The high-level interface in rpy2 is designed to facilitate the use of R by Python programmers. R objects are exposed as instances of Python-implemented classes, with R functions as bound methods to those objects in a number of cases. This section also contains an introduction to graphics with R: *trellis* (*lattice*) plots as well as the grammar of graphics implemented in *ggplot2* let one make complex and informative plots with little code written, while the underlying *grid*graphics allow all possible customization is outlined.

# Overview

This module should be the right pick for casual and general use. Its aim is to abstract some of the details and provide an intuitive interface to both Python and R programmers.

```
>>> import rpy2.robjects as robjects
```

`rpy2.robjects` is written on top of `rpy2.rinterface`, and one not satisfied with it could easily build one's own flavor of a Python-R interface by modifying it (`rpy2.rpy_classic` is another example of a Python interface built on top of `rpy2.rinterface`).

Visible differences with RPy-1.x are:

- no `CONVERSION` mode in **rpy2**, the design has made this unnecessary
- easy to modify or rewrite with an all-Python implementation

# *r*: the instance of **R**

This class is currently a singleton, with its one representation instanciated when the module is loaded:

```
>>> robjects.r
>>> print(robjects.r)
```

The instance can be seen as the entry point to an embedded R process.

Being a singleton means that each time the constructor for `R` is called the same instance is returned; this is required by the fact that the embedded R is stateful.

The elements that would be accessible from an equivalent R environment are accessible as attributes of the instance. Readers familiar with the `ctypes` module for Python will note the similarity with it.

R vectors:

```
>>> pi = robjects.r.pi
>>> letters = robjects.r.letters
```

R functions:

```
>>> plot = robjects.r.plot
```

```
>>> dir = robjects.r.dir
```

This approach has limitation as:

- The actual Python attributes for the object masks the R elements
- '.' (dot) is syntactically valid in names for R objects, but not for

  python objects.

That last limitation can partly be removed by using `rpy2.rpy_classic` if this feature matters most to you.

```
>>> robjects.r.as_null
# AttributeError raised
>>> import rpy2.rpy_classic as rpy
>>> rpy.set_default_mode(rpy.NO_CONVERSION)
>>> rpy.r.as_null
# R function as.null() returned
```

**Note**

The section *Partial use of rpy_classic* outlines how to integrate `rpy2.rpy_classic` code.

Behind the scene, the steps for getting an attribute of *r* are rather straightforward:

1. Check if the attribute is defined as such in the python definition for *r*
2. Check if the attribute is can be accessed in R, starting from *globalenv*

When safety matters most, we recommend using `__getitem__()` to get a given R object.

```
>>> as_null = robjects.r['as.null']
```

Storing the object in a python variable will protect it from garbage collection, even if deleted from the objects visible to an R user.

```
>>> robjects.globalenv['foo'] = 1.2
>>> foo = robjects.r['foo']
>>> foo[0]
1.2
```

Here we *remove* the symbol *foo* from the R Global Environment.

```
>>> robjects.r['rm']('foo')
```

```
>>> robjects.r['foo']
LookupError: 'foo' not found
```

The object itself remains available, and protected from R's garbage collection until *foo* is deleted from Python

```
>>> foo[0]
1.2
```

## Evaluating a string as R code

Just like it is the case with RPy-1.x, on-the-fly evaluation of R code contained in a string can be performed by calling the *r* instance:

```
>>> print(robjects.r('1+2'))
[1] 3
>>> sqr = robjects.r('function(x) x^2')
>>> print(sqr)
function (x)
x^2
>>> print(sqr(2))
[1] 4
```

The astute reader will quickly realize that R objects named by python variables can be plugged into code through their **R** representation:

```
>>> x = robjects.r.rnorm(100)
>>> robjects.r('hist(%s, xlab="x", main="hist(x)")' %x.r_repr())
```

**Warning**

Doing this with large objects might not be the best use of your computing power.

# R objects

The class `rpy2.robjects.RObject` can represent any R object, although it will often be used for objects without any more specific representation in Python/rpy2 (such as `Vector`, `functions.Function`, `Environment`).

The class inherits from the lower-level rpy2.rinterface.Sexp and from rpy2.robjects.robject.RObjectMixin, the later defining higher-level methods for R objects to be shared by other higher-level representations of R objects.

*class* rpy2.robjects.robject.**RObjectMixin**

Bases: object

Class to provide methods common to all RObject instances

**r_repr()**

String representation for an object that can be directly evaluated as R code.

**rclass**

R class for the object, stored as an R string vector.

*class* rpy2.robjects.**RObject**

Bases: rpy2.robjects.robject.RObjectMixin, rpy2.rinterface.Sexp

Base class for all R objects.

# Environments

R environments can be described to the Python user as an hybrid of a dictionary and a scope.

The first of all environments is called the Global Environment, that can also be referred to as the R workspace.

An R environment in RPy2 can be seen as a kind of Python dictionnary.

Assigning a value to a symbol in an environment has been made as simple as assigning a value to a key in a Python dictionary:

```
>>> robjects.r.ls(globalenv)
>>> robjects.globalenv["a"] = 123
>>> print(robjects.r.ls(globalenv))
```

Care must be taken when assigning objects into an environment such as the Global Environment, as this can hide other objects with an identical name. The following example should make one measure that this can mean trouble if no care is taken:

```
>>> globalenv["pi"] = 123
>>> print(robjects.r.pi)
[1] 123
>>>
>>> robjects.r.rm("pi")
>>> print(robjects.r.pi)
[1] 3.1415926535897931
```

The class inherits from the class `rpy2.rinterface.SexpEnvironment`.

An environment is also iter-able, returning all the symbols (keys) it contains:

```
>>> env = robjects.r.baseenv()
>>> [x for x in env]
<a long list returned>
```

**Note**

Although there is a natural link between environment and R packages, one should consider using the convenience wrapper dedicated to model R packages (see *R packages*).

*class* rpy2.robjects.Environment(*o=None*)

Bases: rpy2.robjects.robject.RObjectMixin, rpy2.rinterface.SexpEnvironment

An R environement.

get(*item*, *wantfun=False*)

Get a object from its R name/symol :param item: string (name/symbol) :rtype: object (as returned by `conversion.ri2py()`)

keys()

Return a tuple listing the keys in the object

# Functions

R functions are callable objects, and can be called almost like any regular Python function:

```
>>> plot = robjects.r.plot
>>> rnorm = robjects.r.rnorm
>>> plot(rnorm(100), ylab="random")
```

This is all looking fine and simple until R arguments with names such as *na.rm* are encountered. By default, this is addressed by having a translation of '.' (dot) in the R argument name into a '_' in the Python argument name.

Let's take an example in R:

```
rank(0, na.last = TRUE)
```

In Python one can write:

```
from rpy2.robjects.packages import importr
base = importr('base')

base.rank(0, na_last = True)
```

**Note**

In this example, the object *base.rank* is an instance of `functions.SignatureTranslatedFunction`, a child class of `functions.Function`, and the translation of the argument names is made during the creation of the instance. Making the translation during the creation obviously saves the need to perform translation operations on parameter names, such as replacing . with _, at each function call, and allows *rpy2* to perform sanity checks regarding possible ambiguous translations; the cost of doing it is acceptable cost since this is only performed when the instance is created.

If no translation is desired, the class `functions.Function` can be used. With that class, using the special Python syntax ***kwargs* is one way to specify named arguments to R functions that contain a dot '.'

One will note that the translation is done by inspecting the signature of the R function, and that not much can be guessed from the R ellipsis '...' whenever present. Arguments falling in the '...' will need to have their R names passed to the constructor for`functions.SignatureTranslatedFunction` as show in the example below:

```
>>> graphics = importr('graphics')
>>> graphics.par(cex_axis = 0.5)
Warning message:
In function (..., no.readonly = FALSE)  :
"cex_axis" is not a graphical parameter
<Vector - Python:0xa1688cc / R:0xab763b0>
>>> graphics.par(**{'cex.axis': 0.5})
<Vector - Python:0xae8fbec / R:0xaafb850>
```

There exists a way to specify manually a argument mapping:

```
from rpy2.robjects.functions import SignatureTranslatedFunction
from rpy2.robjects.packages import importr
graphics = importr('graphics')
graphics.par = SignatureTranslatedFunction(graphics.par,
                                           init_prm_translate = {'cex_axis':
'cex.axis'})
>>> graphics.par(cex_axis = 0.5)
<Vector - Python:0xa2cc90c / R:0xa5f7fd8>
```

Translating blindly each '.' in argument names into '_' currently appears to be a risky practice, and is left to one to decide for his own code. (Bad) example:

```
def iamfeelinglucky(**kwargs):
    res = {}
    for k, v in kwargs.iteritems:
        res[k.replace('_', '.')] = v
    return res


graphics.par(**(iamfeelinglucky(cex_axis = 0.5)))
```

Things are also not always that simple, as the use of a dictionary does not ensure that the order in which the arguments are passed is conserved.

R is capable of introspection, and can return the arguments accepted by a function through the function *formals()*, modelled as a method of `functions.Function`.

```
>>> from rpy2.robjects.packages import importr
>>> stats = importr('stats')
>>> rnorm = stats.rnorm
>>> rnorm.formals()
<Vector - Python:0x8790bcc / R:0x93db250>
>>> tuple(rnorm.formals().names)
('n', 'mean', 'sd')
```

**Warning**

Here again there is a twist coming from R, and some functions are "special". rpy2 is exposing as `rpy2.rinterface.SexpClosure`R objects that can be either CLOSXP, BUILTINSXP, or SPECIALSXP. However, only CLOSXP objects will return non-null *formals*.

The R functions as defined in `rpy2.robjects` inherit from the class `rpy2.rinterface.SexpClosure`, and further documentation on the behavior of function can be found in Section *Functions*.

*class* rpy2.robjects.functions.**Function**(**args*, ***kwargs*)

Bases: rpy2.robjects.robject.RObjectMixin, rpy2.rinterface.SexpClosure

Python representation of an R function.

formals()

Return the signature of the underlying R function (as the R function 'formals()' would).

rcall(**args*)

Wrapper around the parent method rpy2.rinterface.SexpClosure.rcall().

*class* rpy2.robjects.functions.**SignatureTranslatedFunction**(**args*, ***kwargs*)

Bases: rpy2.robjects.functions.Function

Python representation of an R function, where the character '.' is replaced with '_' in the R arguments names.

# Formulae

For tasks such as modelling and plotting, an R formula can be a terse, yet readable, way of expressing what is wanted.

In R, it generally looks like:

```
x <- 1:10
y <- x + rnorm(10, sd=0.2)

fit <- lm(y ~ x)
```

In the call to *lm*, the argument is a *formula*, and it can read like *model y using x*. A formula is a **R** language object, and the terms in the formula are evaluated in the environment it was defined in. Without further specification, that environment is the environment in which the the formula is created.

The class `robjects.Formula` is representing an **R** formula.

```
import array
from rpy2.robjects import IntVector, Formula
from rpy2.robjects.packages import importr
stats = importr('stats')

x = IntVector(range(1, 11))
y = x.ro + stats.rnorm(10, sd=0.2)

fmla = Formula('y ~ x')
env = fmla.environment
env['x'] = x
env['y'] = y

fit = stats.lm(fmla)
```

One drawback with that approach is that pretty printing of the *fit* object is note quite as good as what one would expect when working in**R**: the *call* item now displays the code for the function used to perform the fit.

If one still wants to avoid polluting the R global environment, the answer is to evaluate R call within the environment where the function is defined.

```
from rpy2.robjects import Environment

eval_env = Environment()
eval_env['fmla'] = fmla
base = importr('base')

fit = base.eval.rcall(base.parse(text = 'lm(fmla)'), stats._env)
```

Other options are:

- Evaluate R code on the fly so we that model fitting function has a symbol in R
- `fit = robjects.r('lm(%s)' %fmla.r_repr())`
- Evaluate R code where all symbols are defined

*class* rpy2.robjects.**Formula**(*formula*, *environment = rinterface.globalenv*)

Bases: rpy2.robjects.robject.RObjectMixin, rpy2.rinterface.Sexp

environment

Get the environment in which the formula is finding its symbols.

getenvironment()

Get the environment in which the formula is finding its symbols.

setenvironment(*val*)

Set the environment in which a formula will find its symbols.

# R packages

## Importing R packages

In R, objects can be bundled into packages for distribution. In similar fashion to Python modules, the packages can be installed, and then loaded when their are needed. This is achieved by the R functions *library()* and *require()* (attaching the namespace of the package to the R *search path*).

```
from rpy2.robjects.packages import importr
utils = importr("utils")
```

The object `utils` is now a namespace object, in the sense that its `__dict__` contains keys corresponding to the R symbols. For example the R function *data()* can be accessed like:

```
>>> utils.data
<SignatureTranslatedFunction - Python:0x913754c / R:0x943bdf8>
```

Unfortunately, accessing an R symbol can be a little less straightforward as R symbols can contain characters that are invalid in Python symbols. Anyone with experience in R can even add there is a predilection for the dot (.).

In an attempt to address this, during the import of the package a translation of the R symbols is attempted, with dots becoming underscores. This is not unlike what could be found in `rpy`, but with distinctive differences:

- The translation is performed once, when the package is imported, and the results cached. The caching allows us to perform the check below.
- A check that the translation is not masking other R symbols in the package is performed (e.g., both 'print_me' and 'print.me' are present). Should it happen, a `rpy2.robjects.packages.LibraryError` is raised. To avoid this, use the optional argument*robject_translations* in the function `importr()`.

```
d = {'print.me': 'print_dot_me', 'print_me': 'print_uscore_me'}
```

```
thatpackage = importr('thatpackage', robject_translations = d)
```

- Thanks to the namespace encapsulation, translation is restricted to one package, limiting the risk of masking when compared to rpy translating relatively blindly and retrieving the first match

**Note**

There has been (sometimes vocal) concerns over the seemingly unnecessary trouble with not translating blindly '.' into '_' for all R symbols in packages, as rpy was doing it.

Fortunately the R development team is providing a real-life example in R's standard library (the /recommended packages/) to demonstrate the point a final time: the R package *tools* contains a function *package.dependencies* and a function *package_dependencies*, with different behaviour, signatures, and documentation pages.

If using `rpy2.robjects.packages`, we leave how to resolve this up to you. One way is to do:

```
d = {'package.dependencies': 'package_dot_dependencies',
     'package_dependencies': 'package_uscore_dependencies'}
tools = importr('tools', robject_translations = d)
```

The translation of '.' into '_' is clearly not sufficient, as R symbols can use a lot more characters illegal in Python symbols. Those more exotic symbols can be accessed through `__dict__`.

Example:

```
>>> utils.__dict__['?']
<Function - Python:0x913796c / R:0x9366fac>
```

In addition to the translation of robjects symbols, objects that are R functions see their named arguments translated as similar way (with '.' becoming '_' in Python).

```
>>> base = importr('base')
>>> base.scan._prm_translate
{'blank_lines_skip': 'blank.lines.skip',
 'comment_char': 'comment.char',
 'multi_line': 'multi.line',
 'na_strings': 'na.strings',
 'strip_white': 'strip.white'}
```

# Importing arbitrary R code as a package

R packages are not the only way to distribute code. From this author's experience there exists R code circulating as .R files.

This is most likely not a good thing, but as a Python developers this also what you might be given with the task to implement an application (such a web service) around that code. In most working places you will not have the option to refuse the code until it is packaged; fortunately rpy2 is trying to make this situation as simple as possible.

It is possible to take R code in a string, such as for example the content of a .R file and wrap it up as an rpy2 R package. If you are given various R files, it is possible to wrap all of them into their own package-like structure, making concerns such conflicting names in the respective files unnecessary.

```
square <- function(x) {
    return(x^2)
}

cube <- function(x) {
    return(x^3)
}
from rpy2.robjects.packages import SignatureTranslatedAnonymousPackage

string = """
square <- function(x) {
    return(x^2)
}

cube <- function(x) {
    return(x^3)
}
"""

powerpack = SignatureTranslatedAnonymousPackage(string, "powerpack")
```

The R functions *square* and *cube* can be called with *powerpack.square()* and *powerpack.cube*.

Package-less R code can be accessible from an URL, and some R users will just source it from the URL. A recent use-case is to source files from a code repository (for example GitHub).

Using a snippet on stackoverflow:

```
library(devtools)
source_url('https://raw.github.com/hadley/stringr/master/R/c.r')
```

**Note**

If concerned about computer security, you'll want to think about the origin of the code and to which level you trust the origin to be what it really is.

Python has utilities to read data from URLs.

```
import urllib2
from rpy2.robjects.packages import SignatureTranslatedAnonymousPackage

bioc_url = urllib2.urlopen('https://raw.github.com/hadley/stringr/master/R/c.r')
string = ''.join(bioc_url.readlines())

stringr_c = SignatureTranslatedAnonymousPackage(string, "stringr_c")
```

The object *stringr_c* encapsulates the funtions defined in the R file into something like what the rpy2 *importr* is returning.

```
>>> type(stringr_c)
rpy2.robjects.packages.SignatureTranslatedAnonymousPackage
>>> stringr_c._rpy2r.keys()
['str_join', 'str_c']
```

Unlike the R code first shown, this is not writing anything into the the R global environment.

```
>>> from rpy2.robjects import globalenv
>>> globalenv.keys()
()
```

## R namespaces

In R, a *namespace* is describing something specific in which symbols can be exported, or kept internal. A lot of recent R packages are declaring a namespace but this is not mandatory, although recommended in some R development circles.

Namespaces and the ability to control the export of symbols were introduced several years ago in R and were probably meant to address the relative lack of control on symbol

encapsulation an R programmer has. Without it importing a package is in R is like systematically writing *import ** on all packages and modules used in Python, that will predictably create potential problems as the number of packages used is increasing.

Since Python does not generally have the same requirement by default, `importr()` exposes all objects in an namespace, no matter they are exported or not.

## Class diagram

*exception* rpy2.robjects.packages.**LibraryError**

Error occuring when importing an R library

*class* rpy2.robjects.packages.**Package**(*env*, *name*, *translation={}*, *exported_names=None*, *on_conflict='fail'*, *version=None*)

Models an R package (and can do so from an arbitrary environment - with the caution that locked environments should mostly be considered).

*class* rpy2.robjects.packages.**PackageData**(*packagename*, *lib_loc=rpy2.rinterface.NULL*)

Datasets in an R package. In R datasets can be distributed with a package.

Datasets can be:

- serialized R objects
- R code (that produces the dataset)

For a given R packages, datasets are stored separately from the rest of the code and are evaluated/loaded lazily.

