Midterm cheat sheet

29 January 2025 23:45

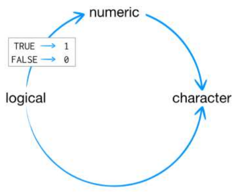
DATA TYPES

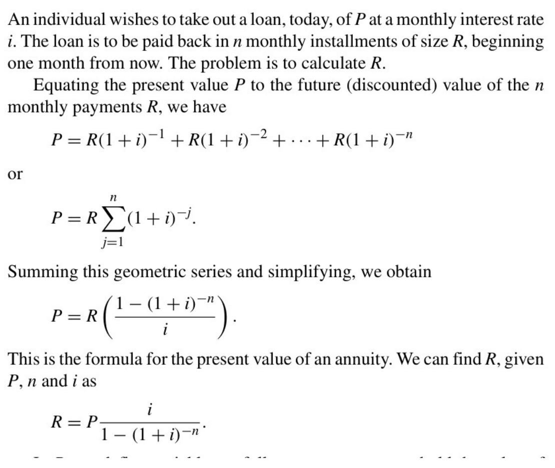
Behavior:

- Can be explicitly casted using as.type()
  - as.logical(3) = TRUE

- Can be checked with is.type()
  - na,nan,numeric,character,double

- Implicit casting follows the cycle below.
  - TRUE + 4 = 5
- Implicit casting that opposes direct flow results in error.
  - "hello" + 2 = Error: non-numeric argument





An individual wishes to take out a loan, today, of P at a monthly interest rate i. The loan is to be paid back in n monthly installments of size R, beginning one month from now. The problem is to calculate R.

Equating the present value P to the future (discounted) value of the n monthly payments R, we have

$$P = R(1+i)^{-1} + R(1+i)^{-2} + \cdots + R(1+i)^{-n}$$

or

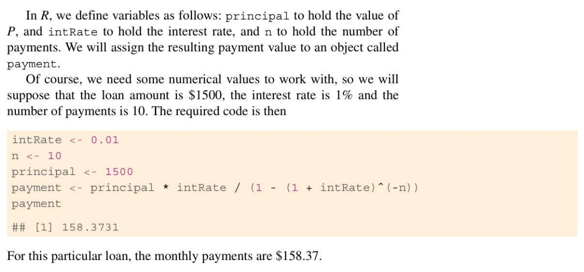
$$P = R\sum_{j=1}^{n}(1+i)^{-j}.$$

Summing this geometric series and simplifying, we obtain

$$P = R\left(\frac{1-(1+i)^{-n}}{i}\right).$$

This is the formula for the present value of an annuity. We can find R, given P, n and i as

$$R = P\frac{i}{1-(1+i)^{-n}}.$$



In *R*, we define variables as follows: `principal` to hold the value of P, and `intRate` to hold the interest rate, and `n` to hold the number of payments. We will assign the resulting payment value to an object called `payment`.

Of course, we need some numerical values to work with, so we will suppose that the loan amount is \$1500, the interest rate is 1% and the number of payments is 10. The required code is then

```
intRate <- 0.01
n <- 10
principal <- 1500
payment <- principal * intRate / (1 - (1 + intRate)^(-n))
payment

## [1] 158.3731
```

For this particular loan, the monthly payments are \$158.37.

Integer:

- Integers are not commonplace in java, hence, any number even without a decimal part is a double (ex. typeof(3) = "double"), except:
  - type-suffix - L
    - typeof(3L) = "integer"
  - sequenced
    - typeof(1:3) = "integer"
  - as.integer()
    - typeof(as.integer(3)) = "integer"

Logical:

- TRUE / T

- FALSE / F

- True/ true / false / False are invalid

Character(categoric data):

- Surrounded by single or double quotations
  - 'hello' == "hello"
- Cannot be indexed with []
- Cannot undergo any arithmetic operations
- To include '\' in a string include '\\' or else error
  - "hello\" = error
  - "hello\\" = "hello\"

Double:

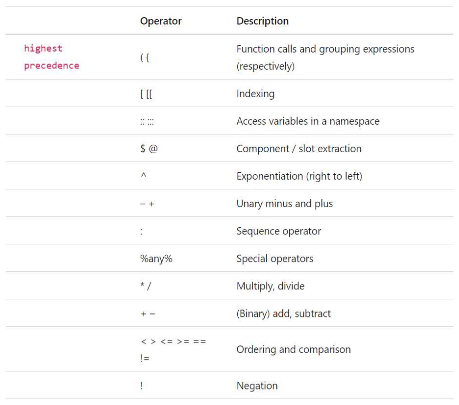
- 15 to 17 significant decimal digits.

OPERATORS

Special operators:

- %% - decimal remainder
  - 6 %% 4 = 2
- %/% - integer remainder
  - 6 %/% 4 = 1

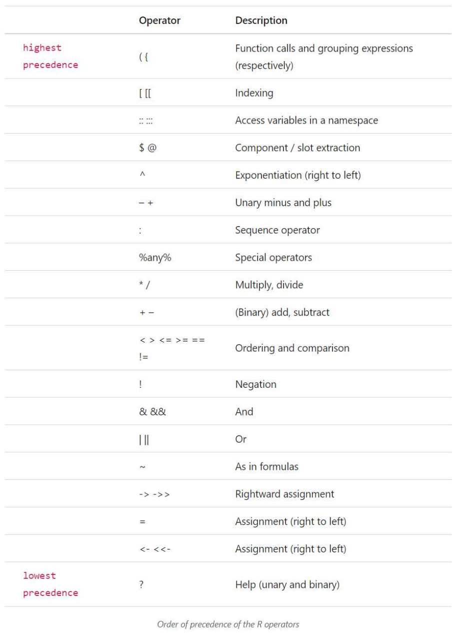
Operator precedence:



	Operator	Description
highest precedence	({	Function calls and grouping expressions (respectively)
	[[[	Indexing
	:: :::	Access variables in a namespace
	$ @	Component / slot extraction
	^	Exponentiation (right to left)
	− +	Unary minus and plus
	:	Sequence operator
	%any%	Special operators
	* /	Multiply, divide
	+ −	(Binary) add, subtract
	< > <= >= == !=	Ordering and comparison
	!	Negation

operators:

- %% - decimal remainder
    - 6 %% 4 = 2
- %/% - integer remainder
    - 6 %/% 4 = 1



	Operator	Description
highest precedence	({	Function calls and grouping expressions (respectively)
	[[[	Indexing
	:: :::	Access variables in a namespace
	$ @	Component / slot extraction
	^	Exponentiation (right to left)
	− +	Unary minus and plus
	:	Sequence operator
	%any%	Special operators
	* /	Multiply, divide
	+ −	(Binary) add, subtract
	< > <= >= == !=	Ordering and comparison
	!	Negation
	& &&	And
	| ||	Or
	~	As in formulas
	-> ->>	Rightward assignment
	=	Assignment (right to left)
	<- <<-	Assignment (right to left)
lowest precedence	?	Help (unary and binary)

Order of precedence of the R operators

FUNCTIONS

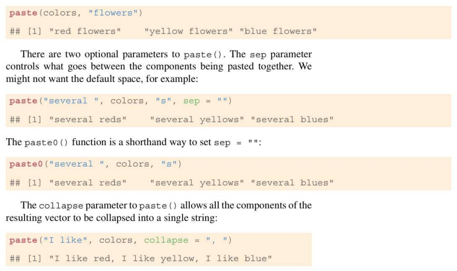
Behavior:

- Passing arguments
  function(par1 = x, par2 = y, par3 = z)
    - Positioning
        - function(, , arg3)
        - function(arg1, arg2)
    - Assignment
        - function(par2 = arg2)
    - Combination
        - function(, arg2, par3 = arg3)

- Most functions in R take arguments as pass by value not reference:
    - purple functions - pass by value
    - pink functions - pass by reference

