R for Biologists - An Introduction to R (Lecture 6)

Classes, generic functions and object orientation

The class of an object determines how it will be treated by what are known as generic functions. Put the other way round, a generic function performs a task or action on its arguments specific to the class of the argument itself. If the argument lacks any class attribute, or has a class not catered for specifically by the generic function in question, there is always a default action provided.

An example makes things clearer. The class mechanism offers the user the facility of designing and writing generic functions for special purposes. Among the other generic functions are `plot()` for displaying objects graphically, `summary()` for summarizing analyses of various types, and `anova()` for comparing statistical models.

The number of generic functions that can treat a class in a specific way can be quite large. For example, the functions that can accommodate in some fashion objects of class "data.frame" include

```
 [     [[<-    any    as.matrix
 [<-   mean    plot   summary
```

A currently complete list can be got by using the methods() function:

```
methods(class="data.frame")
```

Conversely the number of classes a generic function can handle can also be quite large. For example the plot() function has a default method and variants for objects of classes "data.frame", "density", "factor", and more. A complete list can be got again by using the methods() function:

```
methods(plot)
```

```
##  [1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
##  [4] plot.default        plot.dendrogram*    plot.density*
##  [7] plot.ecdf           plot.factor*        plot.formula*
## [10] plot.function       plot.hclust*        plot.histogram*
## [13] plot.HoltWinters*   plot.isoreg*        plot.lm*
## [16] plot.medpolish*     plot.mlm*           plot.ppr*
## [19] plot.prcomp*        plot.princomp*      plot.profile.nls*
## [22] plot.raster*        plot.spec*          plot.stepfun
## [25] plot.stl*           plot.table*         plot.ts
## [28] plot.tskernel*      plot.TukeyHSD*
## see '?methods' for accessing help and source code
```

For many generic functions the function body is quite short, for example

```
coef
```

```
## function (object, ...)
## UseMethod("coef")
## <bytecode: 0x7f8a932240a0>
## <environment: namespace:stats>
```

The presence of UseMethod indicates this is a generic function. To see what methods are available we can use methods()

```r
methods(coef)
```

```
## [1] coef.aov*     coef.Arima*   coef.default* coef.listof*  coef.nls*
## see '?methods' for accessing help and source code
```

In this example there are six methods, none of which can be seen by typing its name. We can read these by either of

```r
getAnywhere("coef.aov")
```

```
## A single object matching 'coef.aov' was found
## It was found in the following places
##   registered S3 method for coef from namespace stats
##   namespace:stats
## with value
##
## function (object, ...)
## {
##     z <- object$coefficients
##     z[!is.na(z)]
## }
## <bytecode: 0x7f8a950f6cd8>
## <environment: namespace:stats>
```

```r
getS3method("coef", "aov")
```

```
## function (object, ...)
## {
##     z <- object$coefficients
##     z[!is.na(z)]
## }
## <bytecode: 0x7f8a950f6cd8>
## <environment: namespace:stats>
```

S4 classes for Object Orientation

S4 classes and methods are implemented in the `methods` package. The core components: classes, generic functions and methods.

Classes

The class defines the structure of an object. The instances of that class represent the objects themselves. Classes can provide an abstraction of complex objects that helps to simplify programming with them. Objects inside of a class are termed slots. A class can extend one or more other classes. The extended classes are considered to be parents of the class that extends them. Extension means that the new class will contain all of the slots that the parent classes have. It is an error to create a class with duplicate slot names (either by inclusion or directly). Classes are something designed by programmers and implemented in packages.

classes can be generated:

From a dataset

```
source("https://bioconductor.org/biocLite.R")
```

```
## Bioconductor version 3.1 (BiocInstaller 1.18.5), ?biocLite for help

## A newer version of Bioconductor is available for this version of R,
##   ?BiocUpgrade for help
```

```
biocLite("graph")
```

```
## BioC_mirror: http://bioconductor.org

## Using Bioconductor version 3.1 (BiocInstaller 1.18.5), R version 3.2.3.

## Installing package(s) 'graph'

##
## The downloaded binary packages are in
##  /var/folders/vv/3xsj6xdd3j7828czt77bcltr0000gn/T//RtmpErDavn/downloaded_packages

## Old packages: 'boot', 'curl', 'devtools', 'dynamicTreeCut', 'evaluate',
##   'formatR', 'ggplot2', 'git2r', 'gridExtra', 'gtable', 'Hmisc',
##   'htmlwidgets', 'latticeExtra', 'leaflet', 'maps', 'Matrix', 'memoise',
##   'mgcv', 'munsell', 'nlme', 'nnet', 'openssl', 'permute', 'scales',
##   'shiny', 'sp', 'vegan', 'WGCNA', 'XML', 'xtable'
```

```
library(graph)
data(apopGraph)
apopGraph
```

```
## A graphNEL graph with directed edges
## Number of Nodes = 50
## Number of Edges = 59
```

you can retrieve the class of an object by using the function `class`

```
class(apopGraph)
```

```
## [1] "graphNEL"
## attr(,"package")
## [1] "graph"
```