The lazy aspect has been conserved and the dataset are only loaded or generated when called through the method 'fetch()'.

**fetch**(*name*)

Fetch the dataset (loads it or evaluates the R associated with it.

In R, datasets are loaded into the global environment by default but this function returns an environment that contains the dataset(s).

**names**()

Names of the datasets

rpy2.robjects.packages.**get_packagepath**(*package*)

return the path to an R package installed

rpy2.robjects.packages.**importr**(*name*, *lib_loc=None*, *robject_translations={}*, *signature_translation=True*, *suppress_messages=True*,*on_conflict='fail'*, *data=True*)

Import an R package.

Arguments:

- name: name of the R package
- lib_loc: specific location for the R library (default: None)
- robject_translations: dict (default: {})
- signature_translation: dict (default: {})
- suppress_message: Suppress messages R usually writes on the console (defaut: True)
- on_conflict: 'fail' or 'warn' (default: 'fail')
- data: embed a PackageData objects under the attribute name __rdata__ (default: True)

Return:

- an instance of class SignatureTranslatedPackage, or of class Package

rpy2.robjects.packages.**quiet_require**(*name*, *lib_loc=None*)

Load an R package /quietly/ (suppressing messages to the console).

rpy2.robjects.packages.**reval**(*string*, *envir=<rpy2.rinterface.SexpEnvironment - Python:0x354fba0 / R:0x4191588>*)

Evaluate a string as R code - string: a string - envir: an environment in which the environment should take place

(default: R's global environment)

rpy2.robjects.packages.**wherefrom**(*symbol*, *startenv=<rpy2.rinterface.SexpEnvironment - Python:0x354fba0 / R:0x4191588>*)

For a given symbol, return the environment this symbol is first found in, starting from 'startenv'

## Finding where an R symbol is coming from

Knowing which object is effectively considered when a given symbol is resolved can be of much importance in R, as the number of packages attached grows and the use of the namespace accessors "::" and ":::" is not so frequent.

The function `wherefrom()` offers a way to find it:

```
>>> import rpy2.robjects.packages as rpacks
>>> env = rpacks.wherefrom('lm')
>>> env.do_slot('name')[0]
'package:stats'
```

**Note**

This does not generalize completely, and more details regarding environment, and packages as environment should be checked Section *SexpEnvironment*.

## Installing/removing R packages

R is shipped with a set of *recommended packages* (the equivalent of a standard library), but there is a large (and growing) number of other packages available.

Installing those packages must be done within R, see the R documentation. As a quick help, installing an R package can be done by

```
sudo R
```

And then in the R console:

```
install.packages('foo')
```

# Working with R's OOPs

Object-Oriented Programming can be achieved in R, but in more than one way. Beside the *official* S3 and S4 systems, there is a rich ecosystem of alternative implementations of objects, like aroma, or proto.

# S3 objects

S3 objects are default R objects (i.e., not S4 instances) for which an attribute “class” has been added.

```
>>> x = robjects.IntVector((1, 3))
>>> tuple(x.rclass)
('integer',)
```

Making the object x an instance of a class *pair*, itself inheriting from *integer* is only a matter of setting the attribute:

```
>>> x.rclass = robjects.StrVector(("pair", "integer"))
>>> tuple(x.rclass)
('pair', 'integer')
```

Methods for *S3* classes are simply R functions with a name such as name.<class_name>, the dispatch being made at run-time from the first argument in the function call.

For example, the function *plot.lm* plots objects of class *lm*. The call *plot(something)* makes R extract the class name of the object something, and see if a function *plot.<class_of_something>* is in the search path.

**Note**

This rule is not strict as there can exist functions with a *dot* in their name and the part after the dot not correspond to an S3 class name.

# S4 objects

S4 objects are a little more formal regarding their class definition, and all instances belong to the low-level R type SEXPS4.

The definition of methods for a class can happen anytime after the class has been defined (a practice something referred to as *monkey patching* or *duck punching* in the Python world).

There are obviously many ways to try having a mapping between R classes and Python classes, and the one proposed here is to make Python classes that inherit `rpy2.rinterface.methods.RS4`.

Before looking at automated ways to reflect R classes as Python classes, we look at how a class definition in Python can be made to reflect an R S4 class. We take the R class *lmList* in the package *lme4* and show how to write a Python wrapper for it.

## Manual R-in-Python class definition

**Note**

The R package *lme4* is not distributed with R, and will have to be installed for this example to work.

First, a bit of boilerplate code is needed. We import the higher-level interface and the function `rpy2.robjects.packages.importr()`. The R class we want to represent is defined in the `rpy2` modules and utilities.

```
import rpy2.robjects as robjects
import rpy2.rinterface as rinterface
from rpy2.robjects.packages import importr

lme4 = importr("lme4")
getmethod = robjects.baseenv.get("getMethod")

StrVector = robjects.StrVector
```

Once done, the Python class definition can be written. In the first part of that code, we choose a static mapping of the R-defined methods. The advantage for doing so is a bit of speed (as the S4 dispatch mechanism has a cost), and the disadvantage is that a modification of the method at the R level would require a refresh of the mappings concerned. The second part of the code is a wrapper to those mappings, where Python-to-R operations prior to calling the R method can be performed. In the last part of the class definition, a*static method* is defined. This is one way to have polymorphic constructors implemented.

```
class LmList(robjects.methods.RS4):
    """ Reflection of the S4 class 'lmList'. """

    _coef = getmethod("coef",
                      signature = StrVector(["lmList", ]),
                      where = "package:lme4")
    _confint = getmethod("confint",
                         signature = StrVector(["lmList", ]),
                         where = "package:lme4")
    _formula = getmethod("formula",
```

```
                          signature = StrVector(["lmList", ]),
                          where = "package:lme4")
    _lmfit_from_formula = getmethod("lmList",
                                    signature = StrVector(["formula",
"data.frame"]),
                                    where = "package:lme4")

    def _call_get(self):
        return self.do_slot("call")
    def _call_set(self, value):
        return self.do_slot("call", value)
    call = property(_call_get, _call_set, None, "Get or set the RS4 slot
'call'.")

    def coef(self):
        """ fitted coefficients """
        return self._coef(self)

    def confint(self):
        """ confidence interval """
        return self._confint(self)

    def formula(self):
        """ formula used to fit the model """
        return self._formula(self)

    @staticmethod
    def from_formula(formula,
                     data = rinterface.MissingArg,
                     family = rinterface.MissingArg,
                     subset = rinterface.MissingArg,
                     weights = rinterface.MissingArg):
        """ Build an LmList from a formula """
        res = LmList._lmfit_from_formula(formula, data,
                                         family = family,
                                         subset = subset,
                                         weights = weights)
        res = LmList(res)
        return res
```

Creating a instance of `LmList` can now be achieved by specifying a model as a `Formula` and a dataset.

```
sleepstudy = lme4.sleepstudy
```

```
formula = robjects.Formula('Reaction ~ Days | Subject')
lml1 = LmList.from_formula(formula,
                           sleepstudy)
```

A drawback of the approach above is that the R "call" is stored, and as we are passing the `DataFrame` *sleepstudy* (and as it is believed to to be an anonymous structure by R) the call is verbose: it comprises the explicit structure of the data frame (try to print *lml1*). This becomes hardly acceptable as datasets grow bigger. An alternative to that is to store the columns of the data frame into the environment for the `Formula`, as shown below:

```
sleepstudy = lme4.sleepstudy
formula = robjects.Formula('Reaction ~ Days | Subject')
for varname in ('Reaction', 'Days', 'Subject'):
    formula.environment[varname] = sleepstudy.rx2(varname)

lml1 = LmList.from_formula(formula)
```

*class* rpy2.robjects.methods.RS4(*sexp*)

Bases: rpy2.robjects.robject.RObjectMixin, rpy2.rinterface.SexpS4

Python representation of an R instance of class 'S4'.

*static* isclass(*name*)

Return whether the given name is a defined class.

slotnames()

Return the 'slots' defined for this object

validobject(*test=False*, *complete=False*)

Return whether the instance is 'valid' for its class.

## Automated R-in-Python class definitions

The S4 system allows polymorphic definitions of methods, that is, there can be several methods with the same name but different number and types of arguments. (This is like Clojure's multimethods). Mapping R methods to Python methods automatically and reliably requires a bit of work, and we chose to concatenate the method name with the type of the parameters in the signature.

Using the automatic mapping is very simple. One only needs to declare a Python class with the following attributes:

| | | |
|---|---|---|
| __rname__ | mandatory | the name of the R class |
| __rpackagename__ | optional | the R package in which the class is declared |
| __attr_translation__ | optional | **dict** to translate |
| __meth_translation__ | optional | **dict** to translate |

Example:

```
from rpy2.robjects.packages import importr
stats4 = importr('stats4')
from rpy2.robjects.methods import RS4, RS4Auto_Type

class MLE(RS4):
  __metaclass__ = RS4Auto_Type
  __rname__ = 'mle'
  __rpackagename__ = 'stats4'
```

The class *MLE* just defined has all attributes and methods needed to represent all slots (*attributes* in the S4 nomenclature) and methods defined for the class when the class is declared (remember that class methods can be declared afterwards, or even in a different R package).

*class* rpy2.robjects.methods.RS4Auto_Type(*sexp*)

Bases: type

This type (metaclass) takes an R S4 class and create a Python class out of it, copying the R documention page into the Python docstring. A class with this metaclass has the following optional attributes: __rname__, __rpackagename__, __attr__translation, __meth_translation__.

## Automated mapping of user-defined classes

Once a Python class mirroring an R classis is defined, the mapping can be made automatic by adding new rules to the conversion system (see Section *Mapping rpy2 objects to arbitrary python objects*).

# Object serialization

The python pickling system can be used to serialize objects to disk, and restore them from their serialized form.

```
import pickle
import rpy2.robjects as ro

x = ro.StrVector(('a', 'b', 'c'))
base_help.fetch('sum')
f = file('/tmp/foo.pso', 'w')
pickle.dump(x, f)
f.close()


f = file('/tmp/foo.pso', 'r')
x_again = pickle.load(f)
f.close()
```

**Warning**

Currently loading an object from a serialized form restores the object in its low-level form (as in `rpy2.rinterface`). Higher-level objects can be restored by calling the higher-level casting function `rpy2.robjects.conversion.ri2py()` (see *Mapping rpy2 objects to arbitrary python objects*).

# Vectors and arrays

Beside functions and environments, most of the objects an R user is interacting with are vector-like. For example, this means that any scalar is in fact a vector of length one.

The class `Vector` has a constructor:

```
>>> x = robjects.Vector(3)
```

*class* rpy2.robjects.**Vector**(*o*)

Bases: rpy2.robjects.robject.RObjectMixin, rpy2.rinterface.SexpVector

Vector(seq) -> Vector.

The parameter 'seq' can be an instance inheriting from rinterface.SexpVector, or an arbitrary Python object. In the later case, a conversion will be attempted using conversion.py2ri().

R vector-like object. Items can be accessed with:

- the method “__getitem__” (“[” operator)
- the delegators rx or rx2

iteritems()

iterate over names and values

sample(*n*, *replace=False*, *probabilities=None*)

Draw a sample of size n from the vector. If ‘replace’ is True, the sampling is done with replacement. The optional argument ‘probabilities’ can indicate sampling probabilities.

## Creating vectors

Creating vectors can be achieved either from R or from Python.

When the vectors are created from R, one should not worry much as they will be exposed as they should by rpy2.robjects.

When one wants to create a vector from Python, either the class Vector or the convenience classes IntVector, FloatVector, BoolVector,StrVector can be used.

*class* rpy2.robjects.vectors.BoolVector(*obj*)

Bases: rpy2.robjects.vectors.Vector, rpy2.rinterface.BoolSexpVector

Vector of boolean (logical) elements BoolVector(seq) -> BoolVector.

The parameter ‘seq’ can be an instance inheriting from rinterface.SexpVector, or an arbitrary Python sequence. In the later case, all elements in the sequence should be either booleans, or have a bool() representation.

*class* rpy2.robjects.vectors.IntVector(*obj*)

Bases: rpy2.robjects.vectors.Vector, rpy2.rinterface.IntSexpVector

Vector of integer elements IntVector(seq) -> IntVector.

The parameter ‘seq’ can be an instance inheriting from rinterface.SexpVector, or an arbitrary Python sequence. In the later case, all elements in the sequence should be either integers, or have an int() representation.

tabulate(*nbins=None*)

Like the R function tabulate, count the number of times integer values are found

*class* rpy2.robjects.vectors.**FloatVector**(*obj*)

Bases: rpy2.robjects.vectors.Vector, rpy2.rinterface.FloatSexpVector

Vector of float (double) elements

FloatVector(seq) -> FloatVector.

The parameter 'seq' can be an instance inheriting from rinterface.SexpVector, or an arbitrary Python sequence. In the later case, all elements in the sequence should be either float, or have a float() representation.

*class* rpy2.robjects.vectors.**StrVector**(*obj*)

Bases: rpy2.robjects.vectors.Vector, rpy2.rinterface.StrSexpVector

Vector of string elements

StrVector(seq) -> StrVector.

The parameter 'seq' can be an instance inheriting from rinterface.SexpVector, or an arbitrary Python sequence. In the later case, all elements in the sequence should be either strings, or have a str() representation.

**factor**() → FactorVector

Construct a factor vector from a vector of strings.

*class* rpy2.robjects.vectors.**ListVector**(*obj*)

Bases: rpy2.robjects.vectors.Vector, rpy2.rinterface.ListSexpVector

R list (vector of arbitray elements)

ListVector(iteritemable) -> ListVector.

The parameter 'iteritemable' can be any object inheriting from rpy2.rlike.container.TaggedList, rpy2.rinterface.SexpVector of type VECSXP, or dict.

*static* **from_length**(*length*)

Create a list of given length

Sequences of date or time points can be stored in POSIXlt or POSIXct objects. Both can be created from Python sequences of time.struct_timeobjects or from R objects.

*class* rpy2.robjects.vectors.POSIXlt(*obj*)

Bases: rpy2.robjects.vectors.POSIXt, rpy2.robjects.vectors.Vector

Representation of dates with a 9-component structure (similar to Python's time.struct_time).

POSIXlt(seq) -> POSIXlt.

The constructor accepts either an R vector or a sequence (an object with the Python sequence interface) of time.struct_time objects.

*class* rpy2.robjects.vectors.POSIXct(*obj*)

Bases: rpy2.robjects.vectors.POSIXt, rpy2.robjects.vectors.FloatVector

Representation of dates as seconds since Epoch. This form is preferred to POSIXlt for inclusion in a DataFrame.

POSIXlt(seq) -> POSIXlt.

The constructor accepts either an R vector floats or a sequence (an object with the Python sequence interface) of time.struct_time objects.

*static* sexp_from_datetime(*seq*)

return a POSIXct vector from a sequence of datetime.datetime elements.

*New in version 2.2.0:* Vectors for date or time points

## FactorVector

R's factors are somewhat peculiar: they aim at representing a memory-efficient vector of labels, and in order to achieve it are implemented as vectors of integers to which are associated a (presumably shorter) vector of labels. Each integer represents the position of the label in the associated vector of labels.

For example, the following vector of labels

| a | b | a | b | b | c |
|---|---|---|---|---|---|

will become

| 1 | 2 | 1 | 2 | 2 | 3 |
|---|---|---|---|---|---|

and

| a | b | c |
|---|---|---|

```
>>> sv = ro.StrVector('ababbc')
>>> fac = ro.FactorVector(sv)
>>> print(fac)
[1] a b a b b c
Levels: a b c
>>> tuple(fac)
(1, 2, 1, 2, 2, 3)
>>> tuple(fac.levels)
('a', 'b', 'c')
```

Since a `FactorVector` is an `IntVector` with attached metadata (the levels), getting items Python-style was not changed from what happens when gettings items from a `IntVector`. A consequence to that is that information about the levels is then lost.

```
>>> item_i = 0
>>> fac[item_i]
1
```

Getting the level corresponding to an item requires using the `levels`,:

```
>>> fac.levels[fac[item_i] - 1]
'a'
```

**Warning**

Do not forget to subtract one to the value in the **FactorVector**. Indexing in Python starts at zero while indexing R starts at one, and recovering the level for an item requires an adjustment between the two.

When extracting elements from a `FactorVector` a sensible default might be to use R-style extracting (see *Extracting items*), as it preserves the integer/string duality.

*class* rpy2.robjects.vectors.FactorVector(*obj*)

Bases: rpy2.robjects.vectors.IntVector

Vector of 'factors'

FactorVector(obj,

levels = rinterface.MissingArg, labels = rinterface.MissingArg, exclude = rinterface.MissingArg, ordered = rinterface.MissingArg) -> FactorVector

obj: StrVector or StrSexpVector levels: StrVector or StrSexpVector labels: StrVector or StrSexpVector (of same length as levels) exclude: StrVector or StrSexpVector ordered: boolean

### isordered

are the levels in the factor ordered ?

### iter_labels()

Iterate the over the labels, that is iterate over the items returning associated label for each item

### nlevels

number of levels

## Extracting items

Extracting elements of sequence/vector can become a thorny issue as Python and R differ on a number of points (index numbers starting at zero / starting at one, negative index number meaning *index from the end* / *everything except*, names cannot / can be used for subsettting).

In order to solve this, the Python way and the R way were made available through two different routes.

### Extracting, Python-style

The python `__getitem__()` method behaves like a Python user would expect it for a vector (and indexing starts at zero).

```
>>> x = robjects.r.seq(1, 5)
>>> tuple(x)
(1, 2, 3, 4, 5)
>>> x.names = robjects.StrVector('abcde')
>>> print(x)
a b c d e
1 2 3 4 5
>>> x[0]
1
```

```
>>> x[4]
5
>>> x[-1]
5
```

## Extracting, R-style

Access to R-style extracting/subsetting is granted though the two delegators *rx* and *rx2*, representing the R functions *[* and *[[*respectively.

