000 people.

Margin of Error in Opinion Polls

- Which do believe? Small samples or large samples?
- Sampling 10,000 people costs more time / money than sampling 1000 people.
- Later in the course we will consider the statistical aspects of this problem and discover:
  - ➡ $p \pm 10\%$ $n = 97$
  - ➡ $p \pm 5\%$ $n = 387$
  - ➡ $p \pm 2\%$ $n = 2401$
  - ➡ $p \pm 1\%$ $n = 9604$

IRAQ WAR: Estimates of Excess Mortality

- In 2004 the Lancet published an article estimating 98,000 excess Iraqi deaths since the 2003 invasion, with a 95% confidence interval 8,000 to 194,000.

IRAQ WAR: Estimates of Excess Mortality

- In 2004 the Lancet published an article estimating 98,000 excess Iraqi deaths since the 2003 invasion, with a 95% confidence interval 8,000 to 194,000.
- In 2006 the Lancet published a second survey estimating 654,965 excess deaths related to the war, with a 95% confidence interval 426,369 to 793,663.

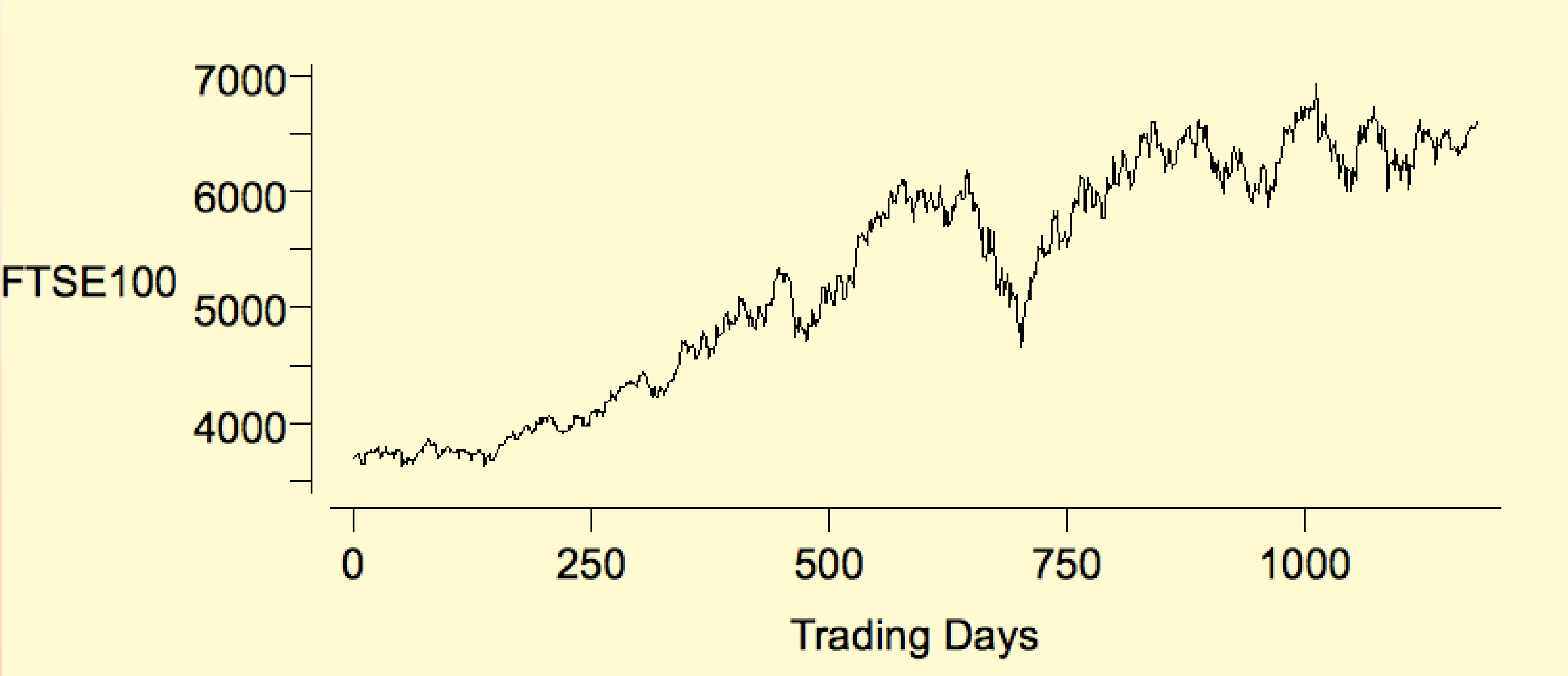
IRAQ WAR: Estimates of Excess Mortality

- In 2004 the Lancet published an article estimating 98,000 excess Iraqi deaths since the 2003 invasion, with a 95% confidence interval 8,000 to 194,000.
- In 2006 the Lancet published a second survey estimating 654,965 excess deaths related to the war, with a 95% confidence interval 426,369 to 793,663.
- Homework: read “Estimating Mortality in War-Time Iraq: A Controversial Survey with Important Lessons for Students” by Fernando De Maio.

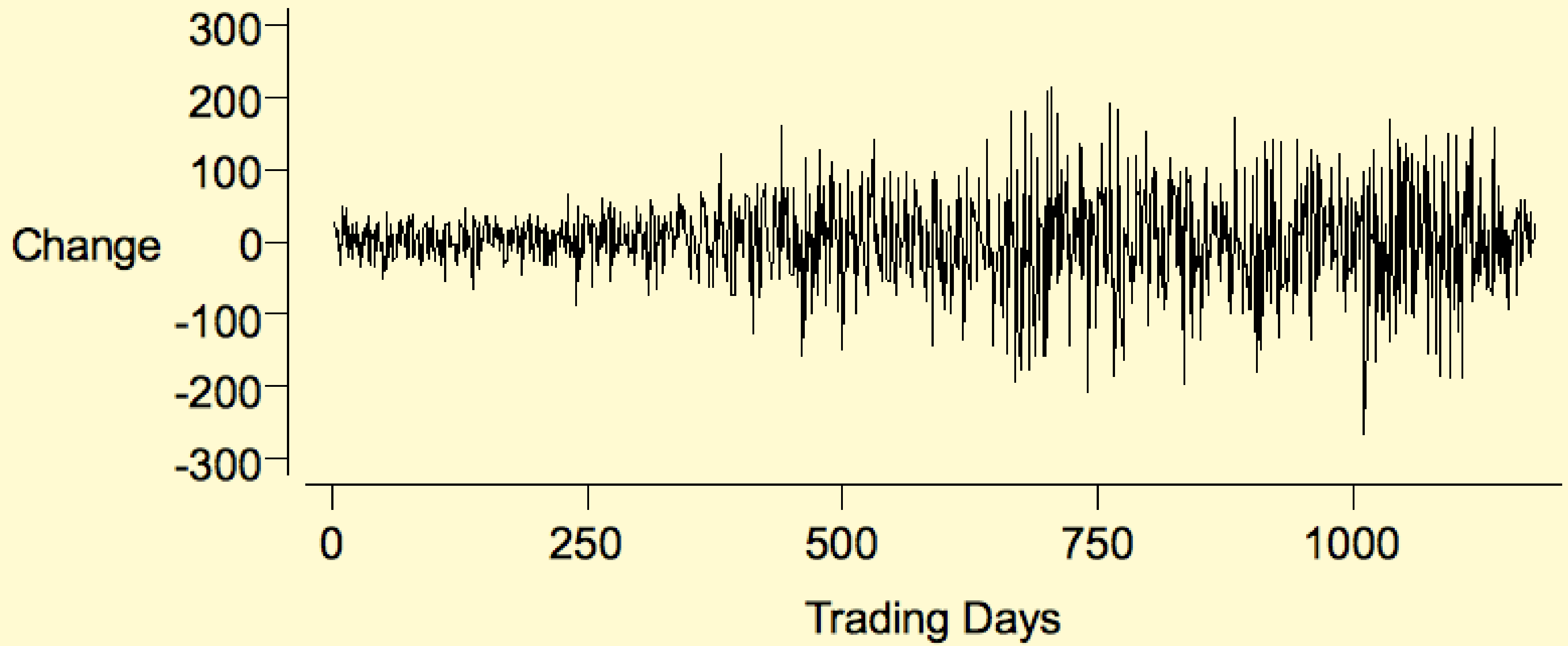
Variation in the Stock Exchange

- FTSE 100
  - 100 most highly capitalised companies
  - representing approximately 80% of the UK market.
  - recognised as THE measure of the UK financial markets.
  - http://www.ftse.co.uk