Special functions:

- c(NULL{default}) - concatenate or combine vectors/literals
    - c("a" = 1, "b" = 2, "c" = 3)
- seq(from = 1, to = 1, by = 2) - sequence
- rep(x, each/times = 1)
    - rep(1:3, 1:3) = 1 2 2 3 3 3
- sample(x, size, replace = FALSE, prob = NULL)
    - sample(5) = 2 1 3 5 4
    - sample(2:6) = 6 5 2 4 3
- substr(x, start, stop)
    - substr("abcde", 2, 10) = b c d e
- paste (..., sep = " ", collapse = NULL, recycle0 = FALSE)
    - paste("a", c()))



```
paste(colors, "flowers")

## [1] "red flowers"    "yellow flowers" "blue flowers"
```

There are two optional parameters to `paste()`. The `sep` parameter controls what goes between the components being pasted together. We might not want the default space, for example:

```
paste("several ", colors, "s", sep = "")

## [1] "several reds"    "several yellows" "several blues"
```

The `paste0()` function is a shorthand way to set `sep = ""`:

```
paste0("several ", colors, "s")

## [1] "several reds"    "several yellows" "several blues"
```

The `collapse` parameter to `paste()` allows all the components of the resulting vector to be collapsed into a single string:

```
paste("I like", colors, collapse = ", ")

## [1] "I like red, I like yellow, I like blue"
```

Informative functions:

- summary(x) - gives the mean, median, min, max, 1st quartile, 2nd quartile of x
- which(x, arr.ind = FALSE, useNames = TRUE) - returns indices of elements in a logical vector that are TRUE.
    - List <- 1:5
      which(List > 4)
      ## 5
- print(x,...) - display the value(s) of an object
- length() - number of elements in an object
- dim() - returns dimensions of an object
- nrow()/NROW() - number of rows
- ncol()/NCOL() - number of columns
- pie(x, labels = names(x), edges = 200, radius = 0.8, clockwise = FALSE, init.angle = if(clockwise) 90 else 0, density = NULL, angle = 45, col = NULL, border = NULL, lty = NULL, main = NULL, ...)
- barplot(height, ...) - Creates a bar plot with vertical or horizontal bars.
- names() - provides the vertical headers of a vector/dataframe
- colnames() - provides the vertical headers of a dataframe
- rownames() - provides the horizontal headers of a dataframe

Mathematical functions:

for co-variance
remove NA in set
method handle fc

- var(x, y = NULL, na.rm = FALSE, use)

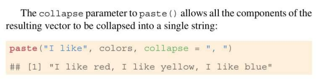
$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - x)^2$$

Generate random numbers from:

- rexp(n, rate = 1)
  the exponential distribution.
- runif(n, min = 0, max = 1)
  the uniform distribution.
- rnorm(n, mean = 0, sd = 1)
  the normal (Gaussian) distribution.
- rpois(n, lambda)
  the Poisson distribution.

- **n**: The number of observations.
- **rate**: The rate parameter (λ). The mean of the exponential distribution is 1/rate1 / \text{rate}.
- **min**: The minimum value of the distribution.
- **max**: The maximum value of the distribution.
- **mean**: The mean of the distribution.
- **sd**: The standard deviation of the distribution.
- **lambda**: The rate parameter (λ), which is the mean of the Poisson distribution.

- exp(x) - x^e
- factorial(x) - x!
- sum(x)
- pi = 3.141592653589793
- sin(x), cos(x), tan(x) - x in radians



The `collapse` parameter to `paste()` allows all the components of the resulting vector to be collapsed into a single string:

```
paste("I like", colors, collapse = ", ")

## [1] "I like red, I like yellow, I like blue"
```

vector/dataframe

- colnames() - provides the vertical headers of a dataframe
- rownames() - provides the horizontal headers of a dataframe

- example(topic = function_name), R searches for the function_name in its help files and executes the code in the "Examples" section of that help file. The results are displayed in the R console.

- pi = 3.141592653589793

- sin(x), cos(x), tan(x) - x in radians
- sinpi(x) … - x in degrees

- log(x, base = exp(1))

- diff(x1,x2,….)
  - diff(c(100,50))
    -50
  - diff(c(50,100))
    50
  - diff(c(10,1,100))
    -9 99

- range(vector) - lowest and highest number
  - range(c(1,2,3,4))
    1 4

- IQR(vector) - interquartile range
  - Datasets with larger IQR are harder to
- cumsum() - return cumulative sum
  - cumsum(c(1,2,3,4))
    1 3 6 10

WORKSPACE & ENVIRONMENT

- q(save = "default", status = 0, runLast = TRUE) - quit session
- objects() / ls() - lists all objects in current workspace
- rm(…) - remove object(s) from workspace
- search() - list of all attached packages
- getwd() - get the current working directory path
- setwd(path) - set the current working directory path to
- set.seeds()
- save.image() - save workspace image
- options(digits = x) - number of significands digits to be printed; default is 7 and maximum is 15.
- options(width = x) - specifying the maximum number of columns per line; default is 80; range from 10 to 10,000

- save() - see below

FILES & PACKAGE

FILES

EXPORT

Save as a single object in .rds binary format

- saveRDS(object, file = "", ascii = FALSE, version = NULL, compress = TRUE, refhook = NULL)
  - saveRDS(anRObject, "name.rds")

Saving one or more objects in .Rdata/rda binary format

- save(..., file, ascii = FALSE, version = NULL, compress = TRUE, compression_level, eval.promises = TRUE, precheck = TRUE)
  - save(rObject1, rObject2, rObject3, rObject4, file="name.rda")

Save a single dataframe/ matrice/ vector as:

BRING TO STATEMENT

Read:

.rds [Paired with saveRDS()]

- readRDS()

.csv, .txt

- read.table(file, header = FALSE, sep = "", quote = "\" ' ", dec = ".", <manymoreargs>)
  - Ensure header = T to retain header row

IMPORT

Import:

.Rdata/.rda [Paired with save()]

- load(file, envir = parent.frame(), verbose = FALSE)

OTHER

- dump(list, file = "", envir = parent.frame(), control = "all")

compression_level, eval.promises = TRUE, precheck = TRUE)
    - save(rObject1, rObject2, rObject3, rObject4, file="name.rda")

Save a single dataframe/ matrice/ vector as:

.csv, .txt

- write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ", eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE, qmethod = c("escape", "double"), fileEncoding = "")

- write.csv(…)

.xlsx

- write.xlsx(df, file) [from xlsx package]

PACKAGES

- search() - list of all attached packages
- install.packages(pkgs, repos = "https://cloud.r-project.org", dependencies = NA, lib = .libPaths()[1], ...) - download packages that are hosted on CRAN.
- library(package, help, pos = 2, lib.loc = NULL, character.only = FALSE, logical.return = FALSE, warn.conflicts = TRUE, quietly = FALSE, verbose = getOption("verbose")) - load and attach add-on packages

.csv, .txt

- read.table(file, header = FALSE, sep = "", quote = "\" ' ", dec = ".", <manymoreargs>)
    - Ensure header = T to retain header row

- read.csv(file, header = TRUE, sep = ",", quote = "\"", dec = ".", fill = TRUE, comment.char = "", ...)

.xlsx

- read.xlsx(xlsxFile, sheet = 1, startRow = 1, colNames = TRUE, rowNames = FALSE, detectDates = FALSE, skipEmptyRows = TRUE, rows = NULL, cols = NULL, check.names = FALSE, namedRegion = NULL)

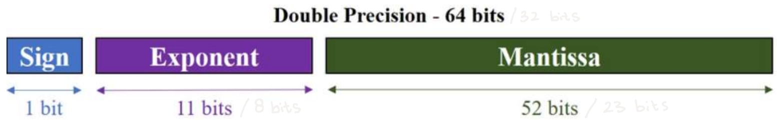
- read_excel(path, sheet = NULL, range = NULL, col_names = TRUE, col_types = NULL, na = "", trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000, n_max), progress = readxl_progress(), .name_repair = "unique")

OTHER

- dump(list, file = "", envir = parent.frame(), control = "all") dump() function creates a text representation of one or more R objects. This text representation can be sourced back into R to recreate the objects, making it useful for sharing, documenting, and version controlling R objects in a human-readable format.