From using the constructor

```
library(IRanges)
```

```
## Loading required package: BiocGenerics

## Loading required package: parallel
```

```
##
## Attaching package: 'BiocGenerics'

## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB

## The following object is masked from 'package:stats':
##
##     xtabs

## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, as.vector, cbind,
##     colnames, do.call, duplicated, eval, evalq, Filter, Find, get,
##     intersect, is.unsorted, lapply, Map, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,
##     rbind, Reduce, rep.int, rownames, sapply, setdiff, sort,
##     table, tapply, union, unique, unlist, unsplit

## Loading required package: S4Vectors

## Loading required package: stats4

## Creating a generic function for 'nchar' from package 'base' in package 'S4Vectors'
```

```
IRanges(start=c(101, 25), end=c(110, 80))
```

```
## IRanges of length 2
##     start end width
## [1]   101 110    10
## [2]    25  80    56
```

From a coercion

```
library(Matrix)
```

```
##
## Attaching package: 'Matrix'

## The following object is masked from 'package:IRanges':
##
##     expand
```

```
m <- matrix(3:-4, nrow=2)
class(m)
```

```
## [1] "matrix"
```

```r
as(m, "Matrix")
```

```
## 2 x 4 Matrix of class "dgeMatrix"
##      [,1] [,2] [,3] [,4]
## [1,]    3    1   -1   -3
## [2,]    2    0   -2   -4
```

From using a specialized high-level constructor

```r
source("https://bioconductor.org/biocLite.R")
```

```
## Bioconductor version 3.1 (BiocInstaller 1.18.5), ?biocLite for help

## A newer version of Bioconductor is available for this version of R,
##   ?BiocUpgrade for help
```

```r
biocLite("GenomicFeatures")
```

```
## BioC_mirror: http://bioconductor.org

## Using Bioconductor version 3.1 (BiocInstaller 1.18.5), R version 3.2.3.

## Installing package(s) 'GenomicFeatures'

##
## The downloaded binary packages are in
##  /var/folders/vv/3xsj6xdd3j7828czt77bcltr0000gn/T//RtmpErDavn/downloaded_packages

## Old packages: 'boot', 'curl', 'devtools', 'dynamicTreeCut', 'evaluate',
##   'formatR', 'ggplot2', 'git2r', 'gridExtra', 'gtable', 'Hmisc',
##   'htmlwidgets', 'latticeExtra', 'leaflet', 'maps', 'Matrix', 'memoise',
##   'mgcv', 'munsell', 'nlme', 'nnet', 'openssl', 'permute', 'scales',
##   'shiny', 'sp', 'vegan', 'WGCNA', 'XML', 'xtable'
```

```r
library(GenomicFeatures)
```

```
## Loading required package: GenomeInfoDb

## Loading required package: GenomicRanges

## Loading required package: AnnotationDbi

## Loading required package: Biobase

## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
```

```
makeTranscriptDbFromUCSC("sacCer2", tablename="ensGene")
```

```
## Warning: 'makeTranscriptDbFromUCSC' is deprecated.
## Use 'makeTxDbFromUCSC' instead.
## See help("Deprecated")

## Download the ensGene table ...

## OK

## Extract the 'transcripts' data frame ...

## OK

## Extract the 'splicings' data frame ...

## OK

## Download and preprocess the 'chrominfo' data frame ...

## OK

## Prepare the 'metadata' data frame ...

## OK

## Make the TxDb object ...

## OK

## TxDb object:
## # Db type: TxDb
## # Supporting package: GenomicFeatures
## # Data source: UCSC
## # Genome: sacCer2
## # Organism: Saccharomyces cerevisiae
## # UCSC Table: ensGene
## # Resource URL: http://genome.ucsc.edu/
## # Type of Gene ID: Ensembl gene ID
## # Full dataset: yes
## # miRBase build ID: NA
## # transcript_nrow: 7130
## # exon_nrow: 7535
## # cds_nrow: 7038
## # Db created by: GenomicFeatures package from Bioconductor
## # Creation time: 2016-03-21 11:41:42 -0700 (Mon, 21 Mar 2016)
## # GenomicFeatures version at creation time: 1.20.6
## # RSQLite version at creation time: 1.0.0
## # DBSCHEMAVERSION: 1.1
```