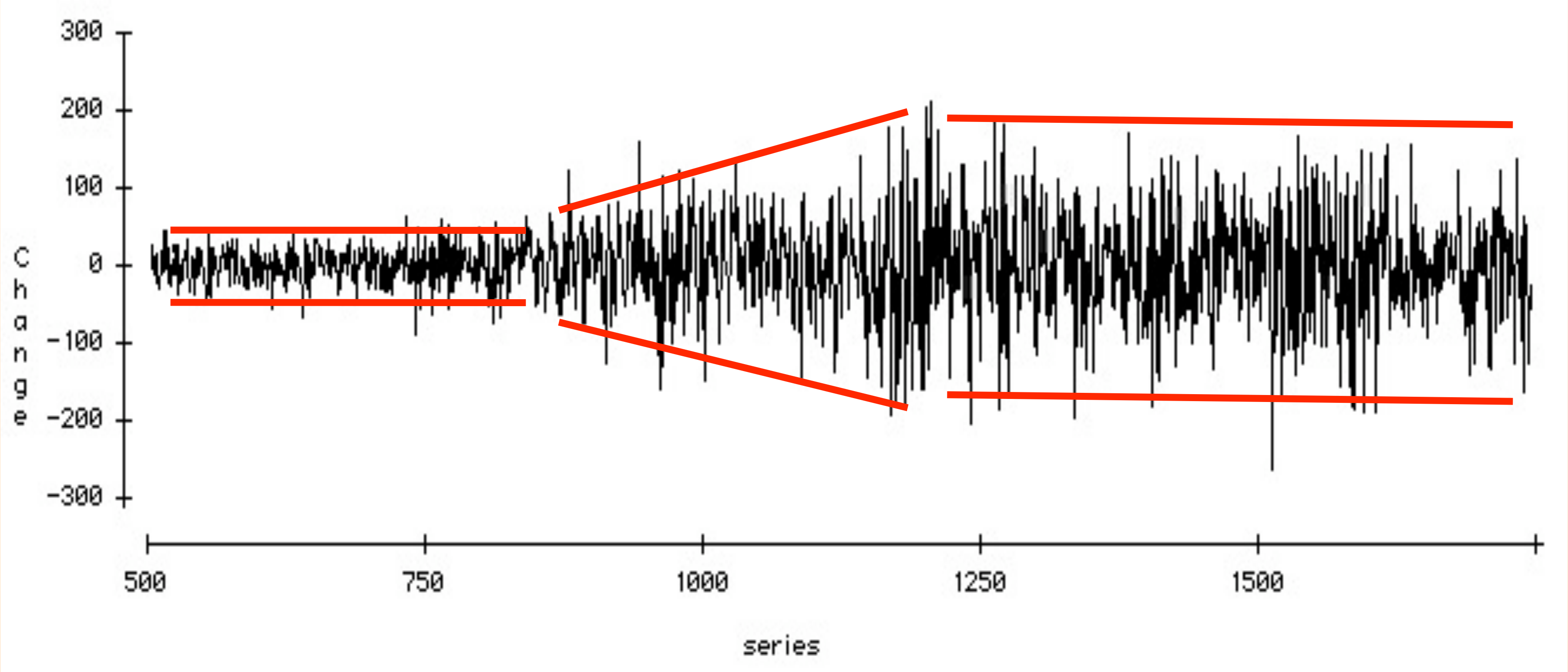
FTSE 100, daily, 1996-2000



Daily Changes in FTSE 100, 1996-2000



Daily Changes in FTSE 100, 1996-2000



Models for the Stock Exchange

Models for the Stock Exchange

- Random walk model:

$$X_t = X_{t-1} + \varepsilon_t$$

Models for the Stock Exchange

- Random walk model:

$$\mathbf{X_t = X_{t-1}} + \varepsilon_{\mathbf{t}}$$

- Efficient Market Hypothesis
  - Diversified portfolios
  - Capital Asset Pricing Model (CAPM)

Models for the Stock Exchange

- Random walk model:

$$X_t = X_{t-1} + \varepsilon_t$$

- Efficient Market Hypothesis
  - Diversified portfolios
  - Capital Asset Pricing Model (CAPM)
- Derivatives trading

  Black-Scholes formula

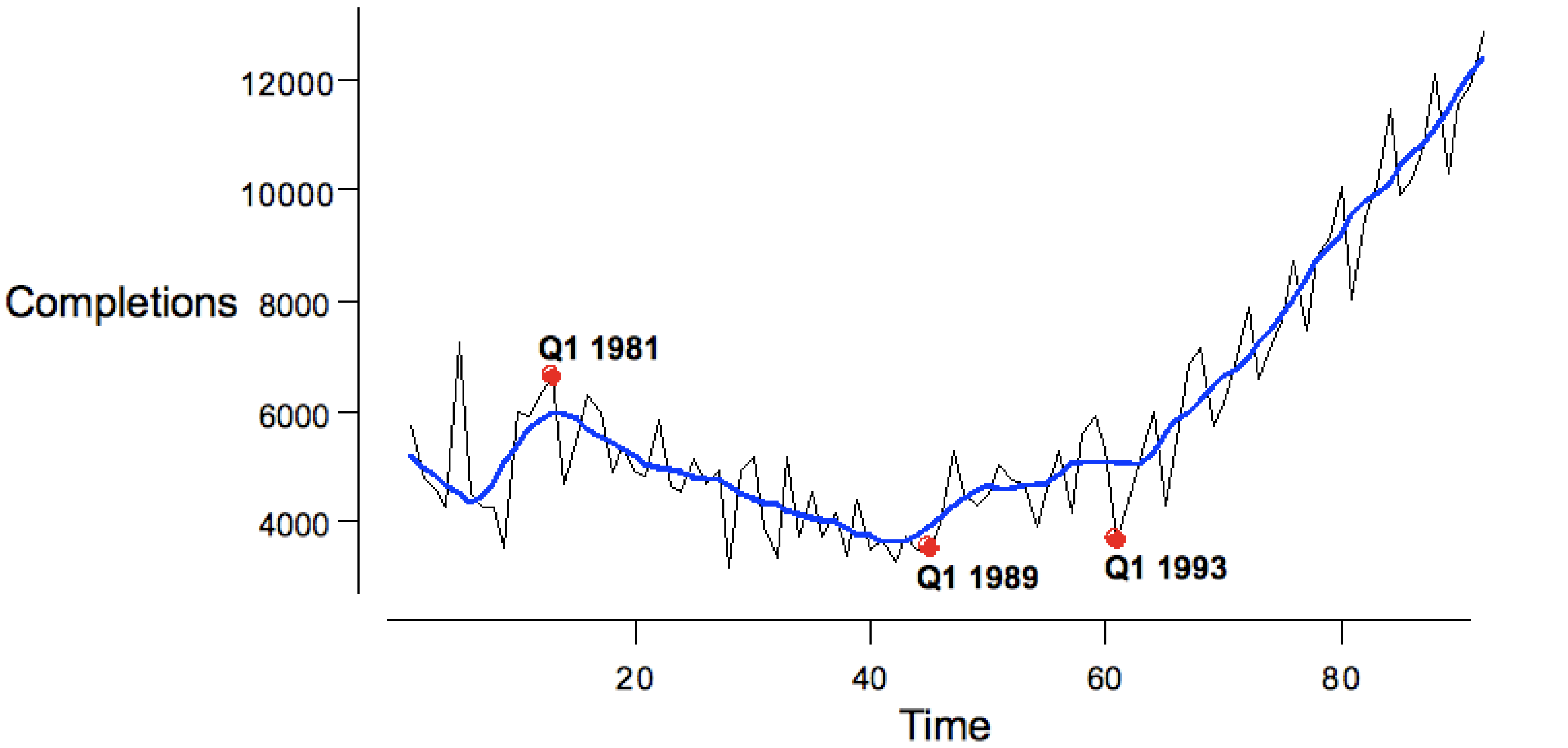
House Completions, Republic of Ireland, Quarterly Totals, 1972-2000