In short, R-style extracting has the following characteristics:

- indexing starts at one
- the argument to subset on can be a vector of
  - integers (negative integers meaning exlusion of the elements)
  - booleans
  - strings (whenever the vector has *names* for its elements)

```
>>> print(x.rx(1))
[1] 1
>>> print(x.rx('a'))
a
1
```

R can extract several elements at once:

```
>>> i = robjects.IntVector((1, 3))
>>> print(x.rx(i))
[1] 1 3
>>> b = robjects.BoolVector((False, True, False, True, True))
>>> print(x.rx(b))
[1] 2 4 5
```

When a boolean extract vector is of smaller length than the vector, is expanded as necessary (this is know in R as the *recycling rule*):

```
>>> print(x.rx(True))
1:5
>>> b = robjects.BoolVector((False, True))
>>> print(x.rx(b))
[1] 2 4
```

In R, negative indices are understood as element exclusion.

```
>>> print(x.rx(-1))
2:5
>>> i = robjects.IntVector((-1, -3))
>>> print(x.rx(i))
[1] 2 4 5
```

That last example could also be written:

```
>>> i = - robjects.IntVector((1, 3)).ro
>>> print(x.rx(i))
[1] 2 4 5
```

This extraction system is quite expressive, as it allows a very simple writting of very commontasks in data analysis such as reordering and random sampling.

```
>>> from rpy2.robjects.packages import importr
>>> base = importr('base')
>>> x = robjects.IntVector((5,3,2,1,4))
>>> o_i = base.order(x)
>>> print(x.rx(o_i))
[1] 1 2 3 4 5
>>> rnd_i = base.sample(x)
>>> x_resampled = x.rx(o_i)
```

R operators are vector operations, with the operator applyied to each element in the vector. This can be used to build extraction indexes.

```
>>> i = x.ro > 3 # extract values > 3
>>> i = (x.ro >= 2 ).ro & (x.ro <= 4) # extract values between 2 and 4
```

(More on R operators in Section *Operators*).

R/S also have particularities, in which some see consistency issues. For example although the indexing starts at 1, indexing on 0 does not return an *index out of bounds* error but a vector of length 0:

```
>>> print(x.rx(0))
integer(0)
```

# Assigning items

## Assigning, Python-style

Since vectors are exposed as Python mutable sequences, the assignment works as for regular Python lists.

```
>>> x = robjects.IntVector((1,2,3))
>>> print(x)
[1] 1 2 3
>>> x[0] = 9
>>> print(x)
[1] 9 2 3
```

In R vectors can be *named*, that is elements of the vector have a name. This is notably the case for R lists. Assigning based on names can be made easily by using the method `Vector.index()`, as shown below.

```
>>> x = robjects.ListVector({'a': 1, 'b': 2, 'c': 3})
>>> x[x.names.index('b')] = 9
```

**Note**

`Vector.index()` has a complexity linear in the length of the vector's length; this should be remembered if performance issues are met.

## Assigning, R-style

Differences between the two languages require few adaptations, and in appearance complexify a little the task. Should other Python-based systems for the representation of (mostly numerical) data structure, such a `numpy` be preferred, one will be encouraged to expose our rpy2 R objects through those structures.

The attributes *rx* and *rx2* used previously can again be used:

```
>>> x = robjects.IntVector(range(1, 4))
>>> print(x)
[1] 1 2 3
>>> x.rx[1] = 9
>>> print(x)
```

```
[1] 9 2 3
```

For the sake of complete compatibility with R, arguments can be named (and passed as a `dict` or `rpy2.rlike.container.TaggedList`).

```
>>> x = robjects.ListVector({'a': 1, 'b': 2, 'c': 3})
>>> x.rx2[{'i': x.names.index('b')}] = 9
```

## Missing values

Anyone with experience in the analysis of real data knows that some of the data might be missing. In S/Splus/R special *NA* values can be used in a data vector to indicate that fact, and `rpy2.robjects` makes aliases for those available as data objects `NA_Logical`, `NA_Real`,`NA_Integer`, `NA_Character`, `NA_Complex`.

```
>>> x = robjects.IntVector(range(3))
>>> x[0] = robjects.NA_Integer
>>> print(x)
[1] NA  1  2
```

The translation of NA types is done at the item level, returning a pointer to the corresponding NA singleton class.

```
>>> x[0] is robjects.NA_Integer
True
>>> x[0] == robjects.NA_Integer
True
>>> [y for y in x if y is not robjects.NA_Integer]
[1, 2]
```

**Note**

`NA_Logical` is the alias for R's *NA*.

**Note**

The NA objects are imported from the corresponding `rpy2.rinterface` objects.

# Operators

Mathematical operations on two vectors: the following operations are performed element-wise in R, recycling the shortest vector if, and as much as, necessary.

To expose that to Python, a delegating attribute `ro` is provided for vector-like objects.

| Python | R |
|---|---|
| + | + |
| - | - |
| * | * |
| / | / |
| ** | ** or ^ |
| or | \| |
| and | & |
| < | < |
| <= | <= |
| == | == |
| != | != |

```
>>> x = robjects.r.seq(1, 10)
>>> print(x.ro + 1)
2:11
```

**Note**

In Python, using the operator `+` on two sequences concatenates them and this behavior hasbeen conserved:

```
>>> print(x + 1)
[1]  1  2  3  4  5  6  7  8  9 10  1
```

**Note**

The boolean operator `not` cannot be redefined in Python (at least up to version 2.5), and its behavior could not be made to mimic R's behavior

# Names

`R` vectors can have a name given to all or some of the elements. The property `names` can be used to get, or set, those names.

```
>>> x = robjects.r.seq(1, 5)
>>> x.names = robjects.StrVector('abcde')
>>> x.names[0]
'a'
>>> x.names[0] = 'z'
>>> tuple(x.names)
('z', 'b', 'c', 'd', 'e')
```

## Array

In *R*, arrays are simply vectors with a dimension attribute. That fact was reflected in the class hierarchy with `robjects.Array` inheriting from `robjects.Vector`.

*class* rpy2.robjects.vectors.**Array**(*obj*)

Bases: `rpy2.robjects.vectors.Vector`

An R array

**dimnames**

names associated with the dimension.

**names**

names associated with the dimension.

## Matrix

A `Matrix` is a special case of `Array`. As with arrays, one must remember that this is just a vector with dimension attributes (number of rows, number of columns).

```
>>> m = robjects.r.matrix(robjects.IntVector(range(10)), nrow=5)
>>> print(m)
     [,1] [,2]
[1,]    0    5
[2,]    1    6
[3,]    2    7
[4,]    3    8
[5,]    4    9
```

**Note**

In *R*, matrices are column-major ordered, although the constructor `matrix()` accepts a boolean argument *byrow* that, when true, will build the matrix *as if* row-major ordered.

## Computing on matrices

Regular operators work element-wise on the underlying vector.

```
>>> m = robjects.r.matrix(robjects.IntVector(range(4)), nrow=2)
>>> print(m.ro + 1)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

For more on operators, see *Operators*.

Matrix multiplication is available as `Matrix.dot()`, transposition as `Matrix.transpose()`. Common operations such as cross-product, eigen values computation , and singular value decomposition are also available through method with explicit names.

```
>>> print( m.crossprod(m) )
     [,1] [,2]
[1,]    1    3
[2,]    3   13
>>> print( m.transpose().dot(m) )
     [,1] [,2]
[1,]    1    3
[2,]    3   13
```

*class* rpy2.robjects.vectors.**Matrix**(*obj*)

Bases: rpy2.robjects.vectors.Array

An R matrix

**colnames**

Column names

**crossprod**(*m*)

crossproduct X'.Y

dot(*m*)

Matrix multiplication

eigen()

Eigen values

ncol

Number of columns

nrow

Number of rows

rownames

Row names

svd(*nu=None*, *nv=None*, *linpack=False*)

SVD decomposition. If nu is None, it is given the default value min(tuple(self.dim)). If nv is None, it is given the default value min(tuple(self.dim)).

tcrossprod(*m*)

crossproduct X.Y'

transpose()

transpose the matrix

## Extracting

Extracting can still be performed Python-style or R-style.

```
>>> m = ro.r.matrix(ro.IntVector(range(2, 8)), nrow=3)
>>> print(m)
     [,1] [,2]
[1,]    2    5
[2,]    3    6
[3,]    4    7
>>> m[0]
```

```
2
>>> m[5]
7
>>> print(m.rx(1))
[1] 2
>>> print(m.rx(6))
[1] 7
```

Matrixes are two-dimensional arrays, and elements can be extracted according to two indexes:

```
>>> print(m.rx(1, 1))
[1] 2
>>> print(m.rx(3, 2))
[1] 7
```

Extracting a whole row, or column can be achieved by replacing an index number by *True* or *False*

Extract the first column:

```
>>> print(m.rx(True, 1))
```

Extract the second row:

```
>>> print(m.rx(2, True))
```

## DataFrame

Data frames are a common way in R to represent the data to analyze.

A data frame can be thought of as a tabular representation of data, with one variable per column, and one data point per row. Each column is an R vector, which implies one type for all elements in one given column, and which allows for possibly different types across different columns.

If we consider for example tre data about pharmacokinetics of theophylline in different subjects, the data table could look like this:

| Subject | Weight | Dose | Time | conc |
|---|---|---|---|---|
| 1 | 79.6 | 4.02 | 0.00 | 0.74 |

| Subject | Weight | Dose | Time | conc |
|---|---|---|---|---|
| 1 | 79.6 | 4.02 | 0.25 | 2.84 |
| 1 | 79.6 | 4.02 | 0.57 | 6.57 |
| 2 | 72.4 | 4.40 | 7.03 | 5.40 |
| ... | ... | ... | ... | ... |

Such data representation shares similarities with a table in a relational database: the structure between the variables, or columns, is given by other column. In the example above, the grouping of measures by subject is given by the column *Subject*.

In `rpy2.robjects`, `DataFrame` represents the *R* class *data.frame*.

## Creating objects

Creating a `DataFrame` can be done by:

- Using the constructor for the class
- Create the data.frame through R
- Read data from a file using the instance method **from_csvfile()**

The `DataFrame` constructor accepts either an `rinterface.SexpVector` (with `typeof` equal to *VECSXP*, that is, an R *list*) or any Python object implementing the method `iteritems()` (for example `dict` or `rpy2.rlike.container.OrdDict`).

Empty *data.frame*:

```
>>> dataf = robjects.DataFrame({})
```

*data.frame* with 2 two columns (not that the order of the columns in the resulting `DataFrame` can be different from the order in which they are declared):

```
>>> d = {'a': robjects.IntVector((1,2,3)), 'b': robjects.IntVector((4,5,6))}
>>> dataf = robject.DataFrame(d)
```

To create a `DataFrame` and be certain of the clumn order order, an ordered dictionary can be used:

```
>>> import rpy2.rlike.container as rlc
>>> od = rlc.OrdDict([('value', robjects.IntVector((1,2,3))),
                      ('letter', robjects.StrVector(('x', 'y', 'z')))])
>>> dataf = robjects.DataFrame(od)
>>> print(dataf.colnames)
```

```
[1] "letter" "value"
```

Creating the data.frame in R can otherwise be achieved in numerous ways, as many R functions do return a *data.frame*, such as the function*data.frame()*.

**Note**

When creating a `DataFrame`, vectors of strings are automatically converted by R into instances of class `Factor`. This behavior can be prevented by wrapping the call into the R base function I.

```
from rpy2.robjects.vectors import DataFrame, StrVector
from rpy2.robjects.packages import importr
base = importr('base')
dataf = DataFrame({'string': base.I(StrVector('abbab')),
                   'factor': StrVector('abbab')})
```

Here the `DataFrame` *dataf* now has two columns, one as a `Factor`, the other one as a `StrVector`

```
>>> dataf.rx2('string')
<StrVector - Python:0x95fe5ec / R:0x9646ea0>
>>> dataf.rx2('factor')
<FactorVector - Python:0x95fe86c / R:0x9028138>
```

## Extracting elements

Here again, Python's `__getitem__()` will work as a Python programmer will expect it to:

```
>>> len(dataf)
2
>>> dataf[0]
<Vector - Python:0x8a58c2c / R:0x8e7dd08>
```

The `DataFrame` is composed of columns, with each column being possibly of a different type:

```
>>> [column.rclass[0] for column in dataf]
['factor', 'integer']
```

Using R-style access to elements is a little richer, with the *rx2* accessor taking more importance than earlier.

Like with Python's `__getitem__()` above, extracting on one index selects columns:

```
>>> dataf.rx(1)
<DataFrame - Python:0x8a584ac / R:0x95a6fb8>
>>> print(dataf.rx(1))
  letter
1      x
2      y
3      z
```

Note that the result is itself of class `DataFrame`. To get the column as a vector, use *rx2*:

```
>>> dataf.rx2(1)
<Vector - Python:0x8a4bfcc / R:0x8e7dd08>
>>> print(dataf.rx2(1))
[1] x y z
Levels: x y z
```

Since data frames are table-like structure, they can be thought of as two-dimensional arrays and can therefore be extracted on two indices.

```
>>> subdataf = dataf.rx(1, True)
>>> print(subdataf)
  letter value
1      x     1
>>> rows_i <- robjects.IntVector((1,3))
>>> subdataf = dataf.rx(rows_i, True)
>>> print(subdataf)
  letter value
1      x     1
3      z     3
```

That last example is extremely common in R. A vector of indices, here *rows_i*, is used to take a subset of the `DataFrame`.

## Python docstrings

*class* rpy2.robjects.vectors.**DataFrame**(*tlist*)

Bases: rpy2.robjects.vectors.ListVector

R ‘data.frame’.

cbind(*args, **kwargs)

bind objects as supplementary columns

*static* from_csvfile(*path*, *header=True*, *sep=', '*, *quote='"'*, *dec='.'*, *row_names=rpy2.rinterface.MissingArg*,*col_names=rpy2.rinterface.MissingArg*, *fill=True*, *comment_char=''*, *as_is=False*)

Create an instance from data in a .csv file.

path : string with a path header : boolean (heading line with column names or not) sep : separator character quote : quote character row_names : column name, or column index for column names (warning: indexing starts at one in R) fill : boolean (fill the lines when less entries than columns) comment_char : comment character as_is : boolean (keep the columns of strings as such, or turn them into factors)

iter_column()

iterator across columns

iter_row()

iterator across rows

ncol

Number of columns. :rtype: integer

nrow

Number of rows. :rtype: integer

rbind(*args, **kwargs)

bind objects as supplementary rows

rownames

Row names

to_csvfile(*path*, *quote=True*, *sep=', '*, *eol='n'*, *na='NA'*, *dec='.'*, *row_names=True*, *col_names=True*,

*qmethod='escape'*, *append=False*)

Save the data into a .csv file.

path : string with a path quote : quote character sep : separator character eol : end-of-line character(s) na : string for missing values dec : string for decimal separator row_names : boolean (save row names, or not) col_names : boolean (save column names, or not) comment_char : method to 'escape' special characters append : boolean (append if the file in the path is already existing, or not)

# R help

*R* has a documentation system that ensures that documentation for code distributed as packages is installed when packages are installed. This documentation can be called and searched from R itself.

Unlike *Python* docstrings, where the documentation string can be found in the special attribute `__doc__`, the R documentation lives outside objects in documentation pages. Each documentation page is associated at minimum one *alias*, aliases often corresponding to the name of an R object defined in a package (function, dataset, etc...).

For example, querying documentation for the R function *sum* becomes a matter of finding which documentation page has the alias *sum*, and retrieve that page.

## Querying on aliases

When working with R, a frequent use case for using the documention is to query on an alias (a function name, a dataset, or a class name) and retrieve the associated documentation.

While the R packaging system will make checks that any given alias is associated with only one page within the same package, it is well possible to have several packages defining a documentation page for the same alias.

With rpy2's interface to the help system, an easy way to retrive pages associated with an alias is to use the function `pages()`, which returns a `tuple` of `Page` instances.

rpy2.robjects.help.pages(*topic*)

Get help pages corresponding to a given topic.

## Package documentation

The documentation for a package is represented with the class `Package`.

*class* rpy2.robjects.help.Package(*package_name*, *package_path = None*)

Bases: object

The R documentation page (aka help) for a package.

fetch(*alias*)

Fetch the documentation page associated with a given alias.

For S4 classes, the class name is *often* suffixed with '-class'. For example, the alias to the documentation for the class AnnotatedDataFrame in the package Biobase is 'AnnotatedDataFrame-class'.

name

Name of the package as known by R

package_path

Path to the installed R package

```
>>> import rpy2.robjects.help as rh
>>> base_help = rh.Package('base')
>>> base_help.fetch('sum')
```

## Documentation page

A documentation page is represented as an instance of class Page.

*class* rpy2.robjects.help.Page(*struct_rdb*)

Bases: object

An R documentation page. The original R structure is a nested sequence of components, corresponding to the latex-like .Rd file

An help page is divided into sections, the names for the sections are the keys for the dict attribute 'sections', and a given section can be extracted with the square-bracket operator.

In R, the S3 class 'Rd' is the closest entity to this class.

### arguments()

Get the arguments and their description as a list of 2-tuples.

### description()

Get the description of the entry

### iteritems()

iterator through the sections names and content in the documentation Page.

### seealso()

Get the other documentation entries recommended

### title()

Get the title

### to_docstring(*section_names=None*)

section_names: list of section names to consider. If None all sections are used.

Returns a string that can be used as a Python docstring.

### usage()

Get the usage for the object

### value()

Get the value returned

```
>>> hp = base_help.fetch('sum')
>>> hp.sections.keys()
('title', 'name', 'alias', 'keyword', 'description', 'usage',
'arguments', 'deta
ils', 'value', 'section', 'references', 'seealso')
```

**Note**

```
>>> print(''.join(hp.to_docstring(('details',))))
details
-------
```

```
This is a generic function: methods can be defined for it
directly or via the  Summary  group generic.
For this to work properly, the arguments   should be
unnamed, and dispatch is on the first argument.

If  na.rm  is  FALSE  an  NA
value in any of the arguments will cause
a value of  NA  to be returned, otherwise
NA  values are ignored.

Logical true values are regarded as one, false values as zero.
For historical reasons,  NULL  is accepted and treated as if it
were  integer(0) .
```

# Graphics

## Introduction

This section shows how to make R graphics from rpy2, using some of the different graphics systems available to R users.

The purpose of this section is to get users going, and be able to figure out by reading the R documentation how to perform the same plot in rpy2.

## Graphical devices

With *R*, all graphics are plotted into a so-called graphical device. Graphical devices can be interactive, like for example *X11*, or non-interactive, like *png* or*pdf*. Non-interactive devices appear to be files. It is possible to create custom graphical devices from Python/rpy2, but this an advanced topic (see *Custom graphical devices*).

By default an interactive R session will open an interactive device when needing one. If a non-interactive graphical device is needed, one will have to specify it.

**Note**

Do not forget to close a non-interactive device when done. This can be required to flush pending data from the buffer.

The module `grdevices` aims at representing the R package grDevices*. Example with the R functions *png* and *dev.off*:

```
from rpy2.robjects.packages import importr
grdevices = importr('grDevices')

grdevices.png(file="path/to/file.png", width=512, height=512)
# plotting code here
grdevices.dev_off()
```

The package contains an `Environment` `grdevices_env` that can be used to access an object known to belong to that R packages, e.g.:

```
>>> palette = grdevices.palette()
>>> print(palette)
[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
[8] "gray"
```

## Getting ready

To run examples in this section we first import `rpy2.robjects` and define few helper functions.

