Base R plot function

Introduction to Data Visualization

Marcus Birkenkrahe

Table of Contents

1. The `plot` function

- `plot(x,y)` is a *generic* function - it adapts to its arguments: e.g.

Plot type	`x`	`y`
Scatterplot	`numeric`	`numeric`
Boxplot	`factor`	`numeric`
Bar chart	`factor`	`NA`
Mosaic plot	`factor`	`factor`

Plot type	x	y

- For an overview of the different methods available, use `methods`.

  Listing 1.1:

```
methods(plot)
```

```
 [1] plot,ANY-method      plot,psi_func-method plot.aareg*
 [4] plot.acf*            plot.constparty*     plot.correspondence*
 [7] plot.cox.zph*        plot.data.frame*     plot.decomposed.ts*
 [10] plot.default         plot.dendrogram*     plot.density*
 [13] plot.ecdf            plot.factor*         plot.formula*
 [16] plot.function        plot.glmtree*        plot.hclust*
 [19] plot.histogram*      plot.HoltWinters*    plot.isoreg*
 [22] plot.lda*            plot.lm*             plot.lmrob*
 [25] plot.lmtree*         plot.lts*            plot.mca*
 [28] plot.mcd*            plot.medpolish*      plot.mlm*
 [31] plot.modelparty      plot.party           plot.ppr*
 [34] plot.prcomp*         plot.princomp*       plot.profile*
 [37] plot.profile.nls*    plot.raster*         plot.ridgelm*
 [40] plot.rpart*          plot.shingle*        plot.simpleparty*
 [43] plot.spec*           plot.spline*         plot.stepfun
 [46] plot.stl*            plot.Surv*           plot.survfit*
 [49] plot.table*          plot.trellis*        plot.ts
 [52] plot.tskernel*       plot.TukeyHSD*       plot.xyVector*
see '?methods' for accessing help and source code
```

- In the extended example, we already applied `plot` to a data frame, a numeric vector, a factor, and a pair of numeric variables, leading to an array of scatterplots, a bar chart, and a scatterplot.

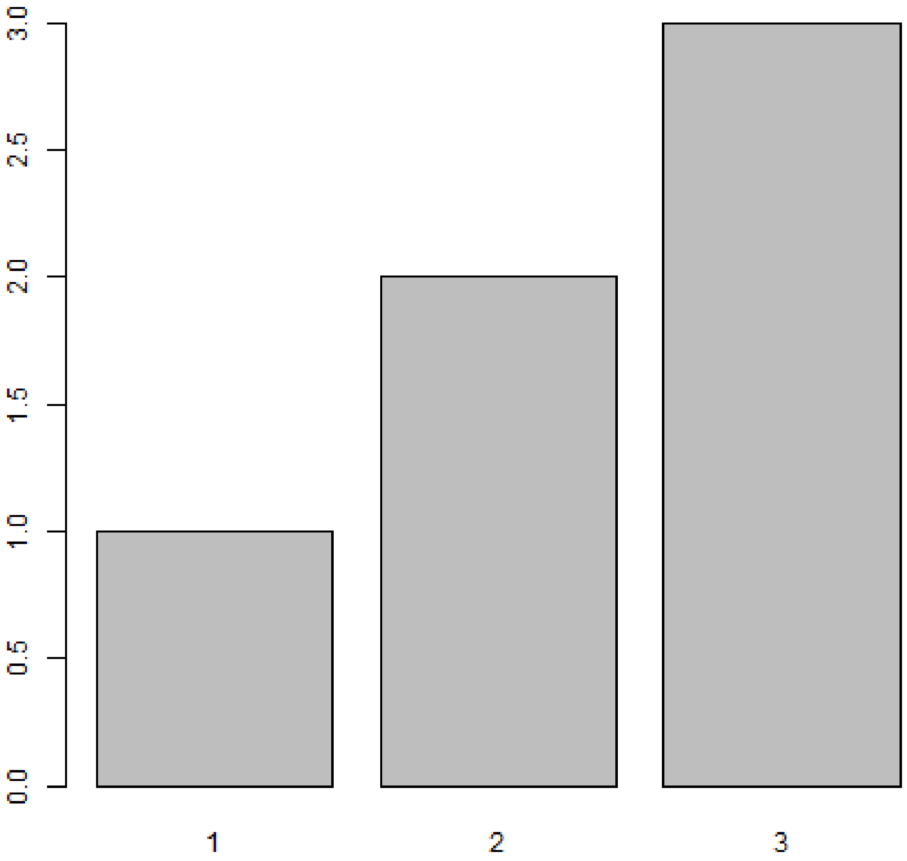
2. Bar chart with `plot`

When only one categorical variable is given, `plot` generates a bar chart with one bar per factor level.

```
x <- c(1,2,2,3,3,3)   # numeric vector
x_f <- factor(x)      # factor vector
x_f
```

```
[1] 1 2 2 3 3 3
Levels: 1 2 3
```

```
plot(x_f)
```



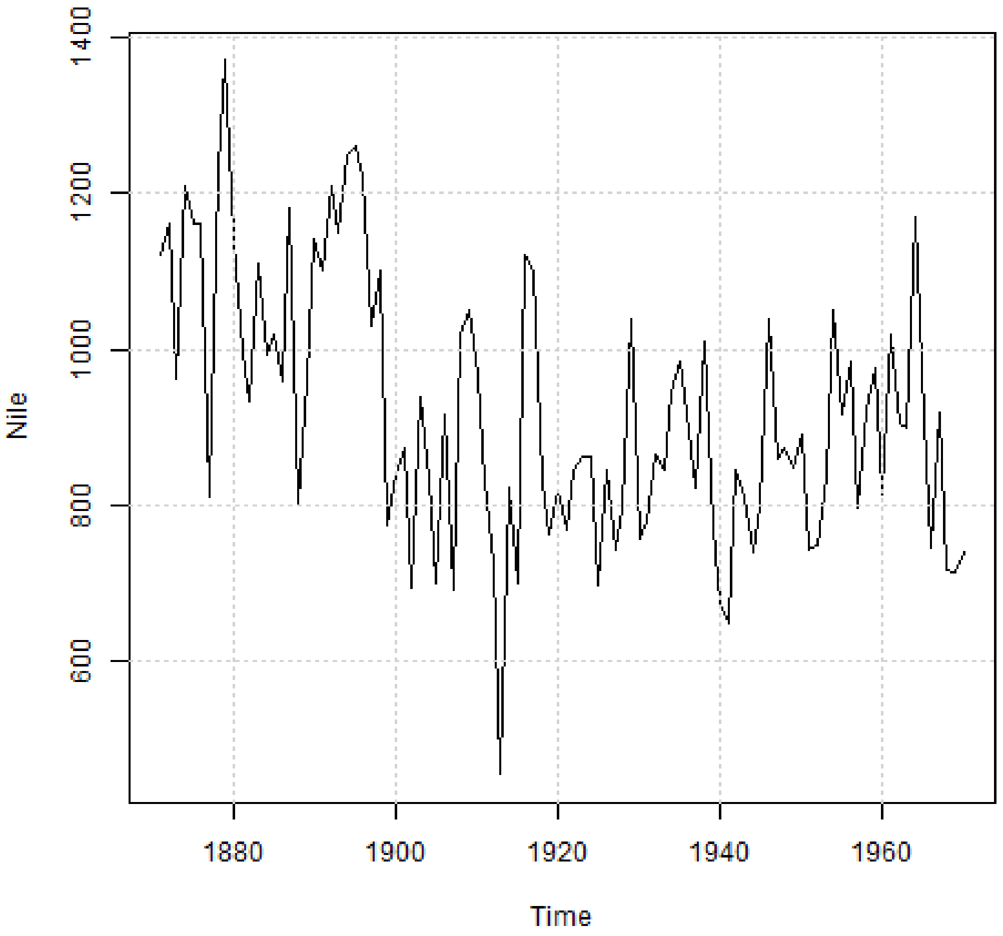
3. Line plot with `plot`

When a time series is given like the data set `Nile`, `plot` generates a line plot.

```
class(Nile)
str(Nile)
```

```
[1] "ts"
Time-Series [1:100] from 1871 to 1970: 1120 1160 963 1210 1160 1160 813 1230
1370 1140 ...
```

```
plot(Nile)  # plotting a time series as line plot
grid()  # draw a grid to make it easier to read a plot
```

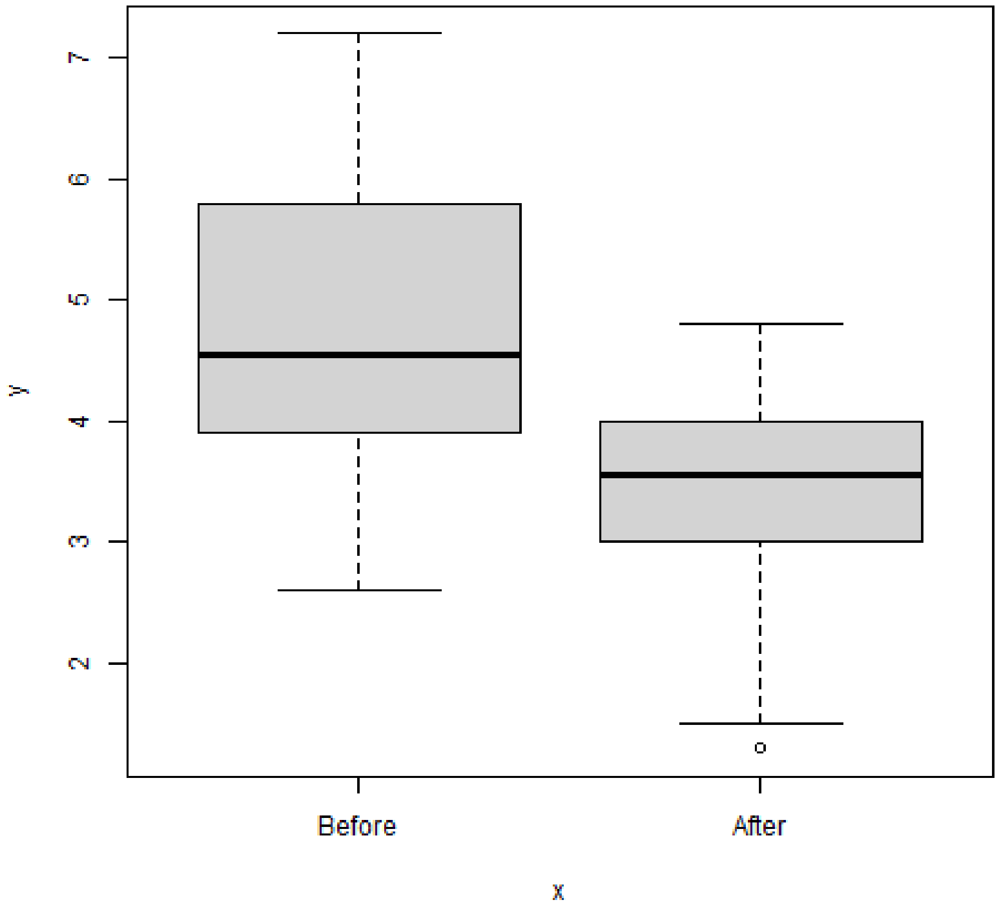


4. Box plots with `plot`

- We used the `boxplot` function to create a boxplot summary of heating gas consumption before and after installation of insulation
- The same result with `plot`:

Listing 4.1:

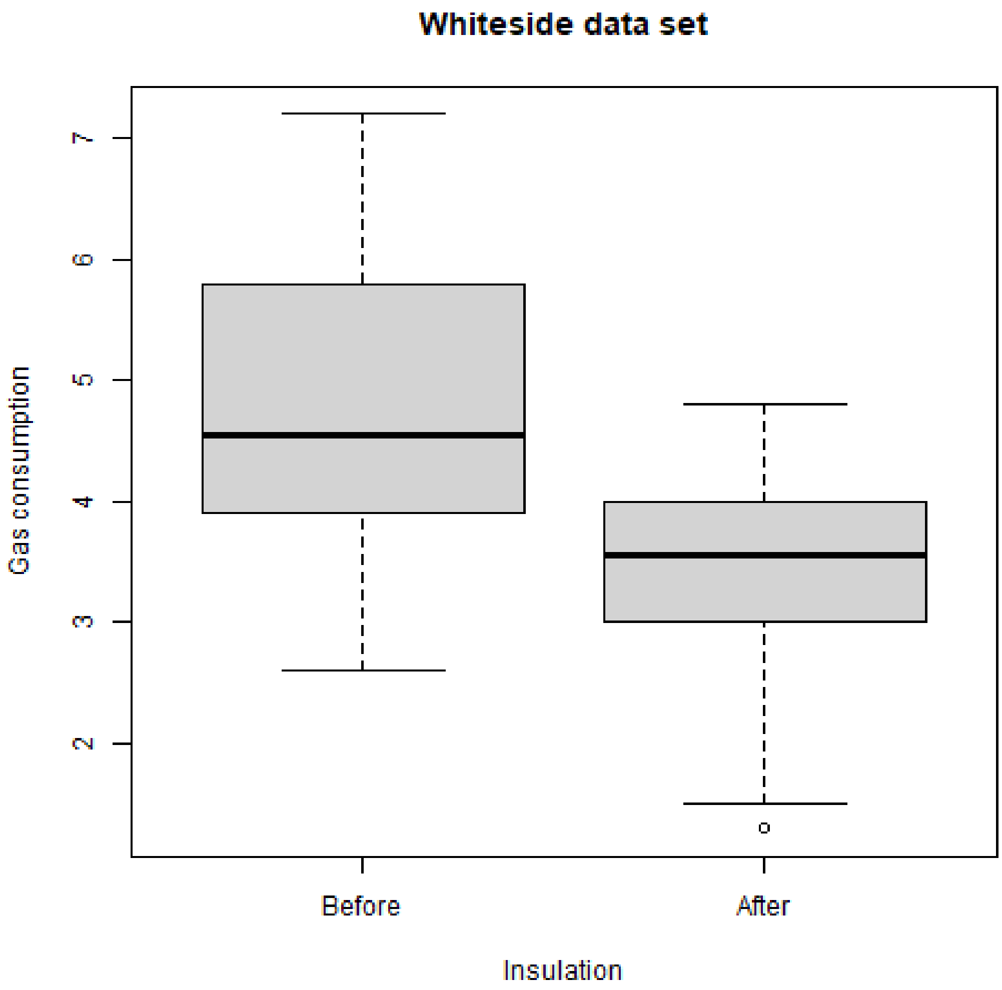
```
plot(whiteside$Insul,whiteside$Gas)
```



- Customization is simple:

Listing 4.2:

```
plot(x=whiteside$Insul, xlab="Insulation",
     y=whiteside$Gas,    ylab="Gas consumption",
     main="Whiteside data set")
```



5. Decision tree models with `plot`

- Many modeling functions in R return a object that can be plotted
- You can store these objects and work with them during analysis
- Example: *decision tree models* from the `whiteside` data frame using the `rpart` package.
- This model predicts the average value of the dependent variable, `Gas`, from the values of the other variables, `Insul` and `Temp`.
- Build the model: the formula only has two parts. The period operator `.` stands for "all other variables".