Midterm 2 cheat sheet

04 March 2025 16:15



Example: convert -42.625 into binary (32 bit)

1 - SIGN

- if number is negative : 1
- if number is positive : 0

$-46.625 < 0$, so:

SIGN BIT: 1

Turn 42.625 in binary representation

NOTE: Ignore the sign for this section

A - INTEGER PART (46)

operand	divisor	quotient	remainder
46	÷2	23	0
23	÷2	11	1
11	÷2	5	1
5	÷2	2	1
2	÷2	1	0
1	÷2	0	1

- Stop when $\boxed{\text{quotient}}$ is zero (or till required precision)
- Read $\boxed{\text{remainder}}$ from $\boxed{\text{bottom to top}}$:

INTEGER BITS: 101110

B - DECIMAL PART (0.625)

operand	multiplicator	quotient	remainder
0.625	x2	1	0.25
0.25	x2	0	0.50
0.5	x2	1	0

- Stop when $\boxed{\text{remainder}}$ is zero (or till required precision)
- Read $\boxed{\text{quotient}}$ from $\boxed{\text{top to bottom}}$:

DECIMAL BITS: .101

C - COMBINE

101110.101

Turn 101110.101 into binary exponent (2^x)

$$\underset{5\,4\,3\,2\,1}{101110.101} \longrightarrow 1.0110101 \cdot 2^{5}$$

(Most likely this is the answer she is looking for)

2 - EXPONENT

- if 32 bit binary precision add 127 to power (127 bias):

$2^{⑤} \rightarrow 5 + 127 = 132$

NOTE: for 64 bit add 1023 (1023 bias)

- convert 132 to binary:

10000100

EXPONENT BITS: 10000100

3 - MANTISSA / SIGNIFICAND

- use the binary representation

101110101

drop 1

- drop the first digit (1); first digit will always be 1 unless number to represent is zero.

01110101

- fill the remaining with zero to make 23

1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	1	1	0	1	0	1	0	0	0	0	0	0

15	16	17	18	19	20	21	22	23
0	0	0	0	0	0	0	0	0

NOTE: for 64 bit fill to 52 bits

MANTISSA BITS: 01110101000000000000000

4 - COMBINE BITS

$$\underbrace{\boxed{1}}_{\text{SIGN}}\underbrace{\boxed{10000100}}_{\text{EXPONENT}}\underbrace{\boxed{01110101000000000000000}}_{\text{MANTISSA}}$$

L10

- Precision errors in R:
-
```
n <- 1:10
1.25 * (n * 0.8) - n

##  [1] 0.000000e+00 0.000000e+00 4.440892e-16 0.000000e+00 0.000000e+00
##  [6] 8.881784e-16 8.881784e-16 0.000000e+00 0.000000e+00 0.000000e+00
```

The `%in%` operator tests whether elements of one vector can be found in another vector.

Note that `y %in% x` is not the same as `x %in% y`.

If we only want to study the measurements of the eggs laid in the robin and wren nests, we can subset the data with the `%in%` operator as in

```
cuckooWrenRobin <- subset(cuckoos[, -4],
     species %in% c("robin", "wren"))
```

not B	A and B	A or B
`!B`	`A & B`	`A	B`

If we attempt logical operations on a numerical vector, 0 is taken to be `FALSE`, and any non-zero value is taken to be `TRUE`:

```
all.equal(x, y) # this function tests approximate equality
```

When reading in a file with columns separated by blanks with blank missing values, you can use code such as

```
dataset1 <- read.table("file1.txt", header=TRUE,
     sep=" ", na.string=" ")
```

This tells R that the blank spaces should be read in as missing values.

```
dataset2
##    x   y  z
## 1 33 223 NA
## 2 32  88  2
## 3  3  NA NA
```

```
summary(x) # computes several summary statistics on the data in x
length(x)  # number of elements in x
min(x)     # minimum value of x
max(x)     # maximum value of x
pmin(x, y) # pairwise minima of corresponding elements of x and y
pmax(x, y) # pairwise maxima of  x and y
range(x)   # difference between maximum and minimum of data in x
IQR(x)     # interquartile range:  difference between 1st and 3rd
           # quartiles of data in x
sd(x)      # computes the standard deviation of the data in x
var(x)     # computes the variance of the data in x
diff(x)    # successive differences of the values in x
sort(x)    # arranges the elements of x in ascending order
```

```
list <- c("apple", "mango", "orange", "guava", "mango", "guava")
```
level(x) - creates non-repeating labels for the vector x
- level creates an order alphabetically

factor(x) - creates levels for the vector x and assigns it to x

```
factor(list)
[1] apple  mango  orange guava  mango  guava
Levels: apple guava mango orange

levels(list)
[1] "apple"  "guava"  "mango"  "orange"

as.character(factor(list))
[1] "apple"  "mango"  "orange" "guava"  "mango"  "guava"
```

dataframe$variable - gives all the values of the column variable

- When comparing median between 2 sets, ensure that the condition for comparison is even. For example:
  - Ensure that the timeframe for both sets are the same.

  Ex: pg6 L09 slides
  - Ensure units are same.
  - Ensure year range of data sets are same.

The function ts is used to create time-series objects.
```
ts(data = NA, start = 1, end = numeric(), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"),
   class = if(nseries > 1) c("mts", "ts", "matrix", "array") else
"ts",
   names = )
```

```
as.character(factor(list))
[1] "apple"  "mango"  "orange" "guava"  "mango"  "guava"
```

dataframe$variable - gives all the values of the column variable

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

```
subset(x, subset, ...)
```

Return subsets of vectors, matrices or data frames which meet conditions.

- Give example with condition

```
head(x, n = 6L, ...)
```

Returns the first or last parts of a vector, matrix, table, data frame or function.

```
## Default S3 method:
aggregate(x, ...)

## S3 method for class 'data.frame'
aggregate(x, by, FUN, ..., simplify = TRUE, drop = TRUE)
```

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

- `aggregate(weight~feed, data = chickwts, FUN = mean) VS`
- `aggregate(feed~weight, data = chickwts, FUN = mean)`

By default, when you create a data.frame in R, it converts character vectors to factors. This behavior can be controlled using the stringsAsFactors argument. If you don't want character vectors to be converted to factors, you can set stringsAsFactors to FALSE

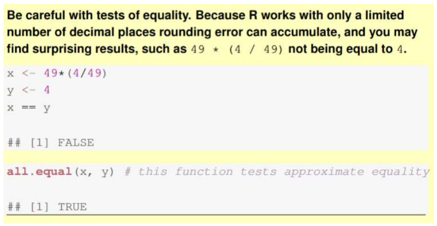
The function ts is used to create time-series objects.

```
ts(data = NA, start = 1, end = numeric(), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"),
   class = if(nseries > 1) c("mts", "ts", "matrix", "array") else
"ts",
   names = )
```

as.ts and is.ts coerce an object to a time-series and test whether an object is a time series.

