CAR: Chapter 1

Contents

1 Arithmetic and basic objects **5**

1.1 Extended arithmetic 12

1.2 Complex numbers too 14

1.3 How numbers are printed 15

1.4 Example of using options and restoring prior state 17

1.5 Printing with `format` 20

1.6 typeof 22

2 Calling Functions **25**

3 Vectors and Variables **29**

3.1 Operations on vectors 33

3.2 Creating and naming objects 36

3.3 Random number generation 38

3.4 Character objects 45

3.5 Logical values 49

3.6 Truth tables 52

3.7 Special logical operators `&&` and `||` 60

3.8 How **coercion** works in R 62

3.9 The hierarchy of atomic types: promotion and demotion 64

3.10 Explicit coercion 66
3.11 Implicit coercion 68

4 Selecting elements of a vector 70

4.1 Selecting by position with positive numbers 70
4.2 Selecting by omission of positions with negative numbers 72
4.3 Selecting with a logical vector 74
4.4 Selecting with names 76
4.5 Selecting elements of a matrix 80

5 Operators are functions 87

6 Valid object names in R 88

7 Writing your own functions 89

8 Quick overview of basic regression and plots in R 94

8.1 Referencing variables in data frames 99

8.2 Added-variable plots 119

8.3 Component-plus-residual plots 123

This version rendered on January 13 2020 10:28
Based on scripts created for J. Fox and S. Weisberg (2019) An R Companion to Applied Regression, 3rd Edition

Load the package written for this textbook:

```
library(car)
```

```
Warning: package 'car' was built under R version 3.6.2
```

```
Loading required package: carData
```

This script introduces a number of principles in R. Understanding these principles goes a long way towards helping you become a productive user of R.

The name of each principle is shown in **bold**. You can search through The R Language Definition to pursue questions that arouse your curiosity.

Post questions and comments on Piazza.

1 Arithmetic and basic objects

Operators grouped in reverse order of **precedence**

```
2 + 3 # addition
```

```
[1] 5
```

```
2 - 3 # subtraction
```

```
[1] -1
```

```r
-3    # unary minus (binary: two arguments, unary: one argument)
```

```
[1] -3
```

```r
2*3   # multiplication
```

```
[1] 6
```

```r
2/3   # division
```

```
[1] 0.6666667
```

```r
2^3   # exponentiation
```

```
[1] 8
```

Examples

```
4^2 - 3*2
```

```
[1] 10
```

```
4 ^ 2-3 * 2  # are spaces important in arithmetic expressions?
```

```
[1] 10
```

```
1 - 6 + 4  # with equal precedence operations
```

```
[1] -1
```

```
# are executed from left to right
1 + 4 - 6
```

```
[1] -1
```

```
4^2 - 3*2
```

```
[1] 10
```

```
(4^2) - (3*2)
```

```
[1] 10
```

```
4^ (2 - 3)*2  # what happens here?
```

```
[1] 0.5
```

```
(4 + 3)^2
```

```
[1] 49
```

```r
4 + 3^2
```

```
[1] 13
```

```r
-2--3    # unary minus
```

```
[1] 1
```

```r
-2 - -3
```

```
[1] 1
```

```r
-2 - - 3
```

```
[1] 1
```

```
-3*-2
```

```
[1] 6
```

Which has higher precedence: exponentiation or unary minus?

Some systems, e.g. Excel, follow the opposite order of precedence

```
-3^2
```

```
[1] -9
```

```
0-3^2
```

```
[1] -9
```

```
(-3)^2
```

```
[1] 9
```

Integer division

```
10 %/% 3
```

```
[1] 3
```

Modulo arithmetic: remainder

```
10 %% 3
```

```
[1] 1
```

```
3 * (10 %/% 3) + 10 %% 3 # why?!
```

```
[1] 10
```

1.1 Extended arithmetic

```
1/0
```

```
[1] Inf
```

```
Inf - 1
```

```
[1] Inf
```

```
Inf + Inf
```

```
[1] Inf
```

```
Inf * 2
```

```
[1] Inf
```

```
Inf - Inf
```

```
[1] NaN
```

```
0/0
```

```
[1] NaN
```

1.2 Complex numbers too

```
0i
```

```
  [1] 0+0i
```

```
1i
```

```
  [1] 0+1i
```

Euler's identity

```
exp(1)
```

```
  [1] 2.718282
```

```
exp(1)^(1i*pi)
```

```
[1] -1+0i
```

```
exp(1)^(1i*pi) + 1  # complex machine 0
```

```
[1] 0+1.224606e-16i
```

1.3 How numbers are printed

```
1.23456789
```

```
[1] 1.234568
```

```r
1.23456789 * 10^8
```

```
[1] 123456789
```

```r
1.23456789e8  # same number in scientific notation
```

```
[1] 123456789
```

```r
1.23456789 * 10^10
```

```
[1] 12345678900
```

```r
1.23456789 * 10^12
```

```
[1] 1.234568e+12
```

```
1.23456789 * 10^-4
```

```
[1] 0.0001234568
```

```
1.23456789 * 10^-5
```

```
[1] 1.234568e-05
```

```
1.23456789 * 10^-10
```

```
[1] 1.234568e-10
```

1.4 Example of using options and restoring prior state

```
opts <- options(scipen = 15)  # penalty 'against' scientific notati
```

```
1.23456789
```

```
  [1] 1.234568
```

```
1.23456789 * 10^8
```

```
  [1] 123456789
```

```
1.23456789 * 10^10
```

```
  [1] 12345678900
```

```
1.23456789 * 10^12
```

```
[1] 1234567890000
```

```
1.23456789 * 10^-4
```

```
[1] 0.0001234568
```

```
1.23456789 * 10^-5
```

```
[1] 0.00001234568
```

```
1.23456789 * 10^-10
```

```
[1] 0.0000000001234568
```

```
opts
```

```
$scipen
[1] 0
```

```
options(opts)  # restoring previous state
```

To see more options in base R use `?options`

1.5 Printing with `format`

```
format(1.23456789 * 10^10)
```

```
[1] "12345678900"
```

```
format(1.23456789 * 10^10, big.mark = ',')
```

```
[1] "12,345,678,900"
```

```
format(123456789.1234, big.mark = ',')
```

```
[1] "123,456,789"
```

```
format(123456789.1234, big.mark = ',', nsmall = 2)
```

```
[1] "123,456,789.12"
```

adding a dollar sign:

```r
paste0('$',format(123456789.1234, big.mark = ',', nsmall = 2))
```

```
[1] "$123,456,789.12"
```

For more possibilities, see `?format` and `?prettyNum`. For people who know C, see `?formatC`

1.6 typeof

Every object in R has a 'type' that identifies its internal representation. You can find the type with the 'typeof' function.

Let's see the types of things we've seen so far:

```r
typeof(1:4)
```

```
[1] "integer"
```

```r
typeof(4)   # looks like an integer
```

```
[1] "double"
```

```r
typeof(4L)   # In 4L, L stands for L i.e. a 64 bit integer
```

```
[1] "integer"
```

```r
typeof(4L + 3L)
```

```
[1] "integer"
```

```r
typeof(4L + 3)
```

```
[1] "double"
```

```r
typeof(Inf)
```

```
[1] "double"
```

```r
typeof(options) # fancy term for a function with baggage
```

```
[1] "closure"
```

```r
typeof(1.23456789)
```

```
[1] "double"
```

```r
typeof(paste0)
```

```
[1] "closure"
```

```
typeof('$')
```

```
[1] "character"
```

```
typeof(typeof)
```

```
[1] "closure"
```

```
typeof(opts)
```

```
[1] "list"
```

2 Calling Functions

```r
log(100)
```

```
[1] 4.60517
```

```r
log(100, base = 10) # why is this different?
```

```
[1] 2
```

For the 'log' function, the second argument is optional because it has a default value.

What do you think the default value is?

```r
log10(100) # equivalent
```

```
[1] 2
```

Functions have arguments.

```
# the possible arguments and their names and defaults, if any
args("log") # the possible arguments and their
```

```
  function (x, base = exp(1))
  NULL
```

```
# names and defaults, if any
args("+") # operators are functions too
```

```
  function (e1, e2)
  NULL
```

```
args("format")
```

```
  function (x, ...)
  NULL
```

```r
args("args")
```

```
function (name)
NULL
```

```r
args("library")
```

```
function (package, help, pos = 2, lib.loc = NULL, character.only =
    logical.return = FALSE, warn.conflicts, quietly = FALSE,
    verbose = getOption("verbose"), mask.ok, exclude, include.only
    attach.required = missing(include.only))
NULL
```

- In a language like C, you need to supply every argument when you call a function. Thank goodness you don't need to do that in R.
- Arguments can be supplied by position and/or by name
- Names can be abbreviated as far as possible to avoid ambiguity

```r
log(100, b=10) # 'b' will do. Why?
```

```
[1] 2
```

An argument can be optional even if it doesn't have a default value provided that value of the argument is not used as the function is evaluated.

```r
log(100, 10) # arguments can be supplied by name or by position
```

```
[1] 2
```

3 Vectors and Variables

```r
c(1, 2, 3, 4, NA, 6)  # c: combine or catenate or concatenate
```

```
[1]  1  2  3  4 NA  6
```

```r
1:4 # integer sequence
```

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

```r
4:1 # descending
```

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

```r
-1:2 # negative to positive
```

```
[1] -1  0  1  2
```

```r
0-1:2 # why is this different? Could you explain why on a test???
```

```
[1] -1 -2
```

```r
seq(1, 4) # equivalent to 1:4
```

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

```r
seq(2, 8, by = 2) # specify interval between elements
```

```
[1] 2 4 6 8
```

```r
seq(0, 1, by = 0.1) # noninteger sequence
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```r
seq(0, 1, length = 11) # specify number of elements
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```r
rep(2, times = 10)  # 'rep' for 'repeat'
```

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

```r
rep(c('a','b'), times= 5) # recycling repeat
```

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

```r
rep(c('a','b'), length.out = 5)
```

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

```r
rep(c('a','b'), each = 5)  # not recycling, each repeated 5 times
```

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

```r
rep(c('a','b'), times = c(5,2)) # like 'each' different times
```

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

```r
rep(c('a','b'), c(5,2)) # what's the default second argument?
```

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

3.1 Operations on vectors

```r
3 * c(1, 2, 3, 4)  # scalar multiplication as in linear algebra
```

```
[1]  3  6  9 12
```

```r
c(1, 2, 3, 4) / 2
```

```
[1] 0.5 1.0 1.5 2.0
```

Dividing a vector by a vector: Is this like linear algebra?

```r
c(1, 2, 3, 4) / c(4, 3, 2, 1)
```

```
[1] 0.2500000 0.6666667 1.5000000 4.0000000
```

What happened? **element-wise operations**. Many operations are applied element by element.

