

# Applied Econometrics

with 

Chapter 2

## Basics

Basics

# Overview

# Basics

## R at various levels:

- Standard arithmetic: R can be used as a (sophisticated) calculator.
- Graphical system: R can create graphics on many devices.
- Full-featured programming language.
- R connects to other languages, programs, and data bases, and also to the operating system.

## In this chapter:

- Illustration of a few typical uses of R.
- Sophisticated shortcuts are avoided here.
- Often solutions are not unique.  
→ Explore alternative solutions by reusing ideas.

Basics

# R as a Calculator

# R as a calculator

**Standard arithmetic operators:** +, -, \*, /, and ^ are available, where  $x^y$  yields  $x^y$ .

```
R> 1 + 1
```

```
[1] 2
```

```
R> 2^3
```

```
[1] 8
```

**Details:** In the output, [1] indicates the position of the first element of the vector returned by R. (Not surprising here, where all vectors are of length 1, but will be useful later.)

**Mathematical functions:** R has `log()`, `exp()`, `sin()`, `asin()`, `cos()`, `acos()`, `tan()`, `atan()`, `sign()`, `sqrt()`, `abs()`, `min()`, `max()`, ....

# R as a calculator

```
R> log(exp(sin(pi/4)^2) * exp(cos(pi/4)^2))  
[1] 1
```

## Details:

- `log(x, base = a)` returns the logarithm of `x` to base `a`.
- `a` defaults to `exp(1)`.
- Convenience functions: `log10()` and `log2()`.
- See `?log` for a full list of all options and related functions.

**Further functions:** `gamma()`, `beta()`, and their logarithms and derivatives, are often useful in statistics and econometrics. See `?gamma` for further information.

# Vector arithmetic

**Basic unit:** Vector. All functions above operate directly on vectors.

**Generation of vectors:** e.g., via `c()`, where `c` stands for “combine” or “concatenate”.

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
R> length(x)

[1] 5
```

**Case-sensitivity:** `x` and `X` are distinct.

**Assignment operators:**

- `<-` (mimicking a single arrow symbol).
- `=` may also be used at the user level, but `<-` is preferred for programming.

# Vector arithmetic

Use  $x$  in subsequent calculations:

```
R> 2 * x + 3
```

```
[1] 6.60 9.28 11.00 179.34 29.00
```

```
R> 5:1 * x + 1:5
```

```
[1] 10.00 14.56 15.00 180.34 18.00
```

```
R> log(x)
```

```
[1] 0.5878 1.1442 1.3863 4.4793 2.5649
```

## Details:

- First statement: scalars 2 and 3 are recycled to the length of  $x$ .
- Second statement:  $x$  is multiplied element-wise by the vector 1 : 5 (the integers from 1 through 5; see below) and then the vector 5 : 1 is added element-wise.
- Third statement: Application of mathematical functions.



# Subsetting vectors

**Subsets of vectors:** Operator `[` can be used in several ways.

```
R> x[c(1, 4)]
```

```
[1] 1.80 88.17
```

```
R> x[-c(2, 3, 5)]
```

```
[1] 1.80 88.17
```

## Details:

- Extract elements by their index.
- Exclude elements with negative index.
- Further specifications are explained later in this chapter.

# Patterned vectors

Vectors with special patterns are needed in statistics and econometrics.

R provides several useful functions for this, including

```
R> ones <- rep(1, 10)
```

```
R> ones
```

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

```
R> even <- seq(from = 2, to = 20, by = 2)
```

```
R> even
```

```
[1] 2 4 6 8 10 12 14 16 18 20
```

```
R> c(ones, even)
```

```
[1] 1 1 1 1 1 1 1 1 1 1 2 4 6 8 10 12 14 16 18 20
```

```
R> trend <- 1981:2005
```

```
R> trend
```

```
[1] 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992  
[13] 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004  
[25] 2005
```

Basics

# Matrix Operations

# Matrix operations

**Creation:** A  $2 \times 3$  matrix containing the elements 1:6, by column, is generated via

```
R> A <- matrix(1:6, nrow = 2)
```

```
R> A
```

```
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

**Alternatively:** Use `nrow` instead of `ncol`.

```
R> matrix(1:6, ncol = 3)
```

```
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

# Basic matrix algebra

**Transpose**  $A^T$  of  $A$  via

```
R> t(A)
```

```
      [,1] [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6
```

**Dimensions:** Access via `dim()`, `nrow()`, and `ncol()`.

```
R> dim(A)
```

```
[1] 2 3
```

```
R> nrow(A)
```

```
[1] 2
```

```
R> ncol(A)
```

```
[1] 3
```

# Basic matrix algebra

**Internally:** Matrices are vectors with an additional dimension attribute enabling row/column-type indexing.

## Indexing:

- $A[i, j]$  extracts element  $a_{ij}$  of matrix  $A$ .
- $A[i, ]$  extracts  $i$ th row.
- $A[, j]$  extracts  $j$ th column.
- Results of these operations are *vectors*, i.e., dimension attribute is dropped (by default).
- $A[i, j, \text{drop} = \text{FALSE}]$  avoids dropping and returns a matrix.

# Basic matrix algebra

## Illustration:

```
R> A1 <- A[1:2, c(1, 3)]
```

```
R> A1
```

```
      [,1] [,2]
[1,]    1    5
[2,]    2    6
```

## Equivalently:

```
R> A[, -2]
```

```
      [,1] [,2]
[1,]    1    5
[2,]    2    6
```

# Basic matrix algebra

**Nonsingularity:** Check determinant or eigenvalues.

```
R> det(A1)
```

```
[1] -4
```

```
R> eigen(A1)
```

```
eigen() decomposition
```

```
$values
```

```
[1] 7.5311 -0.5311
```

```
$vectors
```

```
      [,1] [,2]
```

```
[1,] -0.6079 -0.9562
```

```
[2,] -0.7940  0.2928
```

**Inverse:** `solve()` (if direct computation cannot be avoided).

```
R> solve(A1)
```

```
      [,1] [,2]
```

```
[1,] -1.5  1.25
```

```
[2,]  0.5 -0.25
```



# Basic matrix algebra

**Check:** Using operator for matrix multiplication `%*%`.

```
R> A1 %*% solve(A1)
```

```
      [,1] [,2]  
[1,]    1    0  
[2,]    0    1
```

## Further functionality:

- Adding and subtracting for conformable matrices via `+` and `-`.
- Recycling for non-conformable matrices proceeds along columns.
- Operator `*` returns the element-wise product.
- `kroncker()`: Kronecker product.
- `crossprod()`: Cross product  $A^T B$ .
- `svd()`: Singular-value decomposition.
- `qr()`: QR decomposition.
- `chol()`: Cholesky decomposition.

# Patterned matrices

## Useful functions:

- `diag()`: Create diagonal matrix, or extract diagonal from matrix.
- `upper.tri()` and `lower.tri()`: query the positions of upper or lower triangular elements. Result is matrix of logicals.