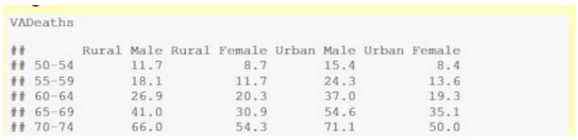
- Know how to:
  - Create a time series
  - Convert times from monthly to yearly or vice-versa
  - Query items in a dataframe
- Range will always return positive values
- Larger IQR have less predicitability
- 0/0 returns NaN - Not a Number
- 1/0 return Inf - Infinity

  The %in% operator tests whether elements of one vector can be found inanother vector

- The %in% operator tests whether elements of one vector can be found in another vector
- 

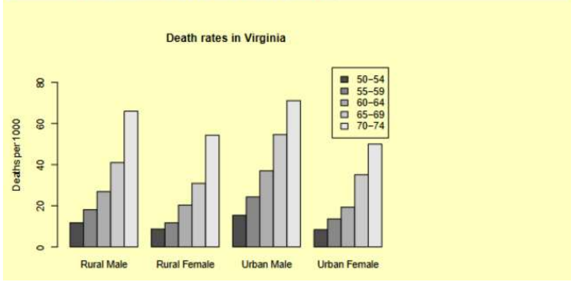
Be careful with tests of equality. Because R works with only a limited number of decimal places rounding error can accumulate, and you may find surprising results, such as `49 * (4 / 49)` not being equal to `4`.

```
x <- 49*(4/49)
y <- 4
x == y

## [1] FALSE

all.equal(x, y) # this function tests approximate equality

## [1] TRUE
```

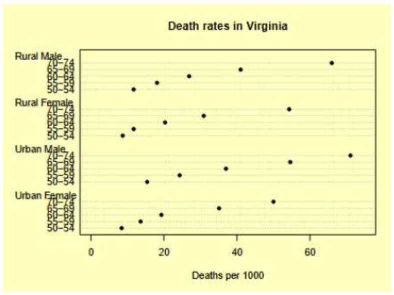


```
VADeaths

##       Rural Male Rural Female Urban Male Urban Female
## 50-54       11.7          8.7       15.4          8.4
## 55-59       18.1         11.7       24.3         13.6
## 60-64       26.9         20.3       37.0         19.3
## 65-69       41.0         30.9       54.6         35.1
## 70-74       66.0         54.3       71.1         50.0
```

```
barplot(VADeaths, beside = TRUE, legend = TRUE, ylim = c(0, 90),
        ylab = "Deaths per 1000",
        main = "Death rates in Virginia")
```

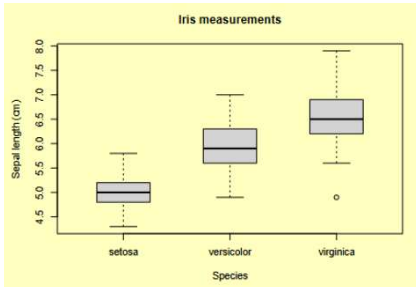


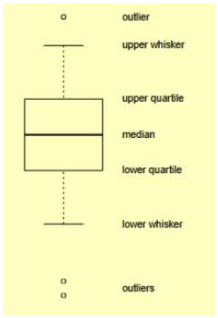
```
dotchart(VADeaths, xlim=c(0, 75),
         xlab="Deaths per 1000",
         main="Death rates in Virginia", pch=16)
```

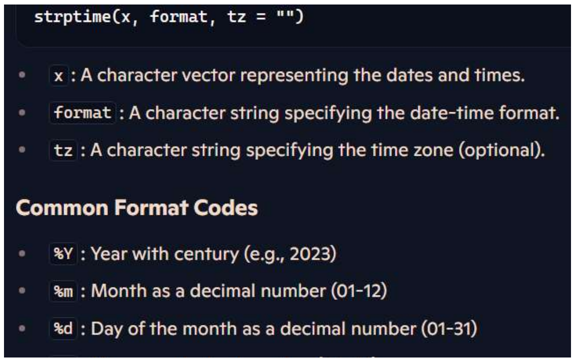


```
boxplot(Sepal.Length ~ Species, data = iris,
        ylab = "Sepal length (cm)",
        main = "Iris measurements", boxwex = 0.5)
```

function: the syntax `Sepal.Length ~ Species` is read as "Sepal.Length depending on Species", where both are columns of the data frame specified by `data = iris`.





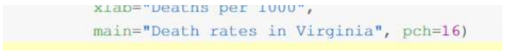


```
strptime(x, format, tz = "")
```

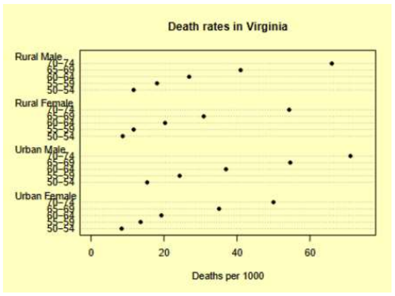
- `x` : A character vector representing the dates and times.
- `format` : A character string specifying the date-time format.
- `tz` : A character string specifying the time zone (optional).

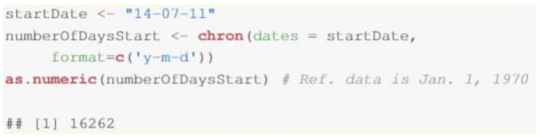
Common Format Codes

- `%Y` : Year with century (e.g., 2023)
- `%m` : Month as a decimal number (01-12)
- `%d` : Day of the month as a decimal number (01-31)

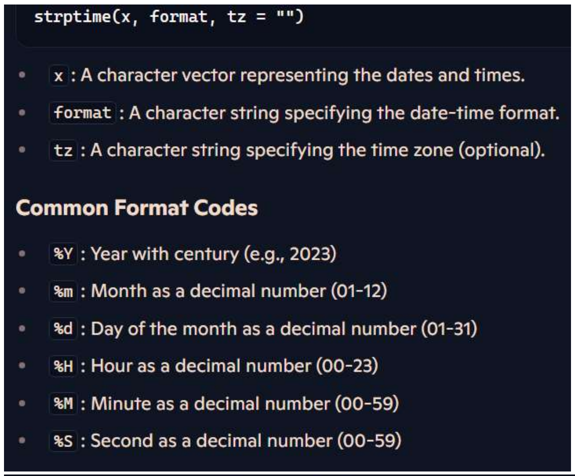


```
        xlab="Deaths per 1000",
        main="Death rates in Virginia", pch=16)
```





```
startDate <- "14-07-11"
numberOfDaysStart <- chron(dates = startDate,
      format=c('y-m-d'))
as.numeric(numberOfDaysStart) # Ref. data is Jan. 1, 1970

## [1] 16262
```

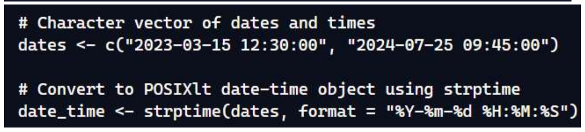


```
strptime(x, format, tz = "")
```

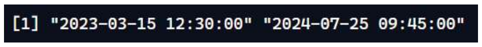
- `x` : A character vector representing the dates and times.
- `format` : A character string specifying the date-time format.
- `tz` : A character string specifying the time zone (optional).

Common Format Codes

- `%Y` : Year with century (e.g., 2023)
- `%m` : Month as a decimal number (01-12)
- `%d` : Day of the month as a decimal number (01-31)
- `%H` : Hour as a decimal number (00-23)
- `%M` : Minute as a decimal number (00-59)
- `%S` : Second as a decimal number (00-59)



```
# Character vector of dates and times
dates <- c("2023-03-15 12:30:00", "2024-07-25 09:45:00")

# Convert to POSIXlt date-time object using strptime
date_time <- strptime(dates, format = "%Y-%m-%d %H:%M:%S")
```



```
[1] "2023-03-15 12:30:00" "2024-07-25 09:45:00"
```

3.1 Simple high level plots

In this section we will discuss several basic plots. The functions to draw these in R are called “high level” because you don’t need to worry about the details of where the ink goes; you just describe the plot you want, and R does the drawing.

3.1.1 Bar charts and dot charts

The most basic type of graph is one that summarizes a single set of numbers. Bar charts and dot charts do this by displaying a bar or dot whose length or position corresponds to the number.