Applying a function to a vector:

```r
log(c(0, 0.1, 1, 10, 100), base=10)
```

```
[1] -Inf   -1    0    1    2
```

Many functions in R are **vectorized**. Sometimes for more than one argument:

```r
log(c(2,10,100), base = c(2,10,100))
```

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

What happens if you do something that doesn't make sense, like adding a long vector to a short vector:

```r
c(1, 2, 3, 4) + c(4, 3) # no warning! What happens?
```

```
[1] 5 5 7 7
```

R usually assumes that you must know what you are doing and tries to do something reasonable. Here R applied what's known as the **recycling** principle: recycle the shorter vector as many times as necessary to provide value to match the longer vector.

However, if the length of the longer vector is not a multiple of the length of the shorter vector, R starts to worry about you and gives you a warning:

```
c(1, 2, 3, 4) + c(4, 3, 2) # R thinks maybe you made a mistake
```

```
Warning in c(1, 2, 3, 4) + c(4, 3, 2): longer object length
is not a multiple of shorter object length
```

```
[1] 5 5 5 8
```

3.2 Creating and naming objects

```r
x <- c(1, 2, 3, 4) # assignment -- does not print result
x # prints value of x
```

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

```r
(x <- c(1, 2, 3, 4)) # assigns and prints at the same time
```

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

Why not = instead of <-? (pronouced 'gets').

We used = to assign arguments to parameters when calling functions.

Actually, = works ... most of the time. But stick with <-. Your fingers will get used to it.

R uses three symbols for three distinct operations that are all represented with = in most other languages. The third, which we will see soon, is logical = which is denoted == in R.

I still regularly make the mistake of typing = when I should have typed ==.

```
x/2  # equivalent to c(1, 2, 3, 4)/2
```

```
[1] 0.5 1.0 1.5 2.0
```

```
(y <- sqrt(x))  # assign AND print
```

```
[1] 1.000000 1.414214 1.732051 2.000000
```

3.3 Random number generation

```
set.seed(372291)  # for reproducibility (chosen randomly)
(x <- rnorm(100))  # 100 standard normal random numbers
```

```
 [1]  0.09833263 -0.89632065 -0.80499585 -0.23440421
 [5] -0.57098627 -0.12431051 -0.35938865  1.19509408
 [9] -0.51327643 -1.16888069  2.04501263  0.96774659
[13] -0.33892531 -0.02385635 -0.13606928 -0.26846724
[17]  0.06233524  0.86995856  1.13066988 -0.34642095
[21]  0.35064032  1.11546736 -0.51577251  0.12023891
[25]  0.26794257  0.81975768 -0.05561499  0.78947930
[29]  0.24476899 -0.23868740  0.72776887  0.24950307
[33]  0.29120391 -0.98124553  0.04353949 -0.85734282
[37]  2.63618842 -1.25254955  0.83030163 -1.95090213
[41] -2.68712959 -0.36168710 -0.58443713  0.81399991
[45] -0.02585634 -1.11835697  1.11836461 -1.29533607
[49]  0.19998422  0.51477512  0.53101946 -0.03766582
[53]  0.98327288 -1.57294475  1.51531106  1.04962419
[57] -0.88333128  2.13595738  1.44940962 -0.84459977
[61] -1.80824601 -0.09673322 -1.00007308  1.63933460
[65] -0.52798368 -0.24537308 -0.31443488  1.15314622
[69] -0.91751726  1.26672906 -1.59600134  0.16465603
[73]  0.85672451  0.82863616 -0.42284364  0.79206310
```

```
 [77] -0.02569929 -0.19991971  1.51607308  0.50764612
 [81]  1.28908616  1.03368892 -0.55391013 -1.29783452
 [85]  1.56289281  0.51518221  0.69617559 -0.86324452
 [89] -0.43990620  1.11514478 -1.40240347  0.10543920
 [93]  0.12556730  1.88108615  1.45922041 -0.33864564
 [97]  0.61101545 -0.23174691 -0.03237697  1.44987907
```

Many distributions are in base R: e.g. 'norm', 'exp', 'poisson', student', 't', 'cauchy', 'f', etc.

The 'extraDistr' package and others have many more.

To generate random numbers from a distribution, prepend the name with and **r**:

```
rf(10, df1 = 2, df2 = 10)
```

```
 [1] 1.31169785 0.07542327 0.44086623 4.30693205 9.62530417
 [6] 1.31581038 0.09651469 0.05892881 0.63146620 0.29580129
```

To get the density for a continuous distribution) or the probability for a discrete distribution, prepend the name with a **d** for density. Note that the probability in the case of a discrete distribution is indeed a density with respect to counting measure.

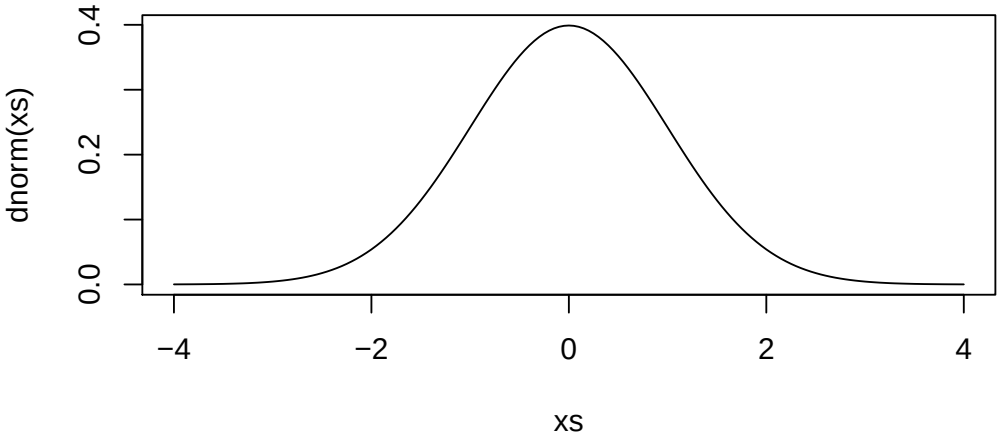
```
dnorm(0)
```

```
[1] 0.3989423
```

```
dnorm(seq(-3,3,1))
```

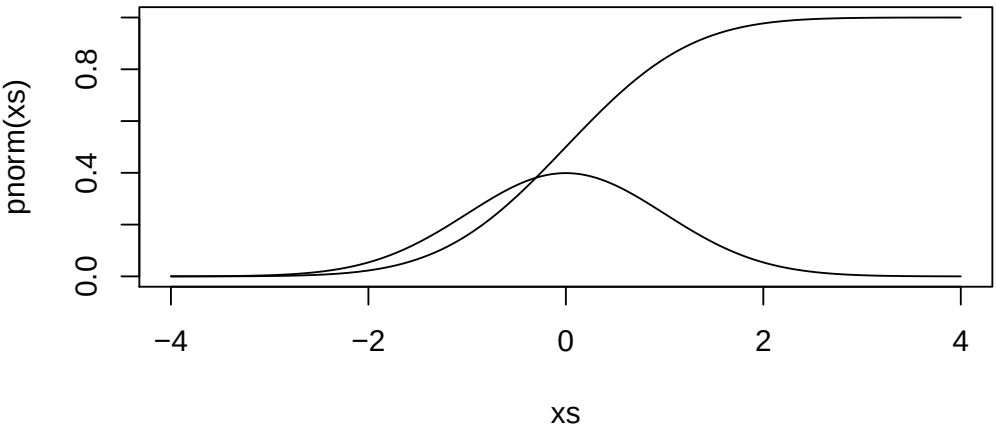
```
[1] 0.004431848 0.053990967 0.241970725 0.398942280
[5] 0.241970725 0.053990967 0.004431848
```

```
xs <- seq(-4, 4, by = .05)
plot(xs, dnorm(xs), type = 'l')
```



Prepend with a **p** for the cumulative distribution function

```
plot(xs, pnorm(xs), type = 'l')
lines(xs, dnorm(xs), type = 'l')
```



To get a quantile from a probability, the inverse CDF, prepend the distribution name with a **q**

Note the ubiquitous summary function:

```
summary(x)
```

```
      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
  -2.68713 -0.51390  0.05294  0.12372  0.83691  2.63619
```

We will see much more about it when we take up OOP.

3.4 Character objects

```
(words <- c("To", "be", "or", "not", "to", "be"))
```

```
  [1] "To"  "be"  "or"  "not" "to"  "be"
```

```r
typeof(words)
```

```
[1] "character"
```

```r
paste(words, collapse=" ")
```

```
[1] "To be or not to be"
```

```r
paste(words)
```

```
[1] "To"  "be"  "or"  "not" "to"  "be"
```

```r
paste(words, words)
```

```
[1] "To To"   "be be"   "or or"   "not not" "to to"
[6] "be be"
```

```r
paste('Variable', 1:10)
```

```
 [1] "Variable 1"  "Variable 2"  "Variable 3"  "Variable 4"
 [5] "Variable 5"  "Variable 6"  "Variable 7"  "Variable 8"
 [9] "Variable 9"  "Variable 10"
```

```r
paste('Variable', 1:10, sep = '_')
```

```
 [1] "Variable_1"  "Variable_2"  "Variable_3"  "Variable_4"
 [5] "Variable_5"  "Variable_6"  "Variable_7"  "Variable_8"
 [9] "Variable_9"  "Variable_10"
```

```r
paste('Var', 1:10, sep = '')
```

```
 [1] "Var1"  "Var2"  "Var3"  "Var4"  "Var5"  "Var6"  "Var7"
 [8] "Var8"  "Var9"  "Var10"
```

```r
paste0('Var', 1:10)   # same as sep = ''
```

```
 [1] "Var1"  "Var2"  "Var3"  "Var4"  "Var5"  "Var6"  "Var7"
 [8] "Var8"  "Var9"  "Var10"
```

Take a column of $ amounts and format accordingly

```r
(amts <- c(123.45,123456.78,12345678.912))
```

```
[1]       123.45   123456.78 12345678.91
```

```r
paste0('$', format(amts, big.mark = ',', nsmall = 2))
```

```
[1] "$        123.45" "$   123,456.78" "$12,345,678.91"
```

OOPS: Look up help on format with '?format' and discover the 'trim' argument.

```
paste0('$', format(amts, big.mark = ',', nsmall = 2, trim = TRUE))
```

```
  [1] "$123.45"        "$123,456.78"    "$12,345,678.91"
```

That's better!