```
R> diag(4)
```

```
      [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

```
R> diag(1:3)
```

```
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

```
R> diag(A1)
```

```
[1] 1 6
```

# Combining matrices

**Combination:** `cbind()` and `rbind()` combine matrices by columns or rows. If necessary, arguments are suitably recycled.

```
R> cbind(1, A1)
```

```
      [,1] [,2] [,3]
[1,]    1    1    5
[2,]    1    2    6
```

```
R> rbind(A1, diag(4, 2))
```

```
      [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    4    0
[4,]    0    4
```

Basics

# R as a Programming Language

# R as a programming language

## R:

- Full-featured, interpreted, object-oriented programming language.
- Designed for “programming with data” (Chambers 1998).
- In-depth treatment of programming in S/R: Venables and Ripley (2000).
- German introduction to programming with R: Ligges (2007).
- More technical: “Writing R Extensions” and “R Language Definition” manuals.

# The mode of a vector

**Basic data structure:** Vector.

**Mode:** All elements of a vector must be of the same type; technically, they must be of the same “mode”.

**Examples:** “numeric”, “logical”, and “character” (there are others).

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
```

```
R> mode(x)
```

```
[1] "numeric"
```

# Logical vectors

**Logical vectors:** Contain the logical constants TRUE and FALSE.

**Aliases:** In a fresh session, the aliases T and F are available for compatibility with S (which uses these as the logical constants).

**Recommendation:** Always use TRUE and FALSE (because T and F can be changed and might have other values – e.g., sample size or *F* statistic).

**Example:** Result of comparisons.

```
R> x
```

```
[1] 1.80 3.14 4.00 88.17 13.00
```

```
R> x > 3.5
```

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

# Character vectors

**Character vectors:** For storing strings.

**Typical basic usage:** Assign labels or names to vectors, matrices, etc.

```
R> names(x) <- c("a", "b", "c", "d", "e")
```

```
R> x
```

```
      a      b      c      d      e
1.80  3.14  4.00 88.17 13.00
```

**Advanced character processing:** Not much used here, but R has powerful character-manipulation facilities, e.g., for computations on text documents or command strings.



# More on subsetting

**Up to now:** Only numeric indices have been introduced.

**Character subsetting:** Can be used if there is a `names` attribute.

**Logical subsetting:** Selects elements corresponding to `TRUE`.

```
R> x[3:5]
```

```
      c      d      e  
4.00 88.17 13.00
```

```
R> x[c("c", "d", "e")]
```

```
      c      d      e  
4.00 88.17 13.00
```

```
R> x[x > 3.5]
```

```
      c      d      e  
4.00 88.17 13.00
```

Subsetting of matrices and data frames etc. works similarly.

# Lists

**So far:** We have only used plain vectors. Lists are related but more flexible data structures.

**Lists:** *Generic vectors*. Each element can be virtually any type of object.

- Vector (of arbitrary mode).
- Matrix.
- Full data frame.
- Function.
- List (again).
- ...

Due to this flexibility, lists are the basis for most complex objects in R; e.g., for data frames or fitted regression models (both described later).

# Lists

**Illustration:** Using `list()`, create description of a sample from a standard normal distribution (generated with `rnorm()`; see below).

```
R> mylist <- list(sample = rnorm(5),
+   family = "normal distribution",
+   parameters = list(mean = 0, sd = 1))
R> mylist

$sample
[1] 0.3771 -0.9346 2.4302 1.3195 0.4503

$family
[1] "normal distribution"

$parameters
$parameters$mean
[1] 0

$parameters$sd
[1] 1
```

# Lists

**Select elements:** Operators `$` or `[[` can be used. `[[` is similar to `[`, but can only select a single element.

```
R> mylist[[1]]
```

```
[1] 0.3771 -0.9346 2.4302 1.3195 0.4503
```

```
R> mylist[["sample"]]
```

```
[1] 0.3771 -0.9346 2.4302 1.3195 0.4503
```

```
R> mylist$sample
```

```
[1] 0.3771 -0.9346 2.4302 1.3195 0.4503
```

Combination of element selection:

```
R> mylist[[3]]$sd
```

```
[1] 1
```

# Logical comparisons

**Logical operators:** `<`, `<=`, `>`, `>=`, `==` (for exact equality) and `!=` (for “not equal”).

If `expr1` and `expr2` are logical expressions,

- `expr1 & expr2` is their intersection (logical “and”),
- `expr1 | expr2` is their union (logical “or”), and
- `!expr1` is the negation of `expr1`.

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
```

```
R> x > 3 & x <= 4
```

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

Assess which elements are TRUE:

```
R> which(x > 3 & x <= 4)
```

```
[1] 2 3
```

**Specialized functions** `which.min()` and `which.max()` for computing the position of the minimum and the maximum.

# Logical comparisons

In addition to `&` and `|`: `all()` and `any()` check whether all or at least some entries of a vector are TRUE:

```
R> all(x > 3)
```

```
[1] FALSE
```

```
R> any(x > 3)
```

```
[1] TRUE
```

Due to coercion (more later!), it is also possible to compute directly on logical vectors using ordinary arithmetic. When coerced to numeric, FALSE becomes 0 and TRUE becomes 1, as in

```
R> 7 + TRUE
```

```
[1] 8
```

# Logical comparisons

**Caution:** Floating-point arithmetic has to be used when assessing exact equality of numerical arguments with `==`.

```
R> (1.5 - 0.5) == 1
```

```
[1] TRUE
```

```
R> (1.9 - 0.9) == 1
```

```
[1] FALSE
```

Use `all.equal()` instead:

```
R> all.equal(1.9 - 0.9, 1)
```

```
[1] TRUE
```

Furthermore, the function `identical()` checks whether two (possibly complex) R objects are exactly identical.

# Coercion

**Coercion functions:** Can convert an object from one type or class to a different one.

**Convention:** If *foo* is the type/class of interest, as *foo()* coerces to *foo* and *is.foo()* checks if an object is *foo*, e.g., `numeric`, `character`, `matrix`, `data.frame`, ...

```
R> is.numeric(x)
```

```
[1] TRUE
```

```
R> is.character(x)
```

```
[1] FALSE
```

```
R> as.character(x)
```

```
[1] "1.8"      "3.14"     "4"        "88.169"  "13"
```

Coercion is enforced automatically in certain situations, e.g.,

```
R> c(1, "a")
```

```
[1] "1" "a"
```



# Random number generation

**Random number generators (RNGs):** Vital for statistical/econometric programming environments for performing Monte Carlo studies.

**In R:** Several algorithms available, see `?RNG`.

**Random seed:** `set.seed()`. Basis for the generation of pseudo-random numbers and making simulations exactly reproducible.

```
R> set.seed(123)
```

```
R> rnorm(2)
```

```
[1] -0.5605 -0.2302
```

```
R> rnorm(2)
```

```
[1] 1.55871 0.07051
```

```
R> set.seed(123)
```

```
R> rnorm(2)
```

```
[1] -0.5605 -0.2302
```

# Random number generation

**Drawing samples:** Sampling with or without replacement from a finite set of values, is available in `sample()`.

**Default:** Draw, without replacement, a vector of the same size as its input argument (i.e., to compute a permutation).