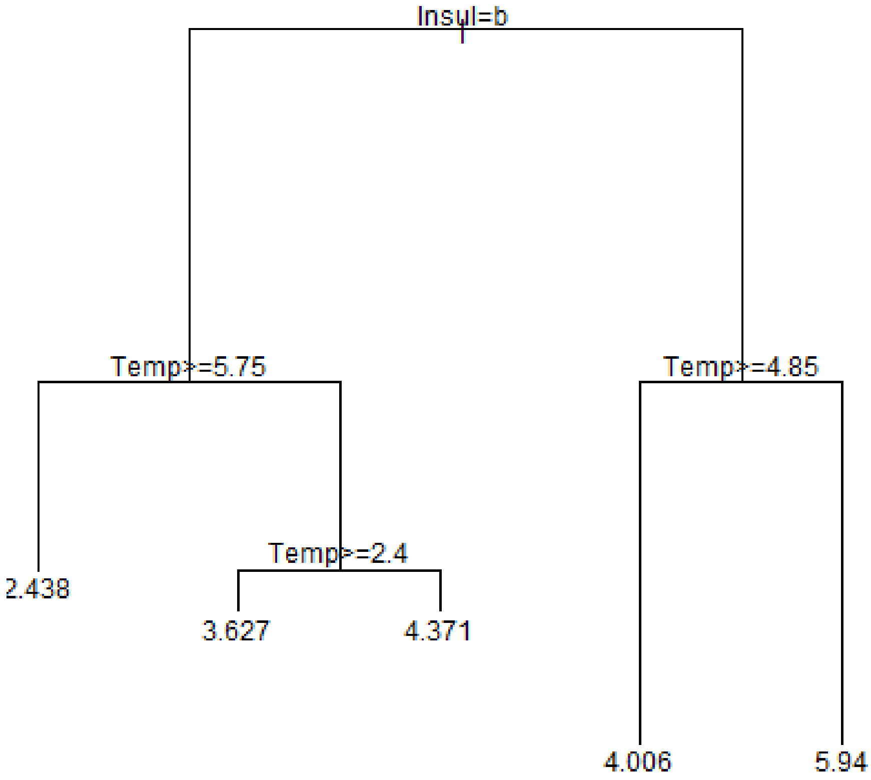
Listing 5.1:

```
library(MASS)   # whiteside data set is in MASS
library(rpart) # rpart: Recursive Partitioning and Regression Trees
rpartModel <- rpart(Gas ~ ., data = whiteside)
class(rpartModel)
```

```
[1] "rpart"
```

- Plot the model and add generic text using `text` for `rpart` models:

```
plot(rpartModel)
text(rpartModel)
```



- [✓] What methods are available for `rpart` and `text`?

```
methods(class = "rpart")
methods(text)
```

```
 [1] as.party     labels       meanvar      model.frame plot         post
 [7] predict      print        prune        residuals   summary      text
see '?methods' for accessing help and source code
 [1] text.default  text.formula* text.rpart*   text.Surv*    text.Surv2*
see '?methods' for accessing help and source code
```

- Let's try `summary`, which generated Tukey's summary for data frames:

```
summary(rpartModel)
```

```
Call:
rpart(formula = Gas ~ ., data = whiteside)
  n= 56

          CP nsplit rel error    xerror      xstd
```

```
   1 0.30233737      0 1.0000000 1.0752448 0.2322884
   2 0.15906314      2 0.3953253 0.7853963 0.1673356
   3 0.03529048      3 0.2362621 0.6116576 0.1260568
   4 0.01000000      4 0.2009716 0.5196344 0.1005721

  Variable importance
  Temp Insul
    67    33

  Node number 1: 56 observations,    complexity param=0.3023374
   mean=4.071429, MSE=1.339541
   left son=2 (30 obs) right son=3 (26 obs)
   Primary splits:
       Insul splits as  RL,       improve=0.2979115, (0 missing)
       Temp  < 7.05 to the right, improve=0.2905559, (0 missing)
   Surrogate splits:
       Temp < 5.35 to the left,  agree=0.679, adj=0.308, (0 split)

  Node number 2: 30 observations,    complexity param=0.1590631
   mean=3.483333, MSE=0.6287222
   left son=4 (8 obs) right son=5 (22 obs)
   Primary splits:
       Temp < 5.75 to the right, improve=0.6326062, (0 missing)

  Node number 3: 26 observations,    complexity param=0.3023374
   mean=4.75, MSE=1.300192
   left son=6 (16 obs) right son=7 (10 obs)
   Primary splits:
       Temp < 4.85 to the right, improve=0.6807166, (0 missing)

  Node number 4: 8 observations
   mean=2.4375, MSE=0.3723437

  Node number 5: 22 observations,    complexity param=0.03529048
   mean=3.863636, MSE=0.1795868
   left son=10 (15 obs) right son=11 (7 obs)
   Primary splits:
       Temp < 2.4  to the right, improve=0.6700458, (0 missing)

  Node number 6: 16 observations
   mean=4.00625, MSE=0.3630859

  Node number 7: 10 observations
   mean=5.94, MSE=0.4984

  Node number 10: 15 observations
   mean=3.626667, MSE=0.04595556

  Node number 11: 7 observations
   mean=4.371429, MSE=0.0877551
```

6. Model-based recursive partitioning with `plot`

- MOB models have tree-based structures like decision tree models
- Each terminal node contains a linear regression model that generates predictions from other covariates (independent variables)
- The code to generate the model looks very similar to `rpart`: the formula has three parts -

Gas is predicted, Temp is the covariate to predict, and Insul is the partitioning variable used to build the tree.

Listing 6.1:

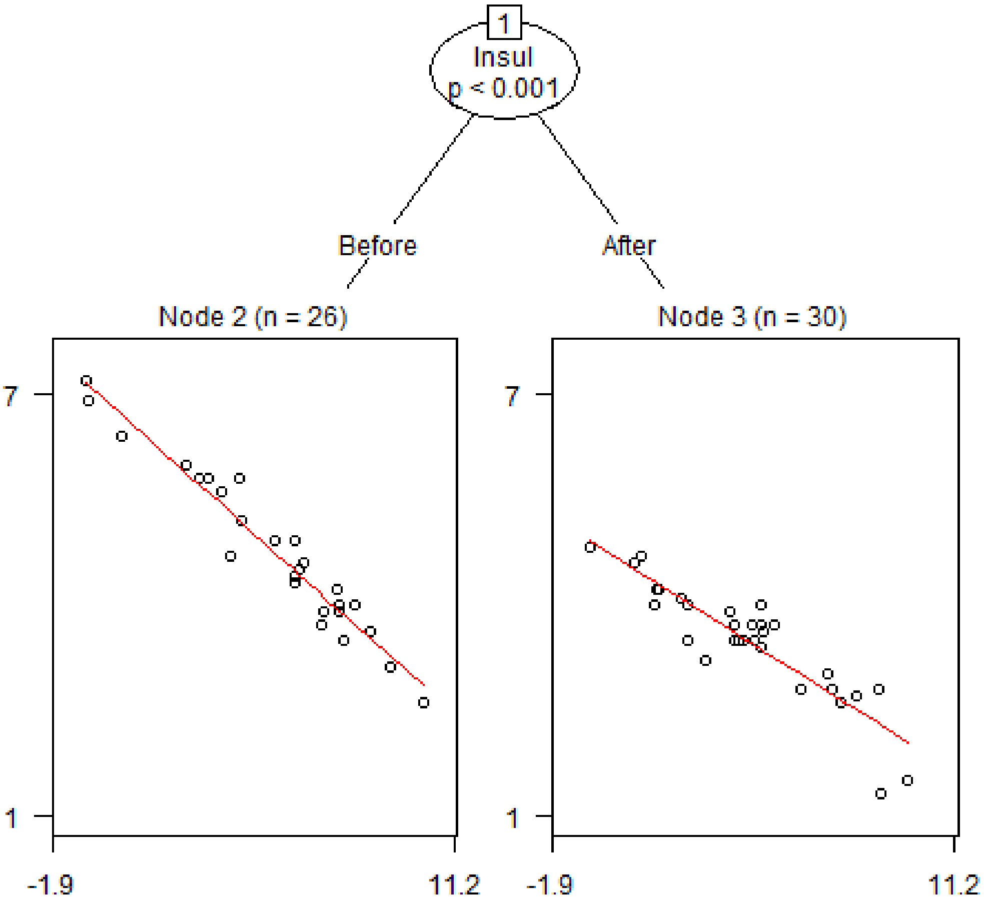
```
library(partykit) # this will load three other required packages
MOBmodel <- lmtree(Gas ~ Temp | Insul, data = whiteside)
class(MOBmodel)
```

```
[1] "lmtree"      "modelparty" "party"
```

- In the plot, all records are assigned to one of the nodes, and a separate linear regression model that predicts Gas from Temp is built for each node.

Listing 6.2:

```
plot(MOBmodel)
```



- What methods are available for lmtree ?

```
methods(class = "lmtree")
```

```
[1] plot     predict print    prune
see '?methods' for accessing help and source code
```

7. What does this mean?

1. Enormous graphics flexibility even within the base R graphics package. Regrettably, there seems to be a "package envy" phenomenon within the R community, fostered by RStudio's aggressive marketing
2. You can define your own object classes and construct methods for generic functions like `plot` or `summary` that make them generate specialized results for our object classes.
3. `methods` reveals the "S3 Object" character of R, dependent on the packages loaded into your R session. S is the language that came before R. In OOP, objects have *methods* and *attributes*.

```
attributes(mtcars)
lmod <- lm(mtcars$wt ~ mtcars$mpg)
lmod
attributes(lmod)
```

```
$names
[1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"

$row.names
[1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
[4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
[7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"

$class
[1] "data.frame"

Call:
lm(formula = mtcars$wt ~ mtcars$mpg)

Coefficients:
(Intercept)   mtcars$mpg
    6.0473      -0.1409
$names
[1] "coefficients" "residuals"     "effects"       "rank"
[5] "fitted.values" "assign"        "qr"            "df.residual"
[9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
```

For an example of the information stored in this model, run `plot(lmod)` in the R console - this will produce several plots at once (click on the graph to move to the next one).

8. Optional parameters for base graphics

- There are 72 optional base graphics parameters set by the `par` function

- Check out the help for `par` right now
- `par` can also be called (like `options` and other system functions)

```
names(par())
```

```
[1] "xlog"      "ylog"      "adj"       "ann"       "ask"       "bg"
[7] "bty"       "cex"       "cex.axis"  "cex.lab"   "cex.main"  "cex.sub"
[13] "cin"       "col"       "col.axis"  "col.lab"   "col.main"
"col.sub"
[19] "cra"       "crt"       "csi"       "cxy"       "din"       "err"
[25] "family"    "fg"        "fig"       "fin"       "font"
"font.axis"
[31] "font.lab"  "font.main" "font.sub"  "lab"       "las"       "lend"
[37] "lheight"   "ljoin"     "lmitre"    "lty"       "lwd"       "mai"
[43] "mar"       "mex"       "mfcol"     "mfg"       "mfrow"     "mgp"
[49] "mkh"       "new"       "oma"       "omd"       "omi"       "page"
[55] "pch"       "pin"       "plt"       "ps"        "pty"       "smo"
[61] "srt"       "tck"       "tcl"       "usr"       "xaxp"      "xaxs"
[67] "xaxt"      "xpd"       "yaxp"      "yaxs"      "yaxt"      "ylbias"
```

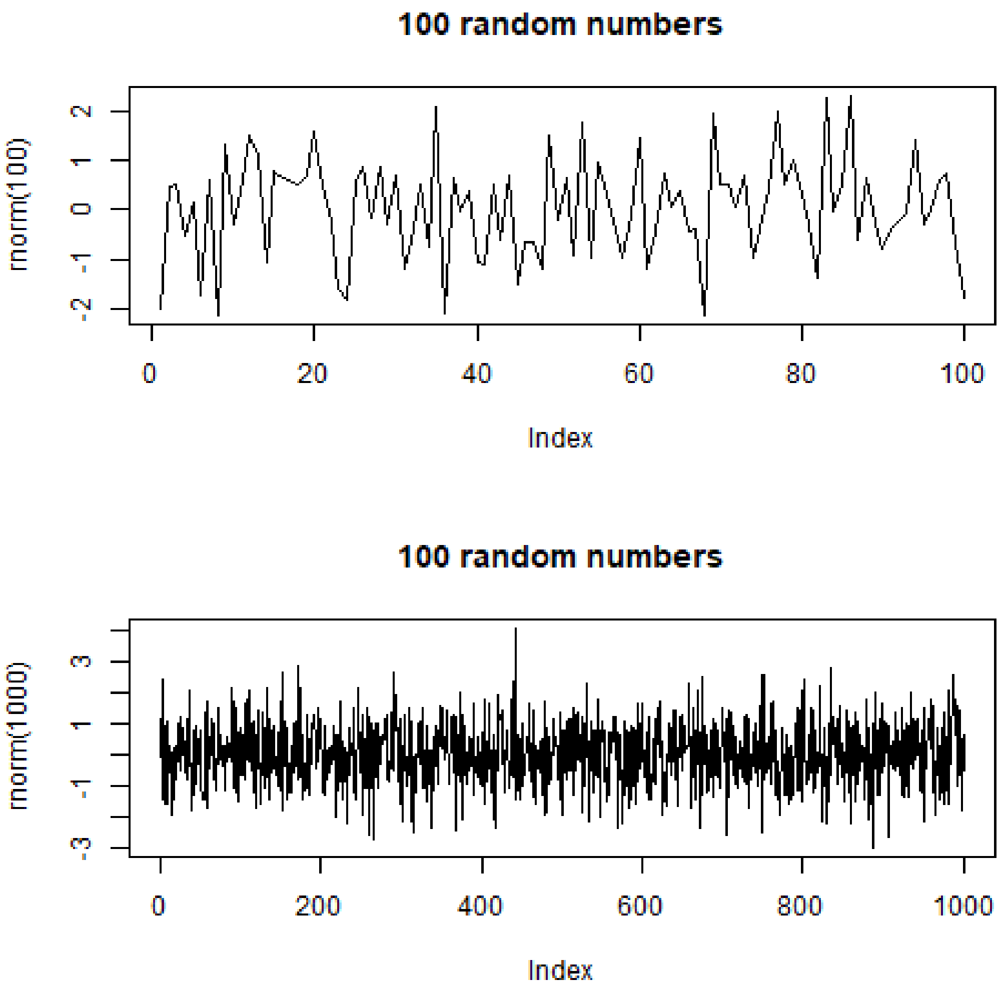
- As you can read in `help(par)`, some of these parameters are read-only (i.e. their values are fixed)

9. Important parameters for `plot` customization

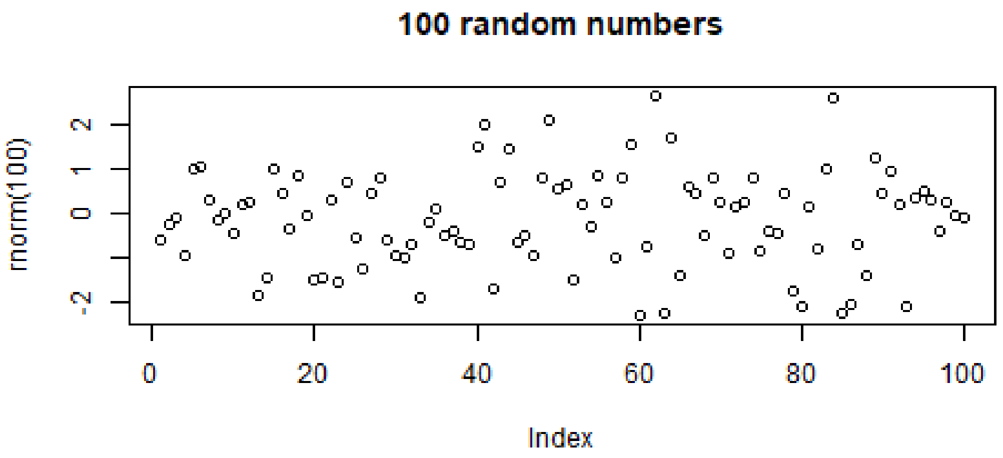
- `mfrow`, a 2-dim vector that sets up an array of plots