3.5 Logical values

```
(logical.values <- c(TRUE, TRUE, FALSE, TRUE))
```

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

```
typeof(logical.values)
```

```
  [1] "logical"
```

Actually R uses '3-valued' logic

```
(logical.values <- c(TRUE, TRUE, FALSE, NA, TRUE))
```

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

```
!logical.values # unary not
```

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

```
logical.values | FALSE  # binary or -- note that FALSE gets recycle
```

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

```
logical.values & FALSE  # binary and
```

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

```
logical.values | TRUE  # binary or -- note that FALSE gets recycled
```

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

```
logical.values & TRUE  # binary and
```

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

```
logical.values == FALSE  # equals
```

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

```
logical.values != FALSE  # not equals
```

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

```
logical.values == TRUE  # equals
```

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

```
logical.values != TRUE  # not equals
```

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

Did the ‘NA" always result in ’NA’ in the corresponding position? If not, why not? Could you explain why on a test?

3.6 Truth tables

```
x <- c(TRUE, FALSE, NA)
x
```

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

```
names(x)   # the names attribute of x is NULL
```

```
NULL
```

```
names(x) <- as.character(x) # use 'names' replacement function
# to give x names
x
```

```
   TRUE FALSE  <NA>
   TRUE FALSE    NA
```

Truth tables

```r
outer(x, x, '|')   # note the value of "TRUE | NA"
```

```
       TRUE FALSE <NA>
  TRUE  TRUE  TRUE TRUE
  FALSE TRUE FALSE   NA
  <NA>  TRUE    NA   NA
```

```r
outer(x, x, '&')
```

```
        TRUE FALSE  <NA>
  TRUE   TRUE FALSE    NA
  FALSE FALSE FALSE FALSE
  <NA>     NA FALSE    NA
```

```r
outer(x, x, '==')
```

```
        TRUE FALSE <NA>
```

```
  TRUE   TRUE FALSE   NA
  FALSE FALSE  TRUE   NA
  <NA>     NA    NA   NA
```

```
outer(x, x, '!=')
```

```
        TRUE FALSE <NA>
  TRUE  FALSE  TRUE   NA
  FALSE  TRUE FALSE   NA
  <NA>     NA    NA   NA
```

Example of logical expressions

```
1 == 2
```

```
  [1] FALSE
```

```r
1 != 2
```

```
[1] TRUE
```

```r
1 <= 2
```

```
[1] TRUE
```

```r
1 < 1:3   # recycling
```

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

```r
3:1 > 1:3
```

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

```r
3:1 >= 1:3
```

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

```r
TRUE & c(TRUE, FALSE)      # logical AND
```

```
[1]  TRUE FALSE
```

```r
c(TRUE, FALSE, FALSE) | c(TRUE, TRUE, FALSE)  # logical OR
```

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

Logical vectors used in 'ifelse' statement

```
(z <- x[1:10]) # first 10 elements of x
```

```
   TRUE FALSE  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
   TRUE FALSE    NA    NA    NA    NA    NA    NA    NA    NA
```

```
z < -0.5 # is each element less than -0.5?
```

```
   TRUE FALSE  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
  FALSE FALSE    NA    NA    NA    NA    NA    NA    NA    NA
```

```
z > 0.5  # is each element greater than 0.5
```

```
   TRUE FALSE  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
   TRUE FALSE    NA    NA    NA    NA    NA    NA    NA    NA
```

```r
z < -0.5 | z > 0.5  #  < and > are of higher precedence than |
```

```
   TRUE FALSE  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
   TRUE FALSE    NA    NA    NA    NA    NA    NA    NA    NA
```

```r
abs(z) > 0.5  # absolute value, equivalent to last expression
```

```
   TRUE FALSE  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
   TRUE FALSE    NA    NA    NA    NA    NA    NA    NA    NA
```

```r
z[abs(z) > 0.5] # values of z for which |z| > 0.5
```

```
  TRUE <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
  TRUE   NA   NA   NA   NA   NA   NA   NA   NA
```

```
z[!(abs(z) > 0.5)] # values z for which |z| <= 0.5
```

```
  FALSE  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
  FALSE    NA    NA    NA    NA    NA    NA    NA    NA
```

3.7 Special logical operators && and ||

- Work only on single expressions, not vectors
- Only evaluate what they need to determine the result

```
TRUE && FALSE   # works only on single expressions, not vectors
```

```
[1] FALSE
```

```
TRUE || FALSE   # only evaluates what it needs to determine result
```

```
[1] TRUE
```

QUESTION: Explain what happens here

```
log(-1)
```

```
Warning in log(-1): NaNs produced
```

```
[1] NaN
```

```
TRUE || log(-1)
```

```
[1] TRUE
```

```r
FALSE || log(-1)
```

```
Warning in log(-1): NaNs produced
```

```
[1] NA
```

3.8 How coercion works in R

What happens when you try to do something that isn't quite right with an object?

```r
sum(logical.values) # logical gets coerced to numeric
```

```
[1] NA
```

TRUE becomes 1 and FALSE becomes 0

```r
sum(!logical.values)
```

```
[1] NA
```

What happens when you catenate things that are of different types? Vectors can only contain elements of the same type.
We'll soon see how to use **list** that can contain elements of different types.

```r
c("A", FALSE, 3.0)
```

```
[1] "A"     "FALSE" "3"
```

```r
c(10, FALSE, -6.5, TRUE)
```

```
[1] 10.0  0.0 -6.5  1.0
```

3.9 The hierarchy of atomic types: promotion and demotion

- logical (lowest)
- numeric:
  - integer
  - double
  - complex
- character (highest)

All the elements of a vector must be of the same type so if you try to mix types in a vector, the 'lower' types get **coerced** (i.e. **promoted**) to the 'higher' types

```
c(TRUE, 1, 'one') # all get coerced to character
```

```
[1] "TRUE" "1"    "one"
```

```r
c(TRUE,1i)        # all get coerced to complex
```

```
[1] 1+0i 0+1i
```

```r
c(TRUE,1i, 'one') # what should happen here?
```

```
[1] "TRUE" "0+1i" "one"
```

QUESTION: Try to guess the difference between the result evaluated from the following 2 lines. Note that the innermost expressions need to be evaluated first.

```r
c(c(TRUE, 2), 'dog')
c(TRUE, c(2, 'dog'))
```

3.10 Explicit coercion

```
as.logical(2)
```

```
[1] TRUE
```

```
as.character(2)
```

```
[1] "2"
```

```
as.integer(2.5)
```

```
[1] 2
```

```
as.integer(2.9)   # truncation, not rounding
```

```
[1] 2
```

```
round(2.5)   # What's happening?
```

```
[1] 2
```

```
round(3.5)
```

```
[1] 4
```

```
typeof(round(2.5))
```

```
[1] "double"
```

```r
as.complex(2)
```

```
[1] 2+0i
```

```r
as.numeric('2')
```

```
[1] 2
```

```r
as.numeric('two')
```

```
Warning: NAs introduced by coercion
```

```
[1] NA
```

3.11 Implicit coercion

```
4 + TRUE
```

```
[1] 5
```

'TRUE' got **promoted** to numeric

```
4 & FALSE
```

```
[1] FALSE
```

4 got **demoted** to logical

```
1 == TRUE # does 1 get demoted, or TRUE get promoted?
```

```
[1] TRUE
```

4 Selecting elements of a vector

4.1 Selecting by position with positive numbers

```
x[12]              # 12th element
```

```
  <NA>
    NA
```

```
words[2]           # second element
```

```
  [1] "be"
```

```
logical.values[3] # third element
```

```
  [1] FALSE
```

```r
x[6:15]       # elements 6 through 15
```

```
 <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
```

```r
x[c(1, 3, 5)] # 1st, 3rd, 5th elements
```

```
 TRUE <NA> <NA>
 TRUE   NA   NA
```

QUESTION: What happens if you go too far?

```r
words[10]
```

```
 [1] NA
```

```
x[100000]
```

```
<NA>
  NA
```

```
logical.values[9]
```

```
[1] NA
```

Although all these NA's look the same, they are subtly different

4.2 Selecting by omission of positions with negative numbers

```r
x[-(11:100)] # omit elements 11 through 100
```

```
   TRUE FALSE  <NA>
   TRUE FALSE    NA
```

```r
letters  # this object comes with R
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```r
letters[-c(1,5,9,15,21)] # what have we done?
```

```
 [1] "b" "c" "d" "f" "g" "h" "j" "k" "l" "m" "n" "p" "q" "r"
[15] "s" "t" "v" "w" "x" "y" "z"
```

4.3 Selecting with a logical vector

```
v <- 1:4
v[c(TRUE, FALSE, FALSE, TRUE)]
```

```
[1] 1 4
```

```
vowels <- c('a','e','i','o','u')
letters %in% vowels  # a very useful operator
```

```
 [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
[10] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[19] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
letters[!(letters %in% vowels)]
```

```
 [1] "b" "c" "d" "f" "g" "h" "j" "k" "l" "m" "n" "p" "q" "r"
[15] "s" "t" "v" "w" "x" "y" "z"
```

```
`[`(v,c(T,F,T,F))   # Everything that happens is a function!
```

```
[1] 1 3
```

```
`[`(v,c(T,F))   # What's happening here??
```

```
[1] 1 3
```

```
v[c(T,F)]
```

```
[1] 1 3
```

4.4 Selecting with names

Vectors can get named:

```
(age <- c(10, 9, 15, 16))
```

```
[1] 10  9 15 16
```

```
names(age) <- c('Paula','Imran','Angela','Jiwon') # replacement fn
age
```

```
  Paula  Imran Angela  Jiwon
     10      9     15     16
```

Selecting by name

```r
age['Angela']
```

```
  Angela
      15
```

```r
age['George']
```

```
  <NA>
    NA
```

```r
age[c('Jiwon','Paula')]
```

```
  Jiwon Paula
     16    10
```

```r
age[c(NA,'Paula')]
```

```
   <NA> Paula
     NA    10
```

```r
names(age)
```

```
  [1] "Paula"  "Imran"  "Angela" "Jiwon"
```

```r
names(age) %in% c('Bob', 'Paula', 'Imran')
```

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

```r
age[ names(age) %in% c('Bob', 'Paula', 'Imran') ]
```

```
  Paula Imran
     10     9
```

```r
age[ !names(age) %in% c('Bob', 'Paula', 'Imran') ]
```

```
  Angela   Jiwon
      15      16
```

Example of a **regular expression**

'^A' matches a capital A at the beginning of a string

'grepl' stands for global regular expression print logical

```r
grepl('^A', names(age))
```

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

```r
age[ grepl('^A', names(age)) ]
```

```
  Angela
      15
```

```r
sort(names(age))
```

```
[1] "Angela" "Imran"  "Jiwon"  "Paula"
```

```r
age[ sort(names(age)) ]
```

```
Angela  Imran  Jiwon  Paula
    15      9     16     10
```

4.5 Selecting elements of a matrix

A **matrix** is like a vector but with two dimensions. If the dimension is higher, it's called an **array**.

```r
mat <- matrix(1:12, nrow = 3, ncol = 4)
mat # notice that it got filled column by column
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

A matrix can have rownames and columns names

```r
rownames(mat) <- c('Algebra','Analysis','Geometry')
colnames(mat) <- c('2013','2014','2015','2016')
mat
```

```
          2013 2014 2015 2016
Algebra      1    4    7   10
Analysis     2    5    8   11
Geometry     3    6    9   12
```

```r
dim(mat)
```

```
[1] 3 4
```

```r
nrow(mat)
```

```
[1] 3
```

```r
ncol(mat)
```

```
[1] 4
```

```r
sum(mat)
```

```
[1] 78
```

Selecting elements is much like vector except that we have two dimensions. Look at the following carefully to see what's happening. Notice what happens if a dimension is blank.