```
R> sample(1:5)
```

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

```
R> sample(c("male", "female"), size = 5, replace = TRUE,  
+       prob = c(0.2, 0.8))
```

```
[1] "female" "male"   "female" "female" "female"
```

The second command draws a sample of size 5, with replacement, from the values "male" and "female", which are drawn with probabilities 0.2 and 0.8, respectively.

# Random number generation

## Distributions:

- Random numbers from specific distributions are typically available in functions of type `rdist()`.
- Examples for *dist* include `norm`, `unif`, `binom`, `pois`, `t`, `f`, `chisq`.
- Functions take sample size `n` as their first argument. Further arguments control parameters of the respective distribution.

Example: `rnorm()` takes `mean` and `sd` as further arguments, defaulting to 0 and 1.

- Further functions for distributions: `ddist()`, `pdist()`, and `qdist()` implementing density, cumulative probability distribution function, and quantile function (inverse distribution function).

# Flow control

**Standard control structures:** Control when a certain expression `expr` is evaluated. (See `?Control` for details.)

- `if/else` statements.
- `for` loops.
- `while` loops.

An `if/else` statement is of the form

```
if(cond) {  
  expr1  
} else {  
  expr2  
}
```

where `expr1` is evaluated if `cond` is `TRUE` and `expr2` otherwise. The `else` branch may be omitted if empty.

# Flow control

**Illustration:** Toy example.

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
R> if(rnorm(1) > 0) sum(x) else mean(x)

[1] 22.02
```

The condition `cond` can only be of length 1. For vectorized evaluation, use `ifelse()`.

```
R> ifelse(x > 4, sqrt(x), x^2)

[1] 3.240 9.860 16.000 9.390 3.606
```

This computes the square root for those values in `x` that are greater than 4 and the square for the remaining ones.

# Flow control

**for loop:** Similar, but the main argument to `for()` is of type variable in sequence.

**Illustration:** Recursively compute first differences in the vector `x`.

```
R> for(i in 2:5) {  
+   x[i] <- x[i] - x[i-1]  
+ }  
R> x[-1]  
  
[1] 1.34 2.66 85.51 -72.51
```

**while loop:** Similar, but the argument to `while()` is a condition that may change in every run of the loop so that it finally can become `FALSE`.

```
R> while(sum(x) < 100) {  
+   x <- 2 * x  
+ }  
R> x  
  
[1] 14.40 10.72 21.28 684.07 -580.07
```

# Writing functions

**Feature of S/R:** Users naturally become developers. Repeated commands can easily be wrapped into functions.

**Simple example:** Deliberately awkward function computing column means in a matrix  $X$  using nested for loops.

```
R> cmeans <- function(X) {  
+   rval <- rep(0, ncol(X))  
+   for(j in 1:ncol(X)) {  
+     mysum <- 0  
+     for(i in 1:nrow(X)) mysum <- mysum + X[i,j]  
+     rval[j] <- mysum/nrow(X)  
+   }  
+   return(rval)  
+ }
```

```
R> X <- matrix(1:20, ncol = 2)
```

```
R> cmeans(X)
```

```
[1] 5.5 15.5
```

# Writing functions

Built-in function `colMeans()`

```
R> colMeans(X)
```

```
[1] 5.5 15.5
```

is clearly preferable.

```
R> X <- matrix(rnorm(2*10^6), ncol = 2)
```

```
R> system.time(colMeans(X))
```

```
   user  system elapsed
0.002   0.000   0.002
```

```
R> system.time(cmeans(X))
```

```
   user  system elapsed
0.112   0.000   0.112
```

`cmeans()` takes only a single argument `X` with no default. If defaults should be defined, use `name = expr` pairs.



# Vectorized calculations

**Vectorized arithmetic:** Can be used to avoid loops.

**Example:** Avoid one for loop by using the vectorized function `mean()`.

```
R> cmeans2 <- function(X) {  
+   rval <- rep(0, ncol(X))  
+   for(j in 1:ncol(X)) rval[j] <- mean(X[,j])  
+   return(rval)  
+ }  
R> system.time(cmeans2(X))
```

```
   user  system elapsed  
0.021  0.004  0.025
```

**More compactly:** `apply(X, 2, mean)`. Looks less cumbersome than for loop (but often performs similarly). `apply()` applies functions along margins (here, columns, the second margin) of an array.

```
R> system.time(apply(X, 2, mean))
```

```
   user  system elapsed  
0.106  0.004  0.109
```

# Vectorized calculations

## Summary:

- 1 Element-wise computations should be avoided if vectorized computations are available.
- 2 Optimized solutions (if available) typically perform better than generic `for` or `apply()` solutions.
- 3 Loops can be written more compactly using `apply()`.

## Several variants:

- `lapply()`: returns a list.
- `tapply()`: returns a table.
- `sapply()`: tries to simplify the result to a vector or matrix where possible.

# Reserved words

**Reserved words:** Basic grammatical constructs of the language that cannot be used in other meanings.

**In R:** `if`, `else`, `for`, `in`, `while`, `repeat`, `break`, `next`, `function`, `TRUE`, `FALSE`, `NA`, `NULL`, `Inf`, `NaN`, ...).

See `?Reserved` for a complete list.

Basics

# Formulas

# Formulas

**Symbolic descriptions:** Relationships among variables can be specified. The  $\sim$  operator is basic to formulas in R.

```
R> f <- y ~ x
R> class(f)

[1] "formula"
```

**Meaning:** Depends on the context, i.e., the function that evaluates the formula.

**Most commonly:**  $y \sim x$  means “ $y$  is explained by  $x$ ”.

**Usage:** Specification of plots, models, etc.

# Formulas

**Illustration:** Artificial data.

```
R> x <- seq(from = 0, to = 10, by = 0.5)
R> y <- 2 + 3 * x + rnorm(21)
```

Use the same formula for plotting and linear regression.

```
R> plot(y ~ x)
R> lm(y ~ x)
```

Call:

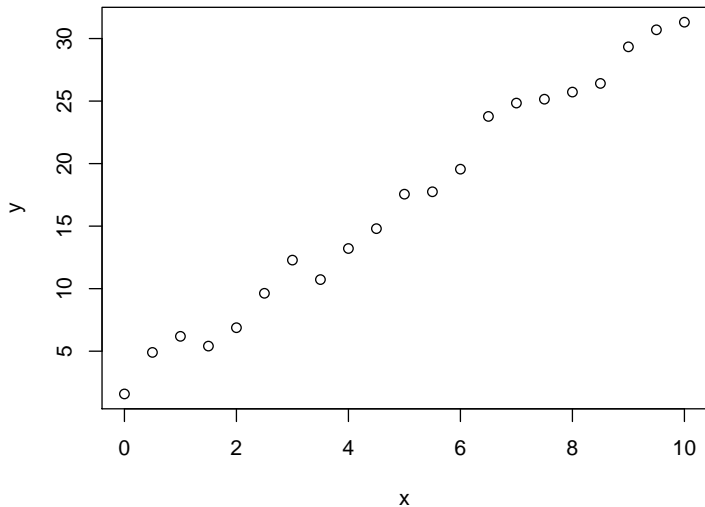
```
lm(formula = y ~ x)
```

Coefficients:

```
(Intercept)          x
          2.00          3.01
```

R formula language is much more powerful, especially for specifying (generalized) linear models (see Chapter 3).