From a high level I/O function

```
library(ShortRead)
```

```
## Loading required package: BiocParallel
```

```
## Loading required package: Biostrings
```

```
## Loading required package: XVector
```

```
##
## Attaching package: 'Biostrings'
```

```
## The following object is masked from 'package:graph':
##
##     complement
```

```
## Loading required package: Rsamtools
```

```
## Loading required package: GenomicAlignments
```

```
lane1 <- readFastq(system.file("extdata","s_1_sequence.txt",package="Biostrings"))
lane1
```

```
## class: ShortReadQ
## length: 256 reads; width: 36 cycles
```

What ojects are stored in a class can be seen by using the fuction `slotNames`, and these objects can be accessed using the accessor operator `@`, however you should never need to use the accessor operator, but instead use the programmer provided acceesor functions.

```
slotNames(lane1)
```

```
## [1] "quality" "sread"   "id"
```

```
lane1@sread
```

```
##   A DNAStringSet instance of length 256
##       width seq
##   [1]    36 GGACTTTGTAGGATACCCTCGCTTTCCTTCTCCTGT
##   [2]    36 GATTTCTTACCTATTAGTGGTTGAACAGCATCGGAC
##   [3]    36 GCGGTGGTCTATAGTGTTATTAATATCAATTTGGGT
##   [4]    36 GTTACCATGATGTTATTTCTTCATTTGGAGGTAAAA
##   [5]    36 GTATGTTTCTCCTGCTTATCACCTTCTTGAAGGCTT
##   ...   ... ...
## [252]    36 GTTTAGATATGAGTCACATTTTGTTCATGGTAGAGT
## [253]    36 GTTTTACAGACACCTAAAGCTACATCGTCAACGTTA
## [254]    36 GATGAACTAAGTCAACCTCAGCACTAACCTTGCGAG
## [255]    36 GTTTGGTTCGCTTTGAGTCTTCTTCGGTTCCGACTA
## [256]    36 GCAATCTGCCGACCACTCGCGATTCAATCATGACTT
```

```
sread(lane1)
```

```
##   A DNAStringSet instance of length 256
##       width seq
##   [1]    36 GGACTTTGTAGGATACCCTCGCTTTCCTTCTCCTGT
##   [2]    36 GATTTCTTACCTATTAGTGGTTGAACAGCATCGGAC
##   [3]    36 GCGGTGGTCTATAGTGTTATTAATATCAATTTGGGT
##   [4]    36 GTTACCATGATGTTATTTCTTCATTTGGAGGTAAAA
##   [5]    36 GTATGTTTCTCCTGCTTATCACCTTCTTGAAGGCTT
##   ...   ... ...
## [252]    36 GTTTAGATATGAGTCACATTTTGTTCATGGTAGAGT
## [253]    36 GTTTTACAGACACCTAAAGCTACATCGTCAACGTTA
## [254]    36 GATGAACTAAGTCAACCTCAGCACTAACCTTGCGAG
## [255]    36 GTTTGGTTCGCTTTGAGTCTTCTTCGGTTCCGACTA
## [256]    36 GCAATCTGCCGACCACTCGCGATTCAATCATGACTT
```

Low-level: getter and setter functions (accessor and assignments)

Typically class developers will define accessor functions for each class slot (unless they want to keep it hidden) and for those slots that are mutable assignment functions.

```
ir <- IRanges(start=c(101, 25), end=c(110, 80))
ir
```

```
## IRanges of length 2
##     start end width
## [1]   101 110    10
## [2]    25  80    56
```

```
width(ir)
```

```
## [1] 10 56
```

```
width(ir) <- width(ir) - 5
ir
```

```
## IRanges of length 2
##     start end width
## [1]   101 105     5
## [2]    25  75    51
```

```
names(ir) <- c("range1","range2")
ir
```

```
## IRanges of length 2
##     start end width  names
## [1]   101 105     5 range1
## [2]    25  75    51 range2
```

High-level: plenty of specialized methods

```
qa1 <- qa(lane1, lane="lane1")
class(qa1)
```

```
## [1] "ShortReadQQA"
## attr(,"package")
## [1] "ShortRead"
```

```
report(qa1)
```

```
## [1] "/var/folders/vv/3xsj6xdd3j7828czt77bcltr0000gn/T//RtmpErDavn/file487751c6fd6b/index.html"
```

Graphical Proceedures

Graphical facilities are an important and extremely versatile component of the R environment. It is possible to use the facilities to display a wide variety of statistical graphs and also to build entirely new types of graph.