Quarter	*1978*	*1979*	*1980*	*1981*	*1982*	*1983*	*1984*	*1985*
Q1	5777	7276	3538	6642	5981	4859	5129	4947
Q2	4772	4510	6001	4710	4883	5862	4671	5188
Q3	4579	4278	5879	5570	5354	4663	4947	3930
Q4	4243	4274	6383	6314	4894	4564	3195	3360

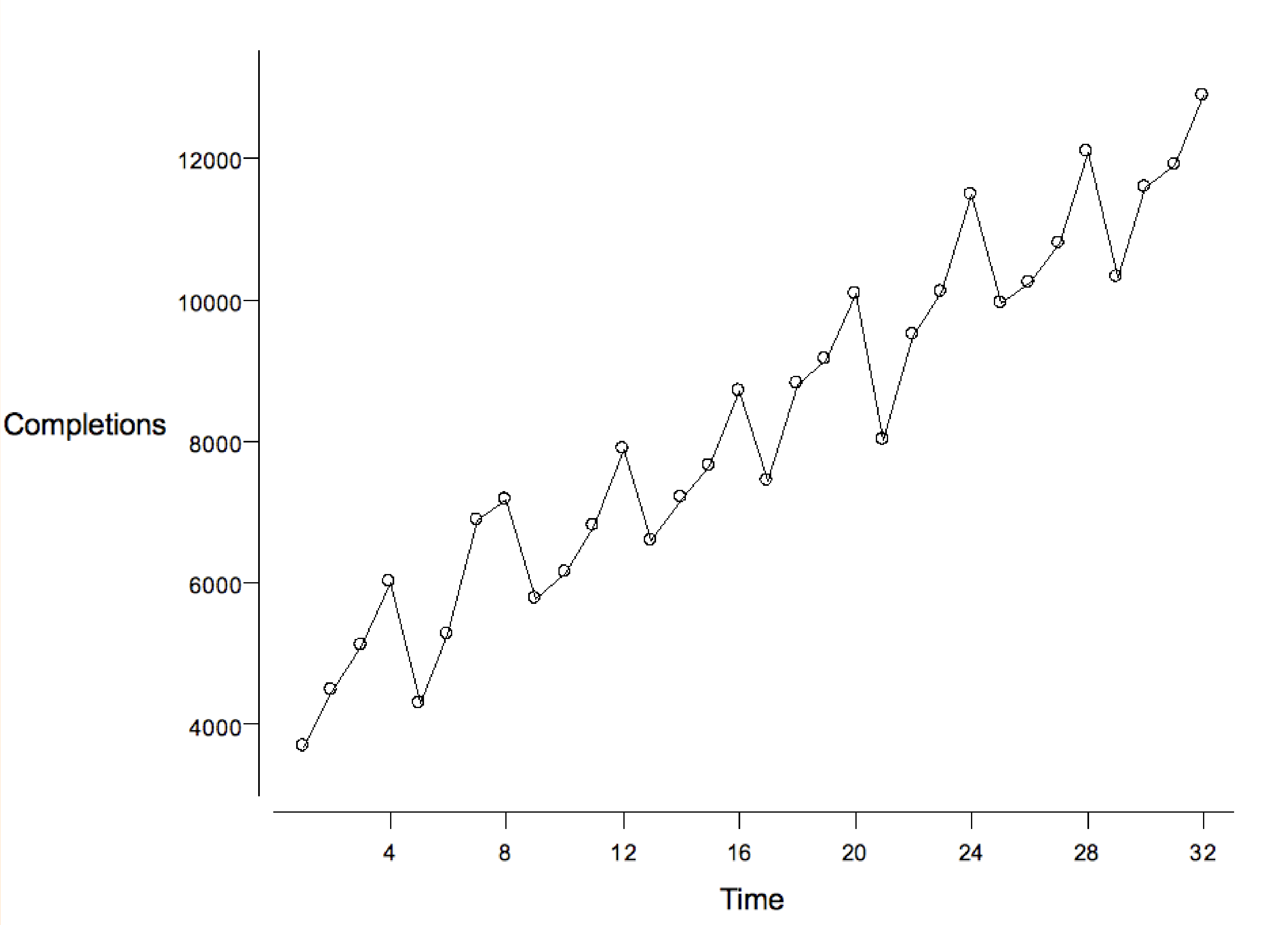
Quarter	*1986*	*1987*	*1988*	*1989*	*1990*	*1991*	*1992*	*1993*
Q1	5186	4144	3682	3554	4296	4692	4155	3684
Q2	3719	3363	3298	3985	4477	3898	5603	4487
Q3	4533	4391	3747	5277	5011	4600	5919	5121
Q4	3726	3478	3477	4484	4752	5282	5305	6009

Quarter	*1994*	*1995*	*1996*	*1997*	*1998*	*1999*	*2000*
Q1	4291	5770	6582	7434	8010	9930	10302
Q2	5266	6149	7203	8799	9506	10227	11590
Q3	6871	6806	7634	9140	10103	10788	11892
Q4	7160	7879	8713	10081	11474	12079	12873