# Formulas



Basics

# Data Management in R



# Creation from scratch

**Data frames:** Basic data structure in R. (In other programs such structures are often called data matrix or data set.)

**Typically:** An array consisting of a list of vectors and/or factors of identical length, i.e., a rectangular format where columns correspond to variables and rows to observations.

**Example:** Artificial data with variables named "one", "two", "three".

```
R> mydata <- data.frame(one = 1:10, two = 11:20, three = 21:30)
```

Alternatively:

```
R> mydata <- as.data.frame(matrix(1:30, ncol = 3))
```

```
R> names(mydata) <- c("one", "two", "three")
```

**Technically:** This data frame is internally represented as a list of vectors (not a matrix).

# Subset selection

**Select columns:** Subsets of variables can be selected via `[` or `$` (for a single variable).

```
R> mydata$two
```

```
[1] 11 12 13 14 15 16 17 18 19 20
```

```
R> mydata[, "two"]
```

```
[1] 11 12 13 14 15 16 17 18 19 20
```

```
R> mydata[, 2]
```

```
[1] 11 12 13 14 15 16 17 18 19 20
```

In all cases: The data frame attributes are dropped (by default).

# Subset selection

**Accessing variables:** Variables can be `attach()`ed. (Technically, this means that the attached data set is added to the `search()` path.)

```
R> mean(two)
```

```
Error in mean(two) : Object "two" not found
```

```
R> attach(mydata)
```

```
R> mean(two)
```

```
[1] 15.5
```

```
R> detach(mydata)
```

**Note:** Attaching data frames can lead to confusion when there are variables with the same name in several data frames or the global environment.

**For a single command:**

```
R> with(mydata, mean(two))
```

```
[1] 15.5
```

# Subset selection

**Select rows:** Subsets of observations (and variables) can be selected again via `[]` or (more conveniently) via `subset()`.

```
R> subset(mydata, two <= 16, select = -two)
```

	one	three
1	1	21
2	2	22
3	3	23
4	4	24
5	5	25
6	6	26

# Import and export

**Export as plain text:** `write.table()`.

```
R> write.table(mydata, file = "mydata.txt", col.names = TRUE)
```

This creates a text file `mydata.txt` in the current working directory. To read again, use:

```
R> newdata <- read.table("mydata.txt", header = TRUE)
```

## Details:

- `read.table()` returns a “`data.frame`” object
- By setting `col.names = TRUE`, `mydata.txt` contains variable names in the first row. Hence, it should be read with `header = TRUE`.
- `write.table()` allows specification of: separation symbol, decimal separator, quotes, and many more. Thus, it can create tab- or comma-separated values etc.

# Import and export

**CSV:** Comma-separated values.

- Convenience interfaces `read.csv()` and `write.csv()` are available.
- CSV is useful format for exchanging data between R and Microsoft Excel.
- On systems with comma (and not the period) as the decimal separator, Excel uses semicolon-separated values (but still calls them CSV).
- These can be read/written with `read.csv2()` and `write.csv2()`.
- More elementary: `scan()` is useful for reading more complex structures.
- See the manual pages and the “R Data Import/Export” manual for further details.

# Import and export

**Binary format:** To write/read R's internal binary format (by convention with extension `.RData` or `.rda`) the commands `load()` and `save()` are available.

```
R> save(mydata, file = "mydata.rda")  
R> load("mydata.rda")
```

## Details:

- Binary files can contain multiple arbitrary objects (not just a single data frame or matrix).
- Upon `load()` all objects are made available in the current environment (by default).

# Import and export

## Data in packages:

- All data sets in the package **AER** are supplied in this binary format.
- Go to the folder `~/AER/data` in your R library to check.
- As they are part of a package, they are made accessible more easily using `data()` (which in this case sets up the appropriate call for `load()`).
- Hence,  

```
data("Journals", package = "AER")
```

loads the `Journals` data frame from the **AER** package, stored in the file `~/AER/data/Journals.rda`.
- If the `package` argument is omitted, all packages currently in the search path are checked whether they provide a file `Journals`.



# Reading and writing foreign binary formats

**Package foreign:** R can also read and write a number of proprietary binary formats, including S-PLUS, SPSS, SAS, Stata, Minitab, Systat, and dBase files.

**Example:** Stata files.

Export:

```
R> library("foreign")  
R> write.dta(mydata, file = "mydata.dta")
```

Import:

```
R> mydata <- read.dta("mydata.dta")
```

# Reading and writing Excel spreadsheets

**Excel spreadsheets:** `.xls` and `.xlsx` files.

- No direct support by base R → exchange via CSV recommended.
- Several CRAN packages also offer support for reading/writing Excel spreadsheets directly.
- **gdata:** `read.xls()` for simple (and quick) reading. Requires Perl (<https://www.perl.org/get.html>).
- **xlsx:** `read.xlsx()` and `write.xlsx()` for simple (but not quite as quick) reading and writing. Requires Java.
- **openxlsx:** `read.xlsx()` and `write.xlsx()` with somewhat less intuitive arguments/default. Requires C++ but binaries on CRAN.
- **readxl:** `read_excel()` with somewhat different arguments. Requires C++ but binaries on CRAN.
- **XLConnect:** Functionality for querying and manipulating (including reading/writing) spreadsheets. Requires Java.

# Interaction with the file system and string manipulations

**Rich functionality:** Interaction with external files and communication with the operating system.

## A few pointers:

- Query files available in a directory or folder: `dir()`.
- Copying and deleting files: `file.copy()` and `file.remove()`.
- These commands are independent of the operating system.
- Potentially system-dependent commands can be issued as strings using `system()`.

**Illustration:** Delete the Stata file created before.

```
R> file.remove("mydata.dta")
```

# Interaction with the file system and string manipulations

Save commands or their output to text files:

- One possibility: `sink()` can direct output to a `file()` connection.
- Strings can be written with `cat()` to a connection.
- In some situations `writeLines()` is more convenient for this.
- Furthermore: `dump()` can create text representations of R objects and write them to a `file()` connection.

Manipulate strings before creating output:

- `strsplit()`: splitting strings.
- `paste()`: paste strings together.
- `grep()` and `gsub()`: pattern matching and replacing.
- `sprintf()`: combining text and variable values.

# Factors

**Categorical information:** Stored in *factors*, an extension of vectors.

**Typical econometric examples:** gender, union membership, or ethnicity.

**In many software packages:** Stored using a numerical encoding (e.g., 0 for males and 1 for females). Especially in regression settings, a single categorical variable with more than two categories is often stored in several such dummy variables.