At startup time R initiates a graphics device driver which opens a special graphics window for the display of interactive graphics. Although this is done automatically, it is useful to know that the command used is X11() under UNIX, windows() under Windows and quartz() under Mac OS X.

Once the device driver is running, R plotting commands can be used to produce a variety of graphical displays and to create entirely new kinds of display.

Plotting commands are divided into three basic groups:

- High-level plotting functions create a new plot on the graphics device, possibly with axes, labels, titles and so on.
- Low-level plotting functions add more information to an existing plot, such as extra points, lines and labels.
- Interactive graphics functions allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse. In addition, R maintains a list of graphical parameters which can be manipulated to customize your plots.

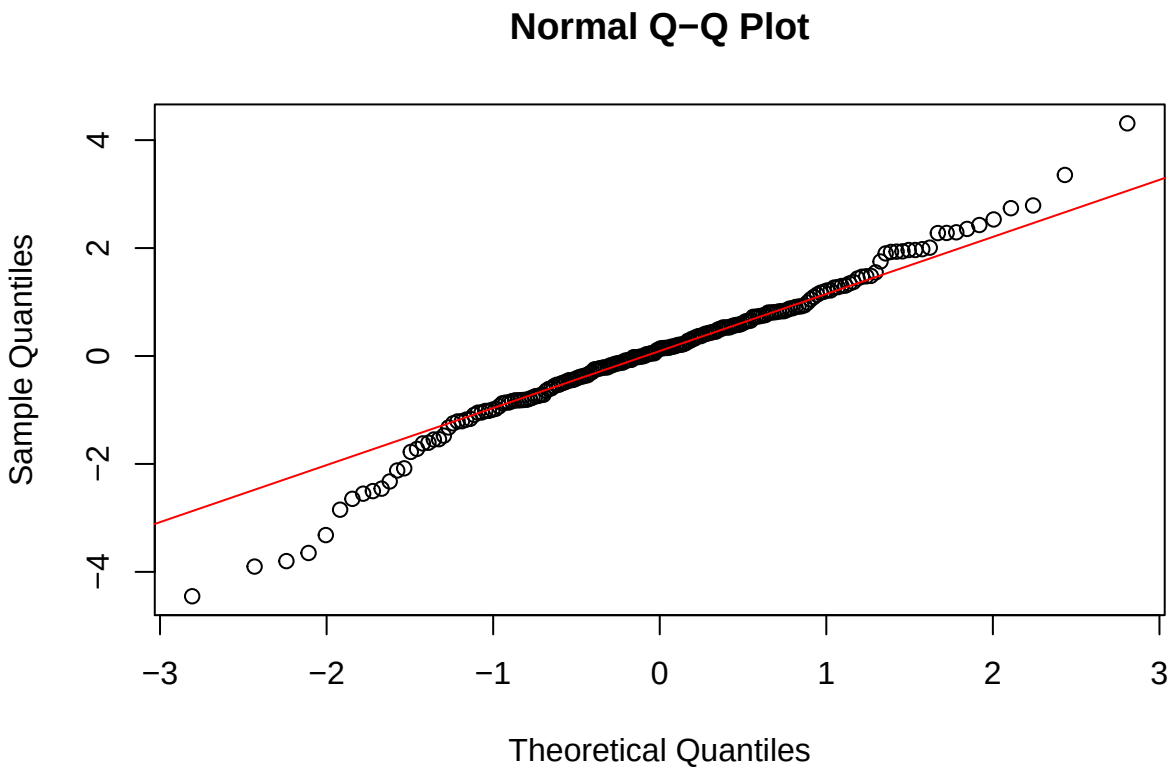
High-level plotting commands

High-level plotting functions are designed to generate a complete plot of the data passed as arguments to the function. Where appropriate, axes, labels and titles are automatically generated (unless you request otherwise.) High-level plotting commands always start a new plot, erasing the current plot if necessary.