```
from rpy2 import robjects
from rpy2.robjects import Formula, Environment
from rpy2.robjects.vectors import IntVector, FloatVector
from rpy2.robjects.lib import grid
from rpy2.robjects.packages import importr

# The R 'print' function
rprint = robjects.globalenv.get("print")
stats = importr('stats')
grdevices = importr('grDevices')
base = importr('base')
datasets = importr('datasets')
```

# Package *lattice*

## Introduction

Importing the package *lattice* is done the same as it is done for other R packages.

```
lattice = importr('lattice')
```

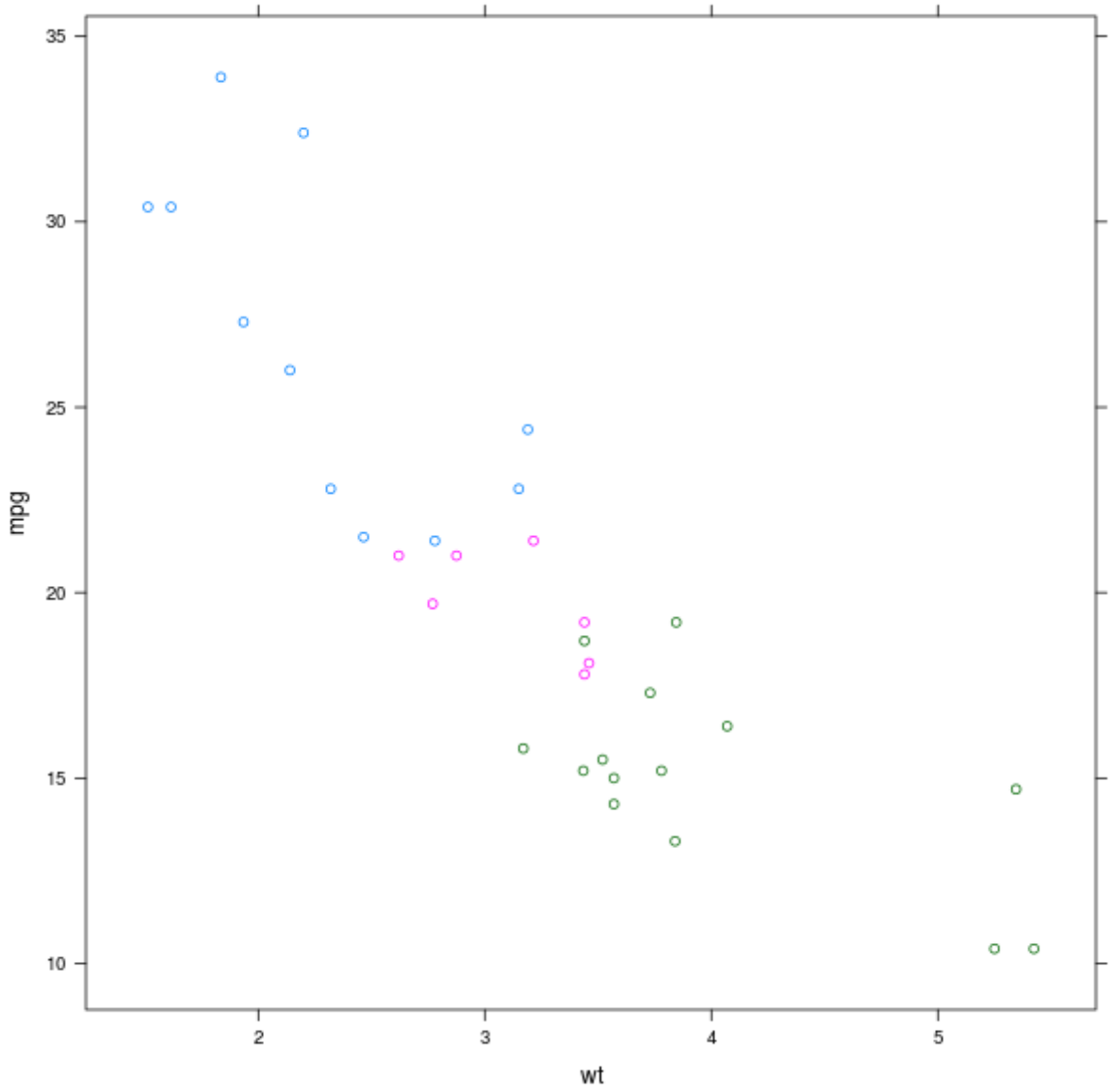
## Scatter plot

We use the dataset *mtcars*, and will use the lattice function *xyplot* to make scatter plots.

```
xyplot = lattice.xyplot
```

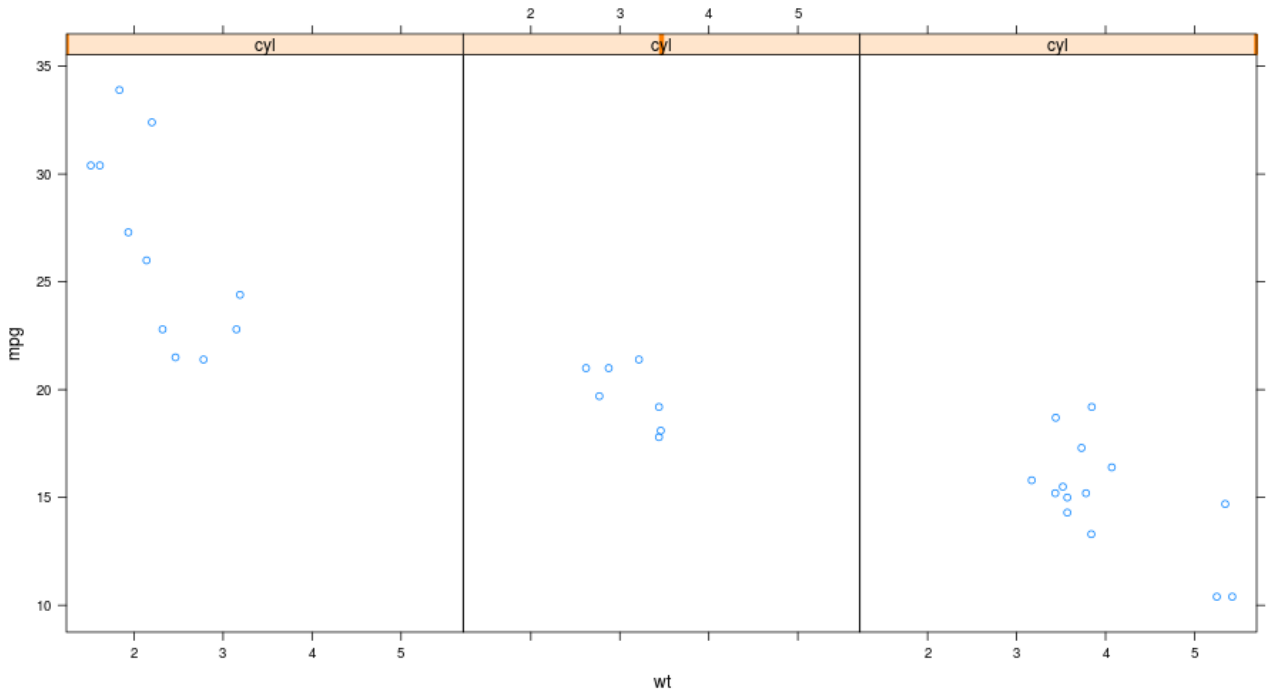
Lattice is working with formulae (see *Formulae*), therefore we build one and store values in its environment. Making a plot is then a matter of calling the function *xyplot* with the *formula* as as an argument.

```
datasets = importr('datasets')
mtcars = datasets.__rdata__.fetch('mtcars')['mtcars']
formula = Formula('mpg ~ wt')
formula.getenvironment()['mpg'] = mtcars.rx2('mpg')
formula.getenvironment()['wt'] = mtcars.rx2('wt')

p = lattice.xyplot(formula)
rprint(p)
```



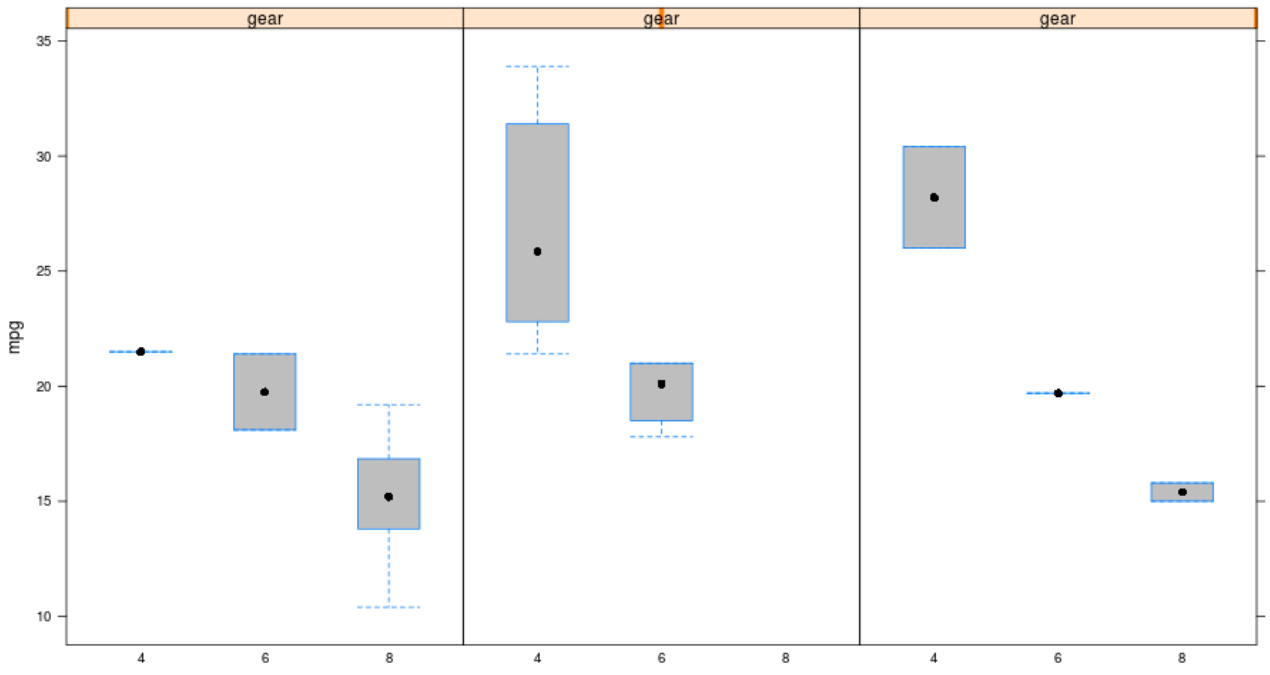

An alternative to color-coding is to have points is different *panels*. In lattice, this done by specifying it in the formula.

```
formula = Formula('mpg ~ wt | cyl')
formula.getenvironment()['mpg'] = mtcars.rx2('mpg')
formula.getenvironment()['wt'] = mtcars.rx2('wt')
formula.getenvironment()['cyl'] = mtcars.rx2('cyl')

p = lattice.xyplot(formula, layout = IntVector((3, 1)))
rprint(p)
```




## Box plot

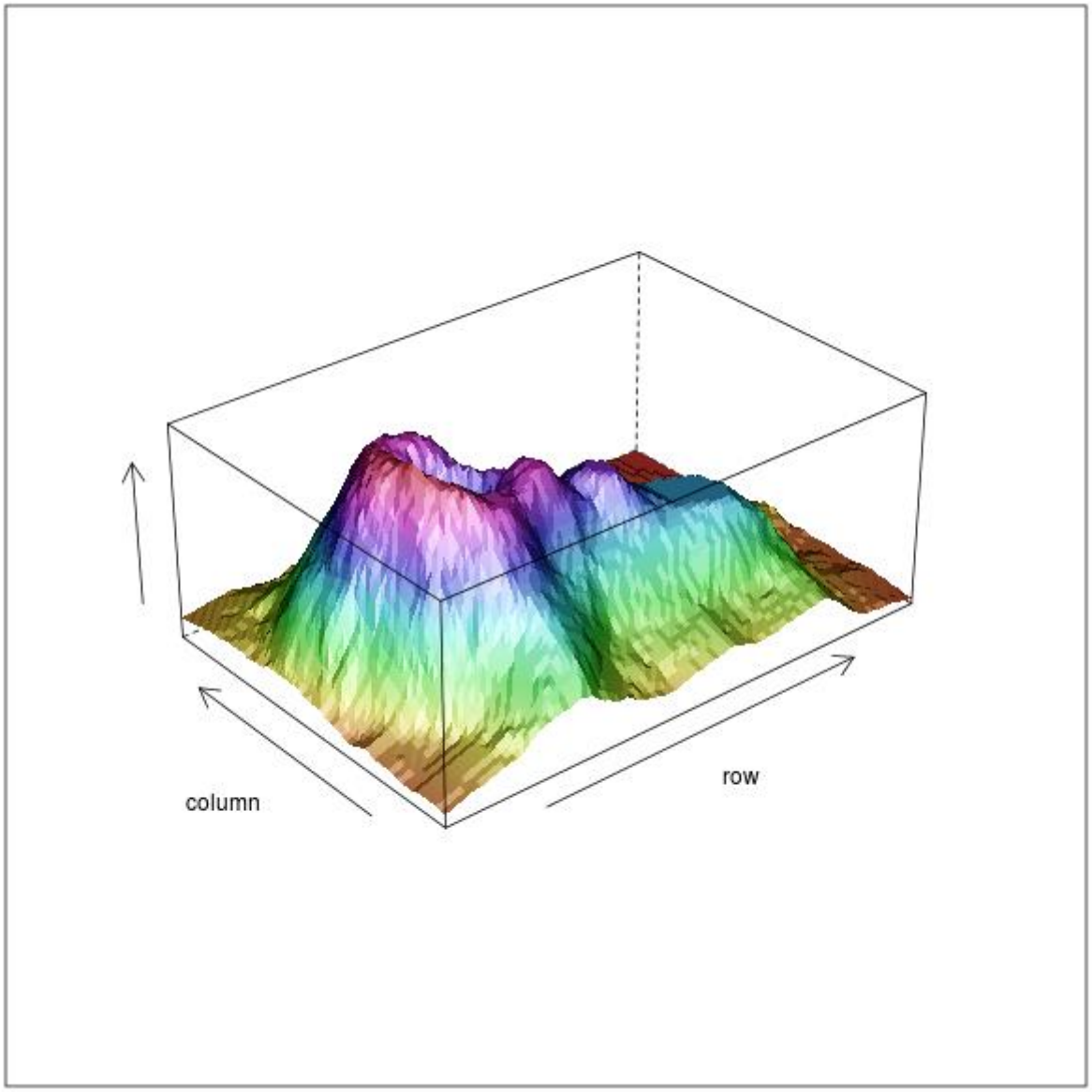
```
p = lattice.bwplot(Formula('mpg ~ factor(cyl) | gear'),
                   data = mtcars, fill = 'grey')
rprint(p, nrow=1)
```



## Other plots

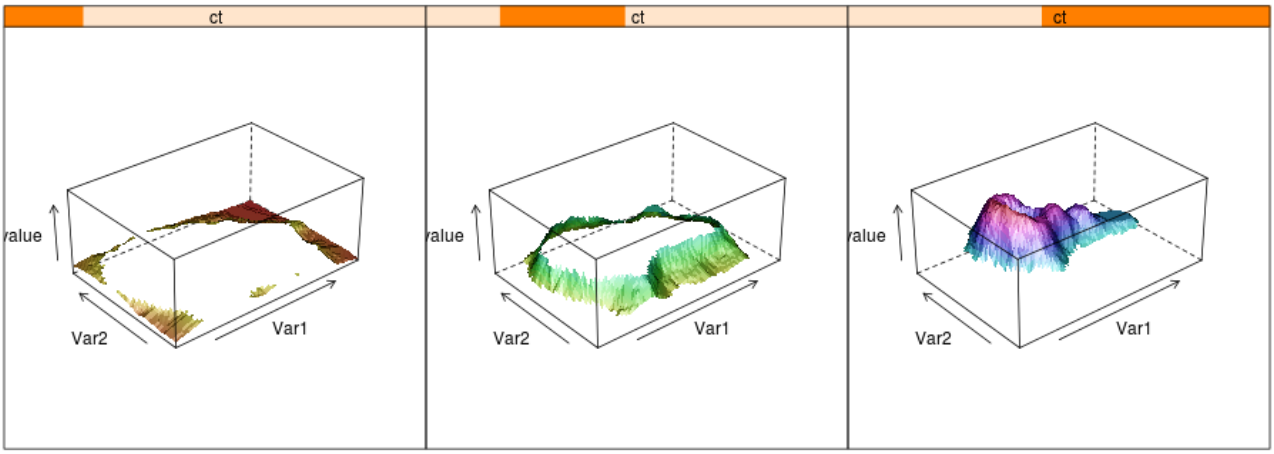
The R package lattice contains a number of other plots, which unfortunately cannot all be detailled here.

```
tmpenv = datasets.__rdata__.fetch("volcano")
volcano = tmpenv["volcano"]

p = lattice.wireframe(volcano, shade = True,
                      zlab = "",
                      aspect = FloatVector((61.0/87, 0.4)),
                      light_source = IntVector((10,0,10)))
rprint(p)
```



Splitting the information into different panels can also be specified in the formula. Here we show an artifial example where the split is made according to the values plotted on the Z axis.