## In R:

```
R> g <- rep(0:1, c(2, 4))
R> g <- factor(g, levels = 0:1, labels = c("male", "female"))
R> g

[1] male   male   female female female female
Levels: male female
```

# Factors

## Details:

- Here, a `factor()` is created from a dummy-coded vector.
- Internally: stored as the integers 1 to  $k$  (= number of levels) plus a character vector of labels, here "male" and "female".
- `factor()` can create the same information from numerical, character or logical vectors.
- For ordinal information, set the argument `ordered = TRUE`.

**Advantage:** R knows that a certain variable is categorical and can choose appropriate methods automatically.

- Labels can be used in printed output.
- Different summary and plotting methods can be chosen.
- Contrast codings (e.g., dummy variables) can be computed in linear regressions.

# Missing values

**In R:** Missing values are coded as `NA` (for “not available”). All standard computations on `NA` become `NA`.

**Caution:** Data sets may have missing values with a different encoding.

**Example:** Sometimes `-99` or `-999` is used in flat text files. These can be converted appropriately via:

```
R> newdata <- read.table("mydata.txt", na.strings = "-999")
```

**Query NAs:** `is.na()`.

Basics

# Object Orientation



# Object orientation:

**Object-oriented programming (OOP):** Paradigm of programming where users/developers can create objects of a certain “class” (that are required to have a certain structure) and then apply “methods” for certain “generic functions” to these objects.

**Simple example:** `summary()` is a generic function choosing different methods based on the class of its argument

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
R> g <- factor(rep(c(0, 1), c(2, 4)), levels = c(0, 1),
+   labels = c("male", "female"))
R> summary(x)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.80	3.14	4.00	22.02	13.00	88.17

```
R> summary(g)

male female
  2      4
```

# Object orientation:

**Several paradigms:** In fact, R has several OOP systems. The base installation already has two, usually called S3 and S4.

**S3:** Much simpler, using a dispatch mechanism based on a naming convention for methods.

**S4:** More sophisticated and closer to other OOP concepts used in computer science.

**For most tasks:** S3 is sufficient and hence briefly discussed here.

**S3 generics:** Functions with a certain list of arguments and then a `UseMethod()` call with the name of the generic function.

```
R> summary  
  
function (object, ...)  
UseMethod("summary")  
<bytecode: 0x560c101d8b48>  
<environment: namespace:base>
```

# Object orientation:

## Details:

- Arguments: `object` (required) plus an arbitrary number of optional arguments passed through `...` to its methods.
- When applied to an object of class `"foo"`: R tries to apply the function `summary.foo()` if it exists. If not, it will call `summary.default()` if such a default method exists (which it does for `summary()`).
- R objects can also have a vector of classes, e.g., `c("foo", "bar")` meaning that the object is of class `"foo"` inheriting from `"bar"`.
- In this case, R first tries to apply `summary.foo()`, then (if this does not exist) `summary.bar()`, and then (if both do not exist) `summary.default()`.

# Object orientation:

## Using methods:

- Methods defined for a certain generic can be queried using `methods()`.
- `methods(summary)` returns a (long) list of methods including `summary.factor()` and `summary.default()` (but not `summary.numeric()`).
- As it is not recommended to call methods directly, some methods are marked as being non-visible to the user and these cannot (easily) be called directly.
- Even if visible, it is preferred to call the generic, i.e., `summary(g)` instead of `summary.factor(g)`.

# Object orientation:

**Illustration:** Definition of a class and methods.

- Create an object of class “normsample” that contains a sample from a normal distribution.
- Define a `summary()` method that reports the empirical mean and standard deviation.

First, we write a simple class creator. In principle, it could have any name, but it is often called like the class itself:

```
R> normsample <- function(n, ...) {  
+   rval <- rnorm(n, ...)  
+   class(rval) <- "normsample"  
+   return(rval)  
+ }
```

# Object orientation:

## Details:

- It takes a required argument `n` (the sample size) and further arguments `...`, which are passed on to `rnorm()` for generating normal random numbers.
- `rnorm()` takes further arguments like the mean and the standard deviation.
- After generation of the vector of normal random numbers, it is assigned the class “normsample” and then returned.

```
R> set.seed(123)
R> x <- normsample(10, mean = 5)
R> class(x)

[1] "normsample"
```

# Object orientation:

**New summary() method:** `summary.normsample()`. It conforms with the argument list of the generic (although `...` is not used) and computes sample size, empirical mean, and standard deviation.

```
R> summary.normsample <- function(object, ...) {  
+   rval <- c(length(object), mean(object), sd(object))  
+   names(rval) <- c("sample size", "mean", "standard deviation")  
+   return(rval)  
+ }
```

This method is found when calling:

```
R> summary(x)
```

sample size	mean	standard deviation
10.0000	5.0746	0.9538

**Typical generics:** `print()`, `plot()`, and `str()`, which print, plot, and summarize the structure are available for most basic classes.

Basics

# R Graphics



# R graphics

Early publications on S and R already emphasized the powerful graphics:

- Beckers and Chambers (1984): “S: An Interactive Environment for Data Analysis and Graphics”.
- Ihaka and Gentleman (1996): “R: A Language for Data Analysis and Graphics”.

**Here:** Brief introduction to “conventional” graphics as implemented in base R.

**Even more flexible: grid** graphics (Murrell 2005), enabling “trellis”-type graphics (Cleveland 1993) in package **lattice**. Not discussed here.

# The function `plot()`

**Generic function:** `plot()`. Methods for many objects, including data frames, time series, and fitted linear models.

**Basic:** Default `plot()` method, creates various types of scatterplots. Many of the explanations below extend to other methods and high-level plotting functions.

**Scatterplot:** `plot(x, y)` produces a scatterplot of  $y$  vs.  $x$ .

# The function `plot()`

**Illustration:** Relationship between the number of subscriptions and the price per citation for economics journals.

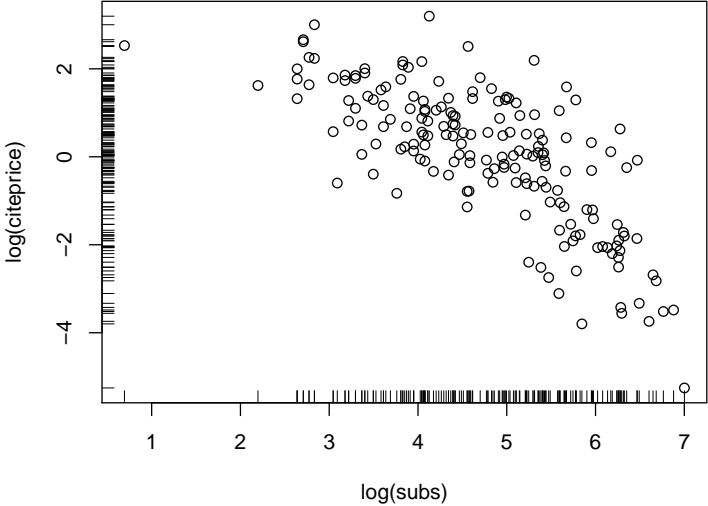
```
R> data("Journals")
R> Journals$citeprice <- Journals$price/Journals$citations
R> attach(Journals)
R> plot(log(subs), log(citeprice))
R> rug(log(subs))
R> rug(log(citeprice), side = 2)
R> detach(Journals)
```