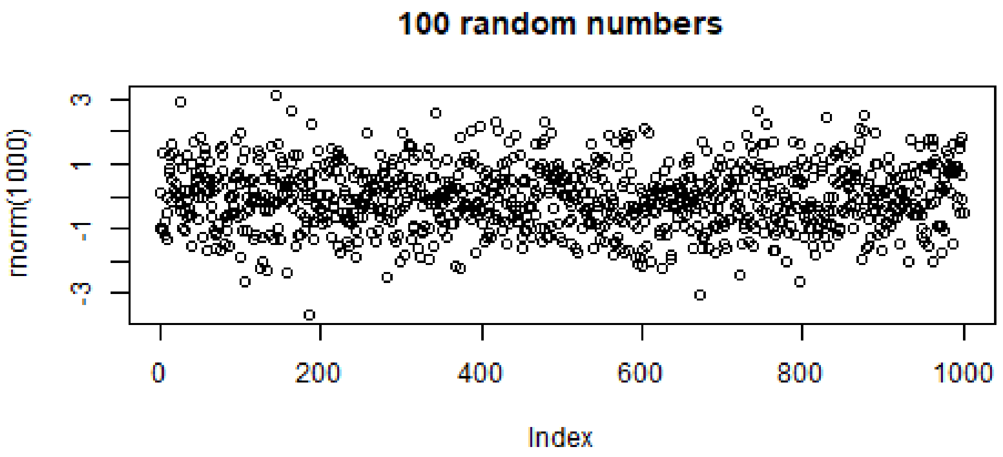
Listing 9.1:

```
par(mfrow=c(2,1))
plot(rnorm(100), type="l")
title("100 random numbers")
plot(rnorm(1000), type="l")
title("100 random numbers")
```



```
    par(mfrow=c(2,1))
    plot(rnorm(100))
    title("100 random numbers")
    plot(rnorm(1000))
    title("100 random numbers")
```


100 random numbers
rnorm(100)
-2
0
1
2
0
20
40
60
80
100
Index

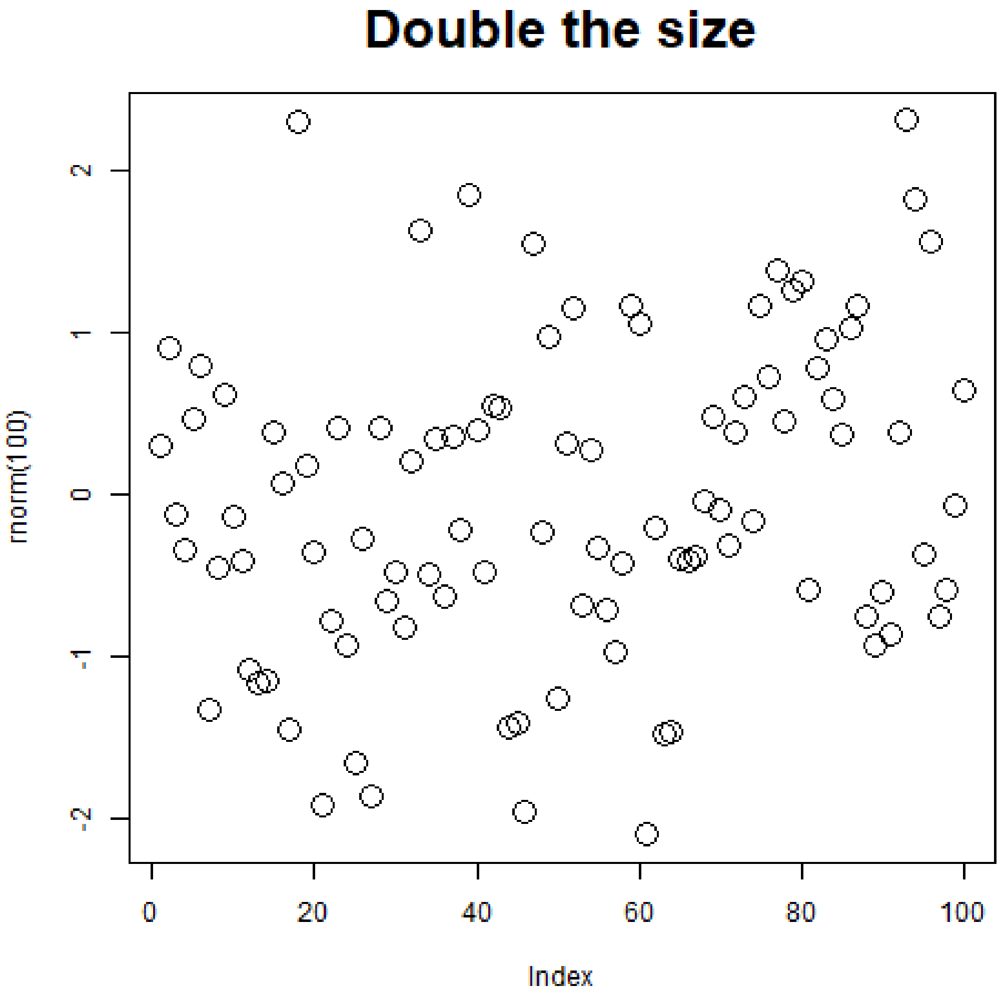

100 random numbers
rnorm(1000)
-3
-1
1
3
0
200
400
600
800
1000
Index

- Several parameters come in groups:

1. The `cex` family scales text and symbols in relation to one another, e.g. `cex.main` scales the main plot title relative to `cex`

Listing 9.2:

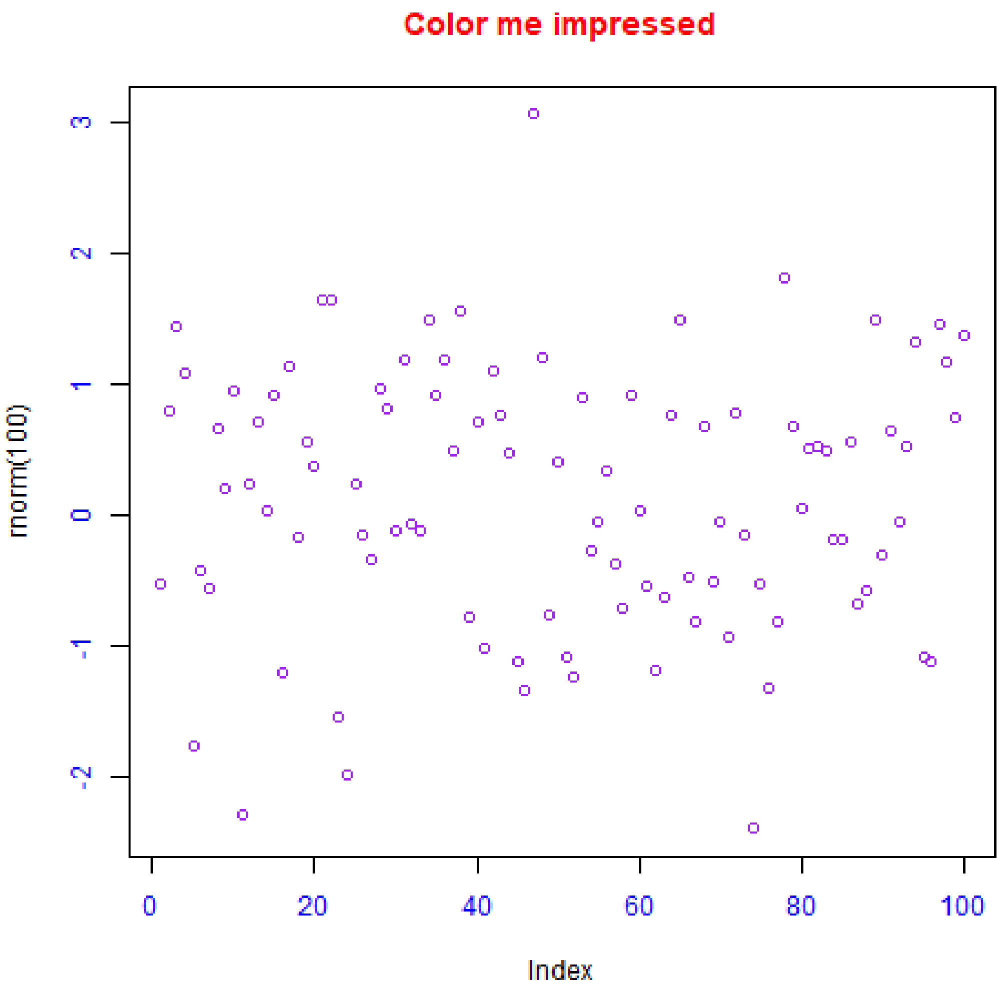
```
plot(rnorm(100),
     main="Double the size",
     cex=2,    # double point symbol size
     cex.main=2 # scale title relative to cex
     )
```



2. The `col` family specifies colors for points, lines, text[1]. To see the complete set of colors, enter `colors()`.

```
plot(rnorm(100),
     main="Color me impressed",
     col      = "purple",  # color points
     col.main = "red",    # color title
     col.axis = "blue"   # color axis labels
     )
```

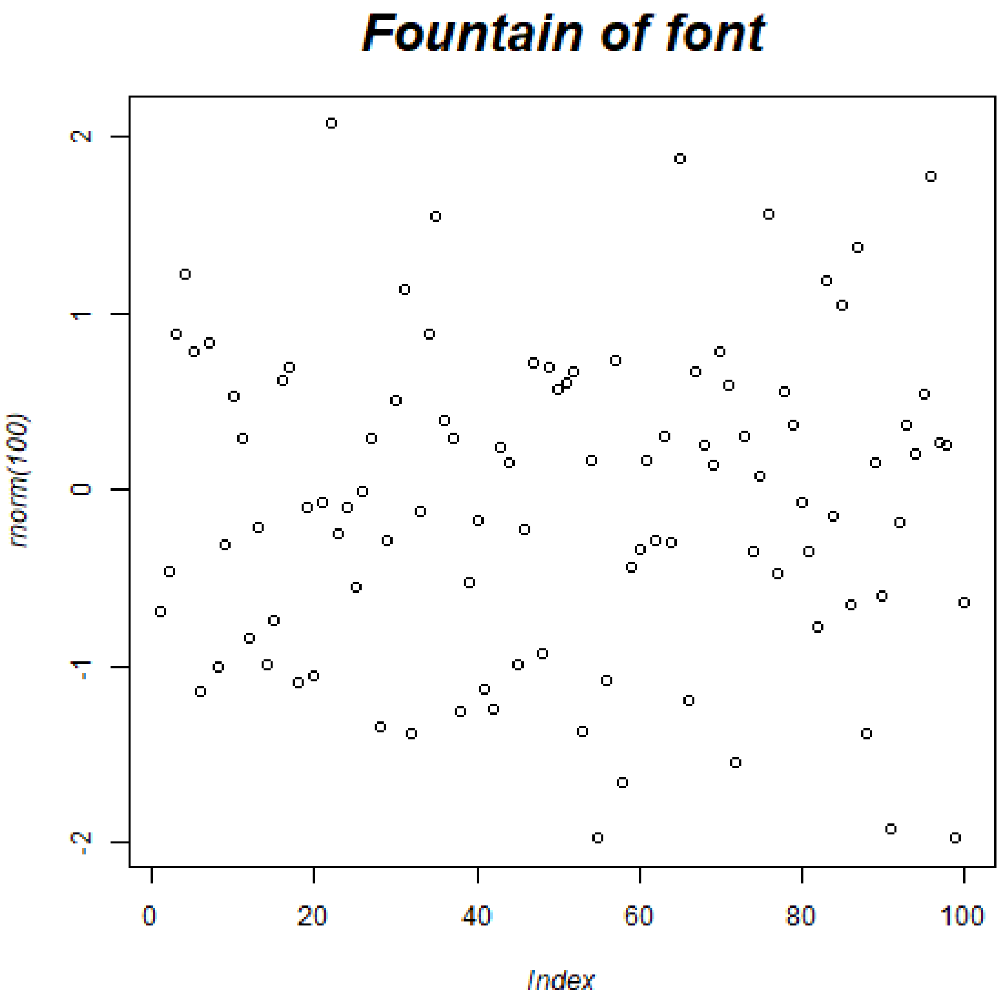
[1] DEFINITION NOT FOUND.



3. The `font` family specifies font types (plain = 1, bold = 2, italic = 3, bold italic = 4).

   Listing 9.3:

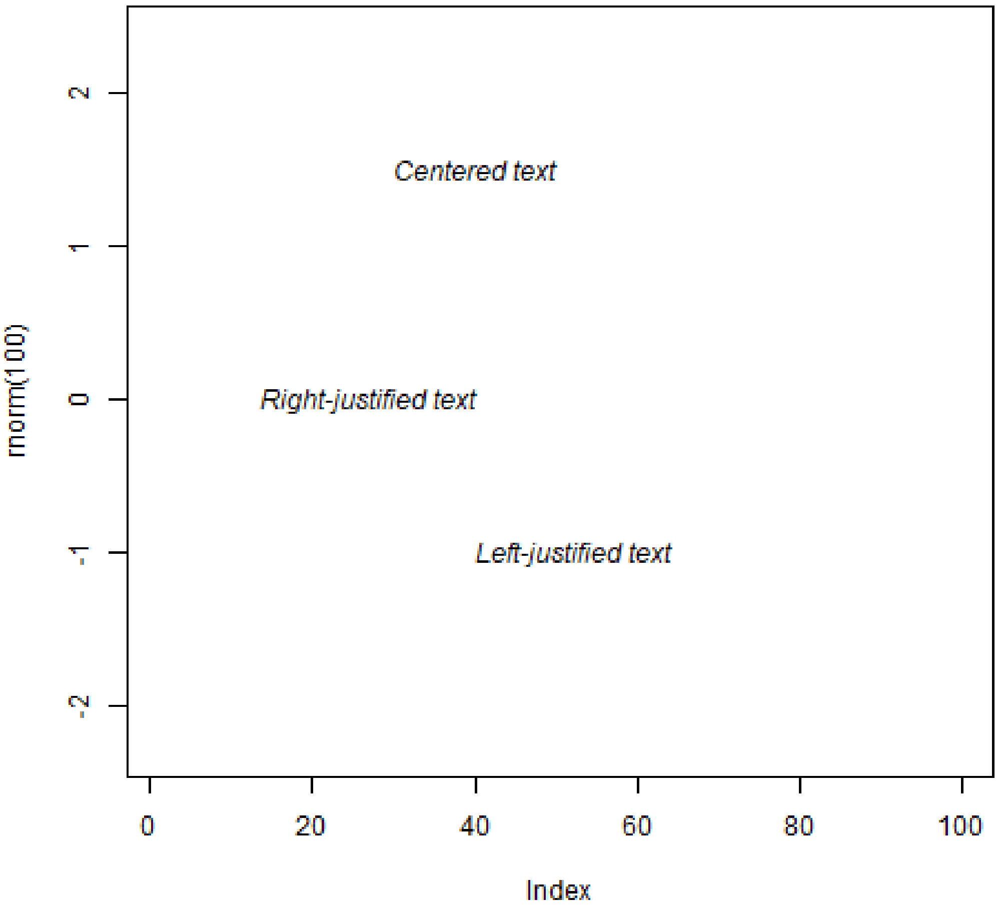
```
plot(rnorm(100),
     main="Fountain of font",
     font.main = 4,  # title font bold italic
     font.lab = 3,   # axis labels in italic
     cex.main = 2    # double title font size
     )
```