```
reshape2 = importr('reshape2')
dataf = reshape2.melt(volcano)
dataf = dataf.cbind(ct = lattice.equal_count(dataf.rx2("value"), number=3,
overlap=1/4))
p = lattice.wireframe(Formula('value ~ Var1 * Var2 | ct'),
                      data = dataf, shade = True,
                      aspect = FloatVector((61.0/87, 0.4)),
                      light_source = IntVector((10,0,10)))
rprint(p, nrow = 1)
```




# Package *ggplot2*

## Introduction

The R package *ggplot2* implements the Grammar of Graphics. While more documentation on the package and its usage with R can be found on the ggplot2 website, this section will introduce the basic concepts required to build plots.

Obviously, the *R* package *ggplot2* is expected to be installed in the *R* used from *rpy2*.

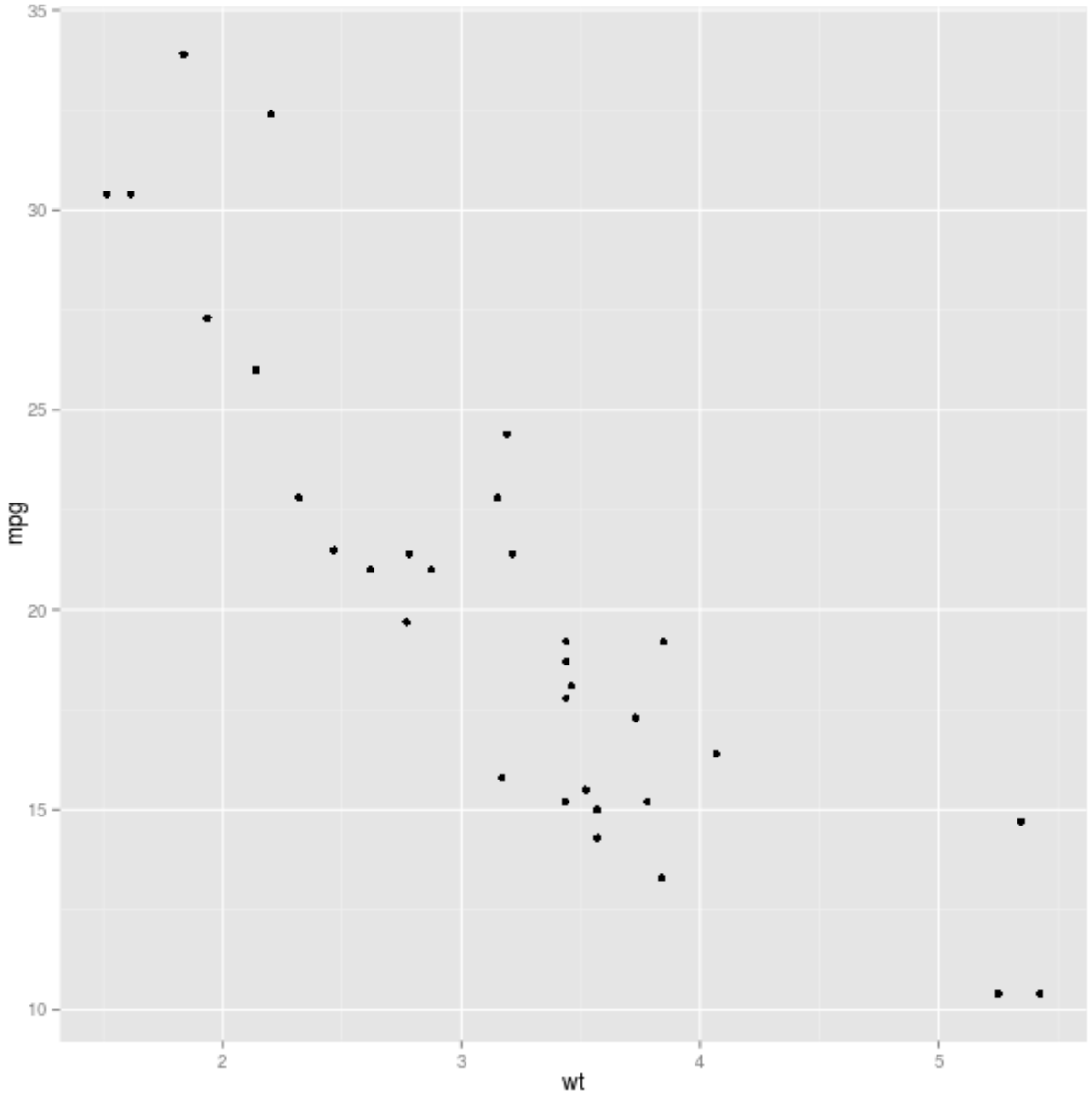
The package is using the *grid* lower-level plotting infrastructure, that can be accessed through the module**rpy2.robjects.lib.grid**. Whenever separate plots on the same device, or arbitrary graphical elements overlaid, or significant plot customization, or editing, are needed some knowledge of *grid* will be required.

Here again, having data in a **DataFrame** is expected (see*DataFrame* for more information on such objects).

```
import math, datetime
import rpy2.robjects.lib.ggplot2 as ggplot2
import rpy2.robjects as ro
from rpy2.robjects.packages import importr
base = importr('base')

mtcars = datasets.__rdata__.fetch('mtcars')['mtcars']
rnorm = stats.rnorm
dataf_rnorm = robjects.DataFrame({'value': rnorm(300, mean=0) + rnorm(100,
mean=3),
                                  'other_value': rnorm(300, mean=0) + rnorm(100,
mean=3),
                                  'mean': IntVector([0, ]*300 + [3, ] * 100)})
```

## Plot

```
gp = ggplot2.ggplot(mtcars)

pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point()

pp.plot()
```



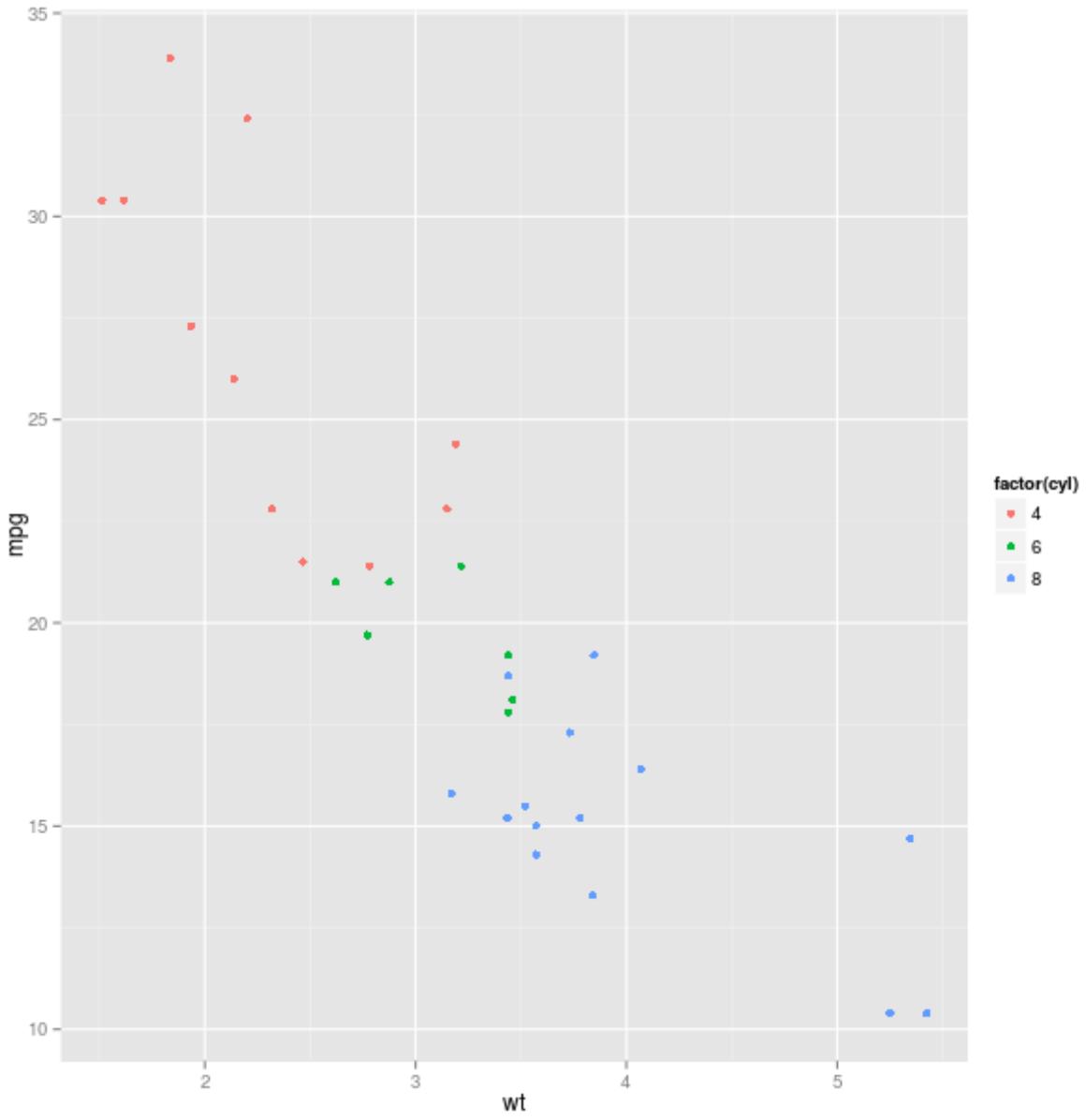

## Aesthethics mapping

An important concept for the grammar of graphics is the mapping of variables, or columns in a data frame, to graphical representations.

Like it was shown for *lattice*, a third variable can be represented on the same plot using color encoding, and this is now done by specifying that as a mapping (the parameter*col* when calling the constructor for the `AesString`).

```
gp = ggplot2.ggplot(mtcars)

pp = gp + \
```

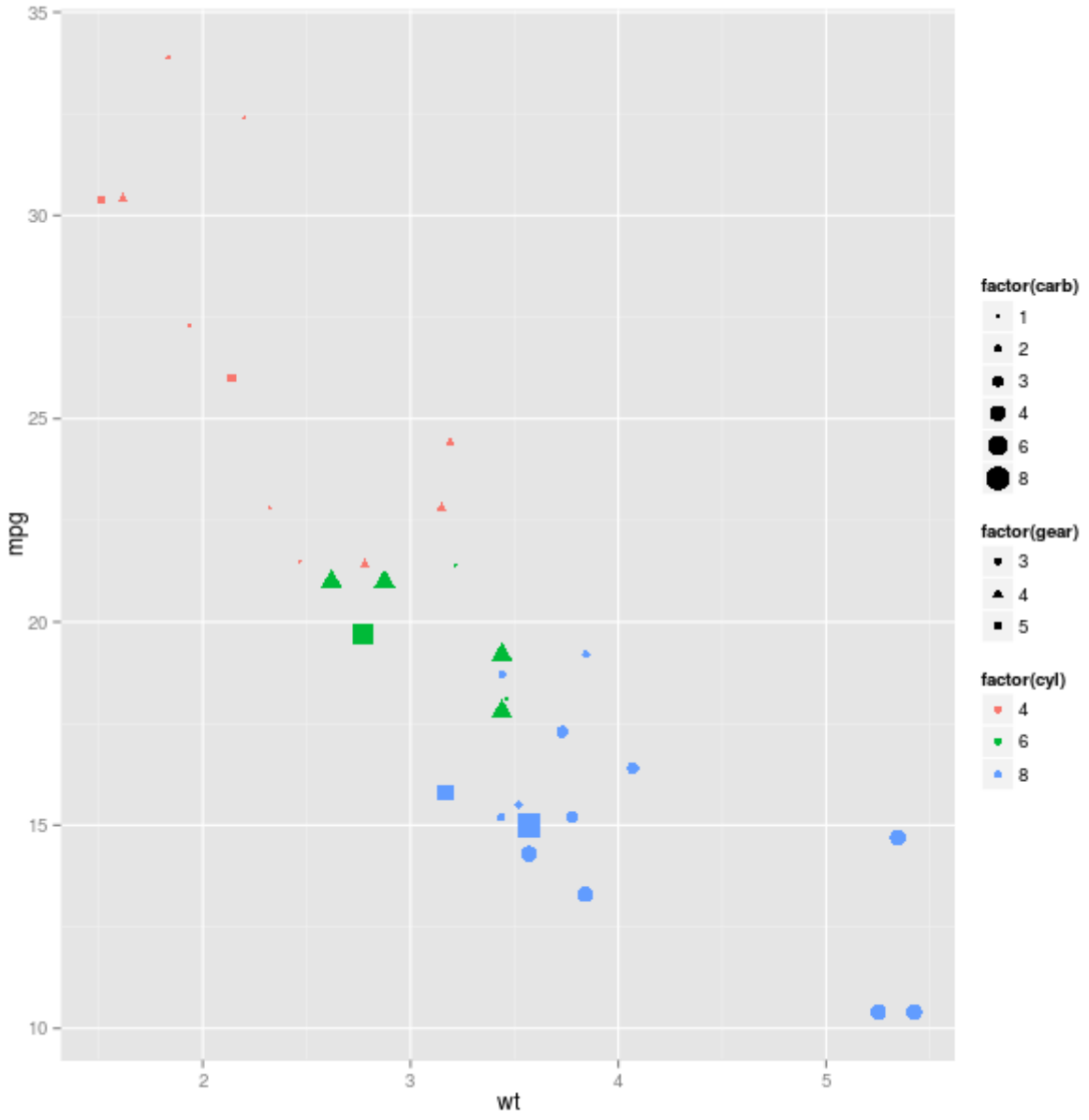
```
    ggplot2.aes_string(x='wt', y='mpg', col='factor(cyl)') + \
    ggplot2.geom_point()

pp.plot()
```




The size of the graphical symbols plotted (here circular dots) can also be mapped to a variable:

```
pp = gp + \
    ggplot2.aes_string(x='wt', y='mpg', size='factor(carb)',
                       col='factor(cyl)', shape='factor(gear)') + \
    ggplot2.geom_point()
```

```
pp.plot()
```



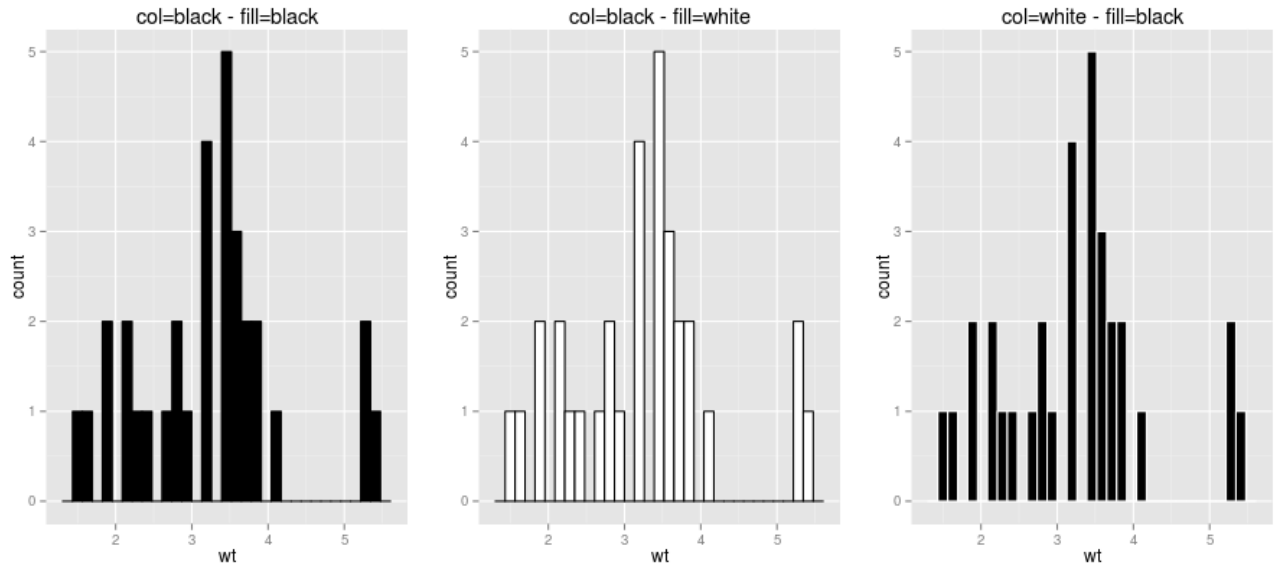

## Geometry

The *geometry* is how the data are represented. So far we used a scatter plot of points, but there are other ways to represent our data.

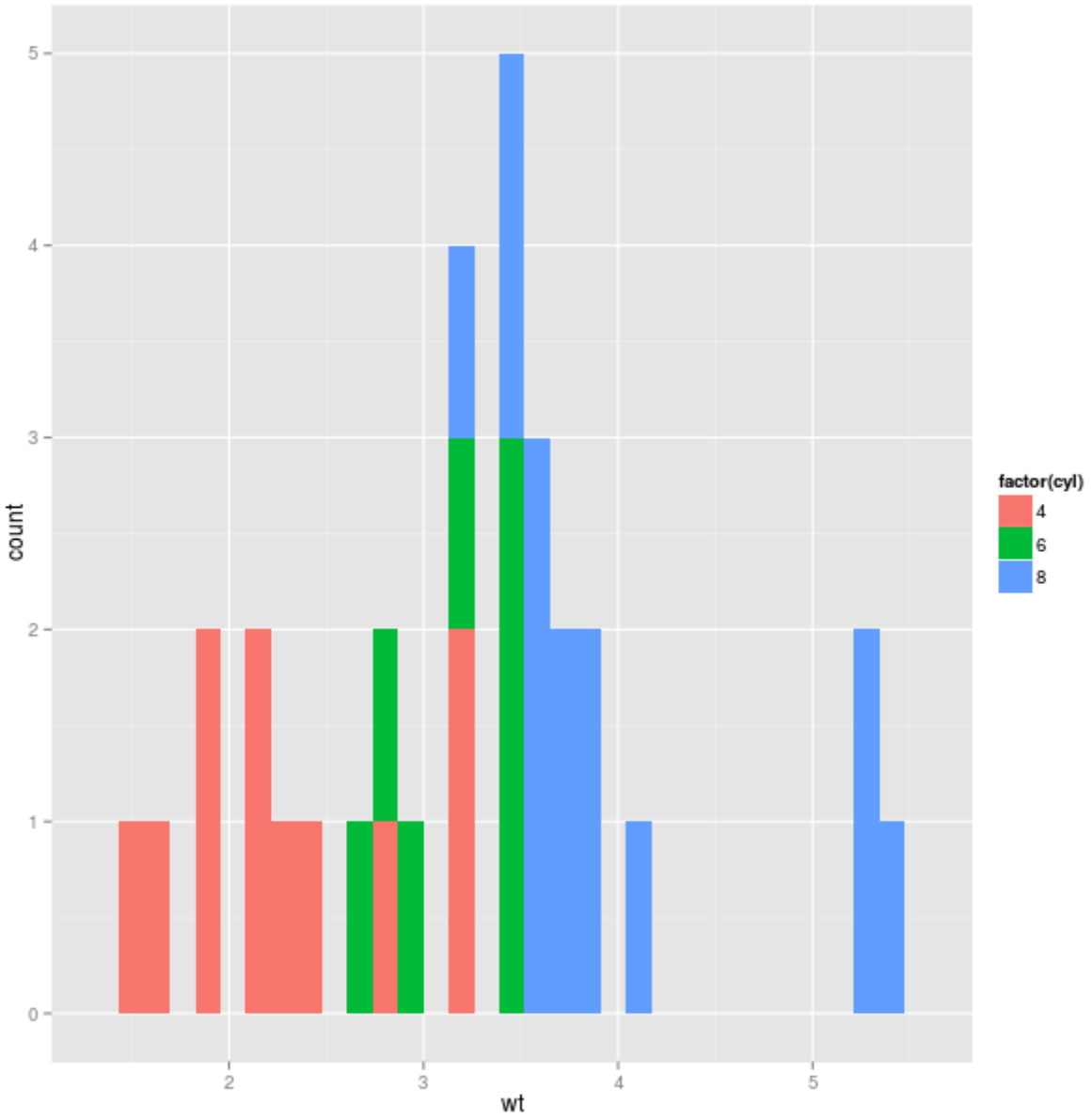
Looking at the distribution of univariate data can be achieved with an histogram:

```
gp = ggplot2.ggplot(mtcars)

pp = gp + \
```

```
    ggplot2.aes_string(x='wt') + \
    ggplot2.geom_histogram()

#pp.plot()
```

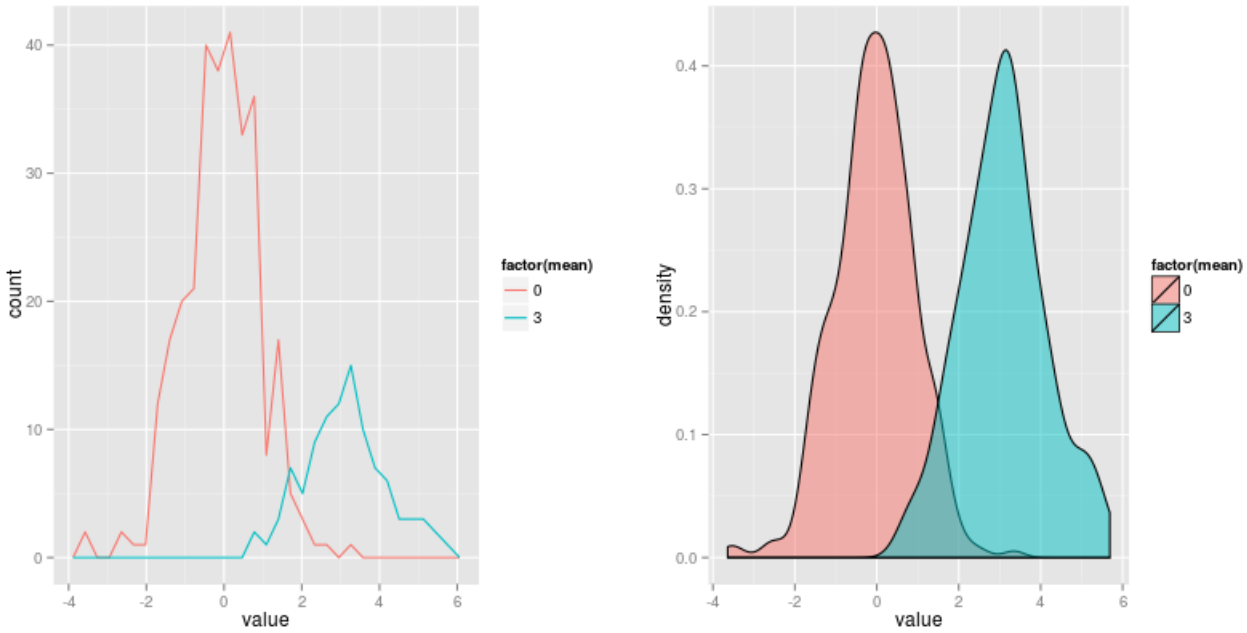



```
gp = ggplot2.ggplot(mtcars)

pp = gp + \
    ggplot2.aes_string(x='wt', fill='factor(cyl)') + \
    ggplot2.geom_histogram()

pp.plot()
```




Barplot-based representations of several densities on the same figure can often be lacking clarity and line-based representation, either `geom_freqpoly()` (representation of the frequency as broken lines) or `geom_density()` (plot a density estimate), can be in better.

```
pp = gp + \
     ggplot2.aes_string(x='value', fill='factor(mean)') + \
     ggplot2.geom_density(alpha = 0.5)
```




Whenever a large number of points are present, it can become interesting to represent the density of “dots” on the scatterplot.

With 2D bins:

```
gp = ggplot2.ggplot(dataf_rnorm)

pp = gp + \
     ggplot2.aes_string(x='value', y='other_value') + \
     ggplot2.geom_bin2d() + \
     ggplot2.ggtitle('geom_bin2d')
pp.plot(vp = vp)
```

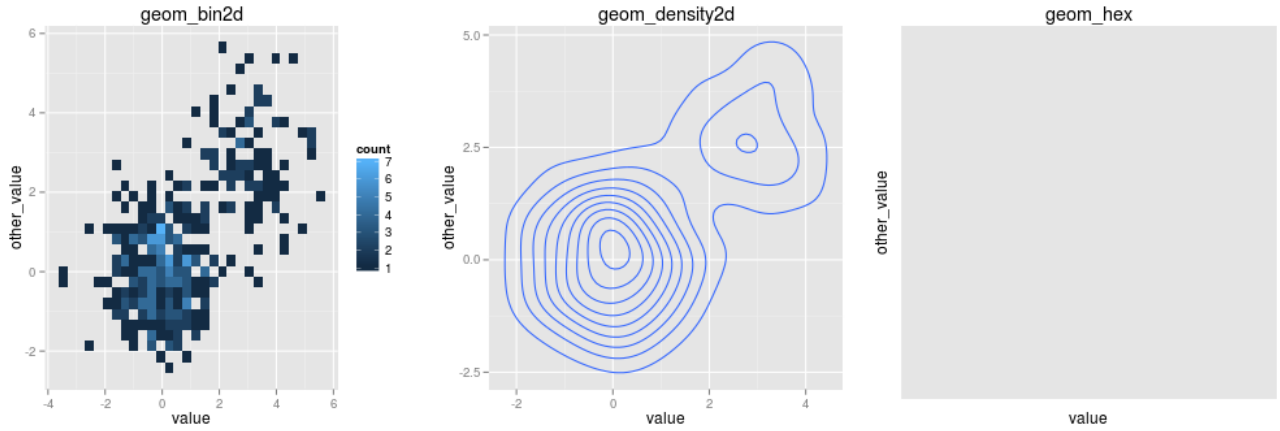
With a kernel density estimate:

```
gp = ggplot2.ggplot(dataf_rnorm)

pp = gp + \
     ggplot2.aes_string(x='value', y='other_value') + \
     ggplot2.geom_density2d() + \
     ggplot2.ggtitle('geom_density2d')
pp.plot(vp = vp)
```

With hexagonal bins:

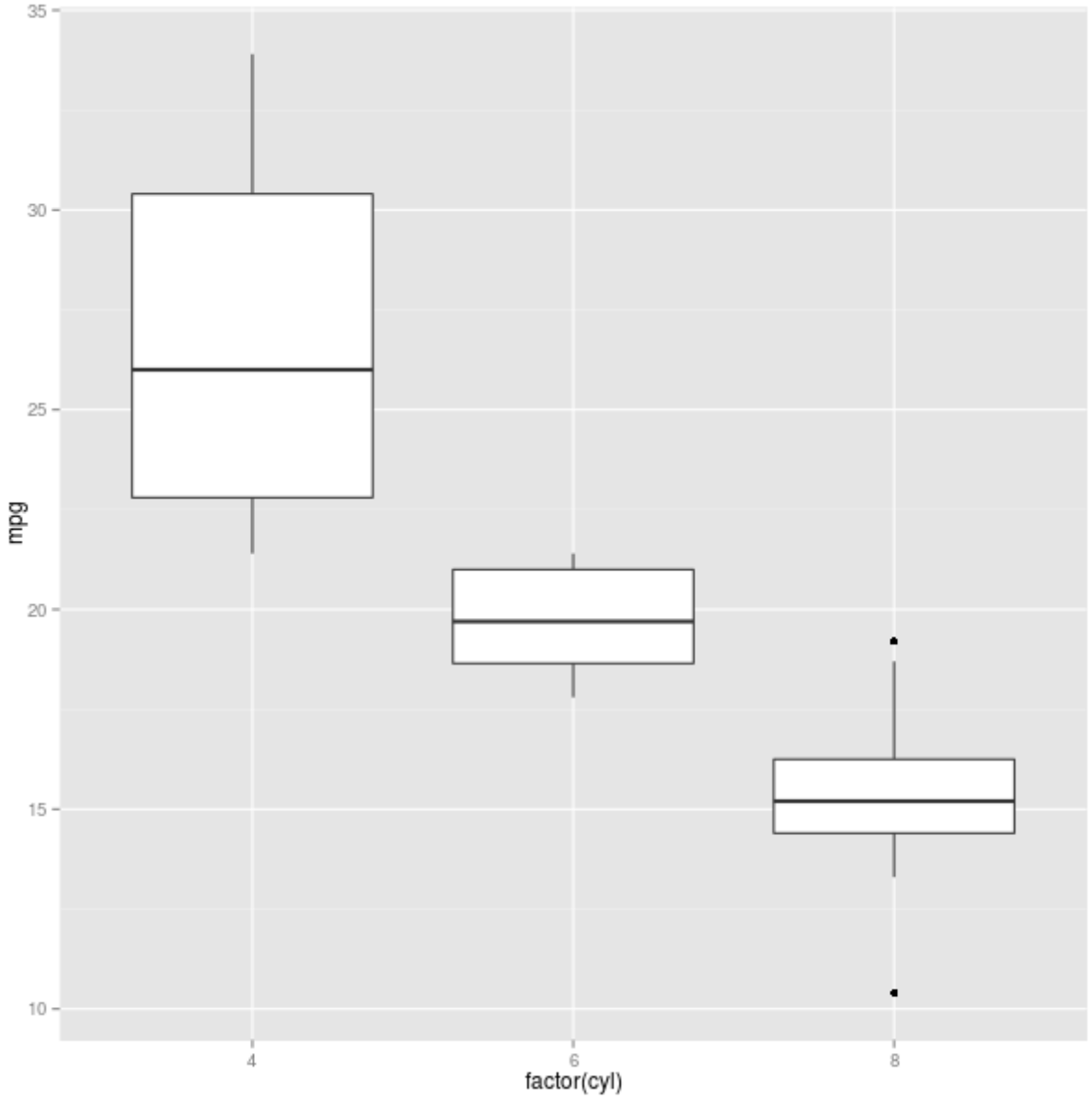
```
gp = ggplot2.ggplot(dataf_rnorm)
```

```
pp = gp + \
     ggplot2.aes_string(x='value', y='other_value') + \
     ggplot2.geom_hex() + \
     ggplot2.ggtitle('geom_hex')
pp.plot(vp = vp)
```



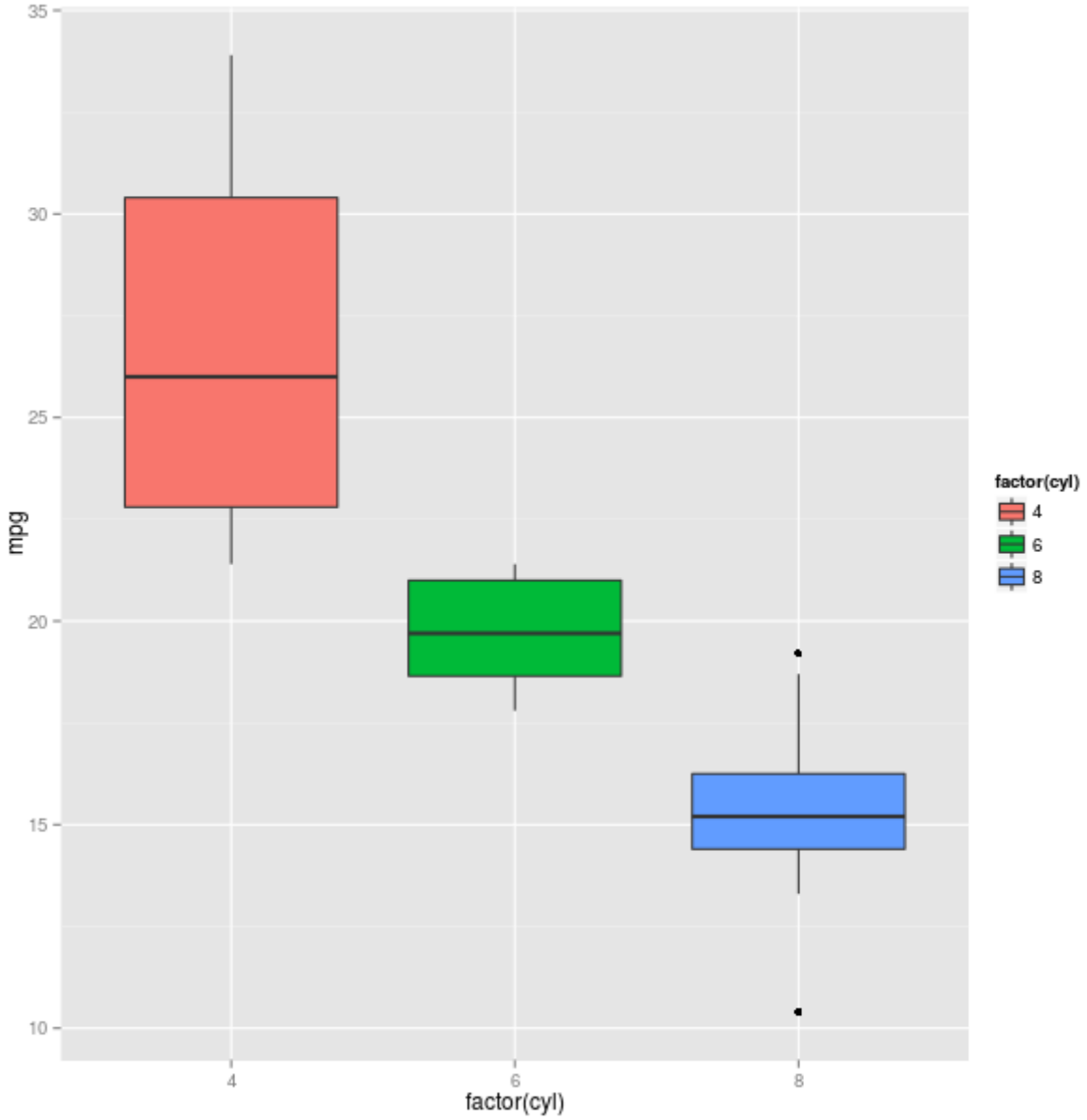

Box plot:

```
gp = ggplot2.ggplot(mtcars)

pp = gp + \
     ggplot2.aes_string(x='factor(cyl)', y='mpg') + \
     ggplot2.geom_boxplot()

pp.plot()
```



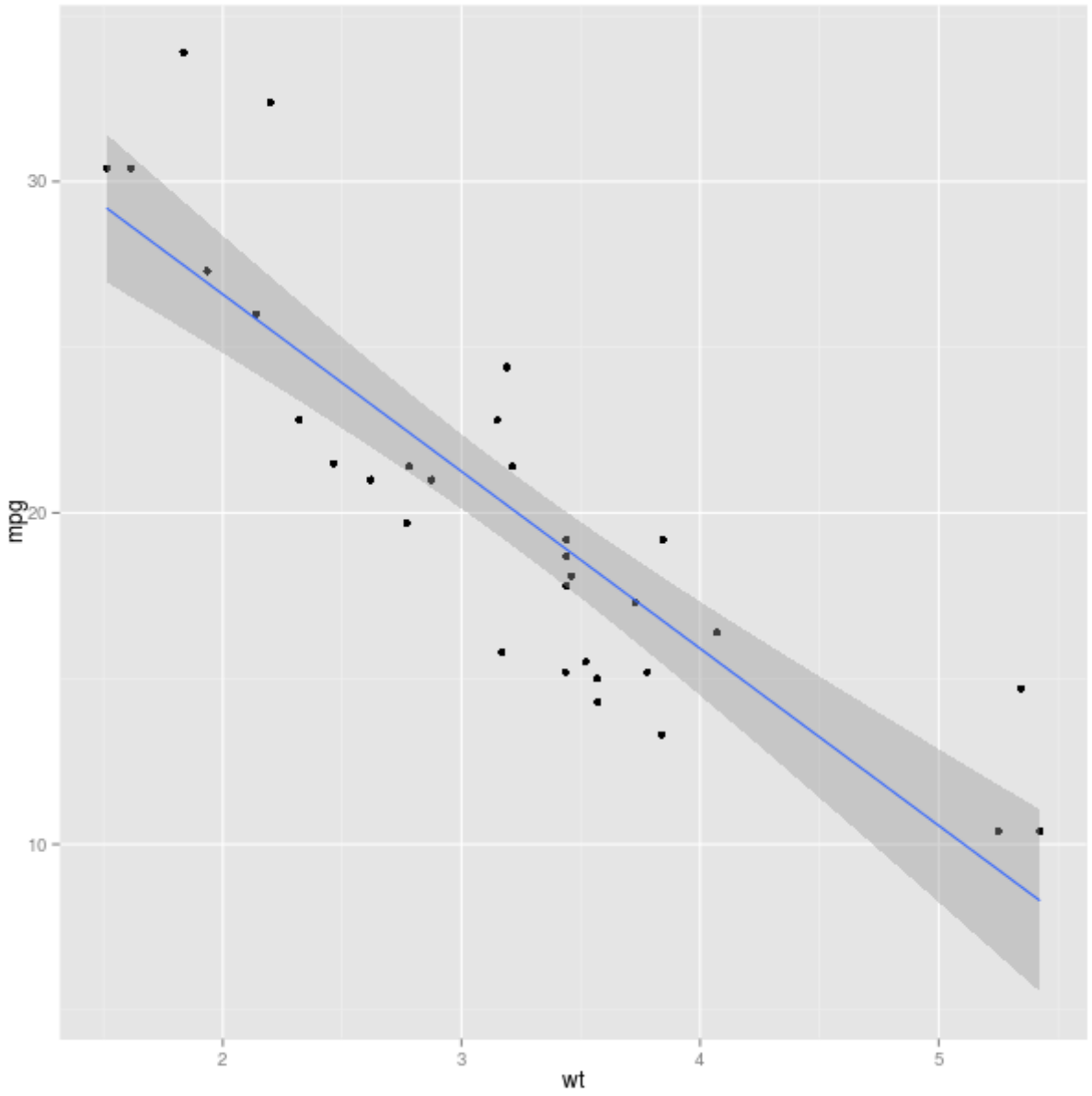

Boxplots can be used to represent a *summary* of the data with an emphasis on location and spread.