Example 3.1

Figure 3.1 displays a basic bar chart based on a built-in data set. The `WorldPhones` matrix holds counts of the numbers of telephones in the major regions of the world for a number of years. The first row of the matrix corresponds to the year 1951. In order to display these data graphically, we first extract that row.

```
WorldPhones51 <- WorldPhones[1, ]
WorldPhones51

##   N.Amer   Europe     Asia   S.Amer  Oceania   Africa Mid.Amer
##    45939    21574     2876     1815     1646       89      555
```

The default code to produce a bar chart of these data using the `barplot()` function is

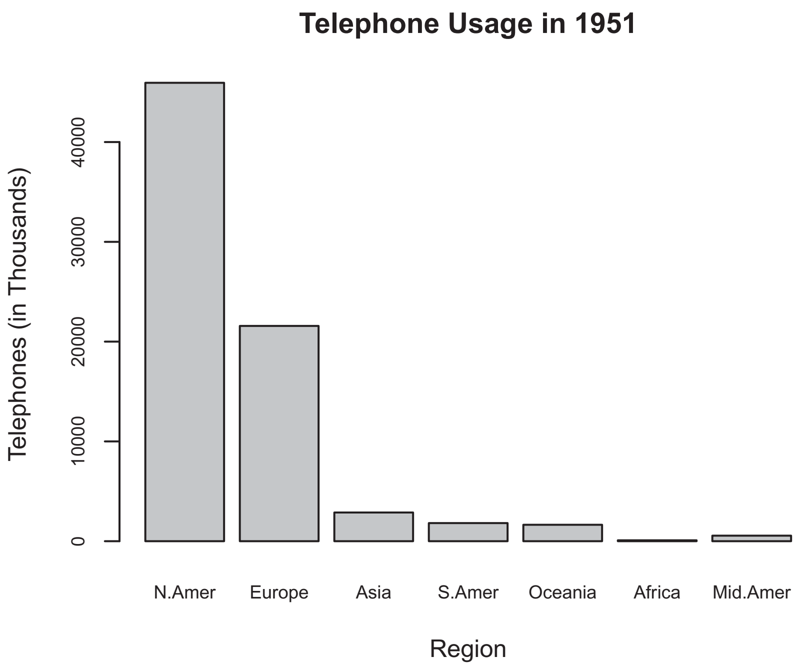


Fig. 3.1 A bar chart displaying the numbers of telephones (in thousands) in the various regions of the world in 1951.

```
barplot(WorldPhones51)
```

Some minor changes are required in order for the plot to be satisfactory: we'd like to display a title at the top, to include informative axis labels, and to reduce the size of the text associated with the axes. These changes can be carried out with the following code, yielding the result that appears in Figure 3.1.

```
barplot(WorldPhones51, main = "Telephone Usage in 1951", cex.names = 0.75,
           cex.axis = 0.75, ylab = "Telephones (in Thousands)", xlab="Region")
```

Understanding the code

The `cex.names = 0.75` argument reduced the size of the region names to 0.75 of their former size, and the `cex.axis = 0.75` argument reduced the labels on the vertical axis by the same amount. The `main` argument sets the main title for the plot, and the `ylab` and `xlab` arguments are used to include axis labels.

An alternative way to plot the same kind of data is in a dot chart (Figure 3.2):

```
dotchart(WorldPhones51, xlab = "Numbers of Phones ('000s)")
```

The values are shown by the horizontal positions of the dots.

Data sets having more complexity can also be displayed using these graphics functions. The `barplot()` function has a number of options which allow for side-by-side or stacked styles of displays, legends can be included using the `legend` argument, and so on.

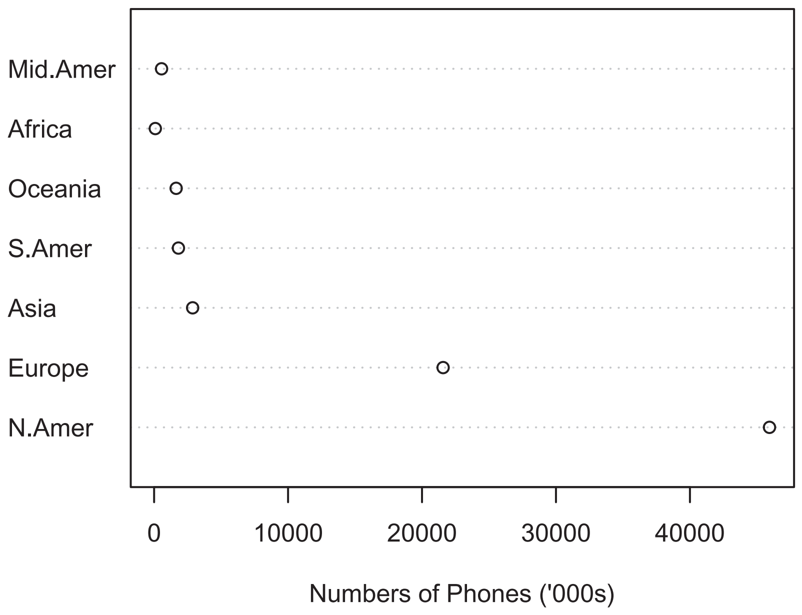


Fig. 3.2 World telephone counts in 1951 displayed as a dot chart.

Example 3.2
The `VADeaths` data set in R contains death rates (number of deaths per 1000 population per year) in various subpopulations within the state of Virginia in 1940.

```
VADeaths

##         Rural Male Rural Female Urban Male Urban Female
## 50-54         11.7          8.7       15.4          8.4
## 55-59         18.1         11.7       24.3         13.6
## 60-64         26.9         20.3       37.0         19.3
## 65-69         41.0         30.9       54.6         35.1
## 70-74         66.0         54.3       71.1         50.0
```

This data set may be displayed as a sequence of bar charts, one for each subgroup (Figure 3.3):

```
barplot(VADeaths, beside = TRUE, legend = TRUE, ylim = c(0, 90),
        ylab = "Deaths per 1000",
        main = "Death rates in Virginia")
```

Understanding the code

The bars correspond to each number in the matrix. The `beside = TRUE` argument causes the values in each column to be plotted side-by-side; `legend = TRUE` causes the legend in the top right to be added. The `ylim = c(0, 90)` argument modifies the vertical scale of the graph to make room for the legend. (We will describe other ways to place the legend in Section 3.3.) Finally, `main = "Death rates in Virginia"` sets the main title for the plot.

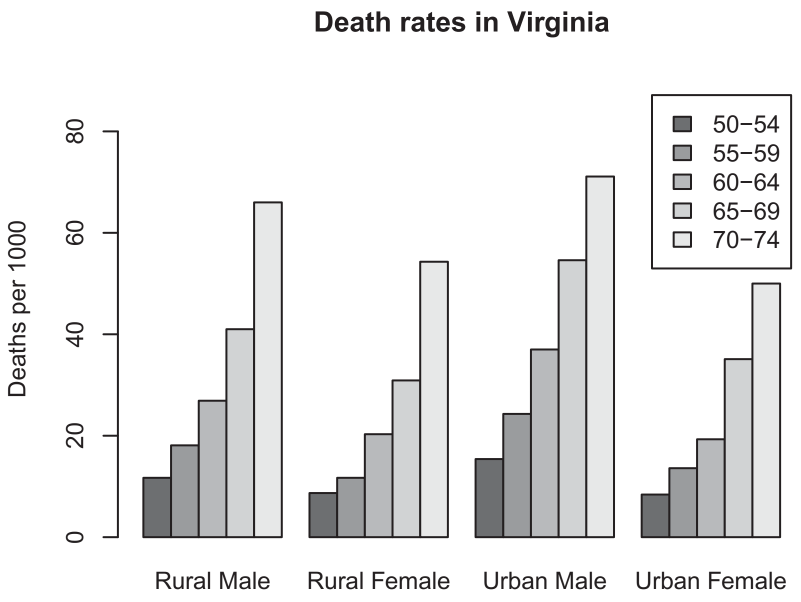


Fig. 3.3 An example of a complex bar chart.