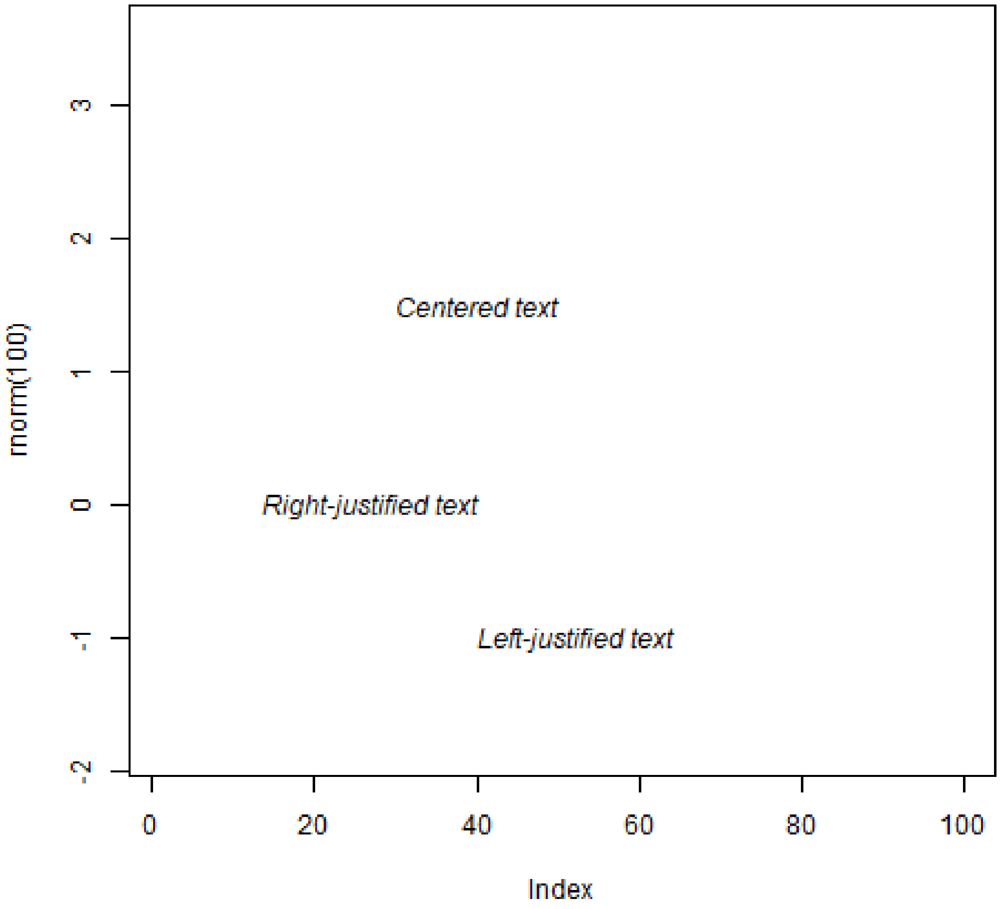

Fountain of font
rnorm(100)
2
1
0
-1
-2
0
20
40
60
80
100
Index

10. Other parameters

- `adj` specifies justification of the text (0=left,1=right, center=0.5)

```
plot(rnorm(100), type="n")
text(x=40,y=0,labels="Right-justified text",adj=1,font=3)
text(x=40,y=-1,labels="Left-justified text",adj=0,font=3)
text(x=40,y=1.5,labels="Centered text",adj=0.5,font=3)
```

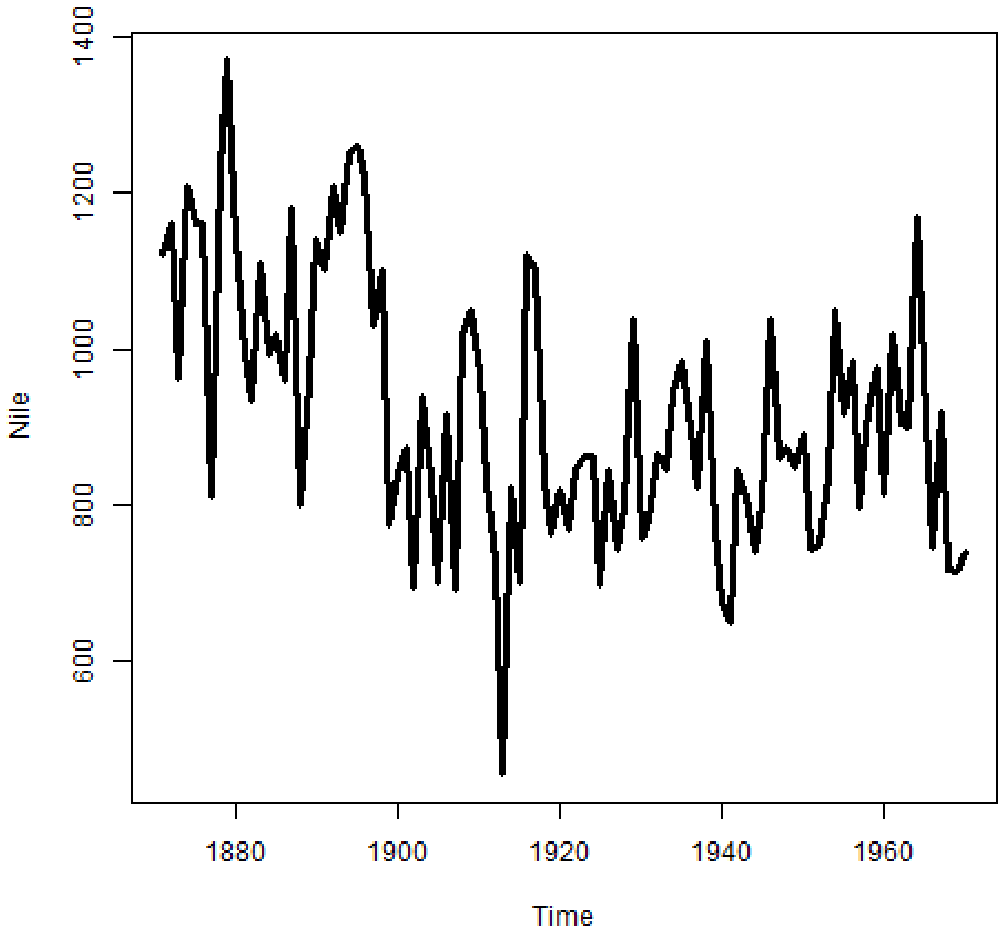




- `lty` and `lwd` specify line type and line width.

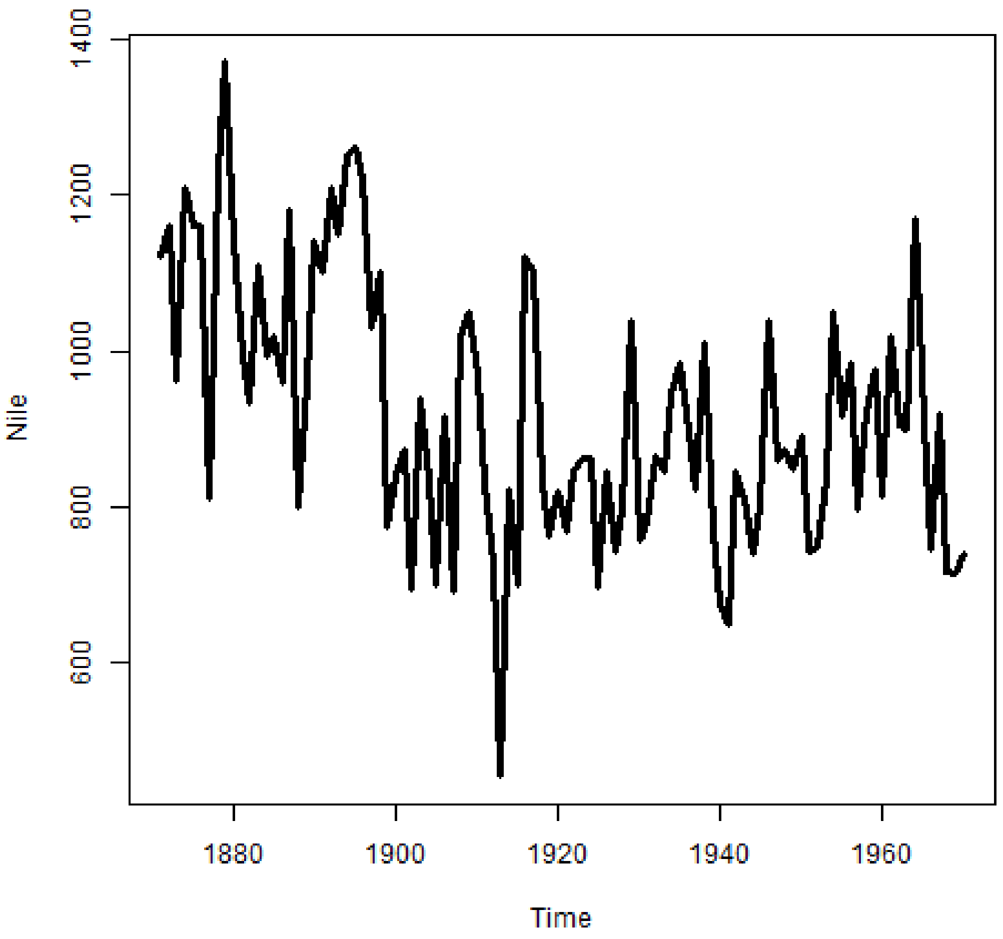
Listing 10.1:

```
plot(Nile, lty=2)
```


Nile
1400
1200
1000
800
600
1880
1900
1920
1940
1960
Time

Listing 10.2:

```
plot(Nile, lwd=3)
```



- [ ] How can you see current values of these parameters? Remember how to do this with `options()`? What type of R object is `par()`?

```
class(par())
par()$lty
par()$adj
```

```
[1] "list"
[1] "solid"
[1] 0.5
```

11. Multiple plots and passing parameters

- The `ask` parameter is a *logical* flag (the default value is `FALSE`) that specifies whether the graphics system should wait for a user response before displaying the next plot.
- [ ] How can I confirm that `ask` is `logical`, and see its value?

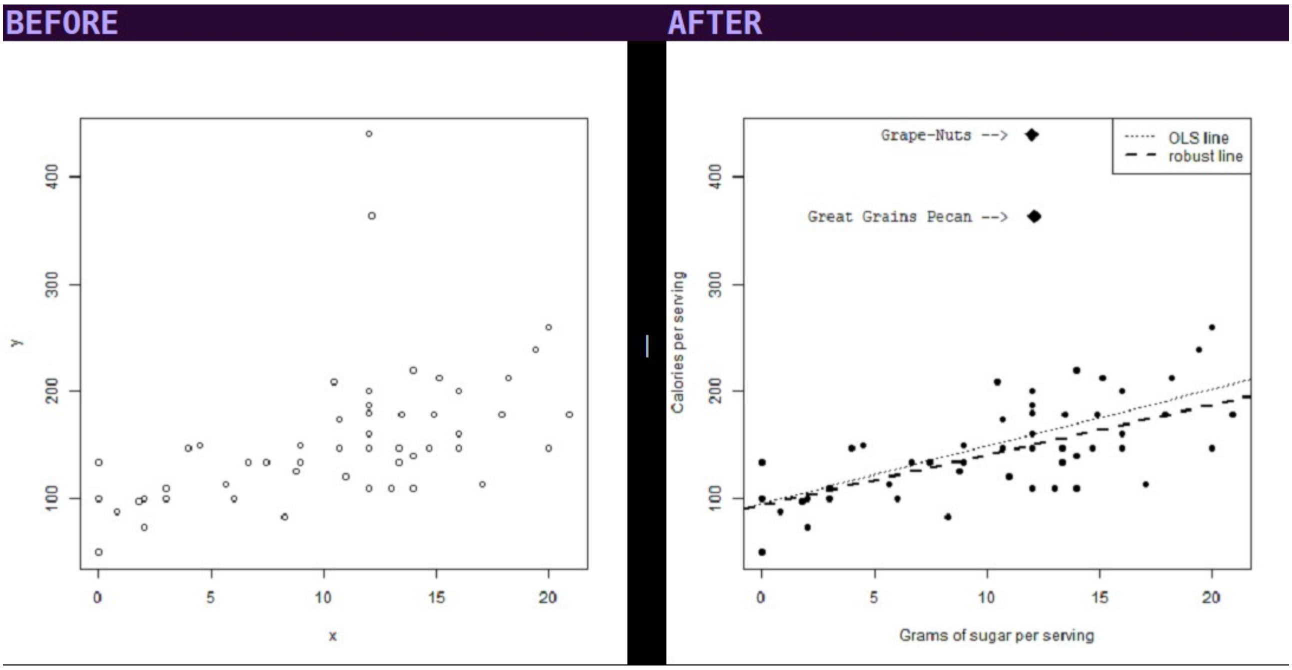
```
class(par()$ask)
```

```
par()$ask
```

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

- [ ] See e.g. `example(Nile)` in the R console (not in Org-mode - because this involves OS shell commands which Emacs cannot render)
- This option is not always set correctly after displaying multiple plots - it must be set back by hand with `par(ask = FALSE)`.
- Some parameters cannot be set as passing parameters, e.g. `las` - usually for different plot types (e.g. mosaic plots).