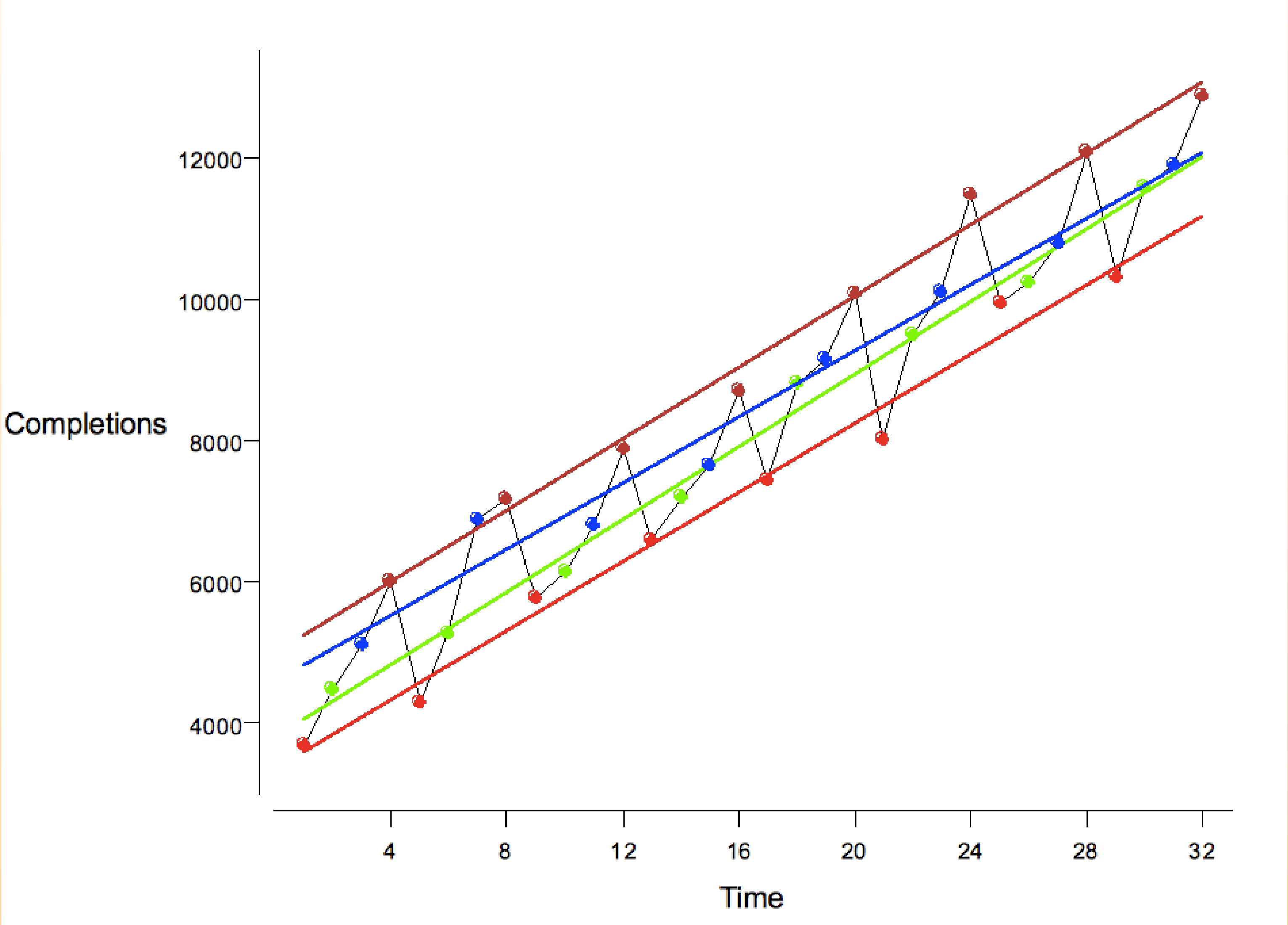
House Completions, Quarterly, 1972-2000



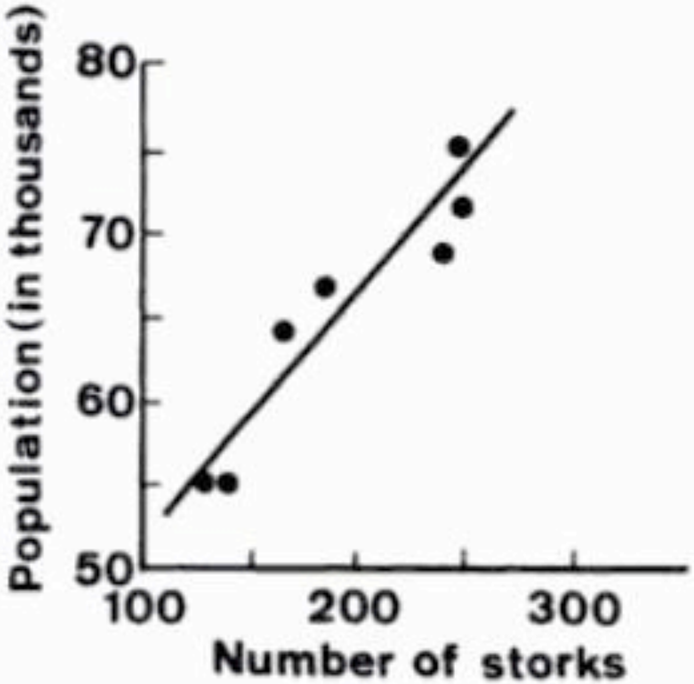
House Completions, Quarterly, 1995-2000



House Completions, Quarterly, 1995-2000



Do Storks Bring Babies?



A plot of the population of Oldenburg in Germany at the end of each year against the number of storks observed in that year, 1930–1936.

R (an oversized calculator)

R (an oversized calculator)

```
> 5+6*3
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1               # and is not executed
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1               # and is not executed
> 1/Inf
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1              # and is not executed
> 1/Inf
[1] 0
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1              # and is not executed
> 1/Inf
[1] 0
> Inf              # Inf stands for infinity
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1               # and is not executed
> 1/Inf
[1] 0
> Inf               # Inf stands for infinity
[1] Inf
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1                # and is not executed
> 1/Inf
[1] 0
> Inf                # Inf stands for infinity
[1] Inf
> exp(-Inf)
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1                # and is not executed
> 1/Inf
[1] 0
> Inf                # Inf stands for infinity
[1] Inf
> exp(-Inf)
[1] 0
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1                # and is not executed
> 1/Inf
[1] 0
> Inf                # Inf stands for infinity
[1] Inf
> exp(-Inf)
[1] 0
> exp(Inf)
```

R (an oversized calculator)

```
> 5+6*3
[1] 23
> exp(log(10))
[1] 10
> pi
[1] 3.141593
> sin(pi)
[1] 1.224647e-16
> sin(2)^2+cos(2)^2 # text after # is treated as a comment
[1] 1               # and is not executed
> 1/Inf
[1] 0
> Inf               # Inf stands for infinity
[1] Inf
> exp(-Inf)
[1] 0
> exp(Inf)
[1] Inf
```

Vectors and Vectorized Calculations

Vectors and Vectorized Calculations

```
> ls()
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"
> x^2        # the square operation is vectorized
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"
> x^2        # the square operation is vectorized
[1]  1  4  9 16 25
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"
> x^2        # the square operation is vectorized
[1]  1  4  9 16 25
> y+x        # the summation is elementwise, vectorized
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"
> x^2        # the square operation is vectorized
[1]  1  4  9 16 25
> y+x        # the summation is elementwise, vectorized
[1] 6 6 6 6 6
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"
> x^2        # the square operation is vectorized
[1]  1  4  9 16 25
> y+x        # the summation is elementwise, vectorized
[1] 6 6 6 6 6
> x^y        # same here
```

Vectors and Vectorized Calculations

```
> ls()
character(0)
> x=1:5 # assigns the vector (1,2,3,4,5) to the object name x
> y=c(5,4,3,2,1) # c for concatenate
> x # typing the object name displays the object content
[1] 1 2 3 4 5
> y
[1] 5 4 3 2 1
> ls()
[1] "x" "y"
> x^2        # the square operation is vectorized
[1]  1  4  9 16 25
> y+x        # the summation is elementwise, vectorized
[1] 6 6 6 6 6
> x^y        # same here
[1]  1 16 27 16  5
```

Functions Creating Vectors

Functions Creating Vectors

```
> 1:10
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)   # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)   # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)  # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
 [1] 1 2 1 2 1 2 1 2 1 2
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
 [1] 1 2 1 2 1 2 1 2 1 2
> rev(1:5)       # reverse the vector 1:5
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
 [1] 1 2 1 2 1 2 1 2 1 2
> rev(1:5)       # reverse the vector 1:5
[1] 5 4 3 2 1
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
 [1] 1 2 1 2 1 2 1 2 1 2
> rev(1:5)       # reverse the vector 1:5
[1] 5 4 3 2 1
> t(1:5) # transpose column vector to row vector (1 row matrix)
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)    # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)   # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)      # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
 [1] 1 2 1 2 1 2 1 2 1 2
> rev(1:5)       # reverse the vector 1:5
[1] 5 4 3 2 1
> t(1:5) # transpose column vector to row vector (1 row matrix)
     [,1] [,2] [,3] [,4] [,5]
```

Functions Creating Vectors

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(-4,4,1)   # sequence from -4 to in increments of 1
[1] -4 -3 -2 -1  0  1  2  3  4
> c(1,2,4,6,7)  # concatenation of 1,2,4,6,7
[1] 1 2 4 6 7
> rep(1,10)     # vector of 10 repeat 1's
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,2),5) # vector of 5 repeat c(1, 2)
 [1] 1 2 1 2 1 2 1 2 1 2
> rev(1:5)      # reverse the vector 1:5
[1] 5 4 3 2 1
> t(1:5) # transpose column vector to row vector (1 row matrix)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
```

Data Modes

We can also use character or logic data in vectors

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
```

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
> x
```

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
> x
[1] "abc"    "ABC"    "xyzXYZ"
```