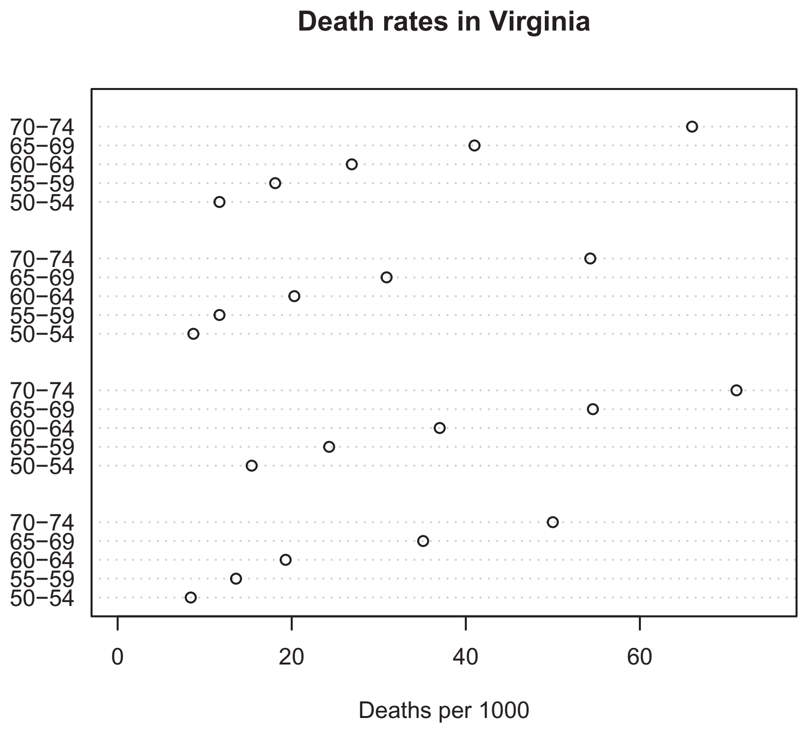


Fig. 3.4 A dot chart interpretation of the Virginia Death Rate data set.

Again, the dot chart offers an alternative way of viewing the data (Figure 3.4). We will discuss criteria to use to choose between these alternatives in Section 3.2.

Example 3.3

```
dotchart(VADeaths, xlim = c(0, 75), xlab = "Deaths per 1000",
         main = "Death rates in Virginia", cex = 0.8)
```

Understanding the code

We set the x-axis limits to run from 0 to 75 so that zero is included, because it is natural to want to compare the total rates in the different groups. We have also set `cex` to 0.8. This shrinks the plotting character to 80% of its default size, but more importantly, shrinks the axis tick labels to 80% of their default size. For this example, the default setting would cause some overlapping of the tick labels, making them more difficult to read.

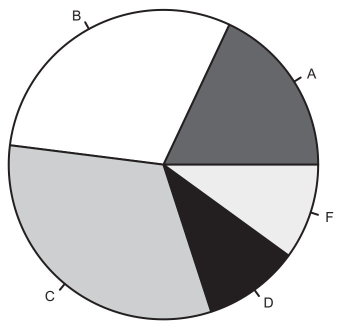


Fig. 3.5 A pie chart showing the distribution of grades in a class.

3.1.2 Pie charts

Pie charts display a vector of numbers by breaking up a circular disk into pieces whose angle (and hence area) is proportional to each number. For example, the letter grades assigned to a class might arise in the proportions, A: 18%, B: 30%, C: 32%, D: 10%, and F: 10%. These data are graphically displayed in Figure 3.5, which was drawn with the R code

```
groupsizes <- c(18, 30, 32, 10, 10)
labels <- c("A", "B", "C", "D", "F")
pie(groupsizes, labels,
    col = c("grey40", "white", "grey", "black", "grey90"))
```

Pie charts are popular in non-technical publications, but they have fallen out of favor with statisticians. Some of the reasons why will be discussed in Section 3.2.

3.1.3 Histograms

A histogram is a special type of bar chart that is used to show the frequency distribution of a collection of numbers. Each bar represents the count of x values that fall in the range indicated by the base of the bar. Usually all bars have the same width; this is the default in R. In this case, the height of each bar is proportional to the number of observations in the corresponding interval. If bars have different widths, then the *area* of the bar should be proportional to the count; in this way the height represents the density (i.e. the frequency per unit of x).

In base graphics, `hist(x, ...)` is the main way to plot histograms. Here `x` is a vector consisting of numeric observations, and optional parameters in `...` are used to control the details of the display.

An example of a histogram (of the areas of the world's largest landmasses) drawn with the default parameter settings was given in Section 2.7.1. The histogram bars decrease in height, roughly exponentially, in the direction of increasing land area (along the horizontal axis). It is often recommended that measurements from such *skewed* distributions be displayed on a logarithmic scale. This is demonstrated in Figure 3.6,

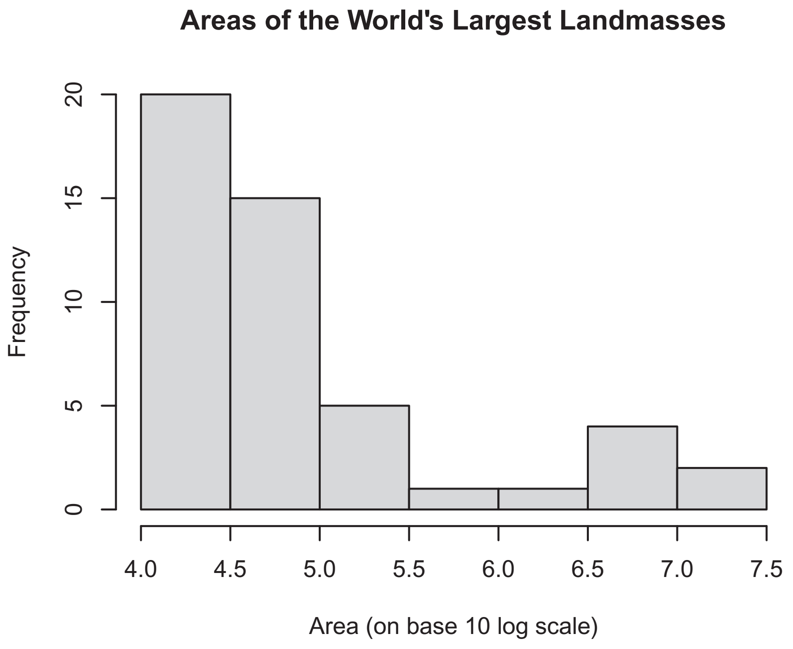


Fig. 3.6 Histogram of landmass areas on the log scale, with better axis labeling and a title.

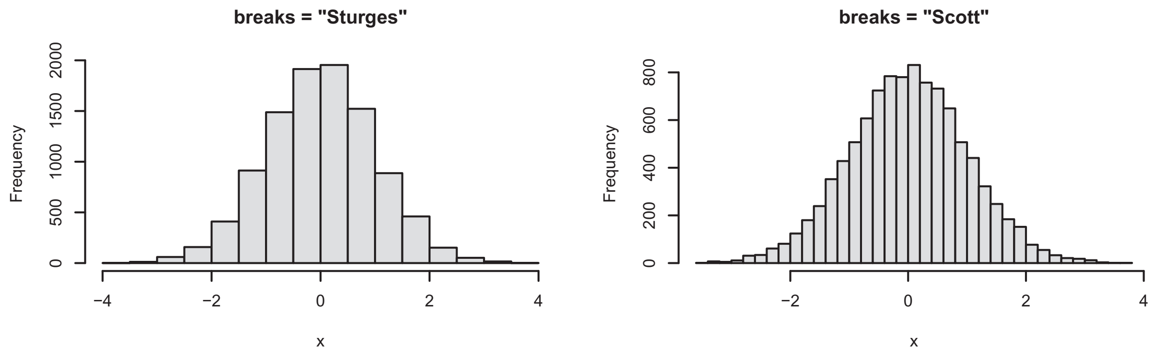


Fig. 3.7 Histograms of the values in a vector `x` of length 10000, using two different rules for setting the breaks.

together with better axis labeling and a title. Further improvements are illustrated in Appendix B.

```
hist(log(1000*islands, 10),  xlab = "Area (on base 10 log scale)",
    main = "Areas of the World's Largest Landmasses")
```

If you have n values of `x`, R, by default, divides the range into approximately $\log_2(n) + 1$ intervals, giving rise to that number of bars.