12. Adding points and lines to a scatterplot

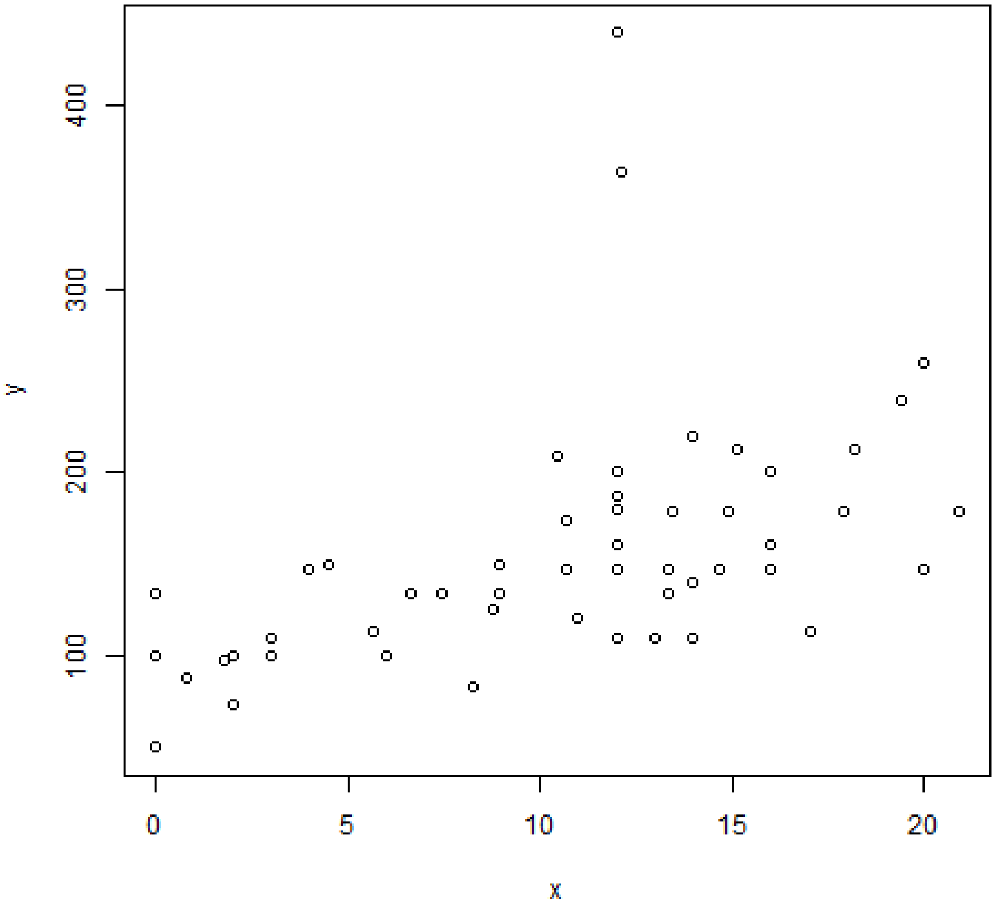


13. Starting `plot` without bells and whistles

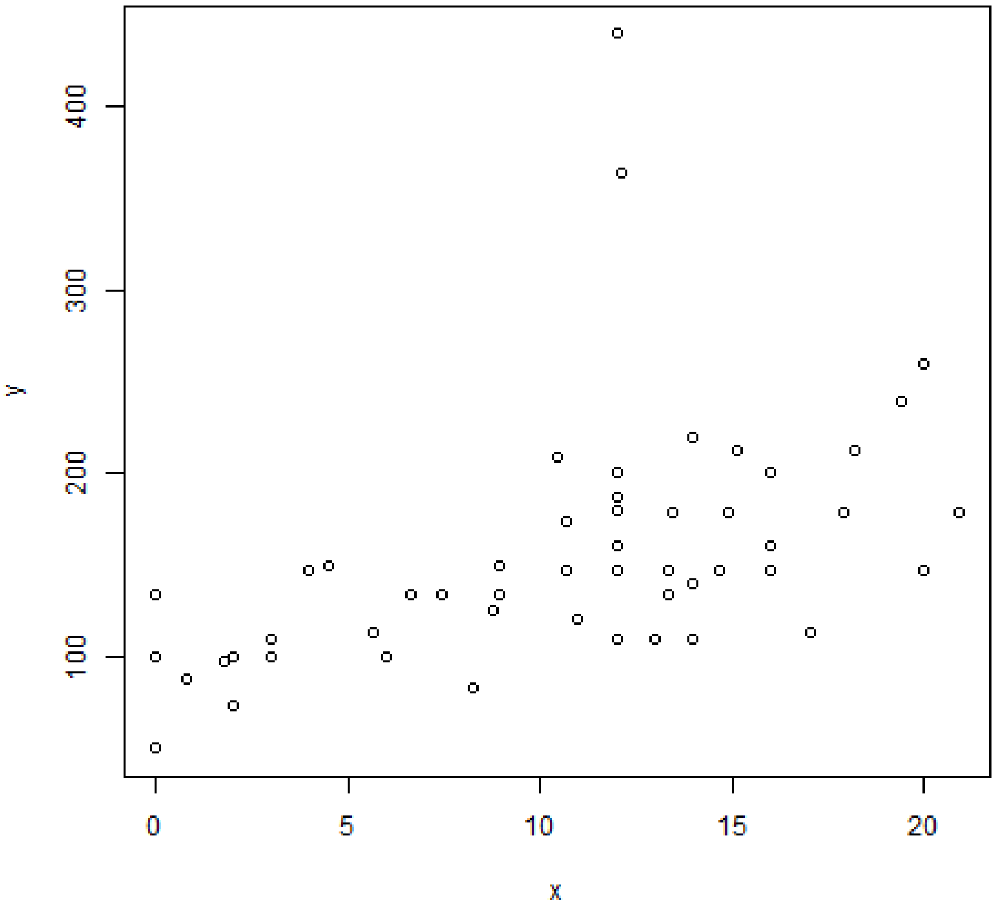
- Starting point is calories vs. sugars from the `UScereal` data frame

Listing 13.1:

```
par(mfrow=c(1,1))
library(MASS)
x <- UScereal$sugars
y <- UScereal$calories
plot(x,y)
```



y
100
200
300
400
0
5
10
15
20
x



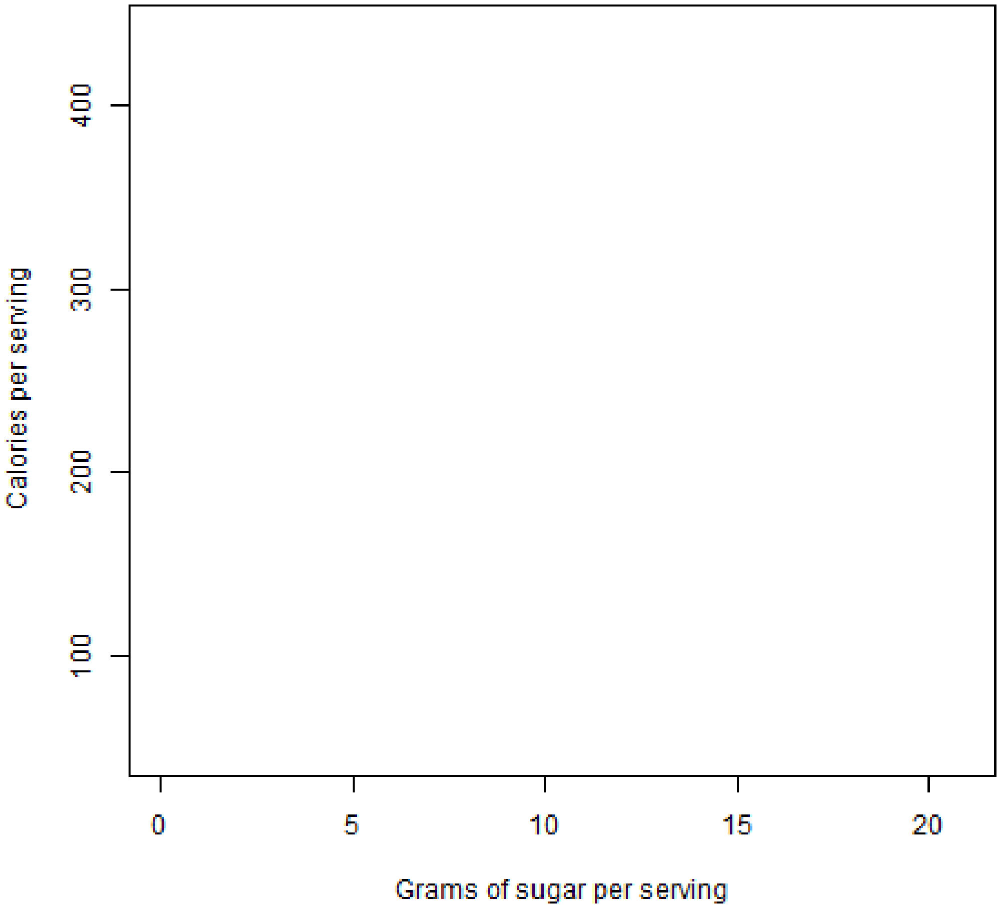
- `mfrow` creates a persistent 1x1 array (single plot)
- Variables `x`, `y` are defined because we use them again
- `plot(x,y)` is invoked to get a simple scatterplot

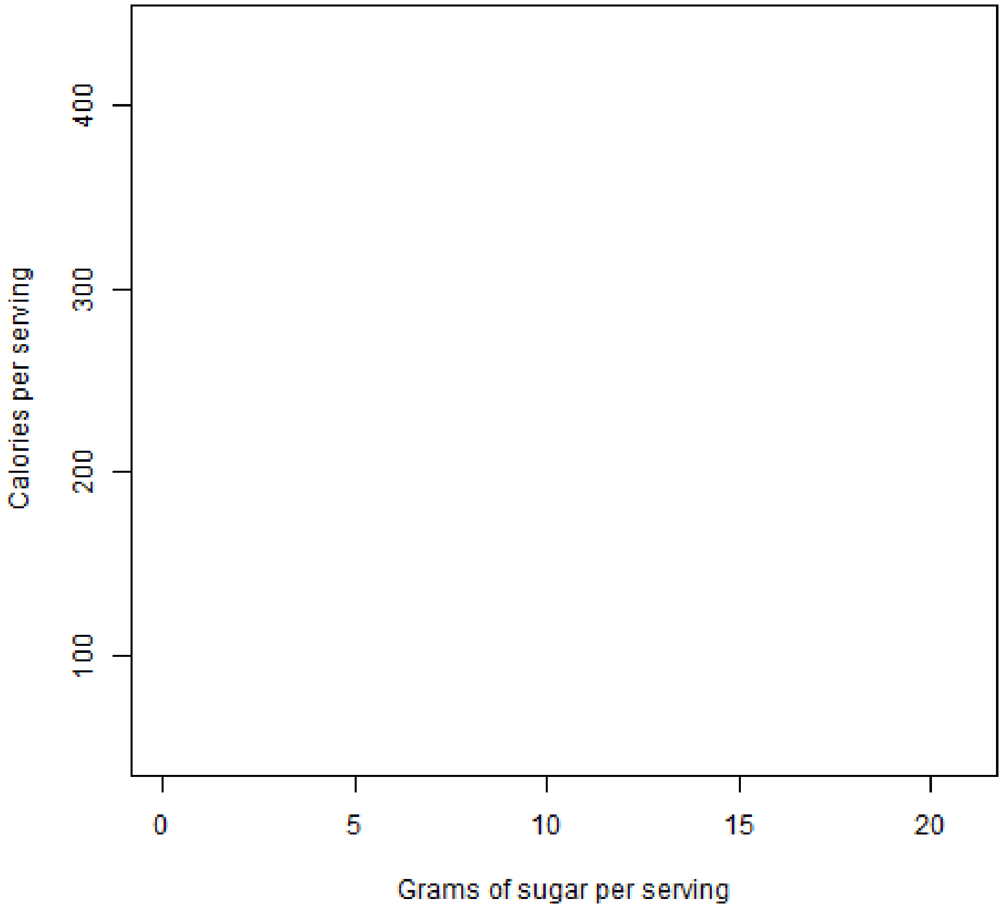
14. Adding axis labels `xlab`, `ylab`, and `type`

- The next block invokes `plot` again, but with three optional arguments:
  1. `xlab`: a `character` string for the x-axis label
  2. `ylab`: a `character` string for the y-axis label
  3. `type = "n"`: specifies that the plot is constructed but *not displayed*

  Listing 14.1:

```
plot(x,y,
     xlab="Grams of sugar per serving",
     ylab="Calories per serving",
     type = "n")
```


Calories per serving
100
200
300
400
0
5
10
15
20
Grams of sugar per serving



15. Highlighting outliers with `points`

- `points` behaves much like `plot`. It adds points to an existing plot.
- `[✓]` Is `points` a *generic* R function?

```
methods(points)
```

```
[1] points.default  points.formula* points.Surv*    points.Surv2*
[5] points.survfit* points.table*
see '?methods' for accessing help and source code
```

- The function takes the coordinate vectors of points to plot
- We want to distinguish outliers and non-outliers. Our definition for outliers: cereals with more than 300 calories per serving.
- To extract the subvectors, we define an index vector - it contains only the index value of the outliers.

```
index <- which(y > 300) # y is our calorie vector
index # vector of outlier value indices
x[index]  # UScereal$sugars[index] outliers
```

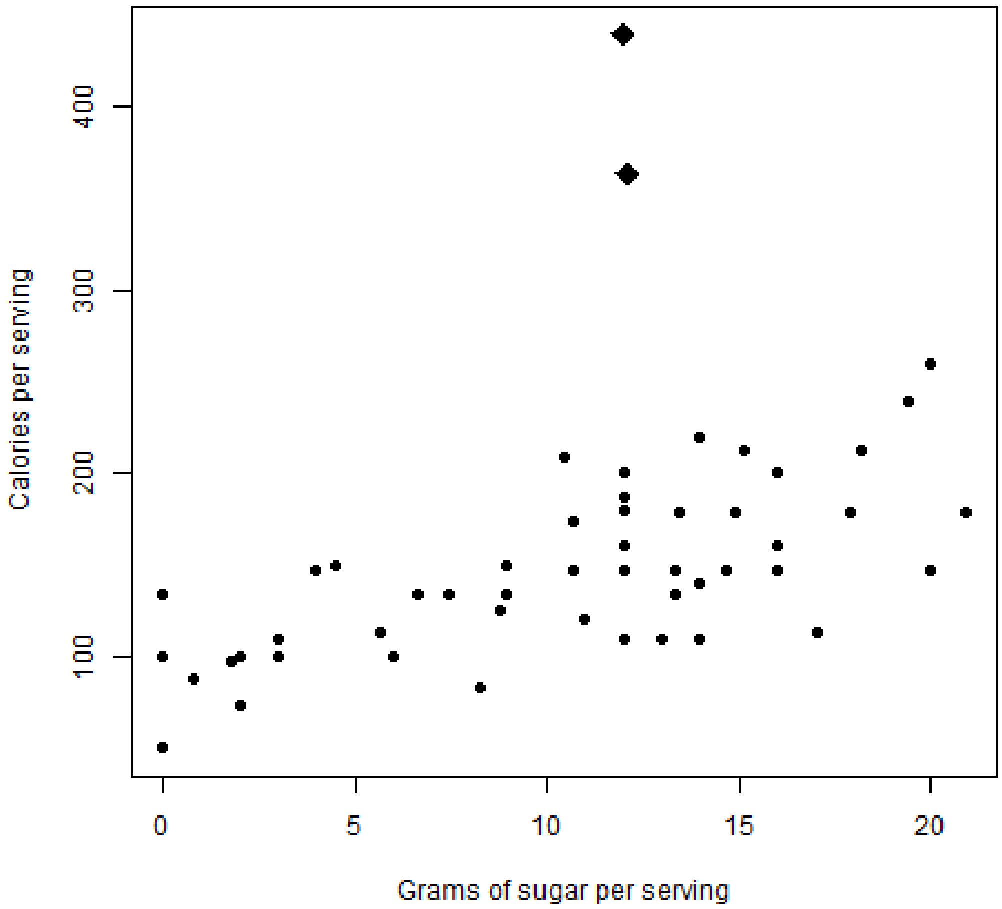
```
y[index]  # UScereal$calories[index] outliers
x[-index]  # UScereal$sugars[index] non-outliers
y[-index]  # UScereal$calories[index] non-outliers
```