```
mat[2,4]
```

```
[1] 11
```

```
mat[2,5]  # different from vectors
```

```
Error in mat[2, 5]: subscript out of bounds
```

```
mat[c(2,3),]  # blank means: take everything
```

```
          2013 2014 2015 2016
Analysis     2    5    8   11
Geometry     3    6    9   12
```

```
mat[c(2,3), 1:4]  # blank means: take everything
```

```
           2013 2014 2015 2016
  Analysis    2    5    8   11
  Geometry    3    6    9   12
```

```
mat[c(2,3),3] # Oops! a dimension got dropped
```

```
  Analysis Geometry
         8        9
```

Dropping a dimension might seem like no big deal but it's one of the things that early designers regret adopting as a default. If your matrix is buried in a function and it drops a dimension that might spell hidden trouble when the function tries to multiply it by another matrix.

```
mat[c(2,3), 3, drop = FALSE]
```

```
           2015
  Analysis    8
  Geometry    9
```

Using 'drop' is the safe way within functions where you don't know whether the selecting vector might have length 1 or 0 when the function is called.

```
mat[c('Algebra','Geometry'),c(2014,2015)] # Need characters
```

```
  Error in mat[c("Algebra", "Geometry"), c(2014, 2015)]: subscript o
```

```
mat[c('Algebra','Geometry'),c('2014','2015')] # Good
```

```
           2014 2015
  Algebra     4    7
  Geometry    6    9
```

```r
mat[c('Algebra','Geometry'),c(NA,'2014','2015')] # NA tilts
```

```
  Error in mat[c("Algebra", "Geometry"), c(NA, "2014", "2015")]: sub
```

```r
mat[c('Algebra','Geometry'),-3]
```

```
           2013 2014 2016
  Algebra     1    4   10
  Geometry    3    6   12
```

```r
mat[c('Algebra','Geometry'),-(1:4)] # a columnless matrix
```

```
  Algebra
  Geometry
```

```
mat[-(1:3),]
```

```
       2013 2014 2015 2016
```

```
mat[-(1:3),-(1:4)]
```

```
  <0 x 0 matrix>
```

5 Operators are functions

Chambers' dictum (main early creator of S at Bell Labs)

- **Everything that exists is an object**
- **Everything that happens is a function call**

```
1 + 2 # `+` is a binary operator
```

```
[1] 3
```

```
`+`(1,2)  # but really a function with two arguments
```

```
[1] 3
```

Note: to call an object with a weird name, just put its names in backticks.

6 Valid object names in R

What names are valid?

You can use:

- letters, case-sensitive
- numbers
- underline
- period
- must start with a letter or period but not a number or underline
- the first non-period character must not be a number

QUESTION: Which of the following assignments use valid names?

7 Writing your own functions

It's easy in R to write your own functions and to build up a toolbox over time. You can document your functions in a package and share it with other users.

```
mean(x)
```

```
[1] NA
```

```
sum(x)/length(x)
```

```
[1] NA
```

Here's a very simple function that duplicates an existing one:

```
myMean <- function(x){
  sum(x)/length(x)
}
myMean(x)
```

```
[1] NA
```

```
y # defined earlier as sqrt(c(1, 2, 3, 4))
```

```
[1] 1.000000 1.414214 1.732051 2.000000
```

```r
myMean(y)
```

```
[1] 1.536566
```

```r
myMean(1:100)
```

```
[1] 50.5
```

```r
myMean(sqrt(1:100))
```

```
[1] 6.714629
```

```r
mySD <- function(x){
  sqrt(sum((x - myMean(x))^2)/(length(x) - 1))
}
mySD(1:100)
```

```
[1] 29.01149
```

```
sd(1:100) # check
```

```
[1] 29.01149
```

```
typeof(mySD)
```

```
[1] "closure"
```

Functions are 'first-class' objects in R

```
mySD
```

```
function(x){
  sqrt(sum((x - myMean(x))^2)/(length(x) - 1))
}
```

```
myMean
```

```
  function(x){
    sum(x)/length(x)
  }
  <bytecode: 0x000000001243e8a0>
```

```
letters
```

```
   [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
  [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
mySD(letters)
```

```
  Error in sum(x): invalid 'type' (character) of argument
```

8 Quick overview of basic regression and plots in R

'Duncan' is a data frame (the term used in R to refer to the kind of object used to contain a data set) in the 'car' package. Since we loaded the 'car' package (with 'library(car)') we can use 'Duncan' typing its name.

```
head(Duncan, n=10) # U.S. data set from the 50s, first 10 lines
```

```
            type income education prestige
accountant  prof     62        86       82
pilot       prof     72        76       83
architect   prof     75        92       90
author      prof     55        90       76
chemist     prof     64        86       90
minister    prof     21        84       87
professor   prof     64        93       93
dentist     prof     80       100       90
```

```
  reporter     wc      67         87        52
  engineer   prof      72         86        88
```

```
dim(Duncan)
```

```
  [1] 45  4
```

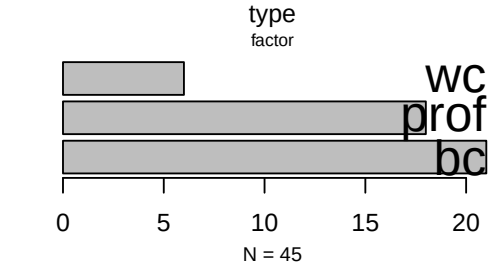
```
summary(Duncan)
```

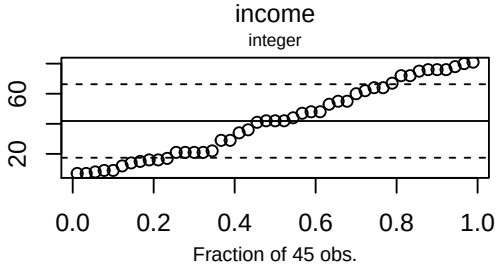
```
     type         income          education         prestige
   bc  :21   Min.   : 7.00   Min.   :  7.00   Min.   : 3.00
   prof:18   1st Qu.:21.00   1st Qu.: 26.00   1st Qu.:16.00
   wc  : 6   Median :42.00   Median : 45.00   Median :41.00
             Mean   :41.87   Mean   : 52.56   Mean   :47.69
             3rd Qu.:64.00   3rd Qu.: 84.00   3rd Qu.:81.00
             Max.   :81.00   Max.   :100.00   Max.   :97.00
```

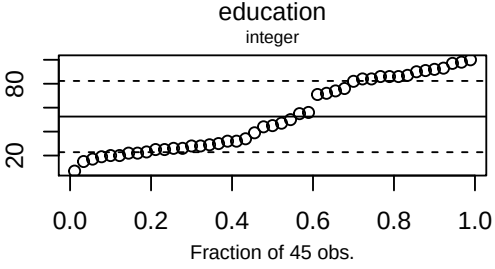
```r
library(spida2)  # install with devtools::install_github('gmonette/
```

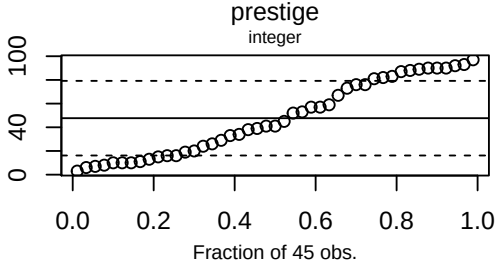
```
  spida2: development branch 0.2.0.9000.
```

```r
xqplot(Duncan)   # uniform quantile plots
```

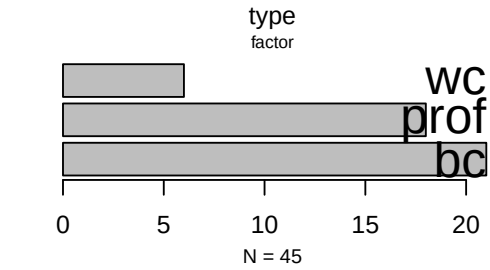

type
factor
wc
prof
bc
0
5
10
15
20
N = 45

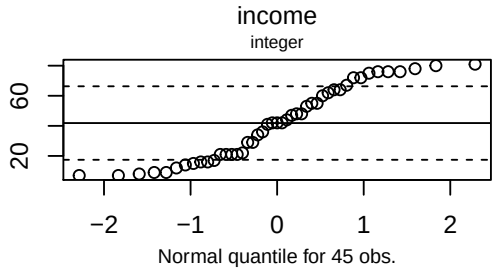

income
integer
20
60
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of 45 obs.

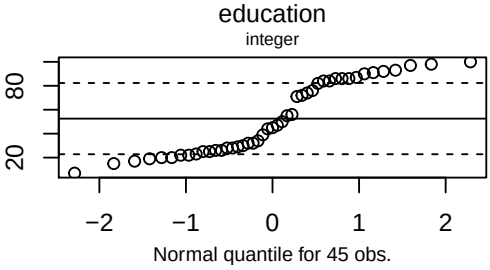

education
integer
20
80
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of 45 obs.

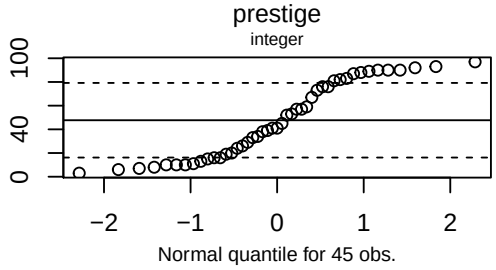

prestige
integer
0
40
100
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of 45 obs.