**Rug:** `rug()` adds ticks, thus visualizing the marginal distributions of the variables.

**Alternatively:** Instead of attaching/detaching the data, one can use

```
R> plot(log(subs) ~ log(citeprice), data = Journals)
```

# The function `plot()`



# Graphical parameters

**Modifications:** `plot()` has many arguments, including

- `type`: modify plot type, e.g., `points` (`type = "p"`, default), `lines` (`type = "l"`), `both` (`type = "b"`), `stair steps` (`type = "s"`).
- `main`, `xlab`, `ylab`: modify title and axis labels.
- Further graphical parameters (see `?par`) can be passed to `plot()` or set separately via `par()`.
- `col`: set `color(s)`.
- `xlim`, `ylim`: adjust plotting ranges.
- `pch`: modify the `plotting character` for points.
- `cex`: corresponding `character extension`.
- `lty`, `lwd`: `line type` and `width`.
- `cex.lab`, `cex.axis`, `cex.foo`: size of labels, axis ticks, etc.

# Graphical parameters

---

Argument	Description
<code>axes</code>	should axes be drawn?
<code>bg</code>	background color
<code>cex</code>	size of a point or symbol
<code>col</code>	color
<code>las</code>	orientation of axis labels
<code>lty, lwd</code>	line type and line width
<code>main, sub</code>	title and subtitle
<code>mar</code>	size of margins
<code>mfcoll, mfrow</code>	array defining layout for several graphs on a plot
<code>pch</code>	plotting symbol
<code>type</code>	types (see text)
<code>xlab, ylab</code>	axis labels
<code>xlim, ylim</code>	axis ranges
<code>xlog, ylog, log</code>	logarithmic scales

---

# Graphical parameters

## Example:

```
R> plot(log(subs) ~ log(citeprice), data = Journals, pch = 20,  
+       col = "blue", ylim = c(0, 8), xlim = c(-7, 4),  
+       main = "Library subscriptions")
```

**Add further layers:** `lines()`, `points()`, `text()`, `legend()`.

```
R> text(-3.798, 5.846, "Econometrica", pos = 2)
```

**Straight line:** `abline(a, b)` with intercept `a` and slope `b`.

**Further plotting functions:** `barplot()`, `pie()` (pie charts),  
`boxplot()`, `qqplot()` (QQ plots), `hist()` (histograms).

**Instructive overview:** `demo("graphics")`.

# Exporting graphics

**Storing graphical results:** e.g., for publication in a report, journal article, or thesis.

**For Microsoft Windows and Microsoft Word:** A simple option is to “copy and paste”.

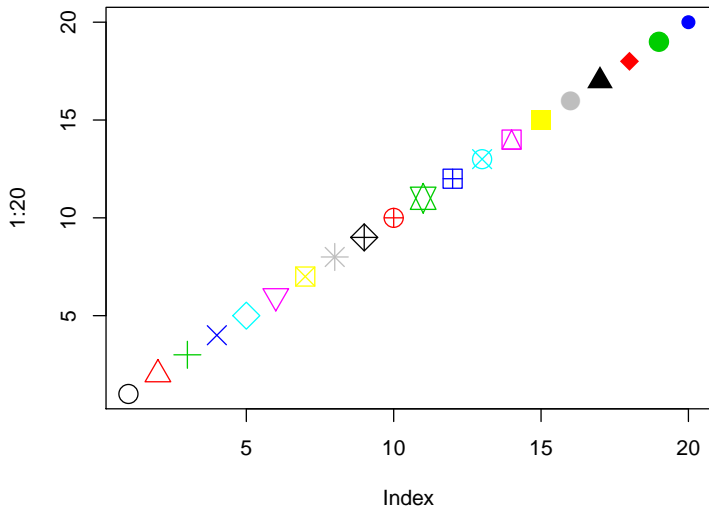
**More generally:** Create external files using a suitable graphics device. Devices available on all platforms include the vector formats PostScript and PDF. The bitmap formats PNG and JPEG and the vector format WMF are system-dependent. See `?Devices` for details.

**Usage:** First the device is opened, then the plot commands are executed, and finally the device is closed by `dev.off()`.

```
R> pdf("myfile.pdf", height = 5, width = 6)
R> plot(1:20, pch = 1:20, col = 1:20, cex = 2)
R> dev.off()
```



# Exporting graphics



# Exporting graphics

## Details:

- Creates the PDF file `myfile.pdf` in the current working directory.
- Contains the graphic generated by the `plot()` call.
- The plot illustrates a few graphical parameters: 20 plotting symbols in double size and different colors (a basic set of colors is numbered).

**Alternatively:** Instead of opening, printing and closing a device, it is possible to print an existing plot in the graphics window to a device using `dev.copy()` and `dev.print()`.

# Mathematical annotation of plots

**Overview:** `?plotmath` and `demo("plotmath")`.

**Syntax:** Somewhat similar to  $\text{\LaTeX}$ .

**Illustration:** Density of the standard normal distribution along with its mathematical definition.

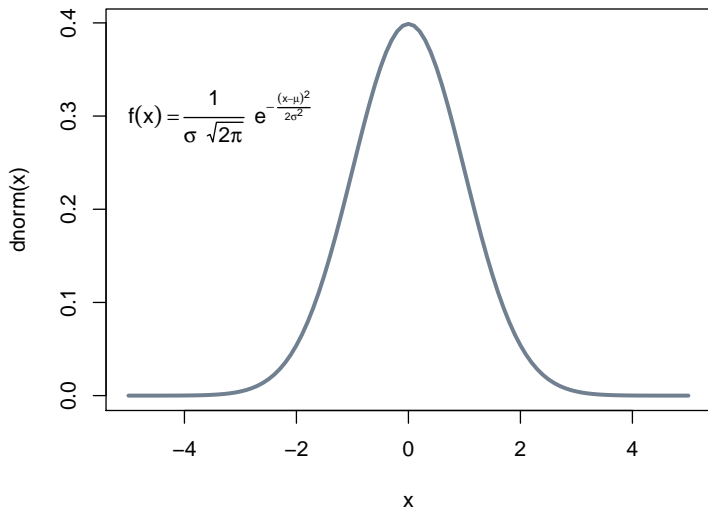
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

**In R:**

```
R> curve(dnorm, from = -5, to = 5, col = "slategray", lwd = 3,  
+       main = "Density of the standard normal distribution")  
R> text(-5, 0.3, expression(f(x) == frac(1, sigma ~  
+   sqrt(2*pi)) ~ e^{- ~ frac((x - mu)^2, 2*sigma^2)}), adj = 0)
```