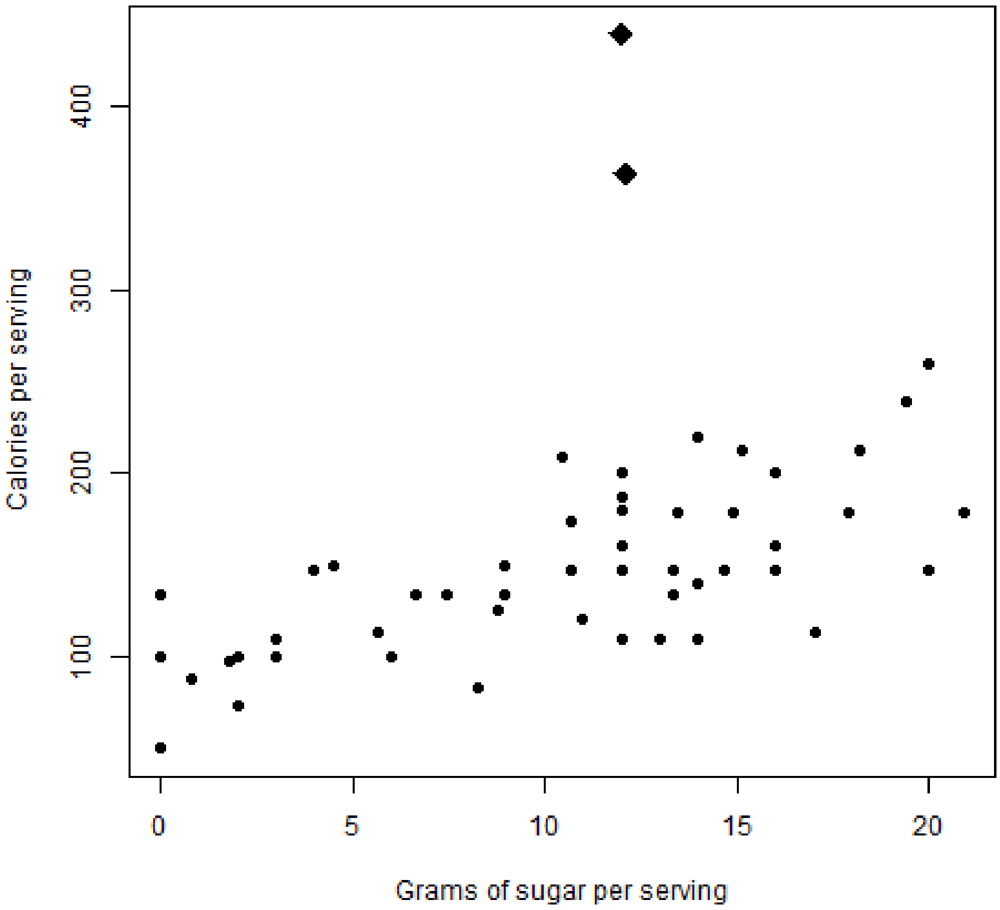
```
[1] 31 32
[1] 12.00000 12.12121
[1] 440.0000 363.6364
[1] 18.181818 15.151515  0.000000 13.333333 14.000000 10.666667  8.955224
[8]  7.462687 16.000000  0.800000 12.000000 14.000000 13.000000  3.000000
[15]  2.000000 12.000000 13.000000 14.000000  3.000000 13.333333
6.666667
[22] 13.000000 14.666667  8.750000 14.925373 17.910448 16.000000
17.045455
[29] 12.000000  5.681818 11.000000 13.333333  8.270677 12.000000
2.000000
[36]  8.955224 12.000000 19.402985  6.000000 13.432836 10.447761
20.000000
[43] 20.895522  3.000000  0.000000 12.000000 16.000000 16.000000
12.000000
[50]  1.769912  3.000000  0.000000  0.000000 20.000000  3.000000
3.000000
[57] 14.000000  3.000000  4.000000 12.000000  4.477612  3.000000
10.666667
[1] 212.12121 212.12121 100.00000 146.66667 110.00000 173.33333 134.32836
[8] 134.32836 160.00000  88.00000 160.00000 220.00000 110.00000 110.00000
[15] 100.00000 110.00000 110.00000 220.00000 110.00000 133.33333
133.33333
[22] 110.00000 146.66667 125.00000 179.10448 179.10448 146.66667
113.63636
[29] 146.66667 113.63636 120.00000 146.66667  82.70677 186.66667
73.33333
[36] 149.25373 110.00000 238.80597 100.00000 179.10448 208.95522
260.00000
[43] 179.10448 100.00000  50.00000 200.00000 160.00000 200.00000
180.00000
[50]  97.34513 110.00000 134.32836 134.32836 146.66667 110.00000
110.00000
[57] 140.00000 100.00000 146.66667 110.00000 149.25373 100.00000
146.66667
```

- We plot the non-outliers with one type of point (`pch=16`), the outliers with another (`pch=18`) and twice as large (`cex=2`).
- Now we're ready to enhance our basic plot.

Listing 15.1:

```
plot(x,y,
     xlab="Grams of sugar per serving",
     ylab="Calories per serving",
     type = "n")
index <- which(y > 300)
points(x[-index], y[-index], pch=16)
points(x[index], y[index], pch=18, cex=2)
```


Calories per serving
100
200
300
400
0
5
10
15
20
Grams of sugar per serving

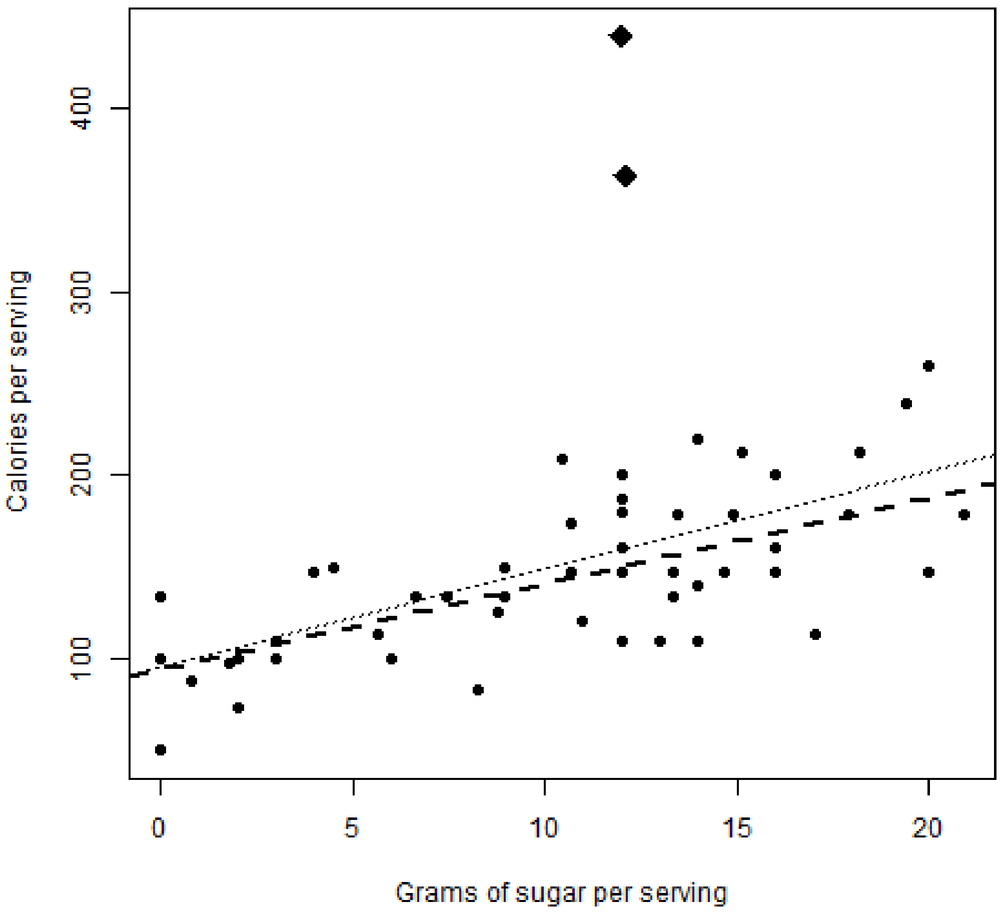


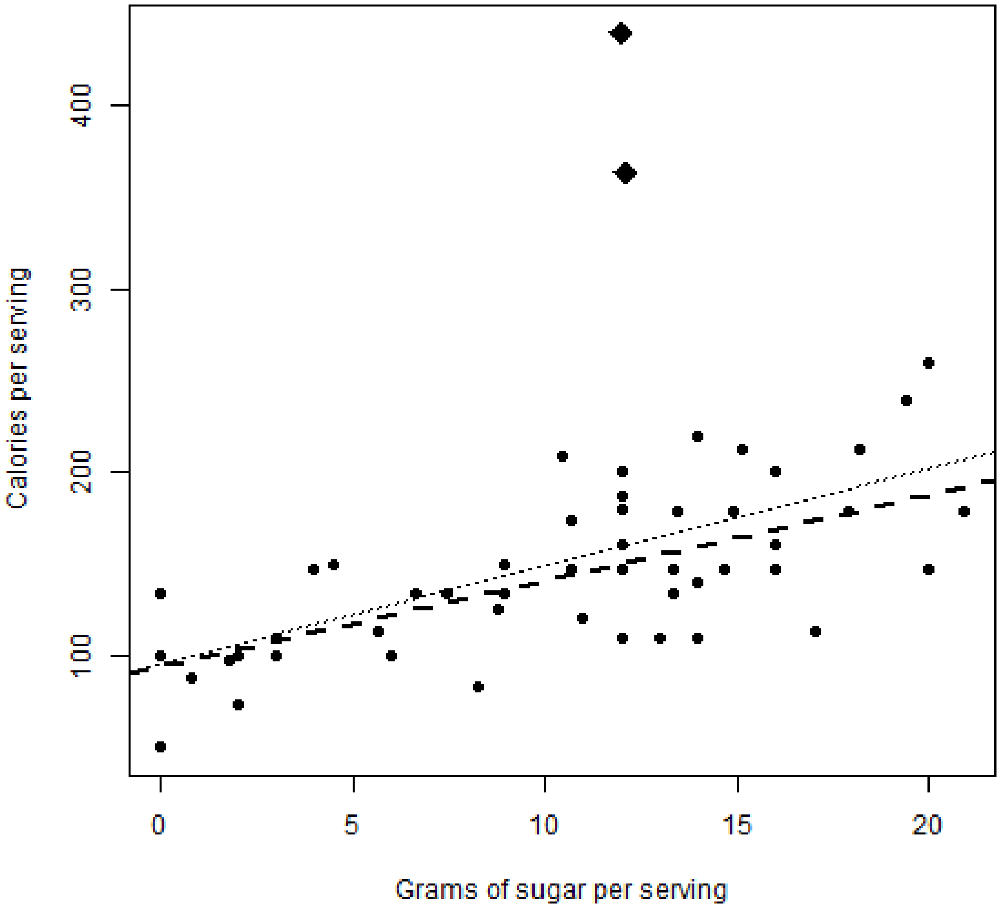
16. Add reference lines with `abline`

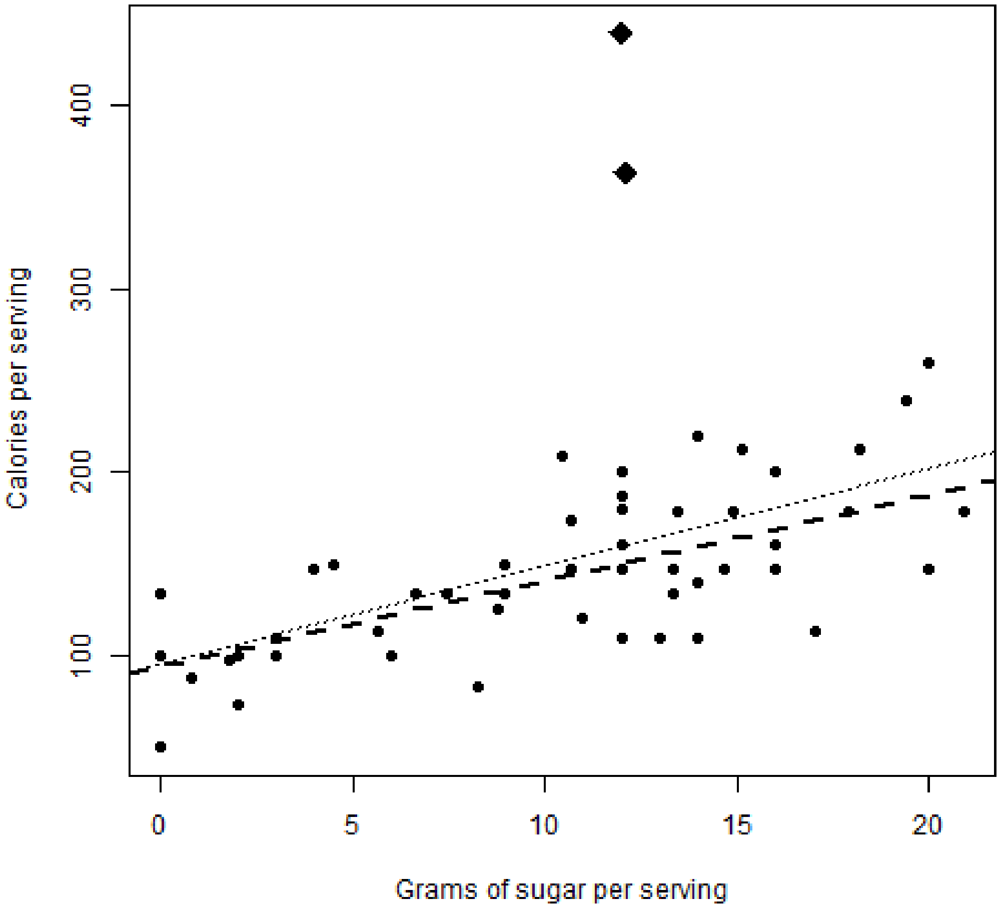
- The final four lines of code add two reference lines using linear regression models.
- Now we're ready to enhance our basic plot.

Listing 16.1:

```
plot(x,y,
     xlab="Grams of sugar per serving",
     ylab="Calories per serving",
     type = "n")
index <- which(y > 300)
points(x[-index], y[-index], pch=16)
points(x[index], y[index], pch=18, cex=2)
olsModel <- lm(y ~ x) # linear regression on y = f(x)
abline(olsModel, lty=3)  # draw thin dotted line
library(robustbase)
robustModel <- lmrob(y ~ x)
abline(robustModel, lty=2, lwd=2)
```


Calories per serving
400
300
200
100
0
5
10
15
20
Grams of sugar per serving


Calories per serving
400
300
200
100
0
5
10
15
20
Grams of sugar per serving



- `lm` is called to construct a linear regression model via the method of ordinary least squares - this is the most popular linear model
- `abline` is called to display the model prediction (based on intercept and slope of the linear function generated). `lty=3` is a dotted line.
- `lmrob` is a robust linear model from the `robustbase` package - it is outlier-resistant. `lty=2, ~lwd=2` gives a thick dashed line
- `[ ]` Do you think `abline` is a generic function?

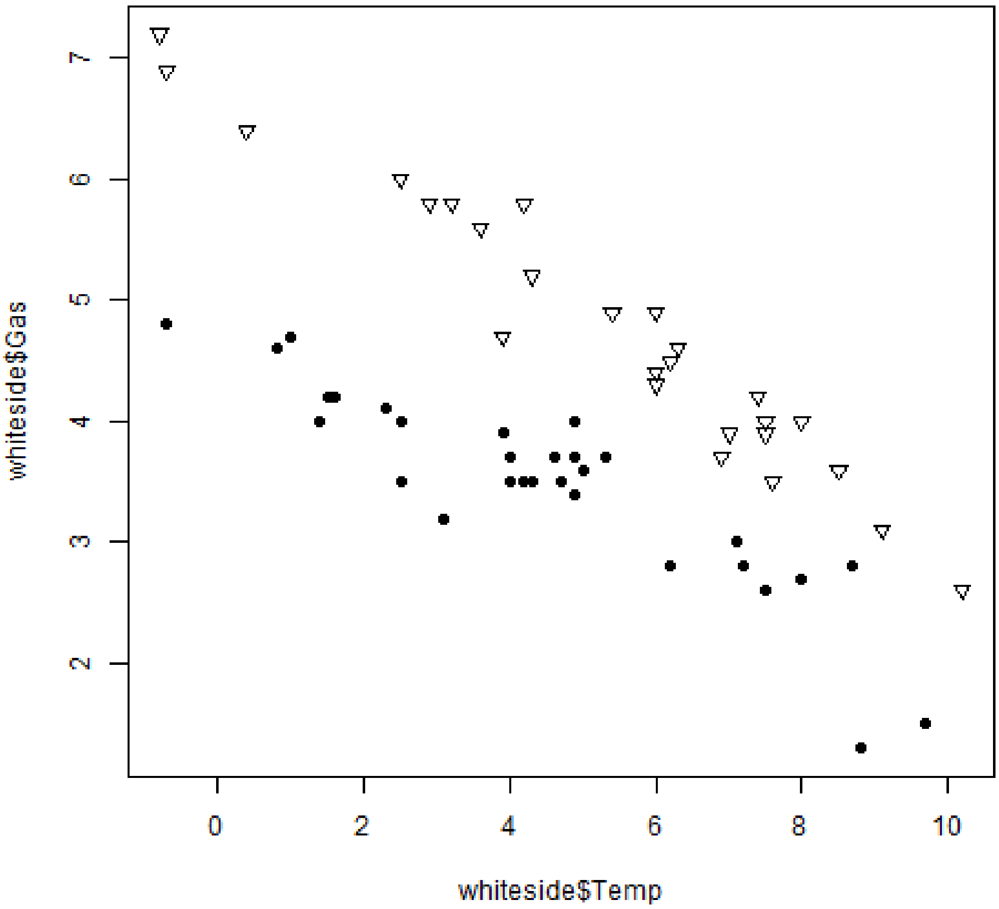
```
methods(abline)
```

```
no methods found
```

17. Customization with vectors

- Consider the following code to plot `whiteside` variables

```
plot(
  x = whiteside$Temp,
  y = whiteside$Gas,
     pch=c(6,16)[whiteside$Insul])
```



- `c(6,16)` defines a 2-dimensional vector of the same length as `Insul` but uses `pch=6` when `Insul = "Before"`, and `pch=16` when `Insul="After"`:

```
 c(6,16)[whiteside$Insul]
```

```
 [1]  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6
 6  6
 [26]  6 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
 16 16
 [51] 16 16 16 16 16 16
```

- The `pch` parameter will accept a vector argument of the same length as the `x`- and `y`-vectors used to create the scatterplot
- Point sizes (`cex`) and colors (`col`) can also be specifed this way

18. Adding text to a plot

- The `text` function works similarly to the other customizing functions
- `text` specifies: x- and y-position of the text, and the text itself.