```
xqplot(Duncan, ptype = 'n')    # normal quantile plots
```







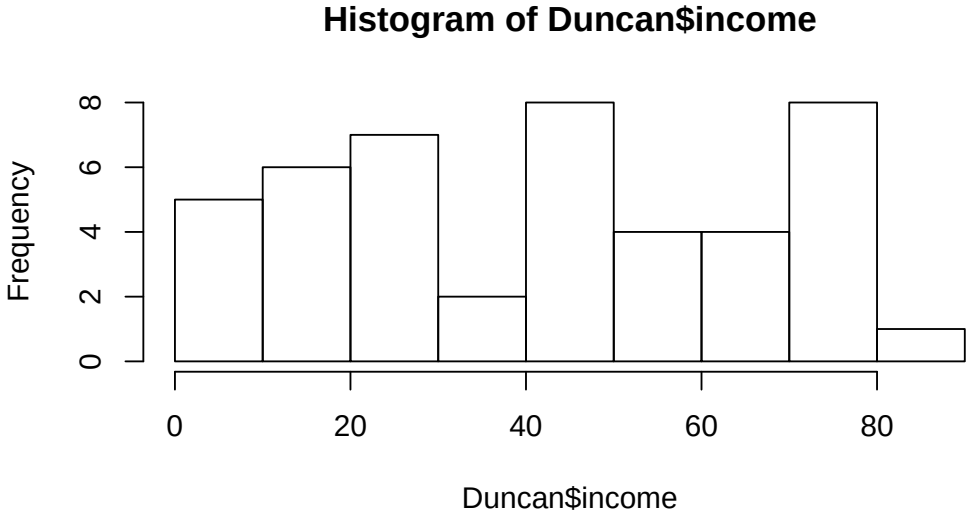


8.1 Referencing variables in data frames

We will see this again later

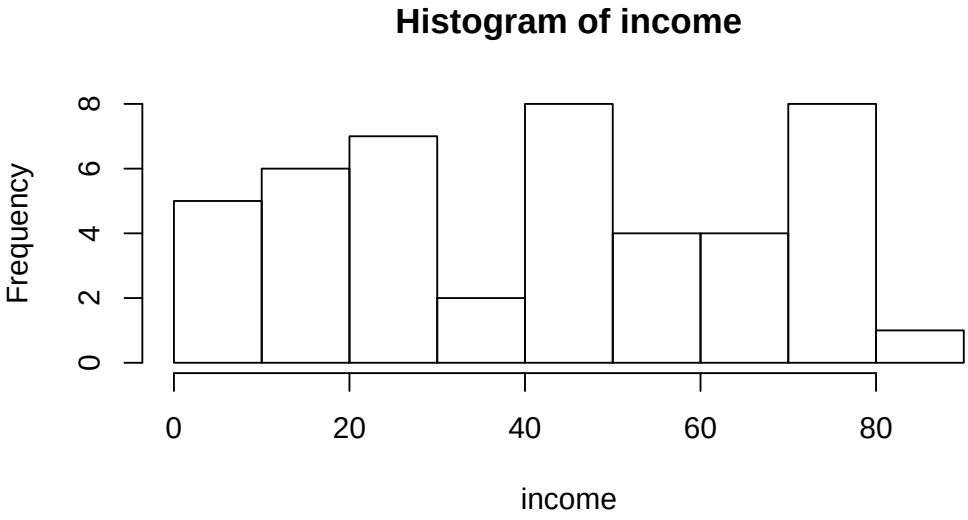
Fully qualified name

```
hist(Duncan$income)
```



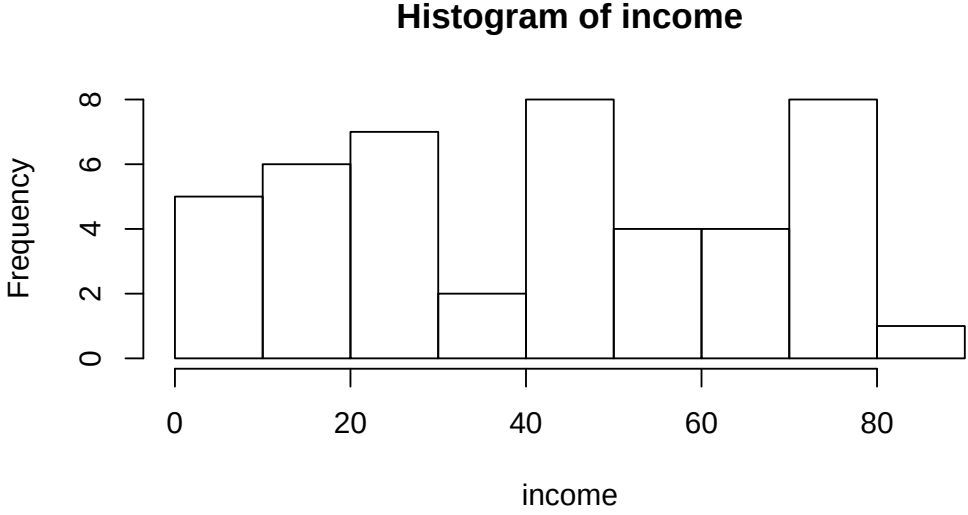
using the 'with' function: the second argument is evaluated in the data frame

```
with(Duncan, hist(income))
```



using ‘attach’ – ___ very highly deprecated___ e.g. this blog post

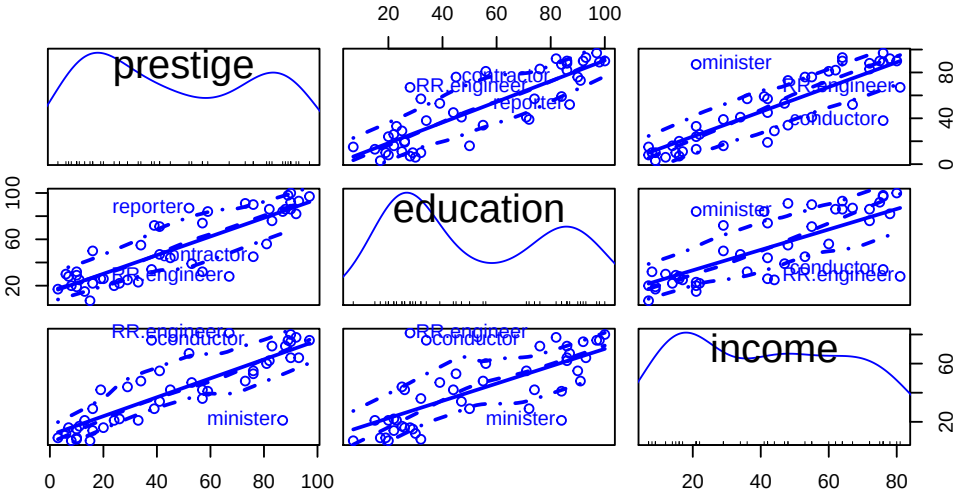
```
attach(Duncan)
hist(income)
```


Histogram of income
Frequency
0
2
4
6
8
0
20
40
60
80
income

```
detach(Duncan)
```

Many 'recent' (past 25 years) modelling functions and graphic functions in R use **formulas**. When a model or a graph is defined by a formula, the data frame in which the terms of the formula are to be found is an argument of the function.

```
scatterplotMatrix( ~ prestige + education + income,
                   data = Duncan, id = list(n=3))
```



Fitting a least-squares regression model

```r
(Duncan.model <- lm(prestige ~ education + income,
                    data = Duncan))
```

```
Call:
lm(formula = prestige ~ education + income, data = Duncan)

Coefficients:
(Intercept)    education       income
    -6.0647       0.5458       0.5987
```

Estimated coefficients and other results of the regression:

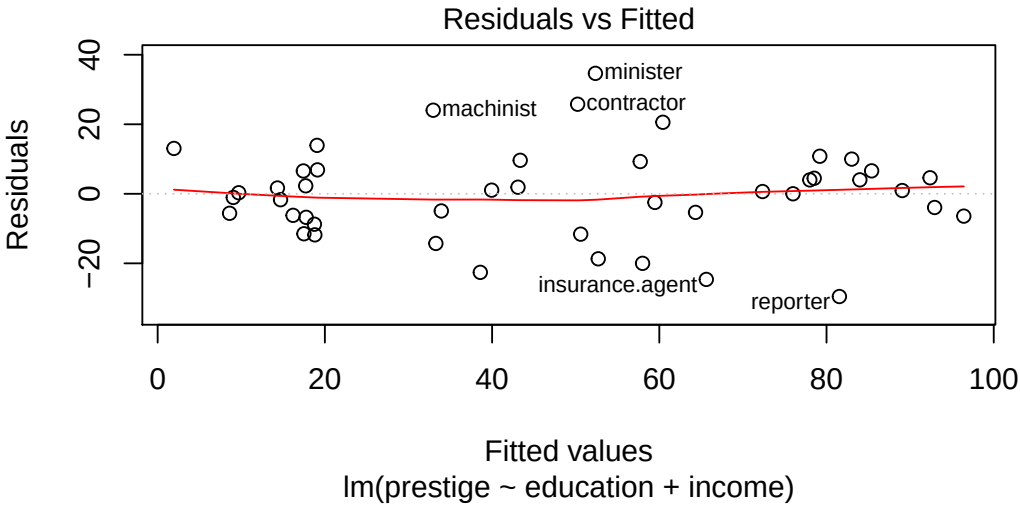
```r
summary(Duncan.model)
```

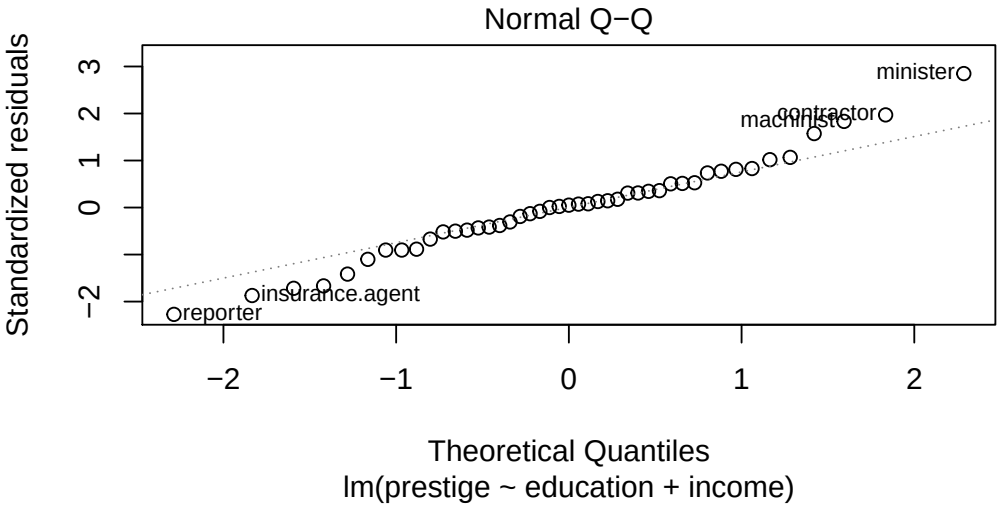
```
Call:
```

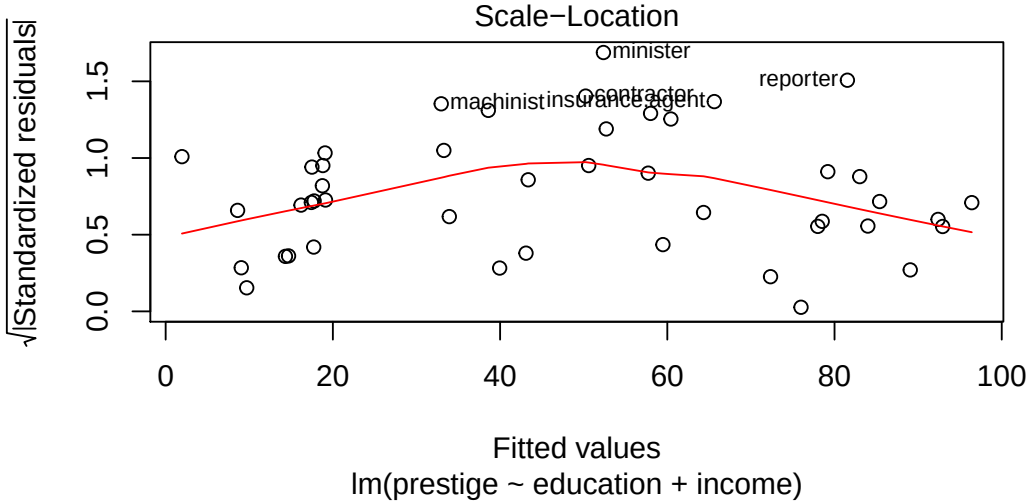
```
lm(formula = prestige ~ education + income, data = Duncan)