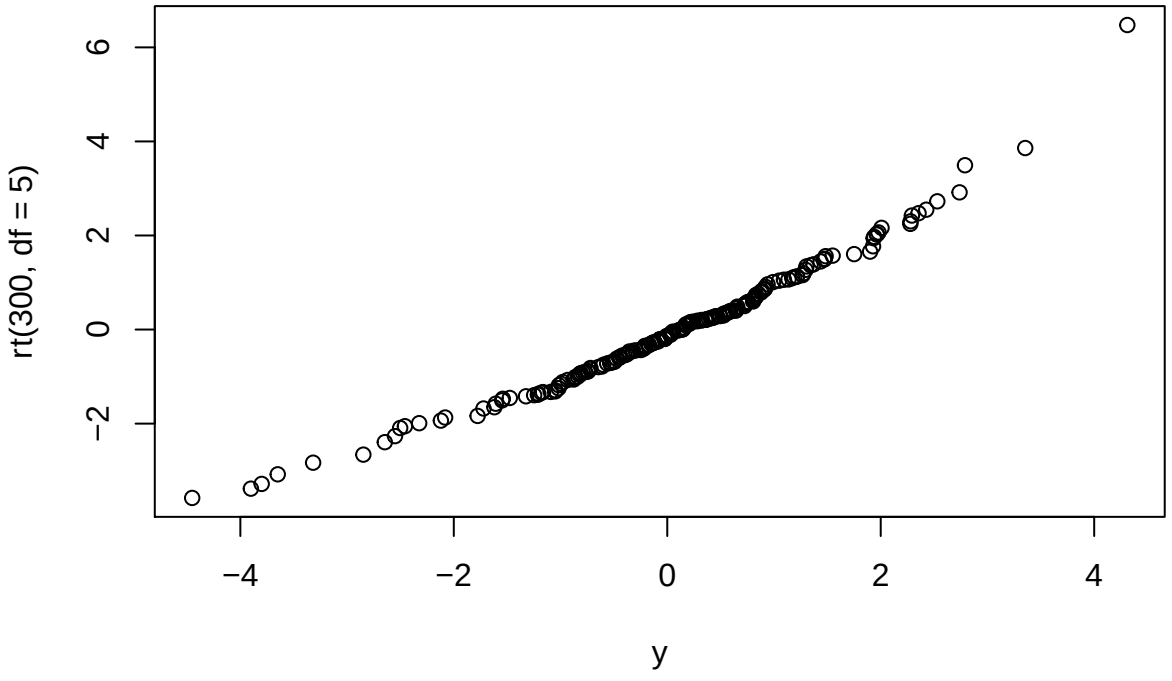
One of the most frequently used plotting functions in R is the `plot()` function. This is a generic function: the type of plot produced is dependent on the type or class of the first argument. Other examples of high-level plotting commands include: pairs and coplot (for multi-variate data), qqnorm, hist, dotchart, image, contour, persp.

```
example(qqnorm)
```

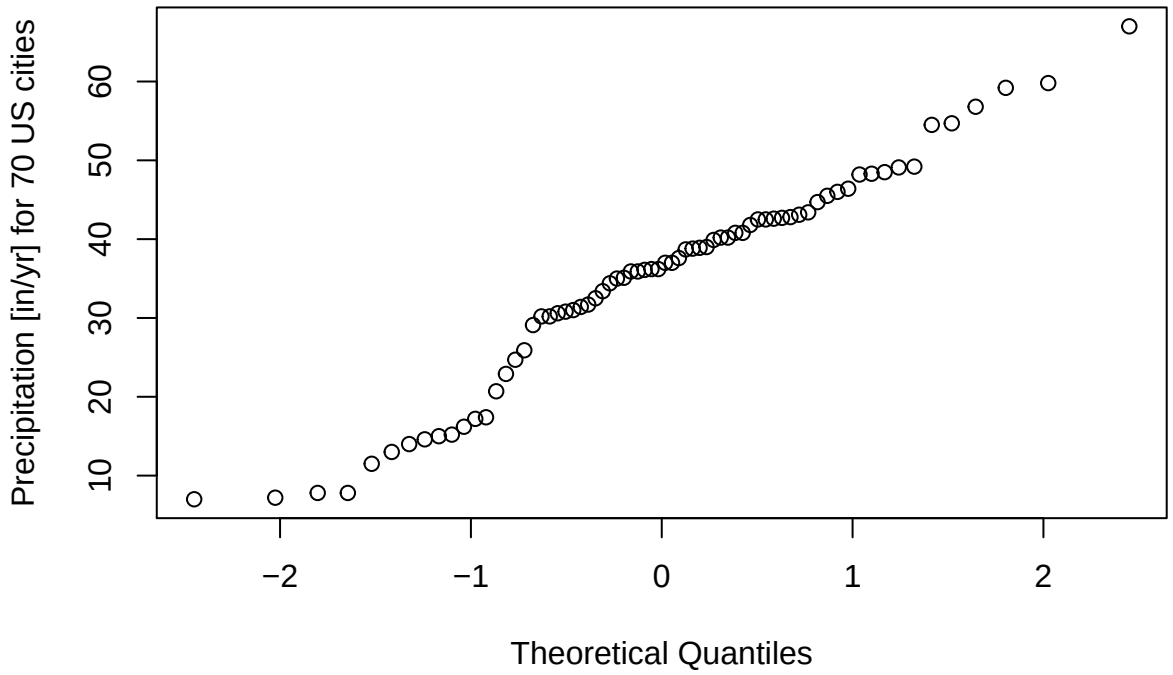
```
##
## qqnorm> require(graphics)
##
## qqnorm> y <- rt(200, df = 5)
##
## qqnorm> qqnorm(y); qqline(y, col = 2)
```