- Left-right alignment is specified by `adj` whose default is centered (`0.5`). To justify right, we need to set it to `adj=1`
- The code block is growing, so let's look at the `text` addition alone:

```
pointLabels <- paste(rownames(UScereal)[index], "-->")
text(
 x=11,  # x-coordinate of text field, ca. x[which(y>300)]-0.5,
 y=y[index], # y-coordinate of text field
 labels=pointLabels,  # text for text field
 adj=1,  # text is right justified
 family="mono", # use mono spaced font family
 font=1) # exact font depends on font family used
```

1. We define a text label. The `paste` function concatenates vectors after converting to `character`. Here, it concatenates a value of the data frame and the character `-->`

```
paste("Result: ", 100)
class(paste("Result: ", 100))
```

```
[1] "Result:  100"
[1] "character"
```

2. Let's take the expression `rownames(UScereal)[index]` apart[2]: You've seen this indexing structure before, in `par()$lty`.

   `rownames` extracts the names of rows of a data frame:

```
rownames(UScereal)  # all row names of data frame = all cereal types
```

```
[1] "100% Bran"
[2] "All-Bran"
[3] "All-Bran with Extra Fiber"
[4] "Apple Cinnamon Cheerios"
[5] "Apple Jacks"
[6] "Basic 4"
[7] "Bran Chex"
[8] "Bran Flakes"
[9] "Cap'n'Crunch"
[10] "Cheerios"
[11] "Cinnamon Toast Crunch"
[12] "Clusters"
[13] "Cocoa Puffs"
[14] "Corn Chex"
[15] "Corn Flakes"
[16] "Corn Pops"
[17] "Count Chocula"
[18] "Cracklin' Oat Bran"
[19] "Crispix"
[20] "Crispy Wheat & Raisins"
[21] "Double Chex"
[22] "Froot Loops"
[23] "Frosted Flakes"
[24] "Frosted Mini-Wheats"
[25] "Fruit & Fibre: Dates Walnuts and Oats"
[26] "Fruitful Bran"
[27] "Fruity Pebbles"
[28] "Golden Crisp"
```

[2] The need to deconstruct, as it were, complex functional expressions in R, is similar to getting a mathematical formula and having to take it apart using your knowledge of the laws of mathematics. If you don't enjoy this type of thing you probably won't study math (or you have to learn a lot more things by heart).

```
[29] "Golden Grahams"
[30] "Grape Nuts Flakes"
[31] "Grape-Nuts"
[32] "Great Grains Pecan"
[33] "Honey Graham Ohs"
[34] "Honey Nut Cheerios"
[35] "Honey-comb"
[36] "Just Right Fruit & Nut"
[37] "Kix"
[38] "Life"
[39] "Lucky Charms"
[40] "Mueslix Crispy Blend"
[41] "Multi-Grain Cheerios"
[42] "Nut&Honey Crunch"
[43] "Nutri-Grain Almond-Raisin"
[44] "Oatmeal Raisin Crisp"
[45] "Post Nat. Raisin Bran"
[46] "Product 19"
[47] "Puffed Rice"
[48] "Quaker Oat Squares"
[49] "Raisin Bran"
[50] "Raisin Nut Bran"
[51] "Raisin Squares"
[52] "Rice Chex"
[53] "Rice Krispies"
[54] "Shredded Wheat 'n'Bran"
[55] "Shredded Wheat spoon size"
[56] "Smacks"
[57] "Special K"
[58] "Total Corn Flakes"
[59] "Total Raisin Bran"
[60] "Total Whole Grain"
[61] "Triples"
[62] "Trix"
[63] "Wheat Chex"
[64] "Wheaties"
[65] "Wheaties Honey Gold"
```

`index` was defined earlier as a subset of `UScereal$calories` values greater than the outlier cutoff value 300: `index <- which(y > 300)`

Hence, `rownames(UScereal)[index]` extracts the cereal names associated with the outlying values and puts them in the text box:

```
rownames(UScereal)[index]
```

```
[1] "Grape-Nuts"          "Great Grains Pecan"
```

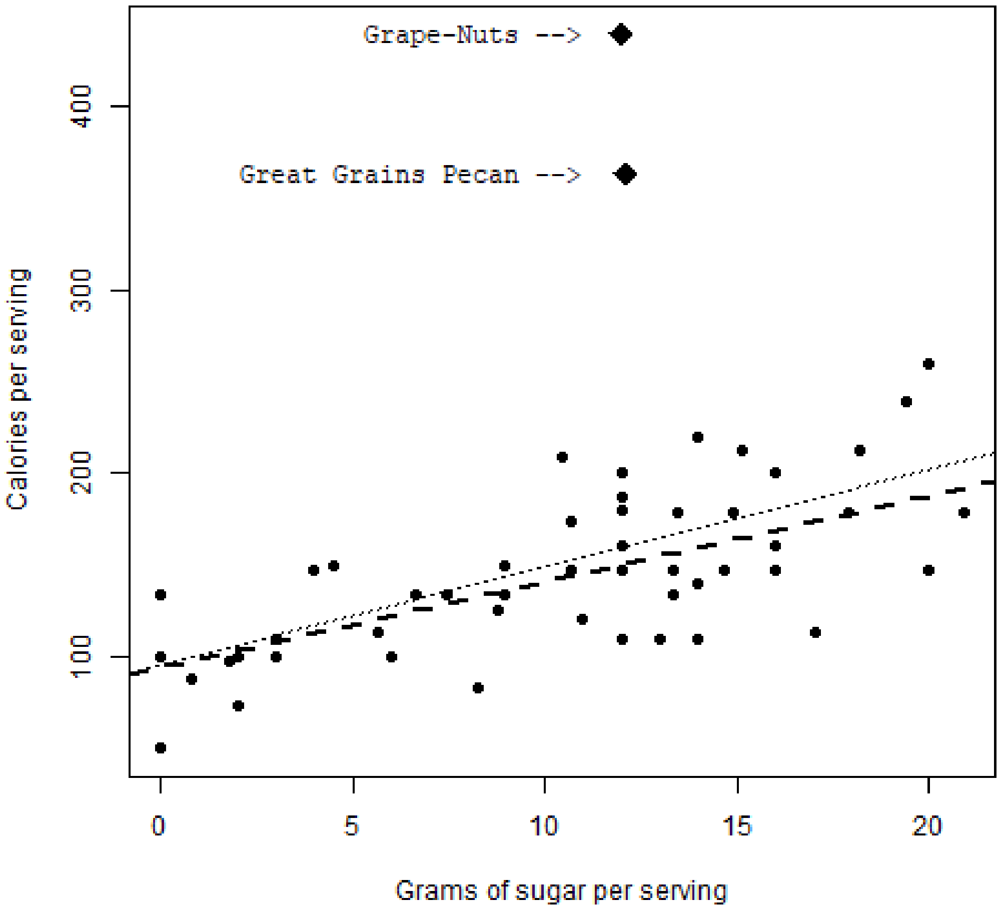
3. Inside the `text` function, we have three sections:
   - coordinates x and y for the text field[3]
   - labels, namely the text to be printed in the plot
   - formatting parameters like text justification, font type etc.

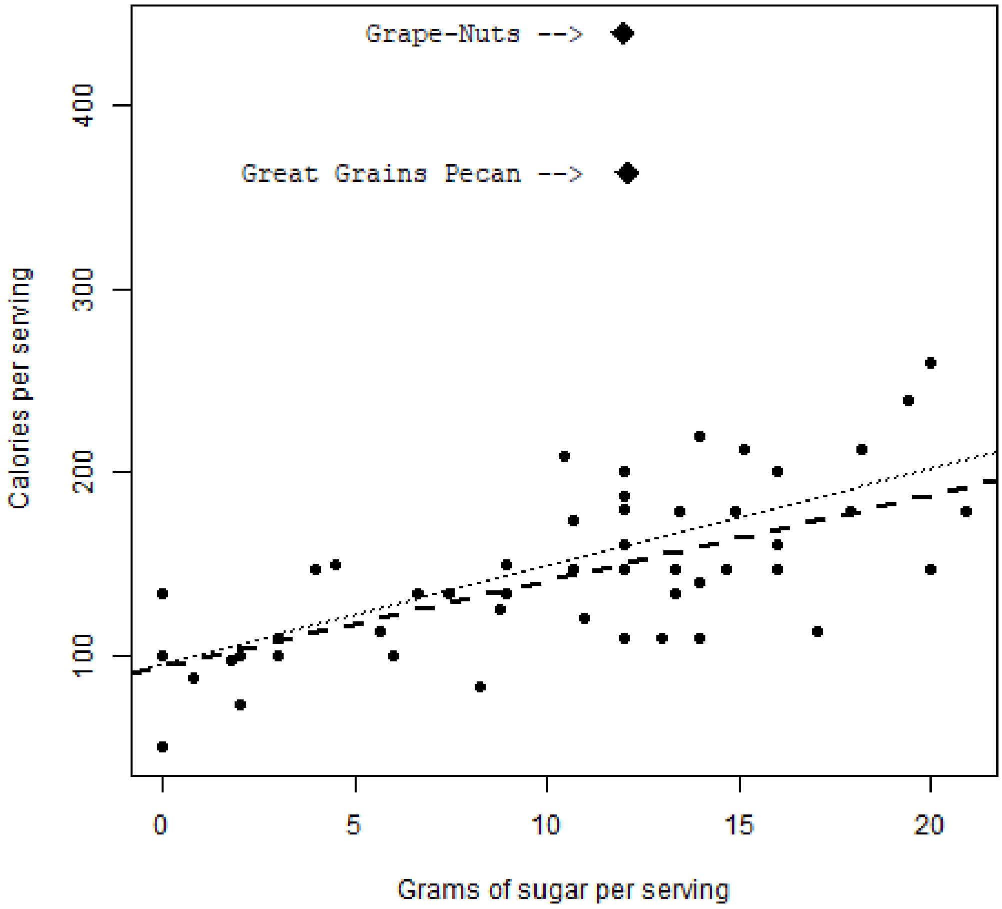
- The complete R code block for printout looks like this now:

[3] It would be better not to have to look at the plot to determine the place of the text box, e.g. with the expression `x[which(y>300)]-0.5` or `(x[which(y>300)[2]]-x[which(y>300)[1]])`, which returns the x-position that belongs to the y-value outlier.

Listing 18.1:

```r
plot(x,y,
     xlab="Grams of sugar per serving",
     ylab="Calories per serving",
     type = "n")
index <- which(y > 300)
points(x[-index], y[-index], pch=16)
points(x[index], y[index], pch=18, cex=2)
olsModel <- lm(y ~ x) # linear regression on y = f(x)
abline(olsModel, lty=3)  # draw thin dotted line
library(robustbase)
robustModel <- lmrob(y ~ x)
abline(robustModel, lty=2, lwd=2)
pointLabels <- paste(rownames(UScereal)[index], "-->")
text(
  x=11,
  y=y[index],
  labels=pointLabels,
  adj=1,
  family="mono",
  font=1)
```


Grape-Nuts -->
Great Grains Pecan -->
400
300
200
100
Calories per serving
0
5
10
15
20
Grams of sugar per serving



19. Adding a legend to a plot

–>> IF YOU MISSED THE LAST 1-2 SESSIONS DOWNLOAD THIS: https://tinyurl.com/55p5yw6f <<–

- The `legend` function adds a boxed explanatory text display
- It can be used like the `text` function: box location and text
- It has many more optional parameters (check out `help(legend)`)

```
legend(x, y = NULL, legend, fill = NULL, col = par("col"),
       border = "black", lty, lwd, pch,
       angle = 45, density = NULL, bty = "o", bg = par("bg"),
       box.lwd = par("lwd"), box.lty = par("lty"), box.col = par("fg"),
       pt.bg = NA, cex = 1, pt.cex = cex, pt.lwd = lwd,
       xjust = 0, yjust = 1, x.intersp = 1, y.intersp = 1,
       adj = c(0, 0.5), text.width = NULL, text.col = par("col"),
       text.font = NULL, merge = do.lines && has.pch, trace = FALSE,
       plot = TRUE, ncol = 1, horiz = FALSE, title = NULL,
       inset = 0, xpd, title.col = text.col, title.adj = 0.5,
       seg.len = 2)
```

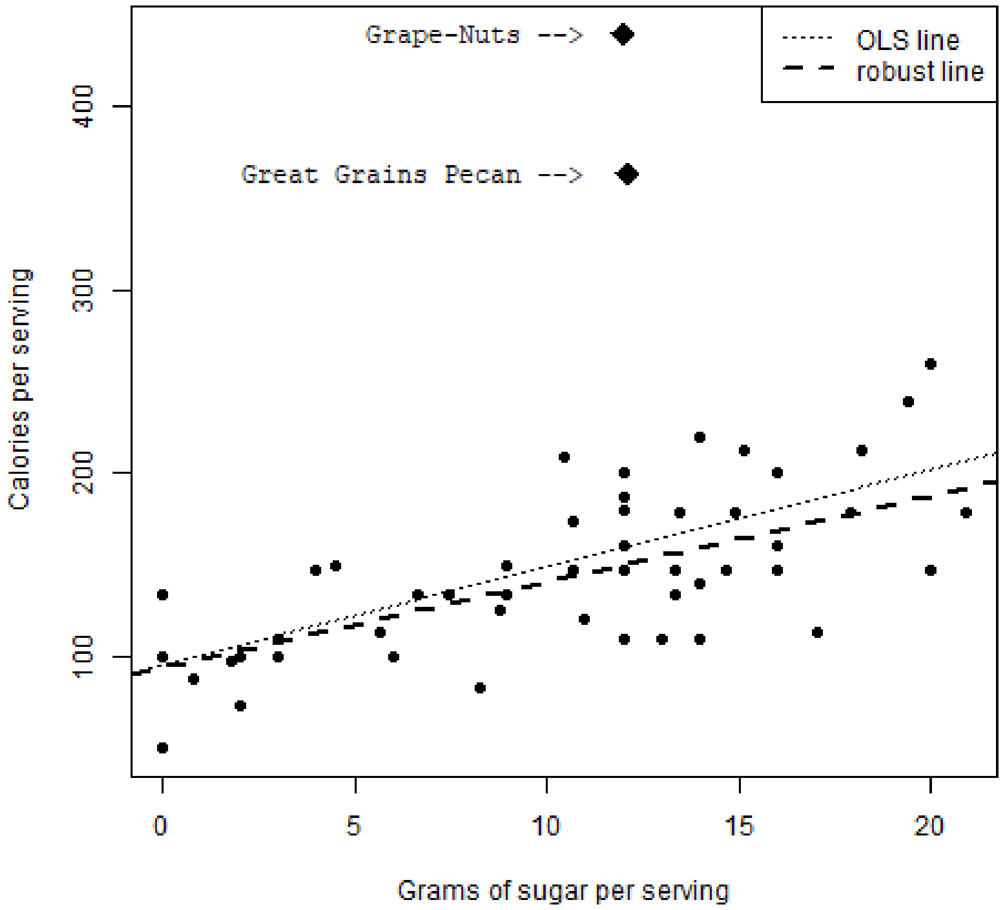
- Let's look at the code fragment for the legend only:

```
legend(
x = "topright",   # set location in plot
legend = c("OLS line", "robust line")) # print legend labels
lty = c(3,2),    # set line type for 2 legend items
lwd = c(1,2))   # set line width for 2 legend items
```