Residuals:
    Min      1Q  Median      3Q     Max
-29.538  -6.417   0.655   6.605  34.641

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.06466    4.27194  -1.420    0.163
education    0.54583    0.09825   5.555 1.73e-06 ***
income       0.59873    0.11967   5.003 1.05e-05 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.37 on 42 degrees of freedom
Multiple R-squared:  0.8282,  Adjusted R-squared:   0.82
F-statistic: 101.2 on 2 and 42 DF,  p-value: < 2.2e-16
```

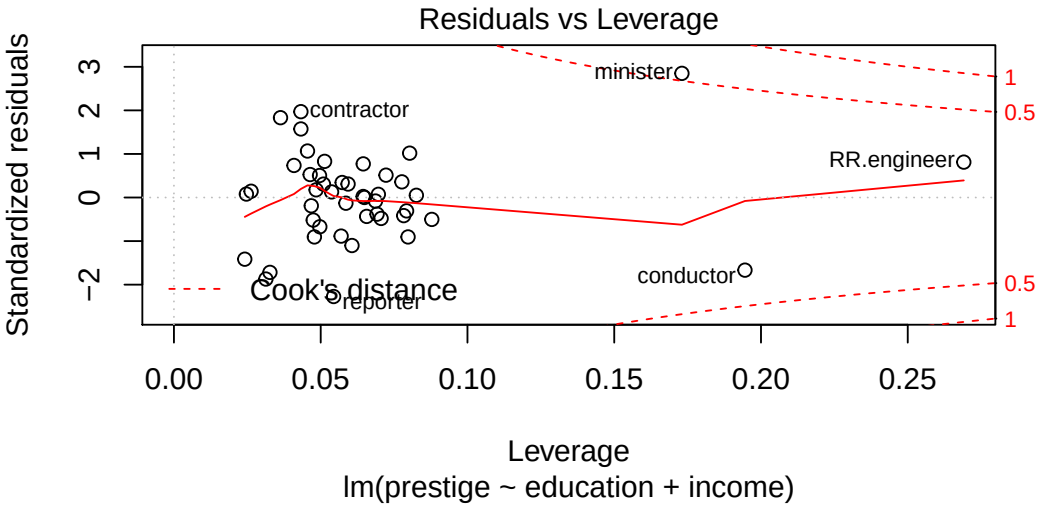
Four important plots to check the regression:

```
plot(Duncan.model, id.n = 5, ask = FALSE)
```


Residuals vs Fitted
Residuals
Fitted values
lm(prestige ~ education + income)
minister
contractor
machinist
insurance.agent
reporter
40
20
0
−20
0
20
40
60
80
100

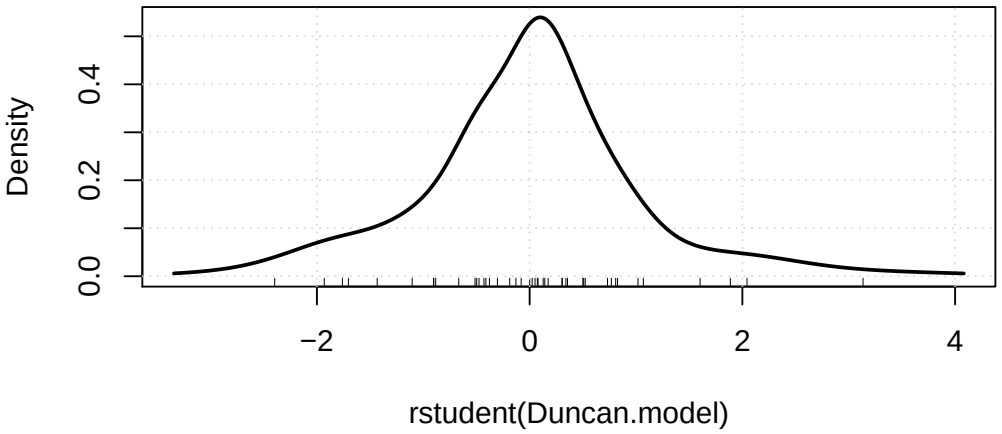

Normal Q−Q
Standardized residuals
3
2
1
0
−2
minister
contractor
machinist
insurance.agent
reporter
−2
−1
0
1
2
Theoretical Quantiles
lm(prestige ~ education + income)


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
minister
reporter
machinist
insurance.agent
contractor
0
20
40
60
80
100
Fitted values
lm(prestige ~ education + income)



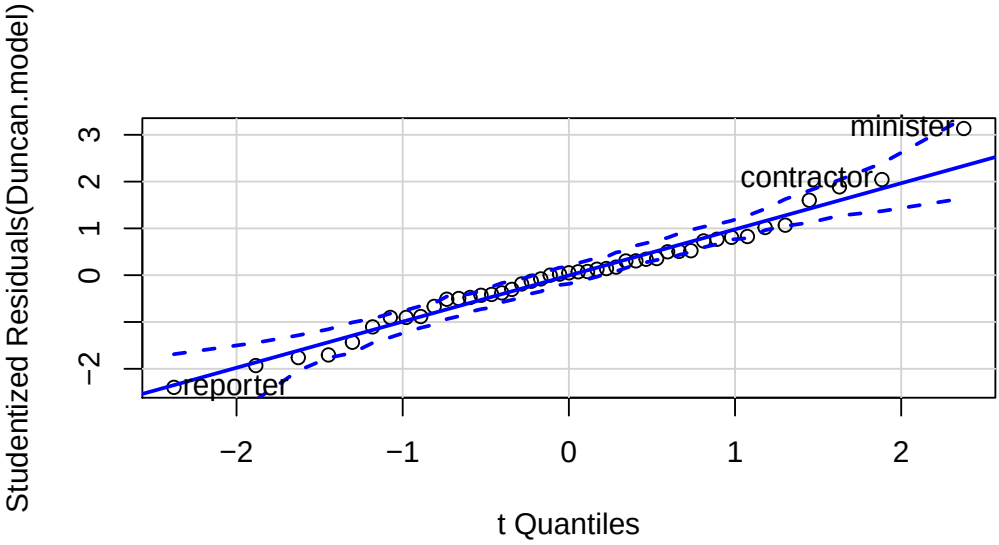
The density of the studentized residuals:

```
densityPlot(rstudent(Duncan.model))
```



Quantile plot of residuals:

```
qqPlot(Duncan.model, id = list(n = 3))
```



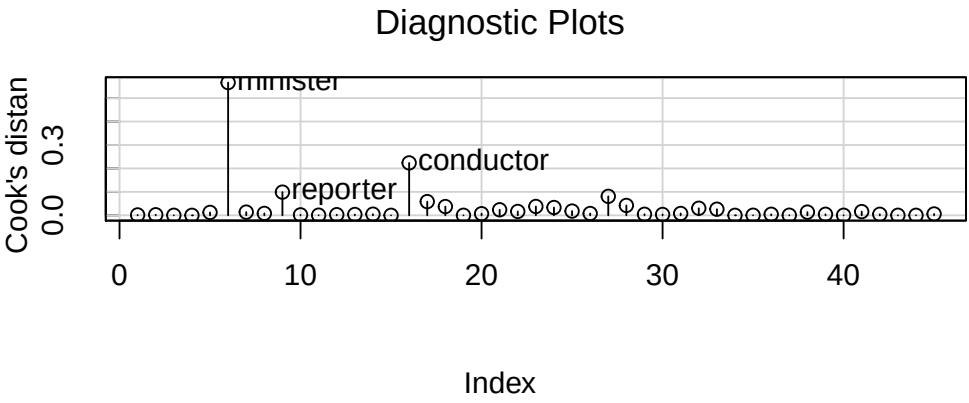