# Mathematical annotation of plots

Density of the standard normal distribution



Basics

# Exploratory Data Analysis with R

# Exploratory data analysis with R

**Here:** Brief introduction. More details in Dalgaard (2008).

**Data:** CPS1985 from Berndt (1991).

```
R> data("CPS1985")
```

```
R> str(CPS1985)
```

```
'data.frame':      534 obs. of  11 variables:
 $ wage      : num  5.1 4.95 6.67 4 7.5 ...
 $ education : num  8 9 12 12 12 13 10 12 16 12 ...
 $ experience: num  21 42 1 4 17 9 27 9 11 9 ...
 $ age       : num  35 57 19 22 35 28 43 27 33 27 ...
 $ ethnicity : Factor w/ 3 levels "cauc","hispanic",...: 2 1 1 1 1 ..
 $ region    : Factor w/ 2 levels "south","other": 2 2 2 2 2 2 1 ..
 $ gender    : Factor w/ 2 levels "male","female": 2 2 1 1 1 1 1 ..
 $ occupation: Factor w/ 6 levels "worker","technical",...: 1 1 1 ..
 $ sector    : Factor w/ 3 levels "manufacturing",...: 1 1 1 3 3 3..
 $ union     : Factor w/ 2 levels "no","yes": 1 1 1 1 1 2 1 1 1 1..
 $ married   : Factor w/ 2 levels "no","yes": 2 2 1 1 2 1 1 1 2 1..
```

# Exploratory data analysis with R

**Inspect top or bottom:** `head()` or `tail()`, returning (by default) the first or last 6 rows.

```
R> head(CPS1985)
```

```
      wage education experience age ethnicity region gender
1      5.10         8         21  35  hispanic  other female
1100  4.95         9         42  57         cauc  other female
2      6.67        12         1  19         cauc  other  male
3      4.00        12         4  22         cauc  other  male
4      7.50        12        17  35         cauc  other  male
5     13.07        13         9  28         cauc  other  male

      occupation      sector union married
1      worker manufacturing    no      yes
1100  worker manufacturing    no      yes
2      worker manufacturing    no      no
3      worker          other    no      no
4      worker          other    no      yes
5      worker          other   yes     no
```

# Exploratory data analysis with R

## Overview: Summary by variable.

```
R> summary(CPS1985)
```

```
      wage      education      experience      age
Min.   : 1.00   Min.     : 2     Min.     : 0.0   Min.     :18.0
1st Qu.: 5.25   1st Qu.:12    1st Qu.: 8.0   1st Qu.:28.0
Median : 7.78   Median :12    Median :15.0   Median :35.0
Mean   : 9.02   Mean      :13   Mean     :17.8   Mean     :36.8
3rd Qu.:11.25   3rd Qu.:15    3rd Qu.:26.0   3rd Qu.:44.0
Max.   :44.50   Max.     :18    Max.     :55.0   Max.     :64.0

      ethnicity      region      gender      occupation
cauc      :440   south:156   male   :289   worker   :156
hispanic: 27   other:378   female:245   technical :105
other    : 67                                     services  : 83
                                                office   : 97
                                                sales    : 38
                                                management: 55

      sector      union      married
manufacturing: 99   no :438   no :184
construction  : 24   yes: 96   yes:350
other          :411
```



# Exploratory data analysis with R

For compactifying input and output:

```
R> levels(CPS1985$occupation)[c(2, 6)] <- c("techn", "mgmt")  
R> attach(CPS1985)
```

**In the following:**

- Exploratory analysis of a single numerical/categorical variable.
- Exploratory analysis of pairs of variables.

# One numerical variable

Distribution of wages: Tukey's five-number summary and sample mean.

```
R> summary(wage)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.00	5.25	7.78	9.02	11.25	44.50

Standalone functions: `mean()`, `median()`, `min()`, `max()`, `fivenum()`.

```
R> mean(wage)
```

```
[1] 9.024
```

Arbitrary quantiles: `quantile()`.

Measures of spread: variance and standard deviation.

```
R> var(wage)
```

```
[1] 26.41
```

```
R> sd(wage)
```

```
[1] 5.139
```

# One numerical variable

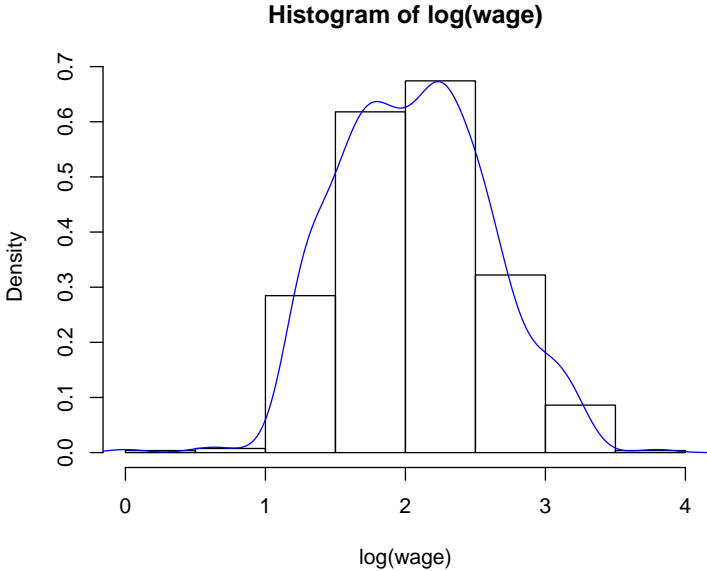
**Graphical summary:** Density visualizations (via histograms or kernel smoothing) and boxplots.

```
R> hist(log(wage), freq = FALSE)
R> lines(density(log(wage)), col = 4)
```

## Details:

- Density of logarithm of wage (i.e., area under curve equals 1).
- Default: absolute frequencies, changed to density via `freq = FALSE`.
- Further fine tuning possible via selection of `breaks`.
- Added kernel density estimate.

# One numerical variable



# One categorical variable

Appropriate summary chosen automatically for “factor” variables.

```
R> summary(occupation)
```

worker	techn	services	office	sales	mgmt
156	105	83	97	38	55

Alternatively: Use `table()` and also compute relative frequencies.

```
R> tab <- table(occupation)
```

```
R> prop.table(tab)
```

```
occupation
  worker   techn services   office   sales   mgmt
0.29213 0.19663 0.15543 0.18165 0.07116 0.10300
```

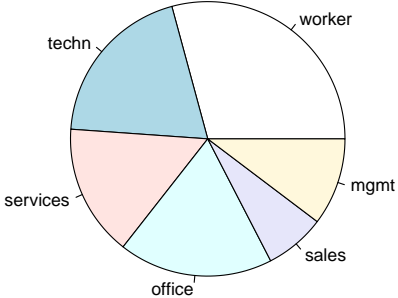
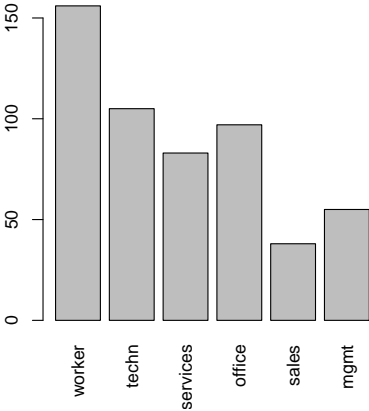
Visualization: `barplot()`. If majorities are to be brought out, `pie()` charts might be useful. Both expect tabulated frequencies as input.