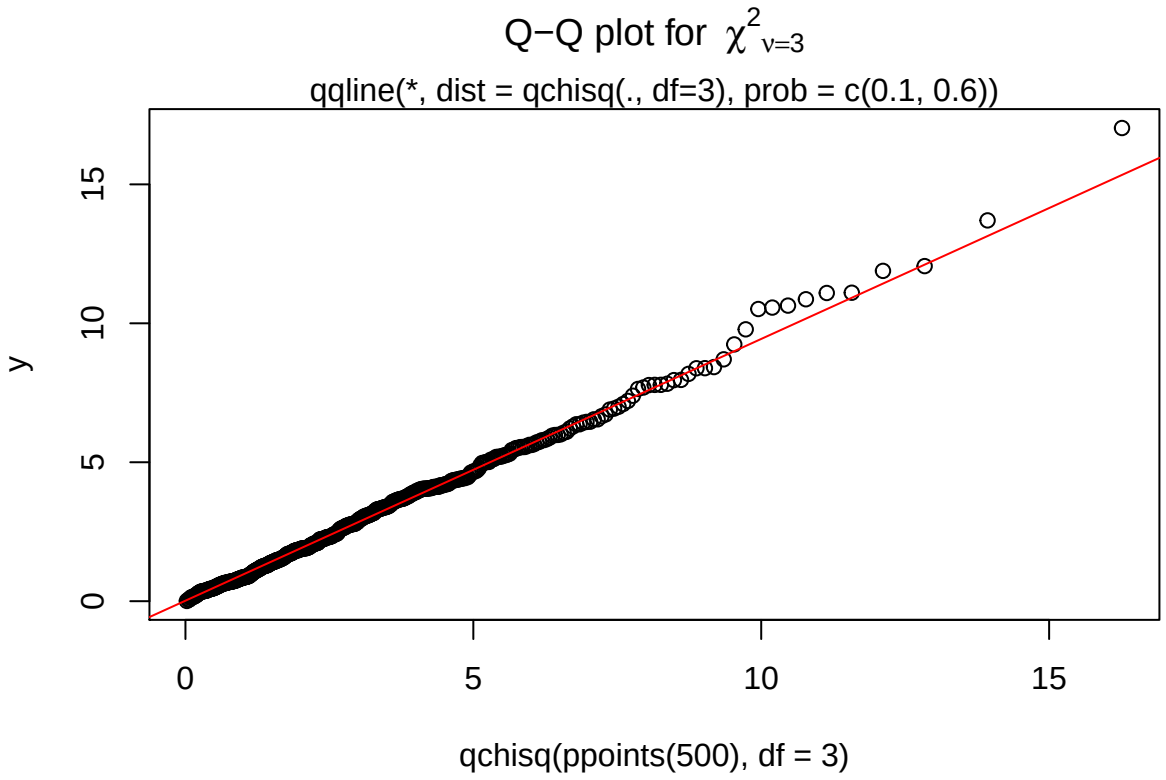

```
##
## qqnorm> qqplot(y, rt(300, df = 5))
```




```
##
## qqnorm> qqnorm(precip, ylab = "Precipitation [in/yr] for 70 US cities")
```




```
##
## qqnorm> ## "QQ-Chisquare" : --------------------------
## qqnorm> y <- rchisq(500, df = 3)
##
## qqnorm> ## Q-Q plot for Chi^2 data against true theoretical distribution:
## qqnorm> qqplot(qchisq(ppoints(500), df = 3), y,
## qqnorm+        main = expression("Q-Q plot for" ~~ {chi^2}[nu == 3]))
```




```
##
## qqnorm> qqline(y, distribution = function(p) qchisq(p, df = 3),
## qqnorm+        prob = c(0.1, 0.6), col = 2)
##
## qqnorm> mtext("qqline(*, dist = qchisq(., df=3), prob = c(0.1, 0.6))")
```