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
> x
[1] "abc"    "ABC"    "xyzXYZ"
> Y=c(TRUE,T,FALSE,F) # example of a logic vector
```

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
> x
[1] "abc"    "ABC"    "xyzXYZ"
> Y=c(TRUE,T,FALSE,F) # example of a logic vector
> Y
```

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
> x
[1] "abc"    "ABC"    "xyzXYZ"
> Y=c(TRUE,T,FALSE,F) # example of a logic vector
> Y
[1]  TRUE  TRUE FALSE FALSE
```

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
> x
[1] "abc"    "ABC"    "xyzXYZ"
> Y=c(TRUE,T,FALSE,F) # example of a logic vector
> Y
[1]  TRUE  TRUE FALSE FALSE
```

T and TRUE are equivalent, same with F and FALSE

Data Modes

We can also use character or logic data in vectors

```
> x=c("abc","ABC","xyzXYZ") # example of a character vector
> x
[1] "abc"    "ABC"    "xyzXYZ"
> Y=c(TRUE,T,FALSE,F) # example of a logic vector
> Y
[1]  TRUE  TRUE FALSE FALSE
```

T and TRUE are equivalent, same with F and FALSE

Note that we use no quotes on T, TRUE, F and FALSE

Logical Operators

Symbol	Function	Symbol	Function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

Logical Operators

Symbol	Function	Symbol	Function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

```
> 1:5<=3
```

Logical Operators

Symbol	Function	Symbol	Function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

```
> 1:5<=3
[1]  TRUE  TRUE  TRUE FALSE FALSE
```

Logical Operators

Symbol	Function	Symbol	Function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

```
> 1:5<=3
[1]  TRUE  TRUE  TRUE FALSE FALSE
> "abc"<"a" # logical operations work on character data as well
```

Logical Operators

Symbol	Function	Symbol	Function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

```
> 1:5<=3
[1]  TRUE  TRUE  TRUE FALSE FALSE
> "abc"<"a" # logical operations work on character data as well
[1] FALSE
```

Logical Operators

Symbol	Function	Symbol	Function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

```
> 1:5<=3
[1]  TRUE  TRUE  TRUE FALSE FALSE
> "abc"<"a" # logical operations work on character data as well
[1] FALSE
> "abc"<"abd" # lexicographical ordering
```

Logical Operators

Symbol	Function	Symbol	Function
<	less than	&&	logical AND
>	greater than	||	logical OR
<=	less than or equal to	!	logical NOT
>=	greater than or equal to		
==	equal to		
!=	not equal to		

```
> 1:5<=3
[1]  TRUE  TRUE  TRUE FALSE FALSE
> "abc"<"a" # logical operations work on character data as well
[1] FALSE
> "abc"<"abd" # lexicographical ordering
[1] TRUE
```

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical $\leftarrow$ numeric $\leftarrow$ character

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical $\leftarrow$ numeric $\leftarrow$ character

```
> c(T,F,0)
```

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical ← numeric ← character

```
> c(T,F,0)
[1] 1 0 0
```

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical ← numeric ← character

```
> c(T,F,0)
[1] 1 0 0
> T+3 # in arithmetic expression T & F
```

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical ← numeric ← character

```
> c(T,F,0)
[1] 1 0 0
> T+3 # in arithmetic expression T & F
[1] 4 # are interpreted as 1 & 0, respectively
```

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical ← numeric ← character

```
> c(T,F,0)
[1] 1 0 0
> T+3 # in arithmetic expression T & F
[1] 4 # are interpreted as 1 & 0, respectively
> c(T,3,"abc")
```

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical ← numeric ← character

```
> c(T,F,0)
[1] 1 0 0
> T+3 # in arithmetic expression T & F
[1] 4 # are interpreted as 1 & 0, respectively
> c(T,3,"abc")
[1] "TRUE" "3"    "abc"
```

Mode Coercion

Mixed mode elements in expressions are coerced to a lowest common denominator mode

mode order: logical ← numeric ← character

```
> c(T,F,0)
[1] 1 0 0
> T+3 # in arithmetic expression T & F
[1] 4 # are interpreted as 1 & 0, respectively
> c(T,3,"abc")
[1] "TRUE" "3"    "abc"
```

When in doubt, experiment!!

Subvectors of Vectors

Subvectors of Vectors

```
> y
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
[1] 2 1     # while the rest are returned
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
[1] 2 1     # while the rest are returned
> y>3
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
[1] 2 1     # while the rest are returned
> y>3
[1]  TRUE  TRUE FALSE FALSE FALSE
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
[1] 2 1     # while the rest are returned
> y>3
[1]  TRUE  TRUE FALSE FALSE FALSE
> y[y>3] # we can also extract desired index positions
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
[1] 2 1     # while the rest are returned
> y>3
[1]  TRUE  TRUE FALSE FALSE FALSE
> y[y>3] # we can also extract desired index positions
[1] 5 4  # by specifying a logic vector of same length as
y
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
[1] 2 1     # while the rest are returned
> y>3
[1]  TRUE  TRUE FALSE FALSE FALSE
> y[y>3] # we can also extract desired index positions
[1] 5 4  # by specifying a logic vector of same length as
y
> y[c(T,T,F,F,F)] # this is an equivalent extraction
```

Subvectors of Vectors

```
> y
[1] 5 4 3 2 1
> y[c(5,3,1)] # subvectors can be extracted by giving
[1] 1 3 5     # the index positions as a vector
> y[3:5]
[1] 3 2 1
> y[6] # an nonexisting index position returns NA
[1] NA
> y[-(1:3)] # negative index positions are omitted
[1] 2 1     # while the rest are returned
> y>3
[1]  TRUE  TRUE FALSE FALSE FALSE
> y[y>3] # we can also extract desired index positions
[1] 5 4  # by specifying a logic vector of same length as
y
> y[c(T,T,F,F,F)] # this is an equivalent extraction
[1] 5 4  # we get those elements with index TRUE or T
```

Matrices

Matrices

```
> mat=cbind(x,y) # cbind combines vectors of same length
> mat            # vertically positioned next to each other
     x y
[1,] 1 5
[2,] 2 4
[3,] 3 3
[4,] 4 2
[5,] 5 1

> mat=cbind(x,y,y^2)
> mat
     x y
[1,] 1 5 25
[2,] 2 4 16
[3,] 3 3  9
[4,] 4 2  4
[5,] 5 1  1
```

Matrices: `dimnames`

```
> dimnames(mat)
[[1]]
NULL

[[2]]
[1] "x" "y" ""
```

Note that the 5 rows of `mat` don't have names, columns 1 & 2 have the original vector names but column 3 has an empty string name. Also note the list nature result of `dimnames(mat)`, list of vectors unequal in length. More on lists later.

```
> dimnames(mat)[[2]][3]="y2"
> mat[1,]
 x  y y2
 1  5 25
```

Sub matrices of Matrices

```
> mat[1:3,1:2]
     x y
[1,] 1 5
[2,] 2 4
[3,] 3 3

> mat[,-2]
     x y2
[1,] 1 25
[2,] 2 16
[3,] 3  9
[4,] 4  4
[5,] 5  1
```

Character Matrices

```
> letters[1:3]
[1] "a" "b" "c"
> LETTERS[1:3]
[1] "A" "B" "C"
> ABC=cbind(letters[1:3],LETTERS[1:3])
> ABC
     [,1] [,2]
[1,] "a"  "A"
[2,] "b"  "B"
[3,] "c"  "C"
> cbind(1:3,letters[1:3])
     [,1] [,2]
[1,] "1"  "a"
[2,] "2"  "b"
[3,] "3"  "c"
> # Numbers were coerced to characters
> # Elements of matrices have to be of same mode.
```

Matrices via `rbind`

```
> rmat=rbind(1:3,2:4,5:7)
> rmat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    3    4
[3,]    5    6    7
> mat[1:3,]
     x y y2
[1,] 1 5 25
[2,] 2 4 16
[3,] 3 3  9
> mat[1:3,]+rmat
     x y y2
[1,] 2 7 28
[2,] 4 7 20
[3,] 8 9 16
```