```
minister   reporter contractor
       6          9         17
```

Outliers:

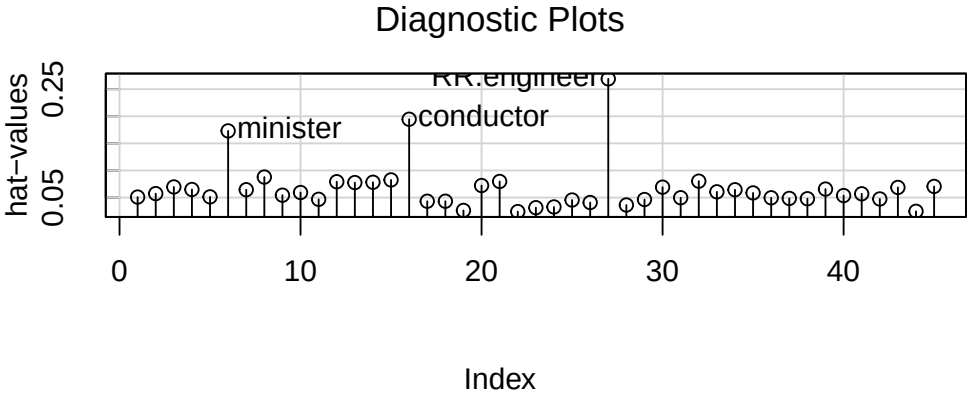
```
outlierTest(Duncan.model)
```

```
  No Studentized residuals with Bonferroni p < 0.05
  Largest |rstudent|:
            rstudent unadjusted p-value Bonferroni p
  minister 3.134519          0.0031772      0.14297
```

```
influenceIndexPlot(Duncan.model, vars = "Cook",
                   id = list(n=3))
```


Diagnostic Plots
Cook's distan
0.0
0.3
minister
conductor
reporter
0
10
20
30
40
Index

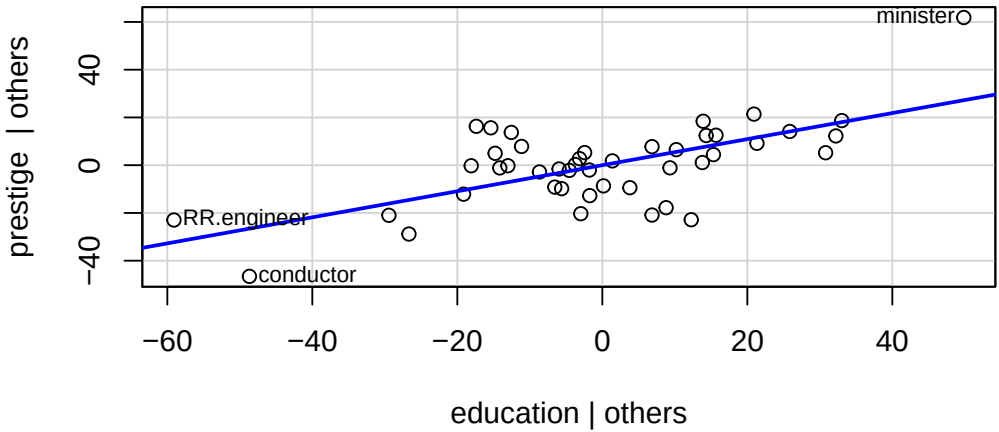
```r
influenceIndexPlot(Duncan.model, vars = "hat",
                   id = list(n=3))
```



8.2 Added-variable plots

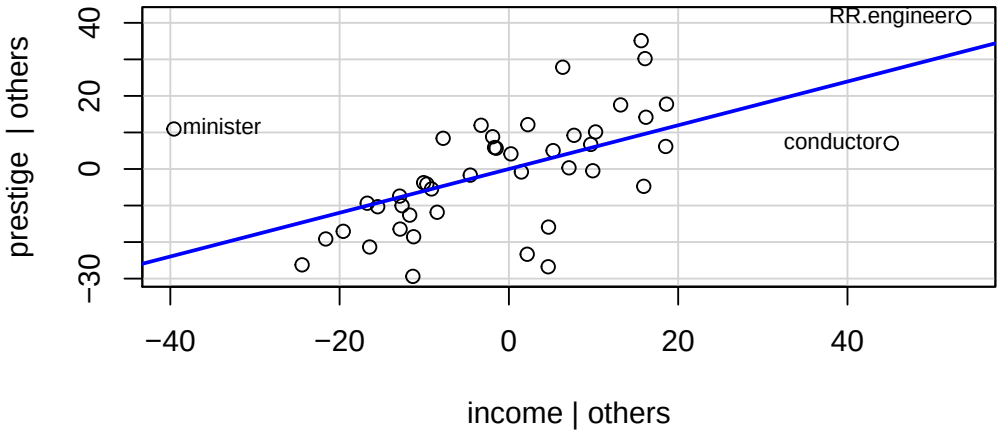
Also know in PROC REG in SAS as 'partial regression leverage plots' not to be confused with partial residual plots (better described as 'component plus residual plots') although the are often called that.

```
avPlots(Duncan.model, ~ education,
        id=list(cex=0.75, n=3, method="mahal"))
```



Imagine the following plot is a simple scatterplot between two variables. What do you see?

```r
avPlots(Duncan.model, ~ income,
        id=list(cex=0.75, n=3, method="mahal"))
```

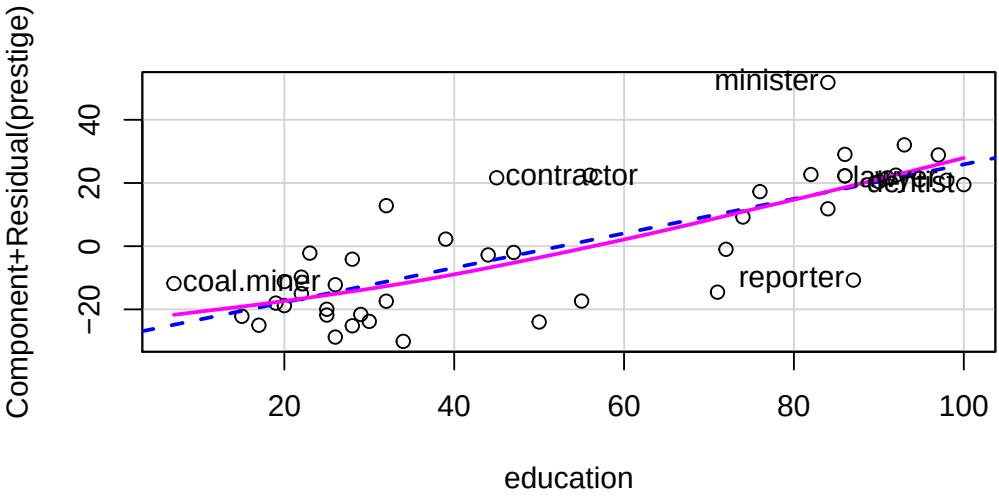


In multiple regression, the AVP is the closest you come to seeing what drives the estimation of each regression coefficient.

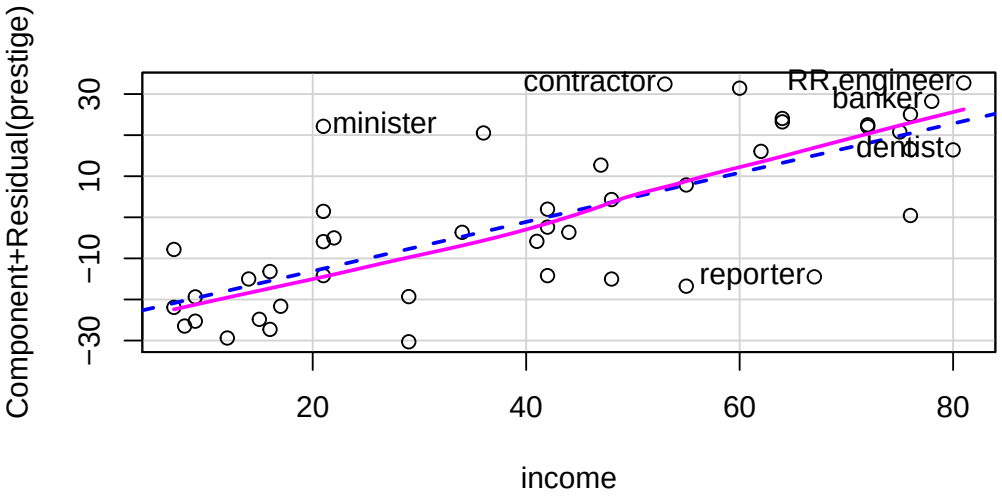
8.3 Component-plus-residual plots

Also known as partial residual plots. These tend to be more widely used but often less informative than AVPs

```
crPlots(Duncan.model, ~ education, id = list(n=3))
```


Component+Residual(prestige)
education
minister
contractor
coal.miner
reporter
dentist
−20
0
20
40
20
40
60
80
100

```r
crPlots(Duncan.model, ~ income, id = list(n=3))
```



Non-constant variance test

```
ncvTest(Duncan.model)
```

```
  Non-constant Variance Score Test
  Variance formula: ~ fitted.values
  Chisquare = 0.3810967, Df = 1, p = 0.53702
```

```
ncvTest(Duncan.model, var.formula = ~ income + education)
```

```
  Non-constant Variance Score Test
  Variance formula: ~ income + education
  Chisquare = 0.6976023, Df = 2, p = 0.70553
```

Position of row names

```r
whichNames(c("minister", "conductor"), Duncan)
```

```
   minister conductor
          6         16
```

```r
duncan.model.2 <- update(Duncan.model, subset = -c(6, 16))
```

An alternative approach that uses the '%in%' operator to creae a logical vector

```r
outliers <- names(Duncan) %in% c("minister", "conductor")
```

```r
duncan.model.2 <- update(Duncan.model, subset = !outliers)
```

```r
summary(duncan.model.2)
```

```
Call:
lm(formula = prestige ~ education + income, data = Duncan, subset

Residuals:
    Min      1Q  Median      3Q     Max
-29.538  -6.417   0.655   6.605  34.641

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.06466    4.27194  -1.420    0.163
education    0.54583    0.09825   5.555 1.73e-06 ***
income       0.59873    0.11967   5.003 1.05e-05 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.37 on 42 degrees of freedom
Multiple R-squared:  0.8282,  Adjusted R-squared:   0.82
F-statistic: 101.2 on 2 and 42 DF,  p-value: < 2.2e-16
```

Comparing coefficients with and without outliers:

```
compareCoefs(duncan.model, duncan.model.2)
```

```
  Error in compareCoefs(duncan.model, duncan.model.2): object 'dunca
```

Type of predictor

```
summary(Duncan$type)
```

```
    bc prof   wc
    21   18    6
```

```
summary(Duncan$prestige)
```

```
     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     3.00   16.00   41.00   47.69   81.00   97.00
```