```
R> barplot(tab)
```

```
R> pie(tab)
```

`plot(occupation)` is equivalent to `barplot(table(occupation))`.

# One categorical variable



# Two categorical variables

## Relationship between two categorical variables:

**Numerical summary:** Contingency table(s) via `xtabs()` (with formula interface) or `table()`. Use `table(gender, occupation)` or

```
R> xtabs(~ gender + occupation, data = CPS1985)
```

	occupation					
gender	worker	techn	services	office	sales	mgmt
male	126	53	34	21	21	34
female	30	52	49	76	17	21

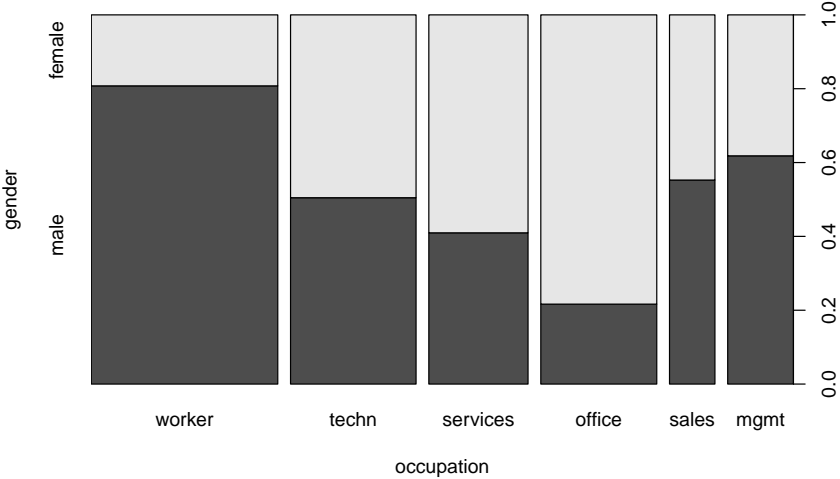
**Graphical summary:** Mosaic plot, a generalization of stacked barplots.

The following variant is also called “spine plot”:

```
R> plot(gender ~ occupation, data = CPS1985)
```

Bar heights correspond to the conditional distribution of `gender` given `occupation`. Bar widths visualize the marginal distribution of `occupation`.

# Two categorical variables





# Two numerical variables

**Numerical summary:** Correlation coefficient(s) via `cor()`. Default is the standard Pearson correlation coefficient, alternatives include the nonparametric Spearman's  $\rho$ .

```
R> cor(log(wage), education)
```

```
[1] 0.3804
```

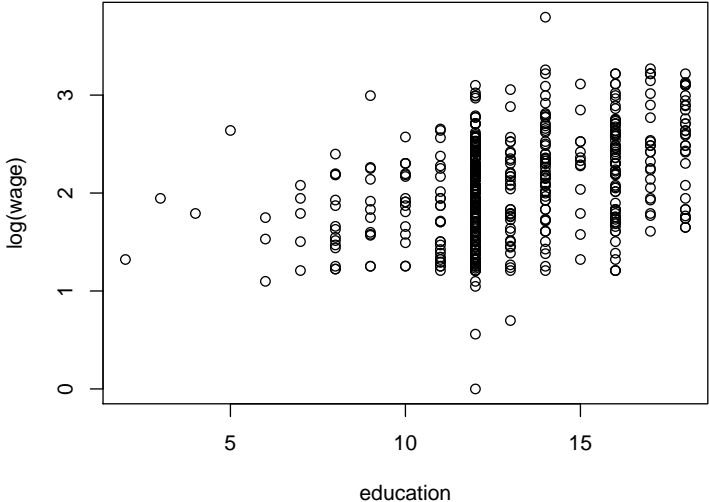
```
R> cor(log(wage), education, method = "spearman")
```

```
[1] 0.3813
```

**Graphical summary:** Scatterplot.

```
R> plot(log(wage) ~ education)
```

# Two numerical variables



# One numerical and one categorical variable

**Numerical summary:** Grouped numerical summaries (for the numerical variable given the categorical variable).

**In R:** `tapply()` applies functions grouped by a (list of) categorical variable(s). Mean wages conditional on gender are available using:

```
R> tapply(log(wage), gender, mean)
```

```
  male female  
2.165  1.934
```

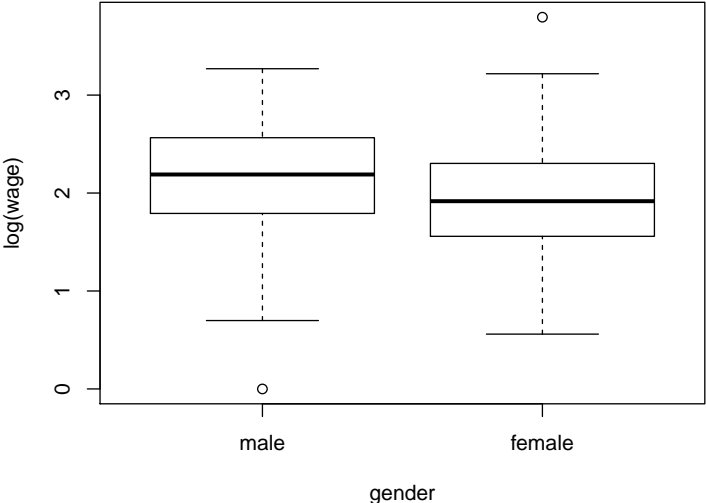
Other measures: Replace `mean` by other function, e.g., `summary`.

**Graphical summary:** Parallel boxplots or quantile-quantile (QQ) plots.

```
R> plot(log(wage) ~ gender)
```

The commands `plot(y ~ x)` and `boxplot(y ~ x)` both yield the same parallel boxplot if `x` is a “factor”.

# One numerical and one categorical variable



# One numerical and one categorical variable

## Boxplot:

- Coarse graphical summary of an empirical distribution.
- Box indicates “hinges” (approximately the lower and upper quartiles) and the median.
- “Whiskers” indicate the largest and smallest observations falling within a distance of 1.5 times the box size from the nearest hinge.
- Observations outside this range are outliers (in an approximately normal sample).

**QQ plot** indicates here that, for most quantiles, male wages are typically higher than female wages.

```
R> mwage <- subset(CPS1985, gender == "male")$wage
R> fwage <- subset(CPS1985, gender == "female")$wage
R> qqplot(mwage, fwage, xlim = range(wage), ylim = range(wage),
+        xaxs = "i", yaxs = "i", xlab = "male", ylab = "female")
R> abline(0, 1)
```

# One numerical and one categorical variable