Matrices via `matrix`

```
> matrix(1:12,ncol=3,nrow=4)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> matrix(1:12,ncol=3,nrow=4,byrow=T)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> # byrow = F is default
```

Objects

- ‘Vectors’ of
  - numbers
  - logical values
  - character strings
  - complex numbers
- Matrices and general n-way arrays
- ‘Lists’ – arbitrary collections of objects of any type
- Data frames – lists with a rectangular structure
- ‘Connections’ – files and similar things
- S functions and other the language objects

Finding objects

- R looks for objects in a sequence of places known as the “search path”.
- The search path is a sequence of “environments” beginning with the “Global Environment”
- You can inspect it at any time (and you should) by the **`search()`** function (or from the Misc menu)
- The **`attach()`** function allows copies of objects to be placed on the search path as individual components
- **`detach()`** removes items from the search path

Looking at the search path: Example

```
> attach(Cars93)
> search()
 [1] ".GlobalEnv"        "Cars93"            "package:methods"
 [4] "package:graphics" "package:utils"     "package:RODBC"
 [7] "package:stats"    "package:MASS"      "Autoloads"
[10] "package:base"
> objects(2)
 [1] "AirBags"            "Cylinders"          "DriveTrain"
 [4] "EngineSize"         "Fuel.tank.capacity" "Horsepower"
....
[22] "RPM"                "Turn.circle"        "Type"
[25] "Weight"             "Wheelbase"          "Width"
> names(Cars93)
 [1] "Manufacturer"       "Model"              "Type"
 [4] "Min.Price"          "Price"              "Max.Price"
 [7] "MPG.city"           "MPG.highway"        "AirBags"
....
[22] "Turn.circle"        "Rear.seat.room"     "Luggage.room"
[25] "Weight"             "Origin"             "Make"

> find(Cars93)
[1] "package:MASS"
```

Factors

- A factor is a vector used to specify a classification
- It is not a character vector, but in some contexts may be used as one
- It is not a numeric vector, but may be used as an index (see later)
- When used in fitting statistical models, it generates the appropriate machinery for a classificaton (e.g. an analysis of variance)
- By default, when a data frame is set up, non-numeric vectors are made into factors.

Data input more formally

- Most data sets will be held as data frames
- The most frequently used input functions are
  - `read.table`
  - `read.csv`
- Simplest way to read data from an excel file:
- Save the sheet you want to become the data frame as a `csv` file
- Use `data <- read.csv("data file.csv")`
- Example: The SS data from Don Heales

Atomic data types

- Numerical vectors
- Character vectors
- Logical vectors
- (Complex number vectors)
  - If given a dimension vector may be treated as a multi-way array (or as a vector, still)
  - If given names, components may be referred to by name or by index

Basic computations with numerical vectors

- Element-by-element operation
- Short vectors are 'recycled' to match long ones:
- Some functions take vectors of values and produce results of the same length:
  - **`sin, cos, tan, asin, acos, atan, log, exp, Arith,`** …
- Some functions return a single value:
  - **`sum, mean, max, min, prod,`** …
- Some functions are a bit special:
  - **`cumsum, sort, range, pmax, pmin,`** …

An artificial example 1

```
> x <- runif(10)
> x
 [1] 0.38520632 0.32295045 0.39109670 0.58721717 0.51926045
 [6] 0.59091389 0.01508866 0.49567887 0.88141482 0.99584085
> y <- 2*x + 1        # recycling short vectors
> y
 [1] 1.770413 1.645901 1.782193 2.174434 2.038521 2.181828
 [7] 1.030177 1.991358 2.762830 2.991682


> z <- (x - mean(x))/sd(x)    # see also 'scale'
> z
 [1] -0.479301232 -0.703218414 -0.458115177  0.247276059
 [5]  0.002854489  0.260572108 -1.810512300 -0.081961974
 [9]  1.305423788  1.716982654

> mean(z)
[1] 4.440892e-17
> sd(z)
[1] 1
>
```

An artificial example 2

```
> m <- rnorm(12)
> m
 [1] -1.3350035  0.3969718 -0.3706427  2.2452284  0.3771235
   0.2324071 -0.7744801
 [8]  1.8709943 -0.4611989  1.1632276  1.0750105 -0.6854324
> dim(m) <- c(3, 4)
> m
           [,1]      [,2]       [,3]       [,4]
[1,] -1.3350035 2.2452284 -0.7744801  1.1632276
[2,]  0.3969718 0.3771235  1.8709943  1.0750105
[3,] -0.3706427 0.2324071 -0.4611989 -0.6854324
> dimnames(m) <- list(letters[1:3], LETTERS[1:4])
> m
           A         B          C          D
a -1.3350035 2.2452284 -0.7744801  1.1632276
b  0.3969718 0.3771235  1.8709943  1.0750105
c -0.3706427 0.2324071 -0.4611989 -0.6854324
```

An artificial example 2 (Cont'd)

```
> m[2,3]
[1] 1.870994
> m["b", "C"]
[1] 1.870994
> m[8]
    <NA>
1.870994
> names(m) <- letters[1:12]
> m
           A         B          C          D
a -1.3350035 2.2452284 -0.7744801  1.1632276
b  0.3969718 0.3771235  1.8709943  1.0750105
c -0.3706427 0.2324071 -0.4611989 -0.6854324
attr(,"names")
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
> m[8]
       h
1.870994
```

Indexing in general

Indexing may be done by

- A vector of positive integers – to indicate inclusion
- A vector of negative integers – to indicate exclusion
- A vector of logical values – to indicate which are in and which are out
- A vector of names – if the object has a names attribute
  - If a zero index occurs on the right no element is selected; if a zero index occurs on the left, no assignment is made
  - An empty index position stands for "the lot".

An artificial example 2 (Cont'd)

```
> names(m) <- NULL
> m
           A         B          C          D
a -1.3350035 2.2452284 -0.7744801  1.1632276
b  0.3969718 0.3771235  1.8709943  1.0750105
c -0.3706427 0.2324071 -0.4611989 -0.6854324
> m[, "A"] <- 0
> m
  A         B          C          D
a 0 2.2452284 -0.7744801  1.1632276
b 0 0.3771235  1.8709943  1.0750105
c 0 0.2324071 -0.4611989 -0.6854324
> m["a", ] <- 0
> m
  A         B          C          D
a 0 0.0000000  0.0000000  0.0000000
b 0 0.3771235  1.8709943  1.0750105
c 0 0.2324071 -0.4611989 -0.6854324
> m[] <- 1:12
> m
  A B C  D
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12
```

An artificial example 3

```
> x <- sample(1:5, 20, rep=T)
> x
 [1] 3 4 1 1 2 1 4 2 1 1 5 3 1 1 1 2 4 5 5 3
> x == 1
 [1] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE
[10]  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[19] FALSE FALSE
> ones <- (x == 1)     # parentheses unnecessary
> x[ones] <- 0
> x
 [1] 3 4 0 0 2 0 4 2 0 0 5 3 0 0 0 2 4 5 5 3
> others <- (x > 1)    # parentheses unnecessary
> y <- x[others]
> y
 [1] 3 4 2 4 2 5 3 2 4 5 5 3

> which(x > 1)
 [1]  1  2  5  7  8 11 12 16 17 18 19 20
```

Sorting

- Usually best done indirectly:
  - Find an index vector that achieves the sort operation and
  - Use it for all vectors that need to remain together
- '**`order`**' is a function that allows sorting with tie-breaking:
- Find an index vector that:
  - arranges the first of its arguments in increasing order,
  - ties are broken by the second argument,