Low-level plotting commands

Sometimes the high-level plotting functions don't produce exactly the kind of plot you desire. In this case, low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot.

Some of the more useful low-level plotting functions are: points, lines. Adds points or connected lines to the current plot. plot()'s type= argument can also be passed to these functions (and defaults to "p" for points() and "l" for lines().) text(x, y, labels), will add text to a plot at points given by x, y. Normally labels is an integer or character vector in which case labels[i] is plotted at point (x[i], y[i]). The default is 1:length(x). the variations abline(a, b), abline(h=y), abline(v=x), abline(lm.obj) will add a line of slope b and intercept a to the current plot. h=y may be used to specify y-coordinates for the heights of horizontal lines to go across a plot, and v=x similarly for the x-coordinates for vertical lines. Also lm.obj may be list with a coefficients component of length 2 (such as the result of model-fitting functions,) which are taken as an intercept and slope, in that order.

polygon(x, y, ...) draws a polygon defined by the ordered vertices in (x, y) and (optionally) shade it in with hatch lines, or fill it if the graphics device allows the filling of figures.

legend(x, y, legend, ...) adds a legend to the current plot at the specified position. Plotting characters, line styles, colors etc., are identified with the labels in the character vector legend. At least one other argument v (a vector the same length as legend) with the corresponding values of the plotting unit must also be given, as follows:

title(main, sub). adds a title main to the top of the current plot in a large font and (optionally) a sub-title sub at the bottom in a smaller font. axis(side, ...) adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom.) Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling plot() with the axes=FALSE argument. Low-level plotting functions usually require some positioning information (e.g., x and y coordinates) to determine where to place the new plot elements. Coordinates are given in terms of user coordinates which are defined by the previous high-level graphics command and are chosen based on the supplied data.

Using graphics parameters

When creating graphics, particularly for presentation or publication purposes, R's defaults do not always produce exactly that which is required. You can, however, customize almost every aspect of the display using graphics parameters. R maintains a list of a large number of graphics parameters which control things such as line style, colors, figure arrangement and text justification among many others. Every graphics parameter has a name (such as 'col', which controls colors,) and a value (a color number, for example.)

A separate list of graphics parameters is maintained for each active device, and each device has a default set of parameters when initialized. Graphics parameters can be set in two ways: either permanently, affecting all graphics functions which access the current device; or temporarily, affecting only a single graphics function call.

The par() function is used to access and modify the list of graphics parameters for the current graphics device.

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

```
##  [1] "xlog"      "ylog"      "adj"       "ann"       "ask"
##  [6] "bg"        "bty"       "cex"       "cex.axis"  "cex.lab"
## [11] "cex.main"  "cex.sub"   "cin"       "col"       "col.axis"
## [16] "col.lab"   "col.main"  "col.sub"   "cra"       "crt"
## [21] "csi"       "cxy"       "din"       "err"       "family"
## [26] "fg"        "fig"       "fin"       "font"      "font.axis"
## [31] "font.lab"  "font.main" "font.sub"  "lab"       "las"
## [36] "lend"      "lheight"   "ljoin"     "lmitre"    "lty"
## [41] "lwd"       "mai"       "mar"       "mex"       "mfcol"
## [46] "mfg"       "mfrow"     "mgp"       "mkh"       "new"
## [51] "oma"       "omd"       "omi"       "page"      "pch"
## [56] "pin"       "plt"       "ps"        "pty"       "smo"
## [61] "srt"       "tck"       "tcl"       "usr"       "xaxp"
## [66] "xaxs"      "xaxt"      "xpd"       "yaxp"      "yaxs"
## [71] "yaxt"      "ylbias"
```

```
par(c("col", "lty"))
```

```
## $col
## [1] "black"
##
## $lty
## [1] "solid"
```

With a character vector argument, returns only the named graphics parameters (again, as a list.)

```
par(col=4, lty=2)
```

With named arguments (or a single list argument), sets the values of the named graphics parameters, and returns the original values of the parameters as a list.