1. The first (x) argument sets the location in the plot
2. The `legend` parameter prints the legend labels (text)
3. The next parameters link the legend to the plot:
   - `lty=3`, `lwd=1` is the dotted thin upper line (OLS fit)
   - `lty=2`, `lwd=2` is the dashed thick lower line (robust fit)

- Our final R code block:

```
plot(x,y,
     xlab="Grams of sugar per serving",
     ylab="Calories per serving",
     type = "n")
index <- which(y > 300)
points(x[-index], y[-index], pch=16)
points(x[index], y[index], pch=18, cex=2)
olsModel <- lm(y ~ x) # linear regression on y = f(x)
abline(olsModel, lty=3)  # draw thin dotted line
library(robustbase)
robustModel <- lmrob(y ~ x)
abline(robustModel, lty=2, lwd=2)
pointLabels <- paste(rownames(UScereal)[index], "-->")
text(
  x=11,
  y=y[index],
  labels=pointLabels,
  adj=1,
  family="mono",
  font=1)
legend(
  x = "topright",
  lty = c(3,2),
  lwd = c(1,2),
  legend = c("OLS line", "robust line"))
```

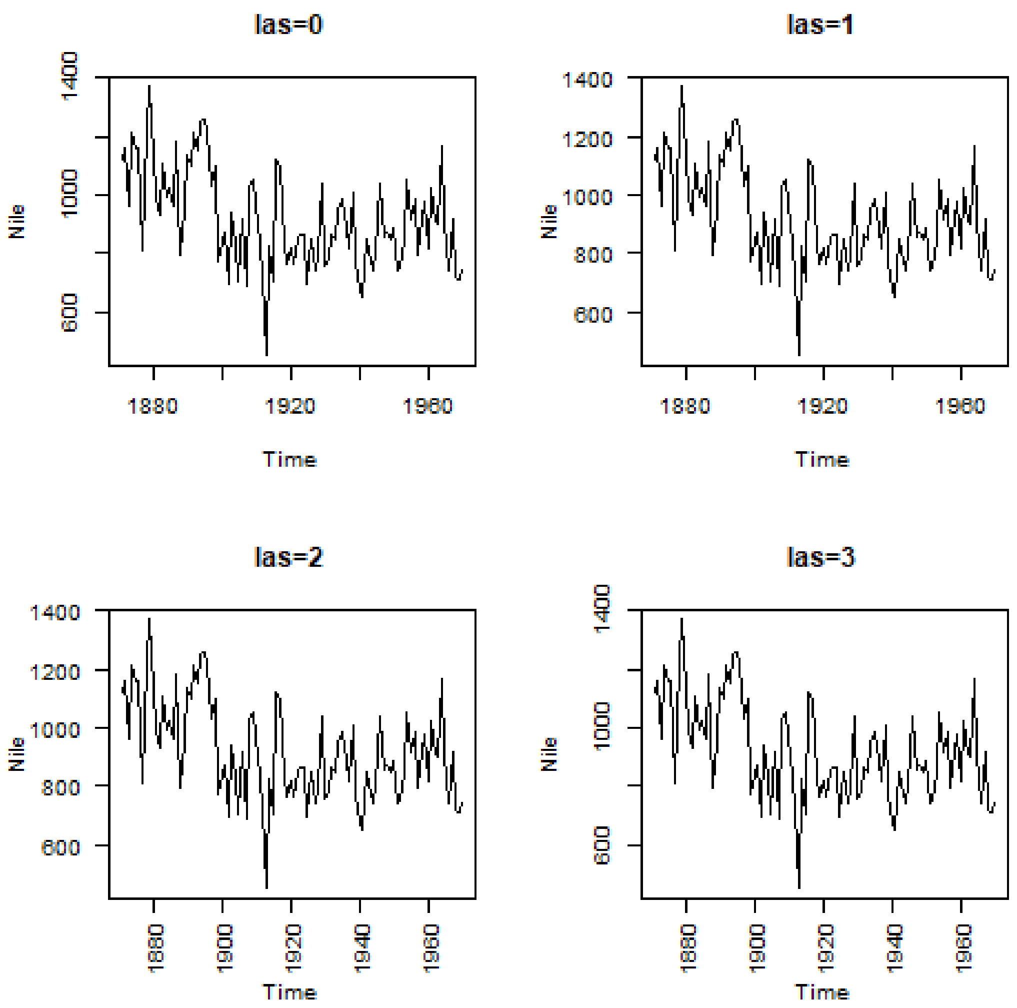


20. Customizing axes

- You already know about labelling axes with `xlab` and `ylab`
- In addition, the limits of the axes can be set with `xlim` and `ylim`
- The `par` function offers additional parameters like `las` and `side`
- `las` specifies the orientation of the axis labels:
  1. `las=0` : labels are displayed axis-parallel (default)
  2. `las=1` : labels are always horizontal
  3. `las=2` : labels are always perpendicular to the axis
  4. `las=3` : labels are always vertical
- To aid readability, you may also have to adjust `cex.lab`
- Example:

```
par(las=0, mfrow=c(2,2))    # labels are axis parallel
plot(Nile, main="las=0")
```

```
par(las=1)                     # labels horizontal
plot(Nile, main="las=1")
par(las=2)                     # labels axis perpendicular
plot(Nile, main="las=2")
par(las=3)                     # labels vertical
plot(Nile, main="las=3")
```



21. Specifying your own axes

- You can specify your own axes in two steps:

  1. execute the base graphics function with `axes = FALSE`
  2. Use the `axis` function to specify your own axes

- The default axes are now suppressed and you need to set:

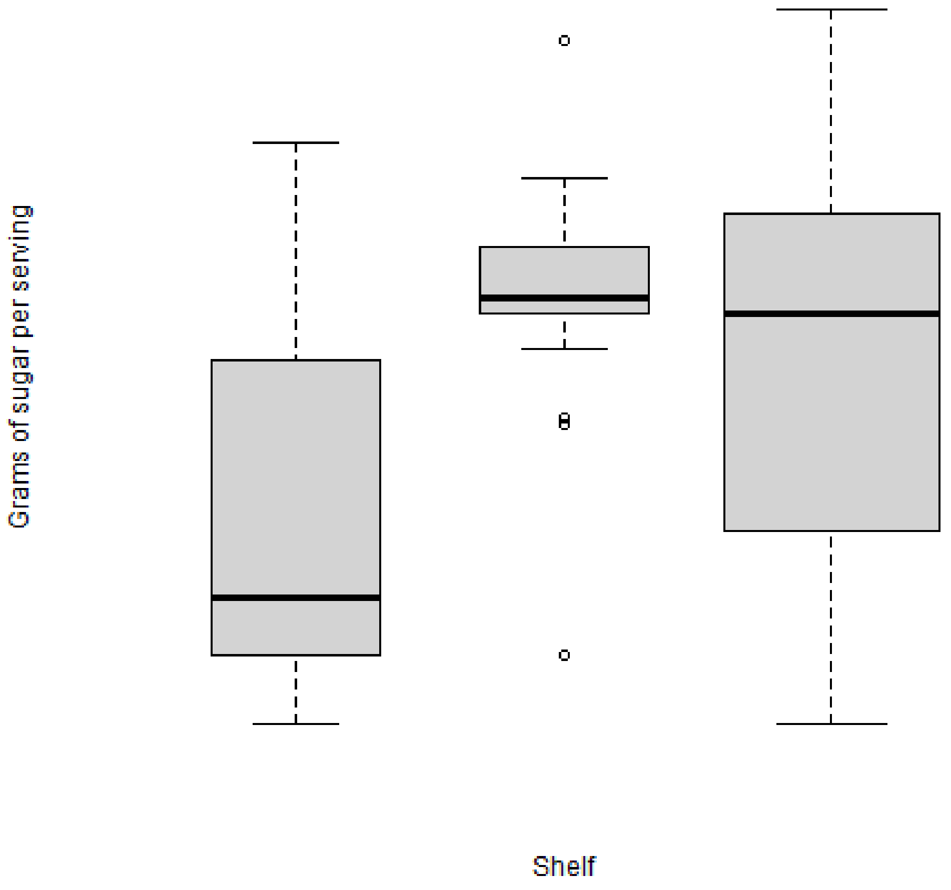
  1. `side = 1` : creates (default) lower x-axis below the plot
  2. `side = 2` : creates (default) y-axis left of the plot
  3. `side = 3` : creates upper x-axis above the plot
  4. `side = 4` : creates a y-axis right of the plot

- Example: boxplot of the range of `sugars` values for each of the three levels of `shelf` value in the `UScereal` dataframe:

  1. `shelf = 1` is at the floor level
  2. `shelf = 2` is the middle shelf (kid-eye-level)
  3. `shelf = 3` is on the top shelf (adult-eye-level)

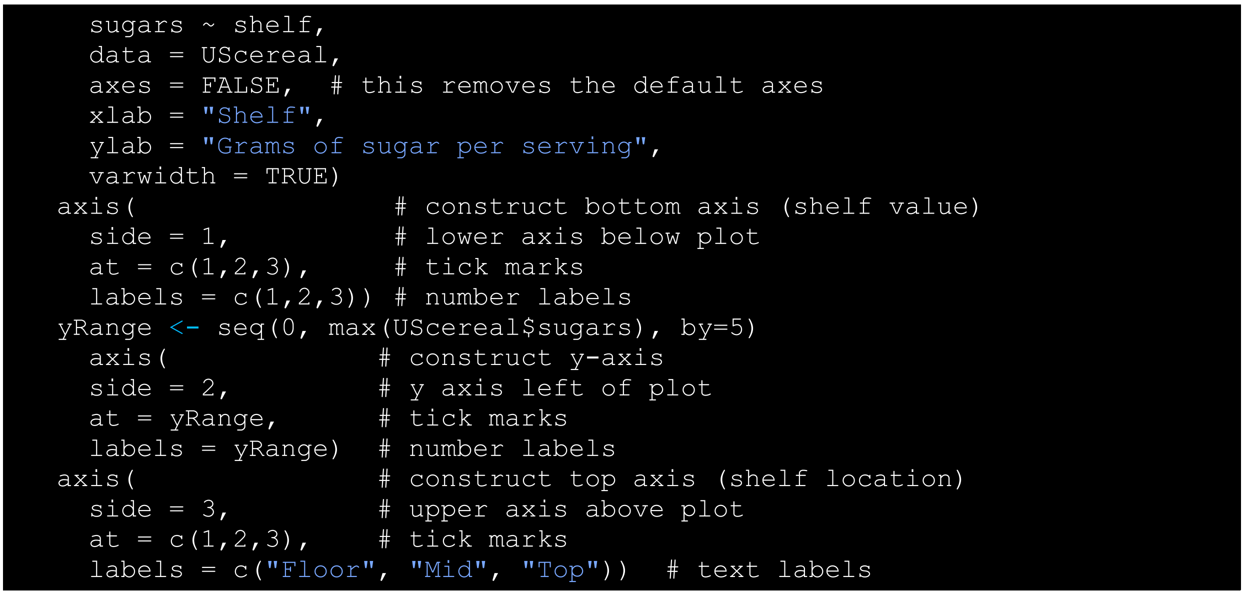
- Box plot without axes: The `varwidth` parameter creates a boxplot of variable width so that the width of each individual boxplot reflects the number of different cereals on each shelf.

```
boxplot(
  sugars ~ shelf,
  data = UScereal,
  axes = FALSE,  # this removes the default axes
  xlab = "Shelf",
  ylab = "Grams of sugar per serving",
  varwidth = TRUE)
```

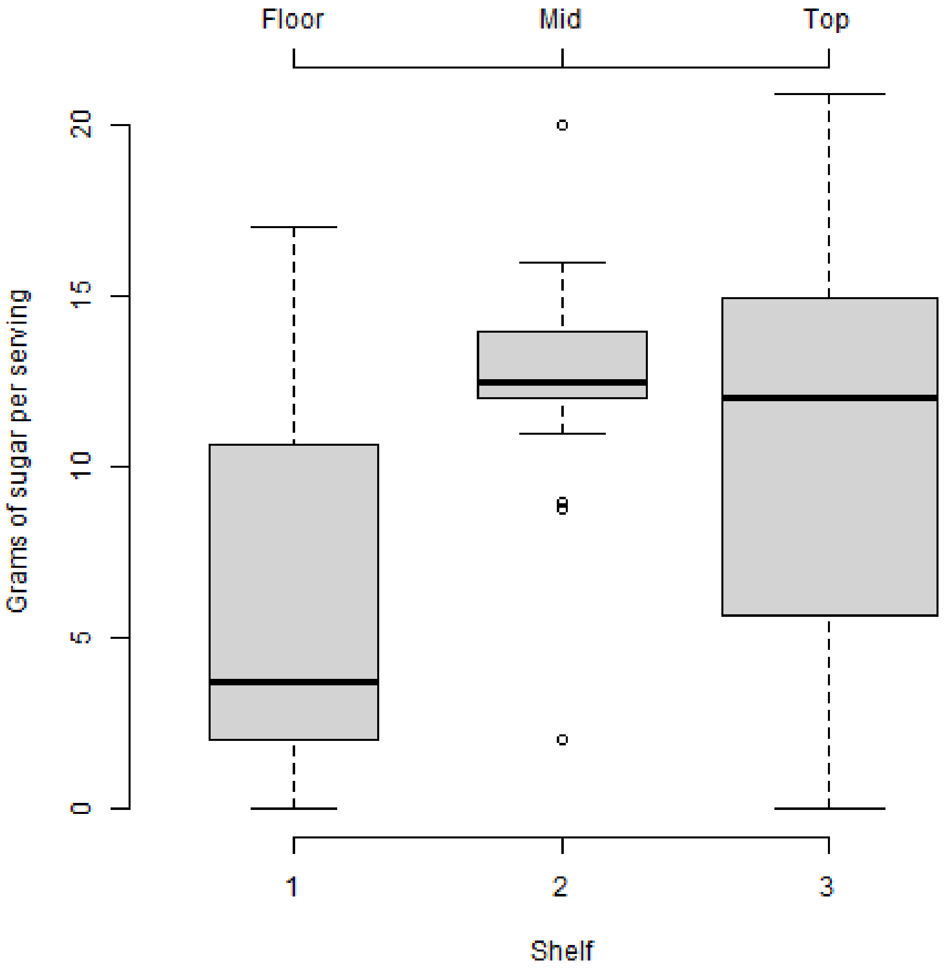


- Box plot with axis specified using `axis`:

```
boxplot(
```



```
  sugars ~ shelf,
  data = UScereal,
  axes = FALSE,  # this removes the default axes
  xlab = "Shelf",
  ylab = "Grams of sugar per serving",
  varwidth = TRUE)
axis(                # construct bottom axis (shelf value)
  side = 1,          # lower axis below plot
  at = c(1,2,3),     # tick marks
  labels = c(1,2,3)) # number labels
yRange <- seq(0, max(UScereal$sugars), by=5)
  axis(              # construct y-axis
  side = 2,          # y axis left of plot
  at = yRange,       # tick marks
  labels = yRange)   # number labels
axis(                # construct top axis (shelf location)
  side = 3,          # upper axis above plot
  at = c(1,2,3),     # tick marks
  labels = c("Floor", "Mid", "Top"))  # text labels
```



22. Lab session: adding details to plots

- Open the raw practice file in GitHub
- Identify yourself as the author and pledge
- Solve the problems using R code blocks
- Submit the completed file to Canvas