An artificial example 4

```
> x <- sample(1:5, 20, rep=T)
> y <- sample(1:5, 20, rep=T)
> z <- sample(1:5, 20, rep=T)
> xyz <- rbind(x, y, z)
> dimnames(xyz)[[2]] <- letters[1:20]
> xyz
  a b c d e f g h i j k l m n o p q r s t
x 2 1 5 2 1 5 5 3 5 4 5 4 3 4 1 2 3 3 3 1
y 1 1 5 5 5 3 4 4 4 5 2 4 4 4 4 3 5 1 2 3
z 2 1 2 4 3 3 5 4 5 1 4 4 3 5 5 4 3 3 5 5
```

An artificial example 4 (Cont'd)

```
> o <- order(x, y, z)
> xyz[, o]
  b t o e a p d r s m h q l n j k f g i c
x 1 1 1 1 2 2 2 3 3 3 3 3 4 4 4 5 5 5 5 5
y 1 3 4 5 1 3 5 1 2 4 4 5 4 4 5 2 3 4 4 5
z 1 5 5 3 2 4 4 3 5 3 4 3 4 5 1 4 3 5 5 2

> xyz         # reminder
  a b c d e f g h i j k l m n o p q r s t
x 2 1 5 2 1 5 5 3 5 4 5 4 3 4 1 2 3 3 3 1
y 1 1 5 5 5 3 4 4 4 5 2 4 4 4 4 3 5 1 2 3
z 2 1 2 4 3 3 5 4 5 1 4 4 3 5 5 4 3 3 5 5
```

Non-atomic (recursive) objects

- Lists
  - an ordered collection of components,
  - components may be arbitrary S objects (including lists)
  - single bracket notation for sublists,
  - double bracket notation for individual components
- Data frames
  - are lists, and may be treated as such
  - have a rectangular structure as well, and may be treated as matrices
  - usually components may only be vectors or factors

Examples

```
> L1 <- list(x = sample(1:5, 20, rep=T), y =
  rep(letters[1:5], 4), z = rpois(20, 1))
> L1
$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1

$y
 [1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a"
  "b" "c"
[14] "d" "e" "a" "b" "c" "d" "e"

$z
 [1] 1 3 0 0 3 1 3 1 0 1 2 2 0 3 1 1 0 1 2 0
```

- Three equivalent ways of accessing the first component

```
> L1[["x"]]
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1[[1]]
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

- A sublist consisting of the first component only

```
> L1[1]
$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

Data Frame example

```
> D1 <- as.data.frame(L1)      # or just data.frame(L1)
> D1                           # numeric, factor, numeric
   x y z
1  2 a 1
2  1 b 3
3  1 c 0
4  4 d 0
5  5 e 3
6  3 a 1
...

16 3 a 1
17 3 b 0
18 2 c 1
19 3 d 2
20 1 e 0
```

```
> fm <- glm(z ~ x + y, poisson, D1, trace = T)
Deviance = 23.38426 Iterations - 1
Deviance = 19.59917 Iterations - 2
Deviance = 19.41966 Iterations - 3
Deviance = 19.41866 Iterations - 4
Deviance = 19.41866 Iterations - 5

> mode(fm)
[1] "list"
> class(fm)
[1] "glm" "lm"


> dropterm(fm, test="Chisq")
...
Single term deletions

Model:
z ~ x + y
       Df Deviance    AIC    LRT Pr(Chi)
<none>      19.419 65.227
x       1   19.672 63.481  0.254  0.6145
```

Dates and times

- R has several mechanisms available for the representation of dates and times. The 'standard' one, however is the **`POSIXct/POSIXlt`** suite of functions and object possibilities
- objects of (old) class "**`POSIXct`**" are numeric vectors with each component the number of seconds since the start of 1970. Such objects are suitable for inclusion in data frames, for example.
- objects of (old) class "**`POSIXlt`**" are lists with the separate parts of the date/time held as separate components, plus a few.

Conversion from one form to another

- **`as.POSIXlt(obj`**) converts from **`POSIXct`** to **`POSIXlt`**
- **`as.POSIXct(obj`**) converts from **`POSIXlt`** to **`POSIXct`**
- **`strptime(char,form)`** generates **`POSIXlt`** objects from suitable character string vectors. You must specify the format used in the input character strings
- **`format(obj,form)`** generates character string vectors from **`POSIXlt`** or **`POSIXct`** objects. You must specify the format to be used in the output character strings **`as.character(obj`**) also generates character string vectors like format(,), but only to the ISO standard time/date format.
- For formatting details see, for example **`?strptime`**

Arithmetic on `POSIXt` objects

- Some arithmetic operations are allowed on date/time objects (POSIXlt or POSIXct). These are
  - `obj + number`
  - `obj - number`
  - `obj1 <lop> obj2`
  - `obj1 - obj2`
- In the first two cases, ‘`number`’ is a number of seconds and each date is augmented by this number of seconds. If you wish to augment by days you need to work with multiples of `60*60*24`.
- In the second case ‘`<lop>`’ is a logical operator and the result is a logical vector
- In the third case the result is a ‘`difftime`’ object, represented as a number of seconds ‘time difference’.

On what day of the week were you born and for how many seconds have you lived?

```
> myBday <- strptime("13-Feb-1944", "%d-%b-%Y")
> class(myBday)
[1] "POSIXt"  "POSIXlt"
> myBday
[1] "1944-02-13"
> weekdays(myBday)
[1] "Sunday"


> Sys.time() – myBday  ### Out of date by now…
Time difference of 22061.61 days

> as.numeric(Sys.time())
[1] 1089261846
> as.numeric(myBday)
[1]  0  0  0 13  1 44  0 43  0
> as.numeric(as.POSIXct(myBday))
[1] -816861600

> as.numeric(Sys.time()) - as.numeric(as.POSIXct(myBday))
[1] 1906124200
```

Notes on lists and their allies

- S is a function language:
  - Most S operations amount to evaluating a function, which return a single object as the result
  - If several pieces of information are needed from the function, the natural way to return them is as a list, perhaps with a special class and other attributes
  - In S, information is stored and transmitted as objects, which in the case of data are often lists
- Other more special recursive objects are available, such as language objects and functions.

The '`apply`' family

- Four members: **`lapply`**, **`sapply`**, **`tapply`**, **`apply`**
  - **`lapply`**: takes any structure, gives a list of results
  - **`sapply`**: like lapply, but 'simplifies' the result if possible
  - **`apply`**: only used for arrays
  - **`tapply`**: used for 'ragged arrays': vectors with an indexing specified by one or more factors.
- Used for

a) efficiency relative to explicit loops and

b) convenience

Example 5: playing hookey in Walgett

```
> find(quine)
[1] "package:MASS"
> data(quine)
> find(quine)
[1] ".GlobalEnv"   "package:MASS"
> names(quine)
[1] "Eth"  "Sex"  "Age"  "Lrn"  "Days"
> tab <- xtabs(Days ~ Sex + Age + Lrn + Eth, quine)

> tab
, , Lrn = AL, Eth = A

   Age
Sex F0  F1  F2  F3
  F  85  57   2 131
  M  65  21 192 190
```

```
, , Lrn = SL, Eth = A

   Age
Sex F0  F1  F2  F3
  F   3 226 291   0
  M  27  27 148   0

, , Lrn = AL, Eth = N

   Age
Sex F0  F1  F2  F3
  F  74  66   1 135
  M  32   7  64 191

, , Lrn = SL, Eth = N

   Age
Sex F0  F1  F2  F3
  F  25  66  56   0
  M  90  43  88   0
```

```
> dim(tab)
[1] 2 4 2 2
> tab1 <- apply(tab, c(2,3,4), sum)
> tab1
, , Eth = A

    Lrn
Age   AL  SL
  F0 150  30
  F1  78 253
  F2 194 439
  F3 321   0

, , Eth = N

    Lrn
Age   AL  SL
  F0 106 115
  F1  73 109
  F2  65 144
```