```r
summary(Duncan)
```

```
    type         income          education         prestige    
 bc   :21   Min.   : 7.00   Min.   :  7.00   Min.   : 3.00  
 prof:18   1st Qu.:21.00   1st Qu.: 26.00   1st Qu.:16.00  
 wc   : 6   Median :42.00   Median : 45.00   Median :41.00  
           Mean   :41.87   Mean   : 52.56   Mean   :47.69  
           3rd Qu.:64.00   3rd Qu.: 84.00   3rd Qu.:81.00  
           Max.   :81.00   Max.   :100.00   Max.   :97.00  
```

Model with interaction

```r
summary(lm(prestige ~ education + income, data = Duncan))
```

```

Call:
lm(formula = prestige ~ education + income, data = Duncan)
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-29.538  -6.417   0.655   6.605  34.641

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.06466    4.27194  -1.420    0.163
education    0.54583    0.09825   5.555 1.73e-06 ***
income       0.59873    0.11967   5.003 1.05e-05 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.37 on 42 degrees of freedom
Multiple R-squared:  0.8282,  Adjusted R-squared:   0.82
F-statistic: 101.2 on 2 and 42 DF,  p-value: < 2.2e-16
```

```r
class(Duncan$type)
```

```
[1] "factor"
```

```r
class(Duncan$prestige)
```

```
[1] "integer"
```

```r
class(Duncan)
```

```
[1] "data.frame"
```

```r
duncan.model <- lm(prestige ~ education + income, data=Duncan)
class(duncan.model)
```

```
[1] "lm"
```

A **generic function**:

```
summary
```

```
function (object, ...)
UseMethod("summary")
<bytecode: 0x0000000018c01c78>
<environment: namespace:base>
```

A **method** for the 'lm' **class** for the generic function

```
args(summary.lm)
```

```
function (object, correlation = FALSE, symbolic.cor = FALSE,
    ...)
NULL
```

The real work happens in the special function that is the method for the generic function 'summary' for the 'lm' class.

```
summary.lm
```

```
function (object, correlation = FALSE, symbolic.cor = FALSE,
    ...)
{
    z <- object
    p <- z$rank
    rdf <- z$df.residual
    if (p == 0) {
        r <- z$residuals
        n <- length(r)
        w <- z$weights
        if (is.null(w)) {
            rss <- sum(r^2)
        }
```

```
    else {
        rss <- sum(w * r^2)
        r <- sqrt(w) * r
    }
    resvar <- rss/rdf
    ans <- z[c("call", "terms", if (!is.null(z$weights)) "weig
    class(ans) <- "summary.lm"
    ans$aliased <- is.na(coef(object))
    ans$residuals <- r
    ans$df <- c(0L, n, length(ans$aliased))
    ans$coefficients <- matrix(NA_real_, 0L, 4L, dimnames = li
        c("Estimate", "Std. Error", "t value", "Pr(>|t|)")))
    ans$sigma <- sqrt(resvar)
    ans$r.squared <- ans$adj.r.squared <- 0
    ans$cov.unscaled <- matrix(NA_real_, 0L, 0L)
    if (correlation)
        ans$correlation <- ans$cov.unscaled
    return(ans)
}
```

```
    if (is.null(z$terms))
        stop("invalid 'lm' object:  no 'terms' component")
    if (!inherits(object, "lm"))
        warning("calling summary.lm(<fake-lm-object>) ...")
    Qr <- qr.lm(object)
    n <- NROW(Qr$qr)
    if (is.na(z$df.residual) || n - p != z$df.residual)
        warning("residual degrees of freedom in object suggest thi
    r <- z$residuals
    f <- z$fitted.values
    w <- z$weights
    if (is.null(w)) {
        mss <- if (attr(z$terms, "intercept"))
            sum((f - mean(f))^2)
        else sum(f^2)
        rss <- sum(r^2)
    }
    else {
        mss <- if (attr(z$terms, "intercept")) {
```

```
        m <- sum(w * f/sum(w))
        sum(w * (f - m)^2)
    }
    else sum(w * f^2)
    rss <- sum(w * r^2)
    r <- sqrt(w) * r
}
resvar <- rss/rdf
if (is.finite(resvar) && resvar < (mean(f)^2 + var(c(f))) *
    1e-30)
    warning("essentially perfect fit: summary may be unreliabl
p1 <- 1L:p
R <- chol2inv(Qr$qr[p1, p1, drop = FALSE])
se <- sqrt(diag(R) * resvar)
est <- z$coefficients[Qr$pivot[p1]]
tval <- est/se
ans <- z[c("call", "terms", if (!is.null(z$weights)) "weights"
ans$residuals <- r
ans$coefficients <- cbind(Estimate = est, `Std. Error` = se,
```

```
    `t value` = tval, `Pr(>|t|)` = 2 * pt(abs(tval), rdf,
        lower.tail = FALSE))
ans$aliased <- is.na(z$coefficients)
ans$sigma <- sqrt(resvar)
ans$df <- c(p, rdf, NCOL(Qr$qr))
if (p != attr(z$terms, "intercept")) {
    df.int <- if (attr(z$terms, "intercept"))
        1L
    else 0L
    ans$r.squared <- mss/(mss + rss)
    ans$adj.r.squared <- 1 - (1 - ans$r.squared) * ((n -
        df.int)/rdf)
    ans$fstatistic <- c(value = (mss/(p - df.int))/resvar,
        numdf = p - df.int, dendf = rdf)
}
else ans$r.squared <- ans$adj.r.squared <- 0
ans$cov.unscaled <- R
dimnames(ans$cov.unscaled) <- dimnames(ans$coefficients)[c(1,
    1)]
```

```
    if (correlation) {
        ans$correlation <- (R * resvar)/outer(se, se)
        dimnames(ans$correlation) <- dimnames(ans$cov.unscaled)
        ans$symbolic.cor <- symbolic.cor
    }
    if (!is.null(z$na.action))
        ans$na.action <- z$na.action
    class(ans) <- "summary.lm"
    ans
}
<bytecode: 0x000000001262aac0>
<environment: namespace:stats>
```

This is R's basic form of Object-Oriented-Programming (OOP). It's what makes it possible for R to have grown through the work of different contributors working relatively independently.

If you create a new statistical method that produces a special kind of object, you don't have to request that 'summary' be changed to work on your object. You just give your object a class and write a summary method for it.

Here's latest discovery of a new statistical method: Use the median to 'fit' data!

```
silly <- function(x) {
  ret <- median(x)
  class(ret) <- 'silly'
  ret
}
fit <- silly(x)
fit
```

```
  [1] NA
  attr(,"class")
  [1] "silly"
```

```
class(fit)
```

```
  [1] "silly"
```

I write a **method** for the **summary** generic function for the **class** 'silly'.

```
summary.silly <- function(fit) {
  cat('This is the fitted value from the silly method: ')
  cat(fit,'\n')
  invisible(fit)
}
```

See what it does:

```
summary(fit)
```

```
  This is the fitted value from the silly method: NA
```

When I called the 'summary' function, it checked the class of its first argument, 'fit'.

Finding that the class was 'silly', summary looks for a function called 'summary.silly' and uses that function. This is called **dispatching**. If 'summary.silly' hadn't existed summary would have used 'summary.default'.

For example, when you use 'glm'

```
mod.mroz <- glm(lfp ~ ., family=binomial, data=Mroz)
class(mod.mroz)
```

```
[1] "glm" "lm"
```

```
summary.glm
```

```
function (object, dispersion = NULL, correlation = FALSE, symbolic
    ...)
{
    est.disp <- FALSE
    df.r <- object$df.residual
    if (is.null(dispersion))
        dispersion <- if (object$family$family %in% c("poisson",
            "binomial"))
            1
```

```
    else if (df.r > 0) {
        est.disp <- TRUE
        if (any(object$weights == 0))
            warning("observations with zero weight not used fo
        sum((object$weights * object$residuals^2)[object$weigh
            0])/df.r
    }
    else {
        est.disp <- TRUE
        NaN
    }
aliased <- is.na(coef(object))
p <- object$rank
if (p > 0) {
    p1 <- 1L:p
    Qr <- qr.lm(object)
    coef.p <- object$coefficients[Qr$pivot[p1]]
    covmat.unscaled <- chol2inv(Qr$qr[p1, p1, drop = FALSE])
    dimnames(covmat.unscaled) <- list(names(coef.p), names(coe
```

```
covmat <- dispersion * covmat.unscaled
var.cf <- diag(covmat)
s.err <- sqrt(var.cf)
tvalue <- coef.p/s.err
dn <- c("Estimate", "Std. Error")
if (!est.disp) {
    pvalue <- 2 * pnorm(-abs(tvalue))
    coef.table <- cbind(coef.p, s.err, tvalue, pvalue)
    dimnames(coef.table) <- list(names(coef.p), c(dn,
        "z value", "Pr(>|z|)"))
}
else if (df.r > 0) {
    pvalue <- 2 * pt(-abs(tvalue), df.r)
    coef.table <- cbind(coef.p, s.err, tvalue, pvalue)
    dimnames(coef.table) <- list(names(coef.p), c(dn,
        "t value", "Pr(>|t|)"))
}
else {
    coef.table <- cbind(coef.p, NaN, NaN, NaN)
```

```
        dimnames(coef.table) <- list(names(coef.p), c(dn,
            "t value", "Pr(>|t|)"))
    }
    df.f <- NCOL(Qr$qr)
}
else {
    coef.table <- matrix(, 0L, 4L)
    dimnames(coef.table) <- list(NULL, c("Estimate", "Std. Err
        "t value", "Pr(>|t|)"))
    covmat.unscaled <- covmat <- matrix(, 0L, 0L)
    df.f <- length(aliased)
}
keep <- match(c("call", "terms", "family", "deviance", "aic",
    "contrasts", "df.residual", "null.deviance", "df.null",
    "iter", "na.action"), names(object), 0L)
ans <- c(object[keep], list(deviance.resid = residuals(object,
    type = "deviance"), coefficients = coef.table, aliased = a
    dispersion = dispersion, df = c(object$rank, df.r, df.f),
    cov.unscaled = covmat.unscaled, cov.scaled = covmat))
```

```
    if (correlation && p > 0) {
        dd <- sqrt(diag(covmat.unscaled))
        ans$correlation <- covmat.unscaled/outer(dd, dd)
        ans$symbolic.cor <- symbolic.cor
    }
    class(ans) <- "summary.glm"
    return(ans)
}
<bytecode: 0x0000000018af6740>
<environment: namespace:stats>
```