In the `islands` data set, there are

```
length(islands)

## [1] 48
```

measurements, so $n = 48$. Since

$$48 > 2^5 = 32$$

$$48 < 2^6 = 64$$

$$5 < \log_2(48) < 6$$

it can be seen that R should choose about 5 or 6 bars. In fact, it chose 8, because it also attempts to put the breaks at round numbers (multiples of 0.5 in this case).

The $\log_2(n) + 1$ rule (known as the “Sturges rule”) is not always satisfactory for large values of n, giving too few bars. Current research suggests that the number of bars should increase proportionally to $n^{1/3}$ instead of $\log_2(n)$. The `breaks = "Scott"` and `breaks = "Freedman-Diaconis"` options provide variations on this choice. Figure 3.7 shows the results for a simulated 10000 point data set, generated from a symmetric distribution, using the “Sturges” and “Scott” rules.

3.1.4 Boxplots

A boxplot (or "box-and-whisker plot") is an alternative to a histogram to give a quick visual display of the main features of a set of data. A rectangular box is drawn, together with lines which protrude from two opposing sides. The box gives an indication of the location and spread of the central portion of the data, while the extent of the lines (the "whiskers") provides an idea of the range of the bulk of the data. In some implementations, outliers (observations that are very different from the rest of the data) are plotted as separate points.

The basic construction of the box part of the boxplot is as follows:

1. A horizontal line is drawn at the median.
2. Split the data into two halves, each containing the median.
3. Calculate the upper and lower quartiles as the medians of each half, and draw horizontal lines at each of these values. Then connect the lines to form a rectangular box.

The box thus drawn defines the *interquartile range* (IQR). This is the difference between the upper quartile and the lower quartile. We use the IQR to give a measure of the amount of variability in the central portion of the data set, since about 50% of the data will lie within the box.

The lower whisker is drawn from the lower end of the box to the smallest value that is no smaller than 1.5 IQR below the lower quartile. Similarly, the upper whisker is drawn from the middle of the upper end of the box to the largest value that is no larger than 1.5 IQR above the upper quartile. The rationale for these definitions is that when data are drawn from the normal distribution or other distributions with a similar shape, about 99% of the observations will fall between the whiskers.

An annotated example of a boxplot is displayed in Figure 3.8. Boxplots are convenient for comparing distributions of data in two or more categories, with a number (say 10 or more) of numerical observations per category. For example, the iris data set in R is a well-studied data set of measurements of 50 flowers from each of three species of iris. Figure 3.9, produced by the code

```
boxplot(Sepal.Length ~ Species, data = iris,
        ylab = "Sepal length (cm)", main = "Iris measurements",
        boxwex = 0.5)
```

compares the distributions of the sepal length measurements between the different species. Here we have used R's formula-based interface to the graphics function: the syntax `Sepal.Length ~ Species` is read as "Sepal.Length depending on Species," where both are columns of the data frame specified by `data = iris`. The `boxplot()` function draws separate side-by-side boxplots for each species. From these, we can see substantial differences between the mean lengths for the species, and that there is one unusually small specimen among the *virginica* samples.

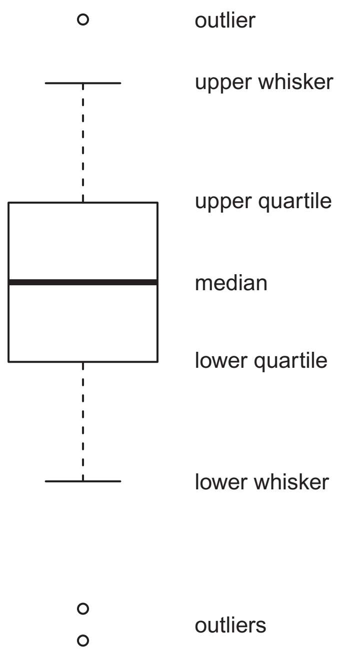


Fig. 3.8 Construction of a boxplot.

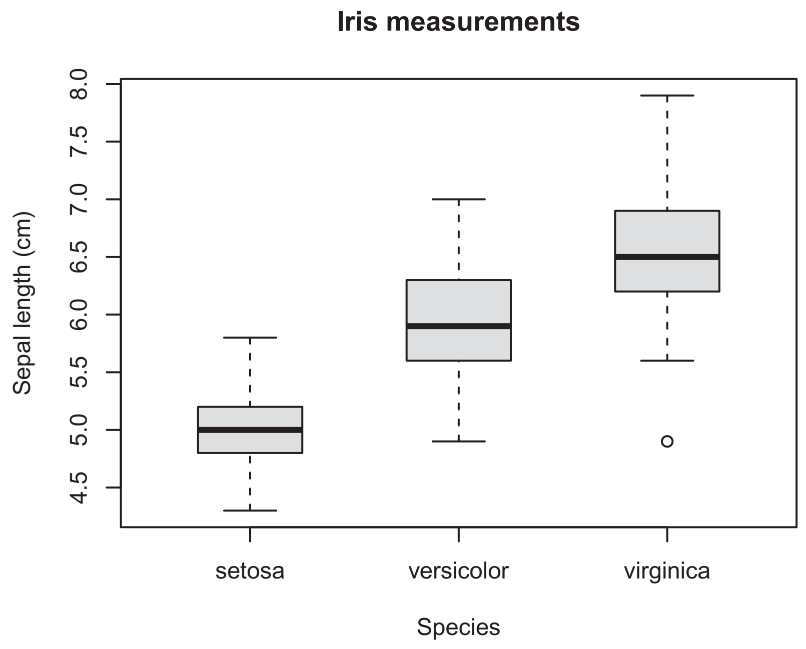


Fig. 3.9 An example of side-by-side boxplots.

3.1.5 Scatterplots

When doing statistics and data science, most of the interesting problems have to do with the relationships between different variables. To study this, one of the most commonly used plots is the scatterplot, in which points (x_i, y_i), $i = 1, \ldots, n$ are drawn using dots or other symbols. These are drawn to show relationships between the x_i and y_i values. In R, scatterplots (and many other kinds of plots) are drawn using the `plot()` function. Its basic usage is `plot(x, y, ...)` where `x` and `y` are numeric vectors of the same length holding the data to be plotted. There are many additional optional arguments, and versions of `plot` designed for non-numerical data as well.

One important optional argument is `type`. The default is `type = "p"`, which draws a scatterplot. Line plots (in which line segments join the (x_i, y_i) points in order from first to last) are drawn using `type = "l"`. Many other types are available, including `type = "n"`, to draw *nothing*: this just sets up the frame around the plot, allowing other functions to be used to draw in it. Some of these other functions will be discussed in Section 3.3.

Many other types of graphs can be obtained with this function. We will show how to explore some of the options using some artificial data. Two vectors of numbers will be simulated, one from a standard normal distribution and the other from a Poisson distribution having mean 30.[1]

[1] See Chapter 5 for more information on the simulation of normal and Poisson random variables.