Example 6: new cars in 1993, again

```
> data(Cars93)
> names(Cars93)
 [1] "Manufacturer"       "Model"
 [3] "Type"               "Min.Price"
 [5] "Price"              "Max.Price"
 [7] "MPG.city"           "MPG.highway"
 [9] "AirBags"            "DriveTrain"
[11] "Cylinders"          "EngineSize"
[13] "Horsepower"         "RPM"
[15] "Rev.per.mile"       "Man.trans.avail"
[17] "Fuel.tank.capacity" "Passengers"
[19] "Length"             "Wheelbase"
[21] "Width"              "Turn.circle"
[23] "Rear.seat.room"     "Luggage.room"
[25] "Weight"             "Origin"
[27] "Make"
> with(Cars93, table(Origin, Type))

         Compact Large Midsize Small Sporty Van
 USA       7      11    10       7     8     5
 non-USA   9       0    12      14     6     4
```

```
> attach(Cars93)
> table(Origin, Type)
         Type
Origin    Compact Large Midsize Small Sporty Van
  USA       7      11    10       7     8     5
  non-USA   9       0    12      14     6     4

> tapply(Weight, list(Origin, Type), mean)
         Compact    Large  Midsize    Small   Sporty  Van
USA     2786.429 3695.455 3355.500 2350.714 3039.375 3779
non-USA 3020.556       NA 3437.083 2293.929 2713.333 3895

> av.gpm <- function(x) mean(100/x)
> round(tapply(MPG.city, list(Origin, Type), av.gpm), 3)
        Compact Large Midsize Small Sporty   Van
USA       4.279  5.48   5.110 3.659  4.921 6.065
non-USA   4.561    NA   5.207 3.370  4.464 5.719
```

What kind of components do we have?

```
> sapply(Cars93, class)
     Manufacturer             Model              Type
         "factor"          "factor"          "factor"
        Min.Price             Price         Max.Price
        "numeric"         "numeric"         "numeric"
         MPG.city       MPG.highway           AirBags
        "integer"         "integer"          "factor"
       DriveTrain         Cylinders        EngineSize
         "factor"          "factor"         "numeric"
       Horsepower               RPM      Rev.per.mile
        "integer"         "integer"         "integer"
   Man.trans.avail Fuel.tank.capacity       Passengers
         "factor"         "numeric"         "integer"
           Length         Wheelbase             Width
        "integer"         "integer"         "integer"
      Turn.circle    Rear.seat.room      Luggage.room
        "integer"         "numeric"         "integer"
           Weight            Origin              Make
        "integer"          "factor"          "factor"
```

Getting stuff in

- `scan(…)` offers a low-level reading facility
- `read.table(…)` can be used to read data frames from formatted text files
- `read.csv(…)` can be used to read data frames from comma separated variable files.
- When reading from excel files, the simplest method is to save each worksheet separately as a csv file and use `read.csv(…)` on each. (Wimpy!)
- A better way is to open a data connection to the excel file directly and use the odbc facilities

Putting stuff out (mostly data frames)

- There is no ***write.csv***
- **write.table** can be used to create text or csv versions on file:

```
> con <- file("myData.csv", "w+")
> write.table(myData, con, sep = ",")
> close(con)

> write.table(myData, "myData.txt")
```

- **print(…)** and **cat(…)** can write to files at a relatively primative level.

Using the RODBC tools

```
> find(odbcConnectExcel)
[1] "package:RODBC"

> con <- odbcConnectExcel("Trawl Data Sets.xls")

> con
RODB Connection 1
Details:
  case=nochange
  DBQ=d:\Data\Monaro\R course\Trawl Data Sets.xls
  DefaultDir=d:\Data\Monaro\R course
  Driver={Microsoft Excel Driver (*.xls)}
  DriverId=790
  MaxBufferSize=2048
  PageTimeout=5
>
```

What are the tables available?

```
> sqlTables(con)
                                 TABLE_CAT TABLE_SCHEM
1 d:\\Data\\Monaro\\R course\\Trawl Data Sets        <NA>
2 d:\\Data\\Monaro\\R course\\Trawl Data Sets        <NA>
3 d:\\Data\\Monaro\\R course\\Trawl Data Sets        <NA>
4 d:\\Data\\Monaro\\R course\\Trawl Data Sets        <NA>

    TABLE_NAME   TABLE_TYPE REMARKS
1      Hopper$ SYSTEM TABLE    <NA>
2 SortingTray$ SYSTEM TABLE    <NA>
3      SS0297$ SYSTEM TABLE    <NA>
4      SS0897$ SYSTEM TABLE    <NA>

> Hopper <- sqlFetch(con, "Hopper")
> SortingTray <- sqlFetch(con, "SortingTray")
> SS0297 <- sqlFetch(con, "SS0297")
> SS0897 <- sqlFetch(con, "SS0897")
```

Stick the Southern Surveyor data together

```
> names(SS0297)
[1] "Cruise"    "Station"   "Box"        "Spcode"
[5] "Subsample" "No"        "Wt"         "Comment"
> names(SS0897)
[1] "Cruise"   "Station" "Box"      "Spcode"   "No"
[6] "Wt"       "Comment"

> nam <- intersect(names(SS0297), names(SS0897))
> nam
[1] "Cruise"   "Station" "Box"      "Spcode"   "No"
[6] "Wt"       "Comment"

> SSData <- rbind(SS0297[, nam], SS0897[, nam])
> names(SSData)
[1] "Cruise"   "Station" "Box"      "Spcode"   "No"
[6] "Wt"       "Comment"
```

The low-level input functon: `scan()`

- The simplest use of scan is to read a vector of numbers:

```
> vec <- scan()
1: 22 35 1.7 2.5e+01 77
6:
Read 5 items
> vec
[1] 22.0 35.0  1.7 25.0 77.0
```

A blank line (two returns) signals the end of the input

Reading characters with `scan()`

```
> chr <- scan(what = "", sep = "\n")
1: This is the first string
2: This is the second
3: and another
4: that's all we need for now
5:
Read 4 items
> chr
[1] "This is the first string"
[2] "This is the second"
[3] "and another"
[4] "that's all we need for now"
>
```

Mixed characters and numbers

```
> lis <- scan(what = list(flag = "", x = 0, y = 0))
1: a 10 3.6
2: a 20 2.4
3: a 30 1.2
4: b 10 5.4
5: b 20 3.7
6: b 30 2.4
7:
Read 6 records
> dat <- as.data.frame(lis)
> dat
  flag  x   y
1    a 10 3.6
2    a 20 2.4
3    a 30 1.2
4    b 10 5.4
5    b 20 3.7
6    b 30 2.4
```

How does R work and how do I work with it?

- R works best if you have a dedicated folder for each separate project – the working folder. Put all data files, &c, in the working folder (or in subfolders of it)
- Start R in the working folder: three ways
  - make an R shortcut pointing to the folder and double-click
  - double-click on the `.RData` file in the folder, when it exists
  - double-click any R shortcut and use `setwd()`
- Load history if you want to have access to what you did last time
- Work on the project – your objects can be automatically saved in the `.RData` file (next slide)
- To quit – use `q()` or just kill the window

How does R work and how do I work with it?

- R creates its objects in memory and saves them in a single file called `.RData` (by default)
- Commands are recorded in an `.Rhistory` file
- Commands may be recalled and reissued using up- and down-arrow in an obvious way
- Recalled commands may be edited using a ‘Windows familiar’ fashion, (with a few extras)
- Flawed commands may be abandoned either by `<Esc>` or `<Home Ctrl-K>` or `<Home #>`
- Copy-and-paste from a ‘script’ file (`<Ctrl-C>`, `<Ctrl-V>`)
- Copy-and-paste from the history window is usual for recalling several commands at once or multiple-line commands.

Customisation

- Some preferences can be changed from the ‘Edit’ menu under ‘GUI Preferences’
- Global actions to be taken every time R is used on this machine may be set in a file `R_HOME/etc/Rprofile.site`
- Actions to happen automatically every time this working folder is used can be set by defining a .First function. e.g.

```
.First <- function() {
    library(MASS)
    library(lattice)
    options(length = 99999)
    loadhistory()
}
```

- Actions to happen automatically every time a session in this working folder ends may be set by a .Last function. e.g.