Setting graphics parameters with the par() function changes the value of the parameters permanently, in the sense that all future calls to graphics functions (on the current device) will be affected by the new value. You can think of setting graphics parameters in this way as setting "default" values for the parameters, which will be used by all graphics functions unless an alternative value is given.

Note that calls to par() always affect the global values of graphics parameters, even when par() is called from within a function. This is often undesirable behavior—usually we want to set some graphics parameters, do some plotting, and then restore the original values so as not to affect the user's R session. You can restore the initial values by saving the result of par() when making changes, and restoring the initial values when plotting is complete.

```
> oldpar <- par(col=4, lty=2)
  ... plotting commands ...
> par(oldpar)
```

To save and restore all settable graphical parameters use

```
> oldpar <- par(no.readonly=TRUE)
  ... plotting commands ...
> par(oldpar)
```

Graphics parameters may also be passed to (almost) any graphics function as named arguments, via the `...` parameter. This has the same effect as passing the arguments to the par() function, except that the changes only last for the duration of the function call. For example:

plot(x, y, pch="+")

produces a scatterplot using a plus sign as the plotting character, without changing the default plotting character for future plots.

Unfortunately, this is not implemented entirely consistently, some high level functions may overwrite your calls, and it is sometimes necessary to set and reset graphics parameters using par().

Device drivers

R can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, R needs to be informed what type of device it is dealing with. This is done by starting a device driver. The purpose of a device driver is to convert graphical instructions from R ("draw a line," for example) into a form that the particular device can understand.

Device drivers are started by calling a device driver function. There is one such function for every device driver: type help(Devices) for a list of them all. For example, issuing the command

```
> postscript()
```

causes all future graphics output to be sent to the printer in PostScript format. Some commonly-used device drivers are:

- X11() - For use with the X11 window system on Unix-alikes
- windows() - For use on Windows
- quartz() - For use on Mac OS X
- postscript() - For printing on PostScript printers, or creating PostScript graphics files.
- pdf() - Produces a PDF file, which can also be included into PDF files.
- png() - Produces a bitmap PNG file. (Not always available: see its help page.)
- jpeg() - Produces a bitmap JPEG file, best used for image plots. (Not always available: see its help page.)

When you have finished with a device, be sure to terminate the device driver by issuing the command

```
> dev.off()
```

This ensures that the device finishes cleanly; for example in the case of hardcopy devices this ensures that every page is completed and has been sent to the printer. (This will happen automatically at the normal end of a session.)

Session Info

```
sessionInfo()
```

```
## R version 3.2.3 (2015-12-10)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.11.3 (El Capitan)
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats4    parallel  stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] ShortRead_1.26.0        GenomicAlignments_1.4.2
##  [3] Rsamtools_1.20.5        Biostrings_2.36.4
##  [5] XVector_0.8.0           BiocParallel_1.2.22
##  [7] GenomicFeatures_1.20.6  AnnotationDbi_1.30.1
##  [9] Biobase_2.28.0          GenomicRanges_1.20.8
## [11] GenomeInfoDb_1.4.3      Matrix_1.2-3
## [13] IRanges_2.2.9           S4Vectors_0.6.6
## [15] BiocGenerics_0.14.0     graph_1.46.0
## [17] BiocInstaller_1.18.5
##
## loaded via a namespace (and not attached):
##  [1] RColorBrewer_1.1-2   futile.logger_1.4.1  bitops_1.0-6
##  [4] futile.options_1.0.0 tools_3.2.3          zlibbioc_1.14.0
##  [7] biomaRt_2.24.1       digest_0.6.9         RSQLite_1.0.0
## [10] evaluate_0.8         lattice_0.20-33      DBI_0.3.1
## [13] yaml_2.1.13          hwriter_1.3.2        rtracklayer_1.28.10
## [16] stringr_1.0.0        knitr_1.12.3         grid_3.2.3
```

```
## [19] XML_3.98-1.3         rmarkdown_0.9.5      latticeExtra_0.6-26
## [22] lambda.r_1.1.7       magrittr_1.5         htmltools_0.3
## [25] stringi_1.0-1        RCurl_1.95-4.8
```