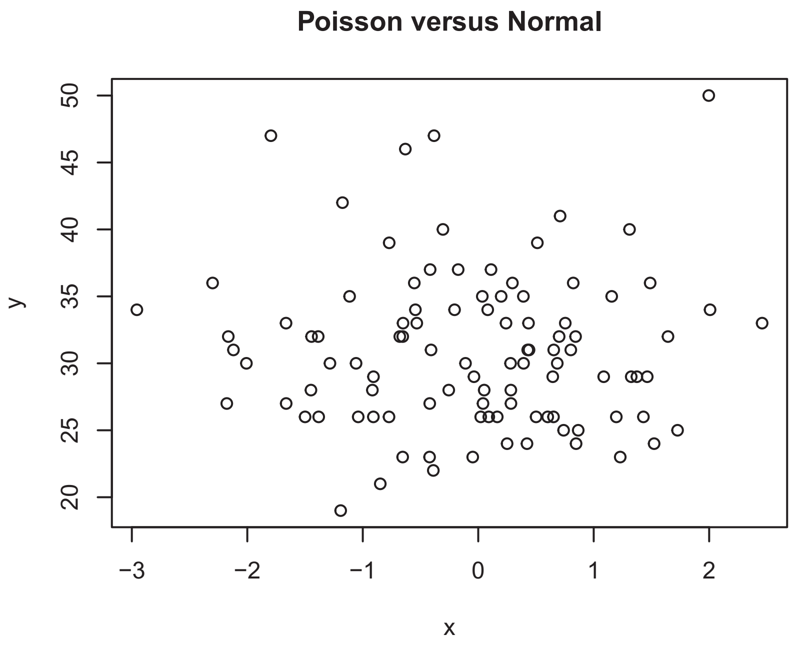


Fig. 3.10 An example of a scatterplot.

```
x <- rnorm(100)        # assigns 100 random normal observations to x
y <- rpois(100, 30)    # assigns 100 random Poisson observations
                       # to y; mean value is 30
mean(y)                # the resulting value should be near 30

## [1] 30.91
```

The `main` argument sets the main title for the plot. Figure 3.10 shows the result of

```
plot(x, y, main = "Poisson versus Normal")
```

Other possibilities you should try:

```
plot(x, y, pch = 16)      # changes the plot symbol to a solid dot
plot(x, y, type = 'l')    # plots a broken line (a dense tangle of line
                          # segments here)
plot(sort(x), sort(y), type = 'l')   # a plot of the sample "quantiles"
```

3.1.6 Plotting data from data frames

Example 3.4
The `Orange` data frame is in the `datasets` package installed with R. It consists of 35 observations on the age (in days since December 31, 1968) and the corresponding circumference of five different orange trees, with identifiers

```
unique(as.character(Orange$Tree))

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

(Since `Orange$Tree` is a factor, we use `as.character()` to get the displayed form, and `unique()` to select the unique values.)

To get a sense of how circumference relates to age, we might try the following:

```
plot(circumference ~ age, data = Orange)
```

Understanding the code

We have used the graphics formula and the `data` argument as in the earlier boxplot example. The plot function finds `circumference` and `age` in the `Orange` data frame, and plots the ordered pairs of (age, circumference) observations.

Figure 3.11 hides important information: the observations are not all from the same tree, and they are not all from different trees; they are from five different trees, but we cannot tell which observations are from which tree. One way to remedy this problem is to use a different plotting symbol for each tree. The `pch` parameter controls the plotting character. The default setting `pch = 1` yields the open circular dot. Other numerical values of this parameter will give different plotting characters. We can also ask for different characters to be plotted; for example, `pch = "A"` causes R to plot the character `A`.

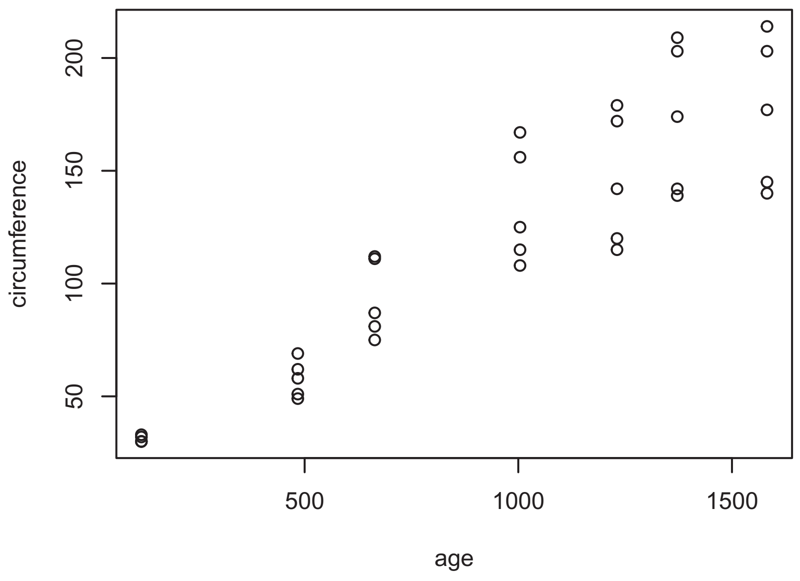


Fig. 3.11 A scatterplot of circumference versus age for five different orange trees.

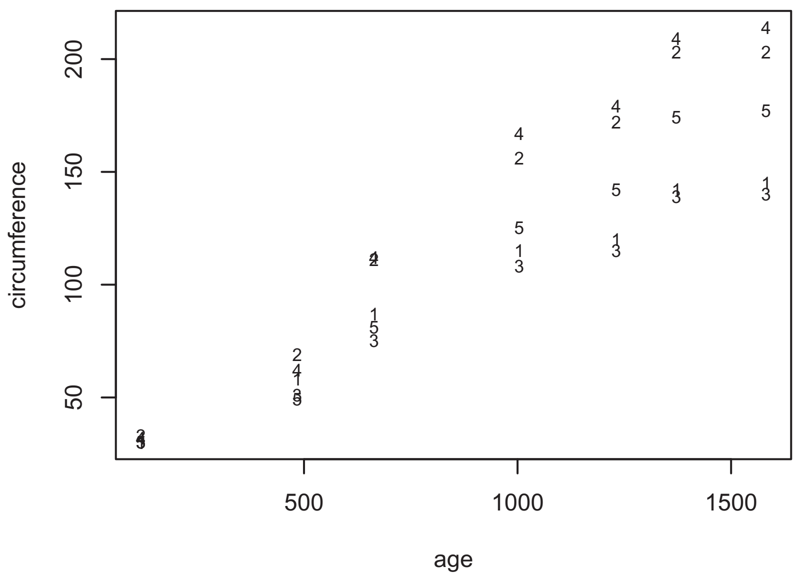


Fig. 3.12 A scatterplot of circumference versus age for five different orange trees.

Example 3.5
The following code can be used to identify the individual trees (Figure 3.12).

```
plot(circumference ~ age, data = Orange, pch = as.character(Tree),
     cex = 0.75)
```

Understanding the code

The `cex` parameter controls the size of the plotting character, and the `pch` parameter has been assigned the levels of the `Tree` column; because `Tree` is a factor, care must be taken in order that the level values are used, and not the factor codes, hence the use of `as.character()`.

There are many more optional arguments to the `plot()` function, described on the `?plot` and `?par` help pages.

3.1.7 QQ plots

Quantile-quantile plots (otherwise known as QQ plots) are a type of scatterplot used to compare the distributions of two groups or to compare a sample with a reference distribution.

In the case where there are two groups of equal size, the QQ plot is obtained by first sorting the observations in each group: $X[1] \leq \cdots \leq X[n]$ and $Y[1] \leq \cdots \leq Y[n]$. Next, draw a scatterplot of $(X[i], Y[i])$, for $i = 1, \ldots, n$.

When the groups are of different sizes, some scheme must be used to artificially match them. R reduces the size of the larger group to the size