```
gp = ggplot2.ggplot(mtcars)

pp = gp + \
     ggplot2.aes_string(x='factor(cyl)', y='mpg', fill='factor(cyl)') + \
     ggplot2.geom_boxplot()

pp.plot()
```

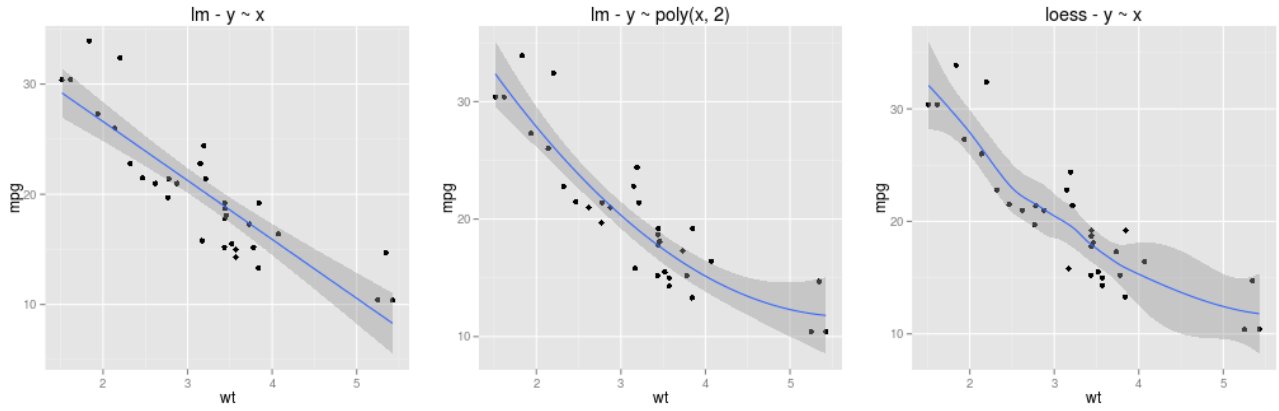



Models fitted to the data are also easy to add to a plot:

```
pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point() + \
     ggplot2.stat_smooth(method = 'lm')
pp.plot()
```

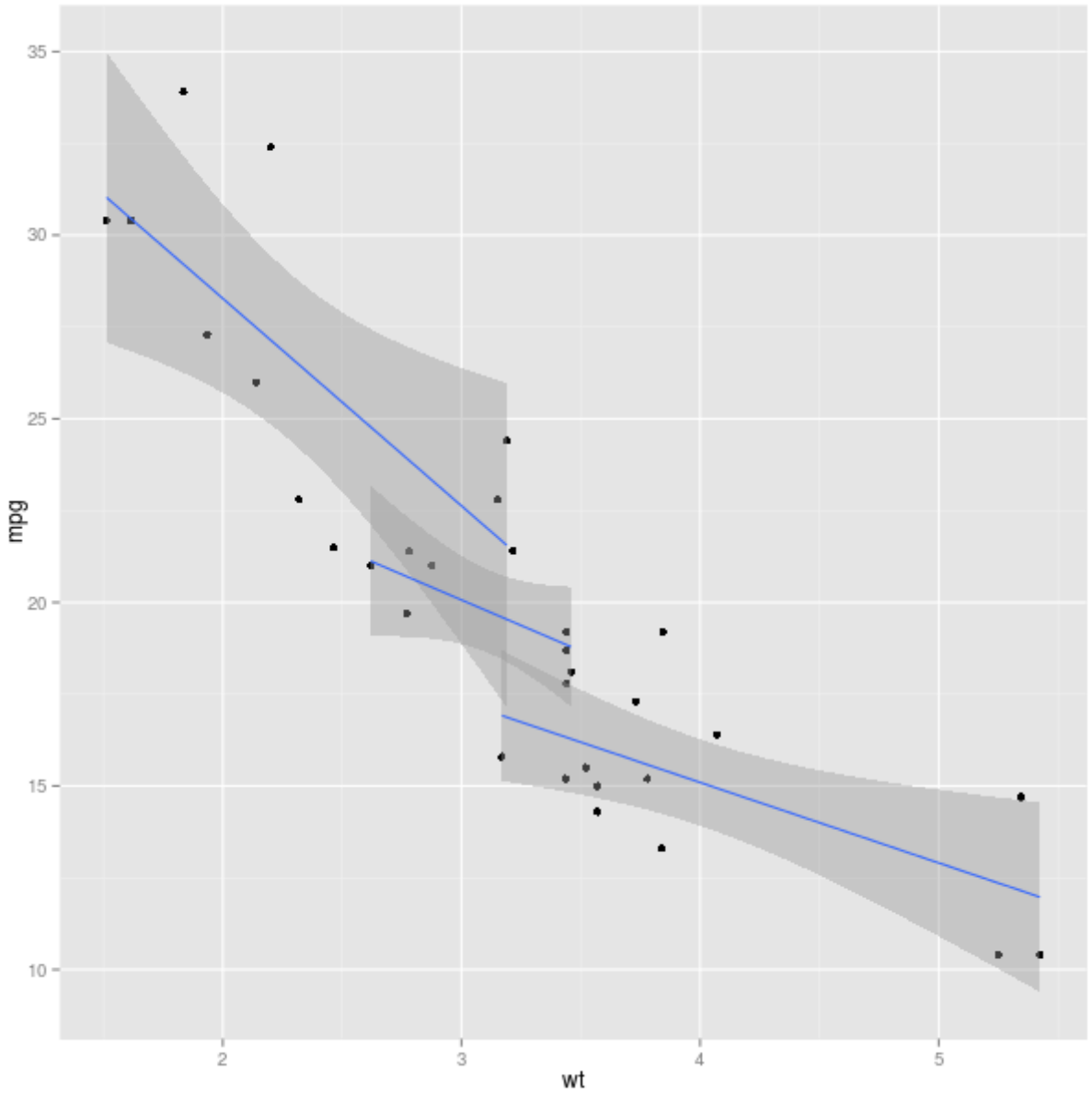



The *method* can be one of {*glm*, *gam*, *loess*, *rlm*}, and formula can be specified to declared the fitting (see example below).



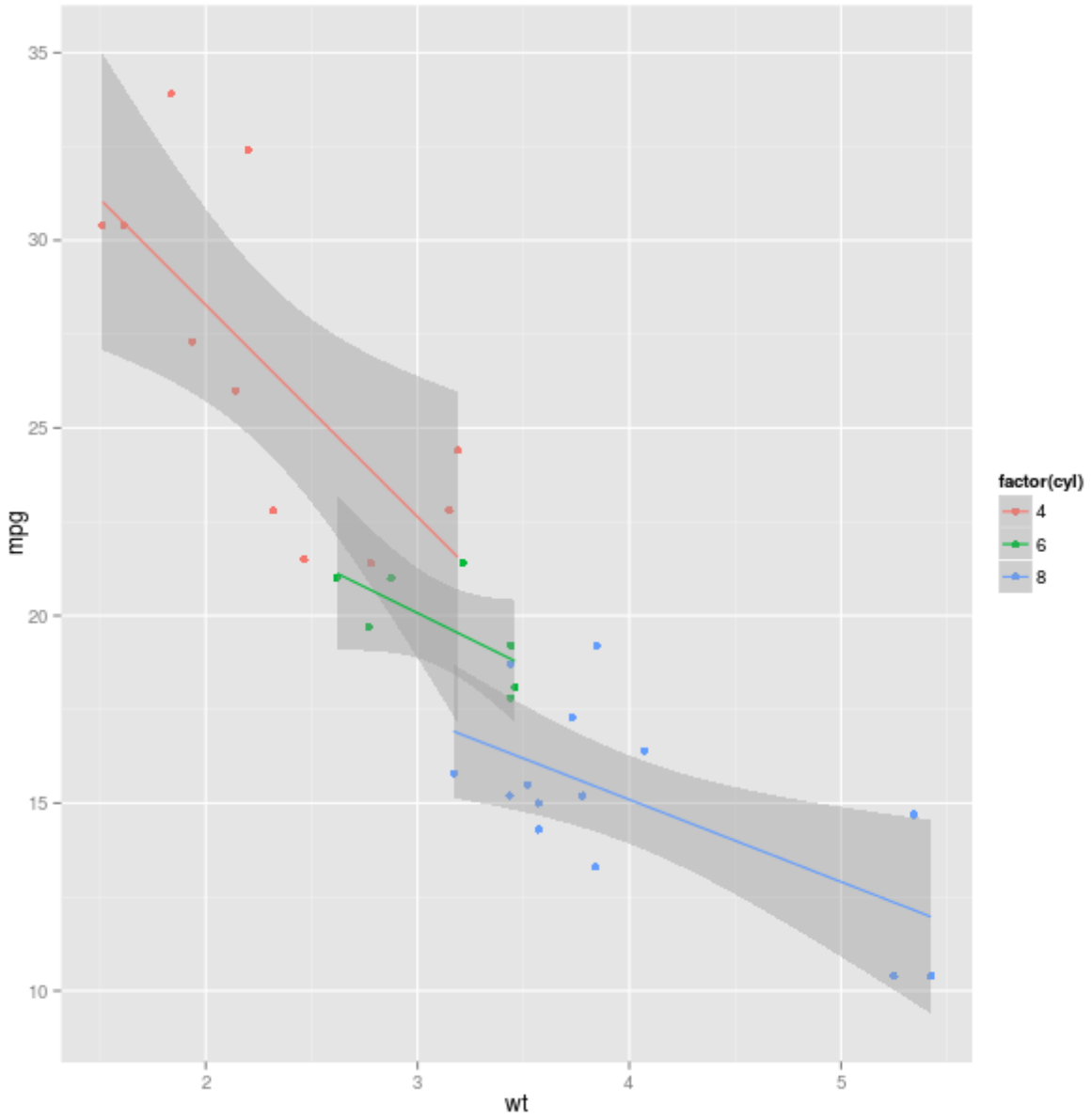

The constructor for `GeomSmooth` also accepts a parameter*groupr* that indicates if the fit should be done according to groups.

```
pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point() + \
     ggplot2.geom_smooth(ggplot2.aes_string(group = 'cyl'),
                         method = 'lm')
pp.plot()
```



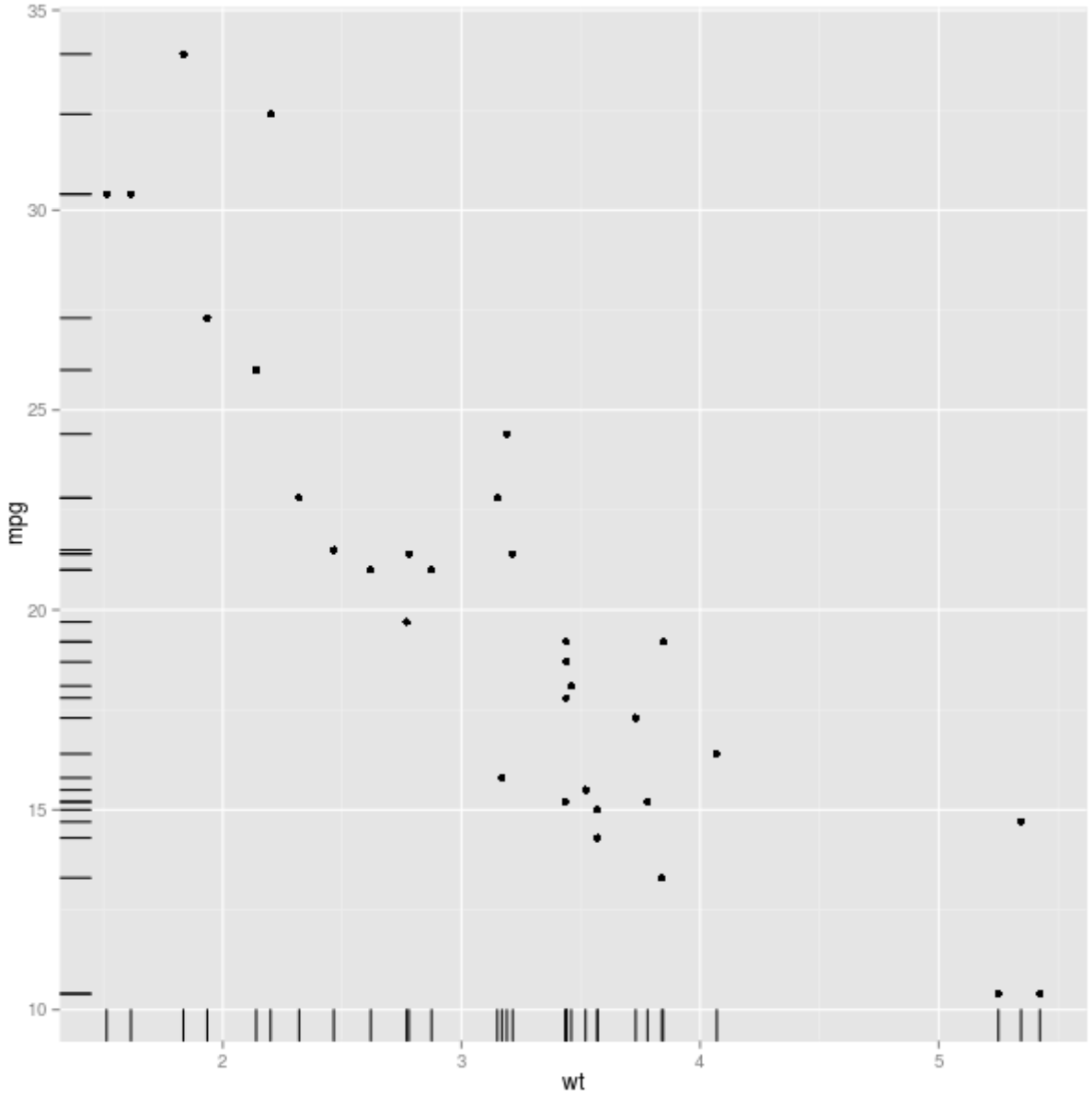

Encoding the information in the column *cyl* is again only a matter of specifying it in the `AesString` mapping.

```
pp = ggplot2.ggplot(mtcars) + \
     ggplot2.aes_string(x='wt', y='mpg', col='factor(cyl)') + \
     ggplot2.geom_point() + \
     ggplot2.geom_smooth(ggplot2.aes_string(group = 'cyl'),
                         method = 'lm')
pp.plot()
```

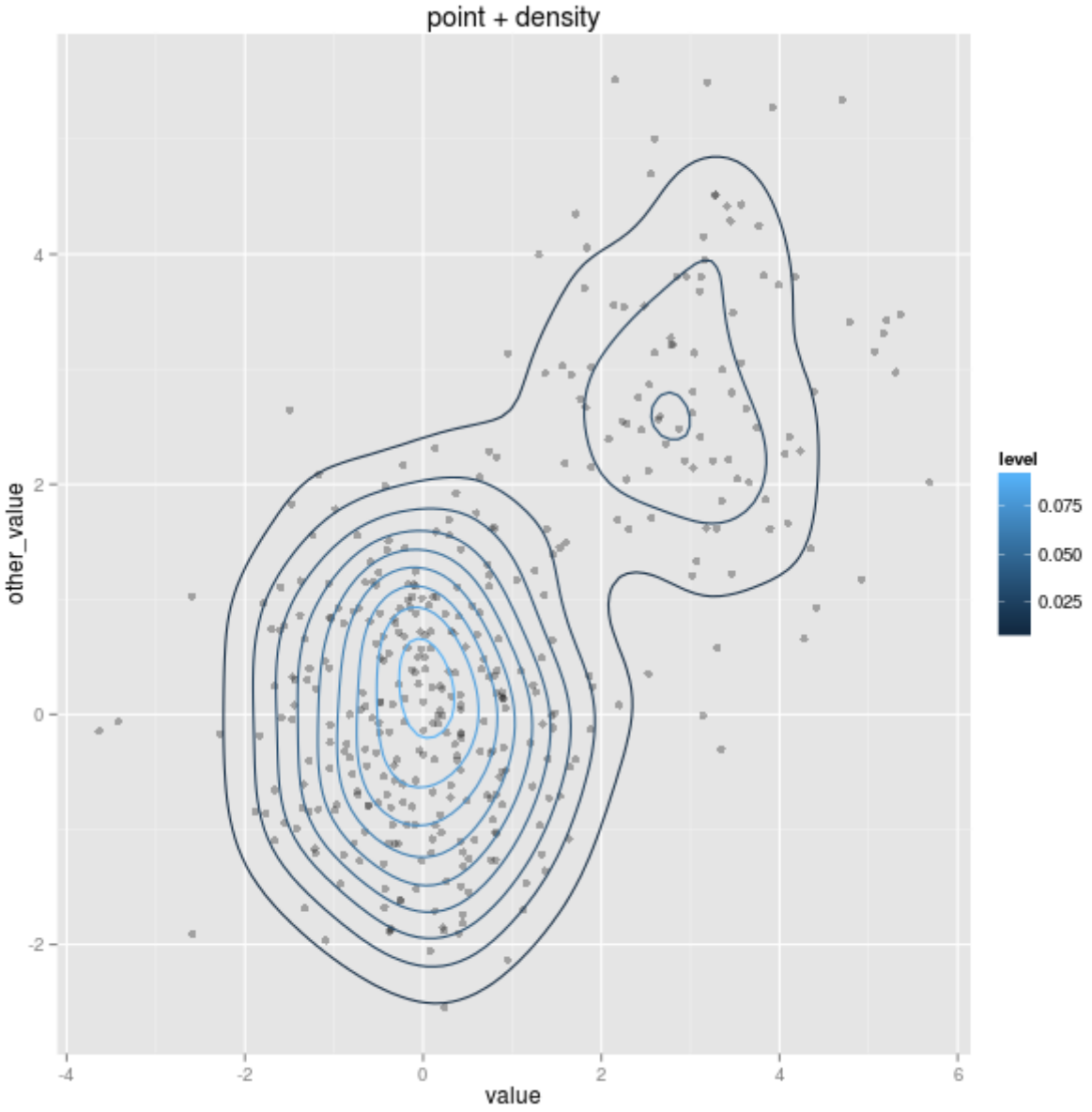



As can already be observed in the examples with `GeomSmooth`, several *geometry* objects can be added on the top of each other in order to create the final plot. For example, a marginal *rug* can be added to the axis of a regular scatterplot:

```
gp = ggplot2.ggplot(mtcars)

pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point() + \
     ggplot2.geom_rug()

pp.plot()
```

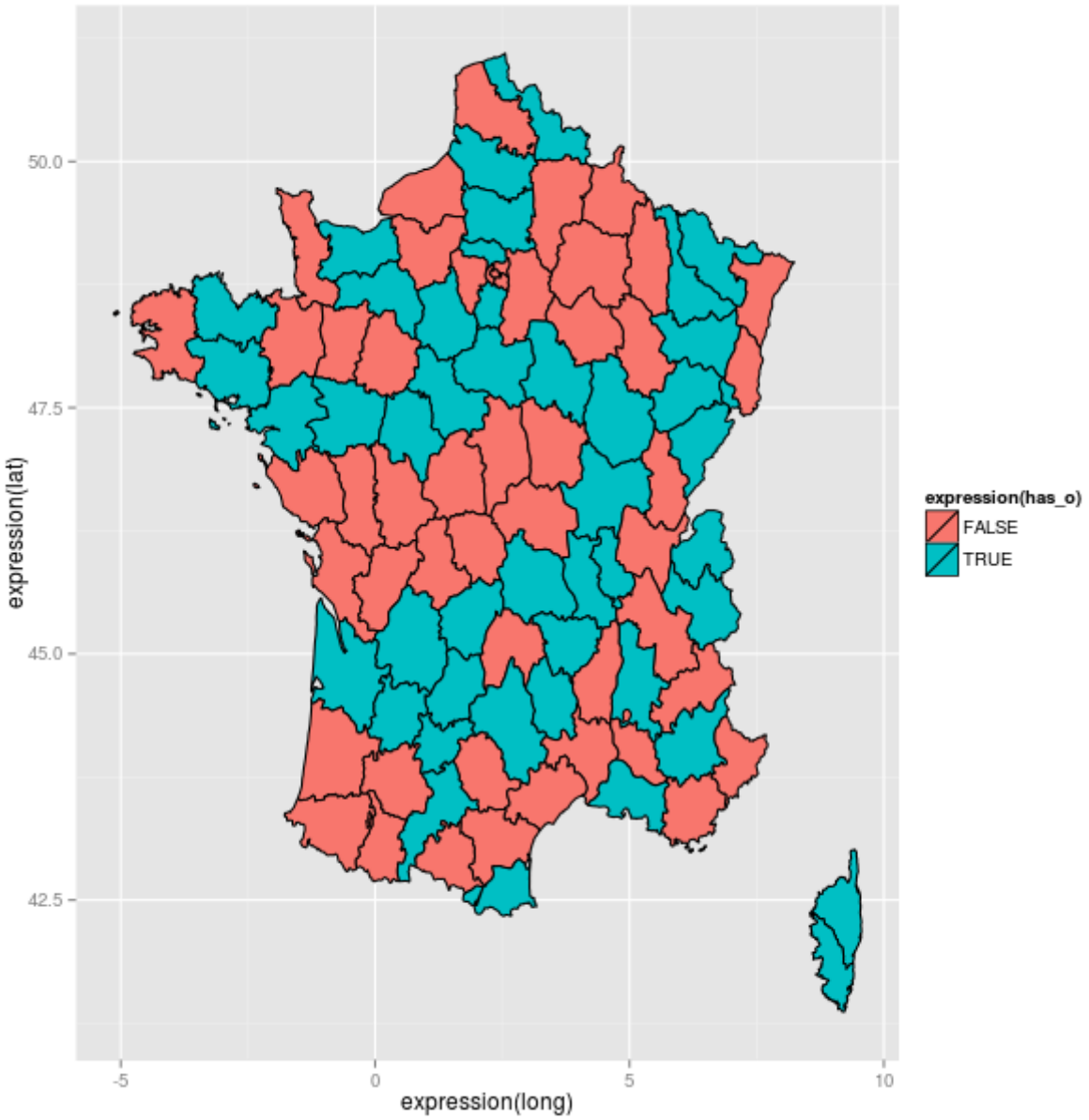



```
gp = ggplot2.ggplot(dataf_rnorm)

pp = gp + \
     ggplot2.aes_string(x='value', y='other_value') + \
     ggplot2.geom_point(alpha = 0.3) + \
     ggplot2.geom_density2d(ggplot2.aes_string(col = '..level..')) + \
     ggplot2.ggtitle('point + density')
pp.plot()
```




Polygons can be used for maps, as shown in the relatively artificial example below:

```
map = importr('maps')
fr = ggplot2.map_data('france')

# add a column indicating which region names have an "o".
fr = fr.cbind(fr, has_o = base.grepl('o', fr.rx2("region"),
                                     ignore_case = True))
p = ggplot2.ggplot(fr) + \
    ggplot2.geom_polygon(ggplot2.aes(x = 'long', y = 'lat',
                                     group = 'group', fill = 'has_o'),
                         col="black")
p.plot()
```



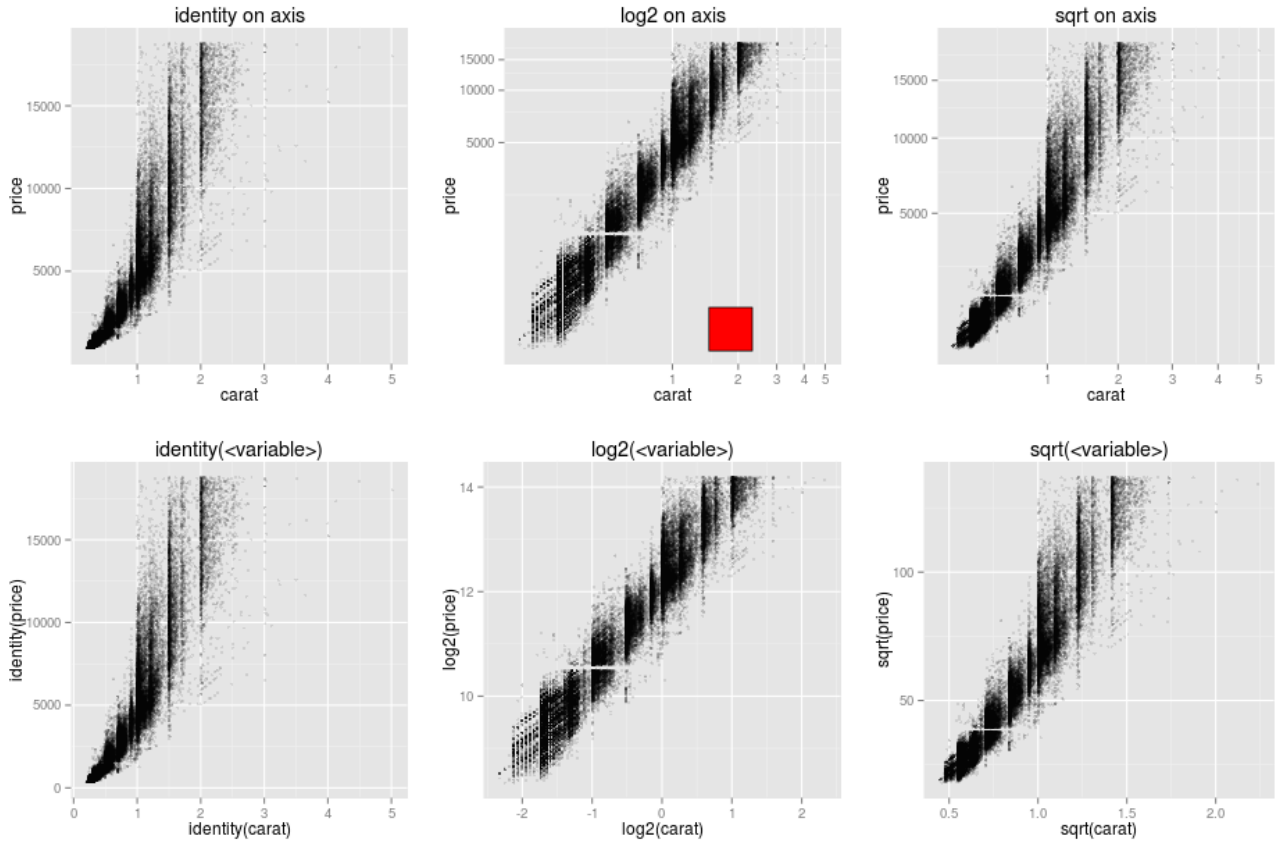

## xes

Axes can be transformed and configured in various ways.

A common transformation is the log-transform of the coordinates.

```
from rpy2.robjects.lib import grid
grid.newpage()
grid.viewport(layout=grid.layout(2, 3)).push()

diamonds = ggplot2.ggplot2.__rdata__.fetch('diamonds')['diamonds']
gp = ggplot2.ggplot(diamonds)
```

```
for col_i, trans in enumerate(("identity", "log2", "sqrt")):
    # fetch viewport at position col_i+1 on the first row
    vp = grid.viewport(**{'layout.pos.col':col_i+1, 'layout.pos.row': 1})
    pp = gp + \
        ggplot2.aes_string(x='carat', y='price') + \
        ggplot2.geom_point(alpha = 0.1, size = 1) + \
        ggplot2.coord_trans(x = trans, y = trans) + \
        ggplot2.ggtitle("%s on axis" % trans)
    # plot into the viewport
    pp.plot(vp = vp)

    # fetch viewport at position col_i+1 on the second row
    vp = grid.viewport(**{'layout.pos.col':col_i+1, 'layout.pos.row': 2})
    pp = gp + \
        ggplot2.aes_string(x='%s(carat)' % trans, y='%s(price)' % trans) + \
        ggplot2.geom_point(alpha = 0.1, size = 1) + \
        ggplot2.ggtitle("%s(<variable>)" % trans)
    pp.plot(vp = vp)
```




**Note**

The red square is an example of adding graphical elements to a ggplot2 figure.

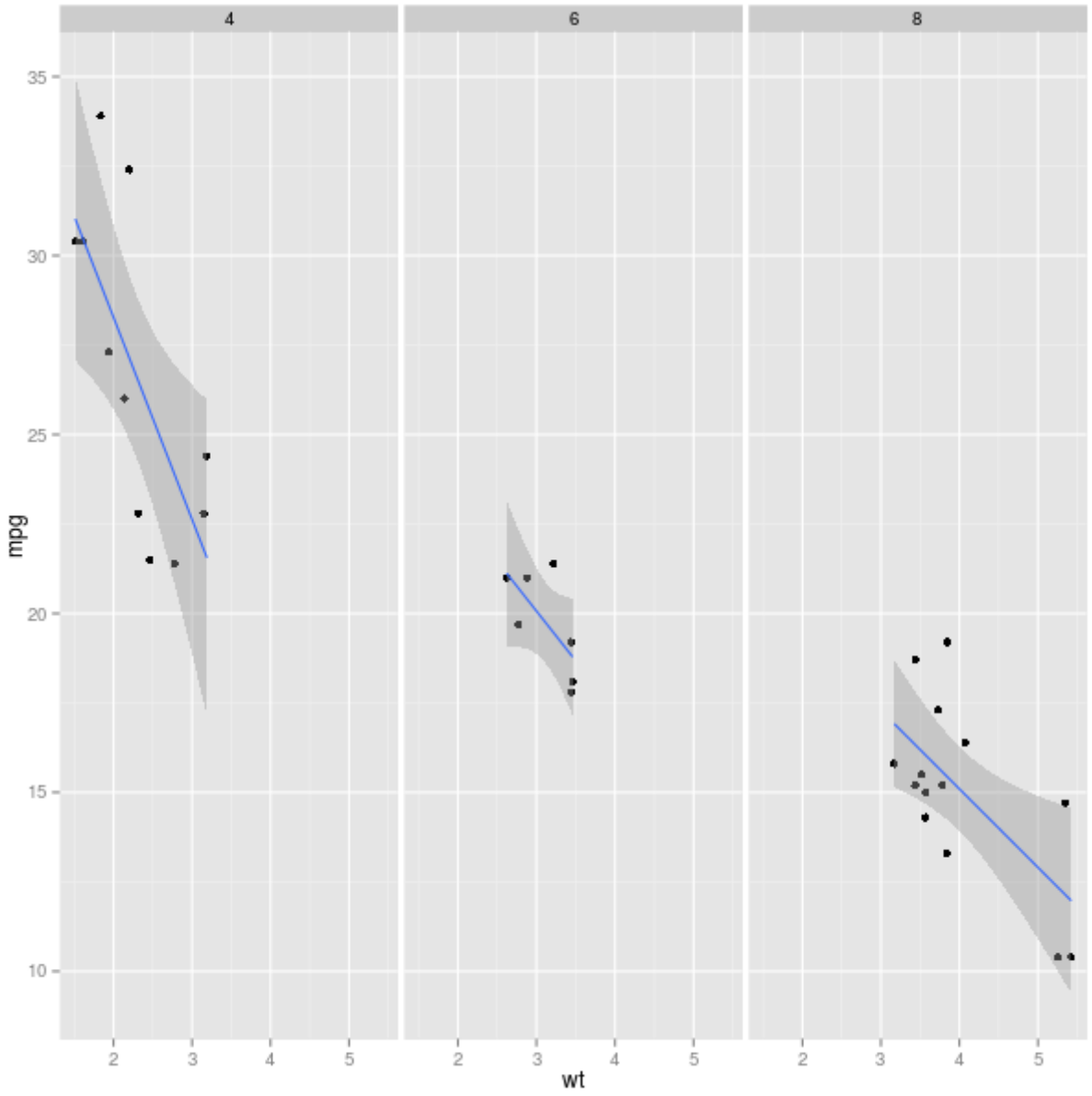
```
vp = grid.viewport(**{'layout.pos.col':2, 'layout.pos.row': 1})
grid.rect(x = grid.unit(0.7, "npc"),
          y = grid.unit(0.2, "npc"),
          width = grid.unit(0.1, "npc"),
          height = grid.unit(0.1, "npc"),
          gp = grid.gpar(fill = "red"),
          vp = vp).draw()
```

## Facets

Splitting the data into panels, in a similar fashion to what we did with *lattice*, is now a matter of adding *facets*. A central concept to *ggplot2* is that plot are made of added graphical elements, and adding specifications such as “I want my data to be split in panel” is then a matter of adding that information to an existing plot.

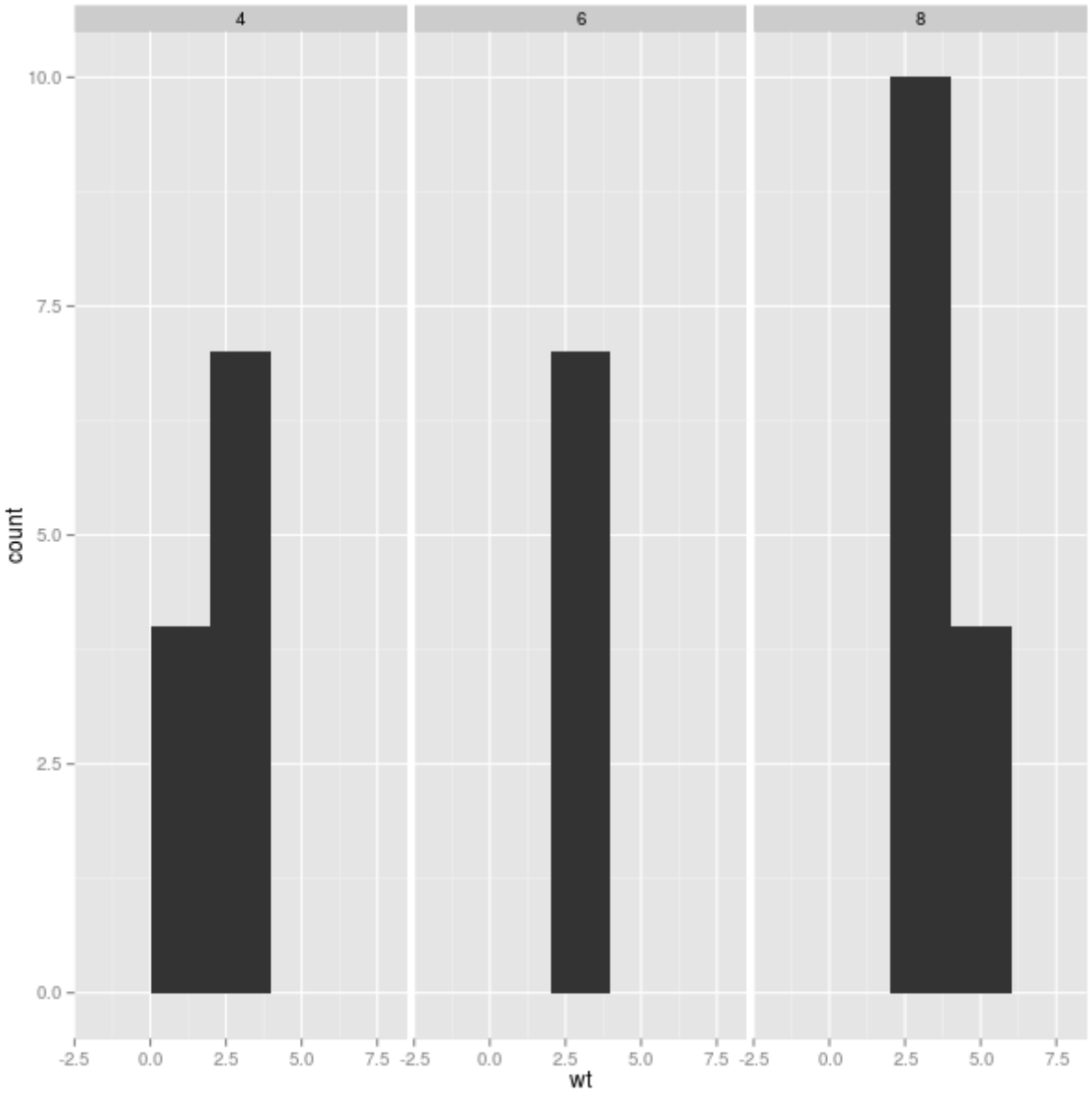
For example, splitting the plots on the data in column *cyl*is still simply done by adding a `FacetGrid`.

```
pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point() + \
     ggplot2.facet_grid(ro.Formula('. ~ cyl')) + \
     ggplot2.geom_smooth(ggplot2.aes_string(group="cyl"),
                         method = "lm",
                            data = mtcars)

pp.plot()
```

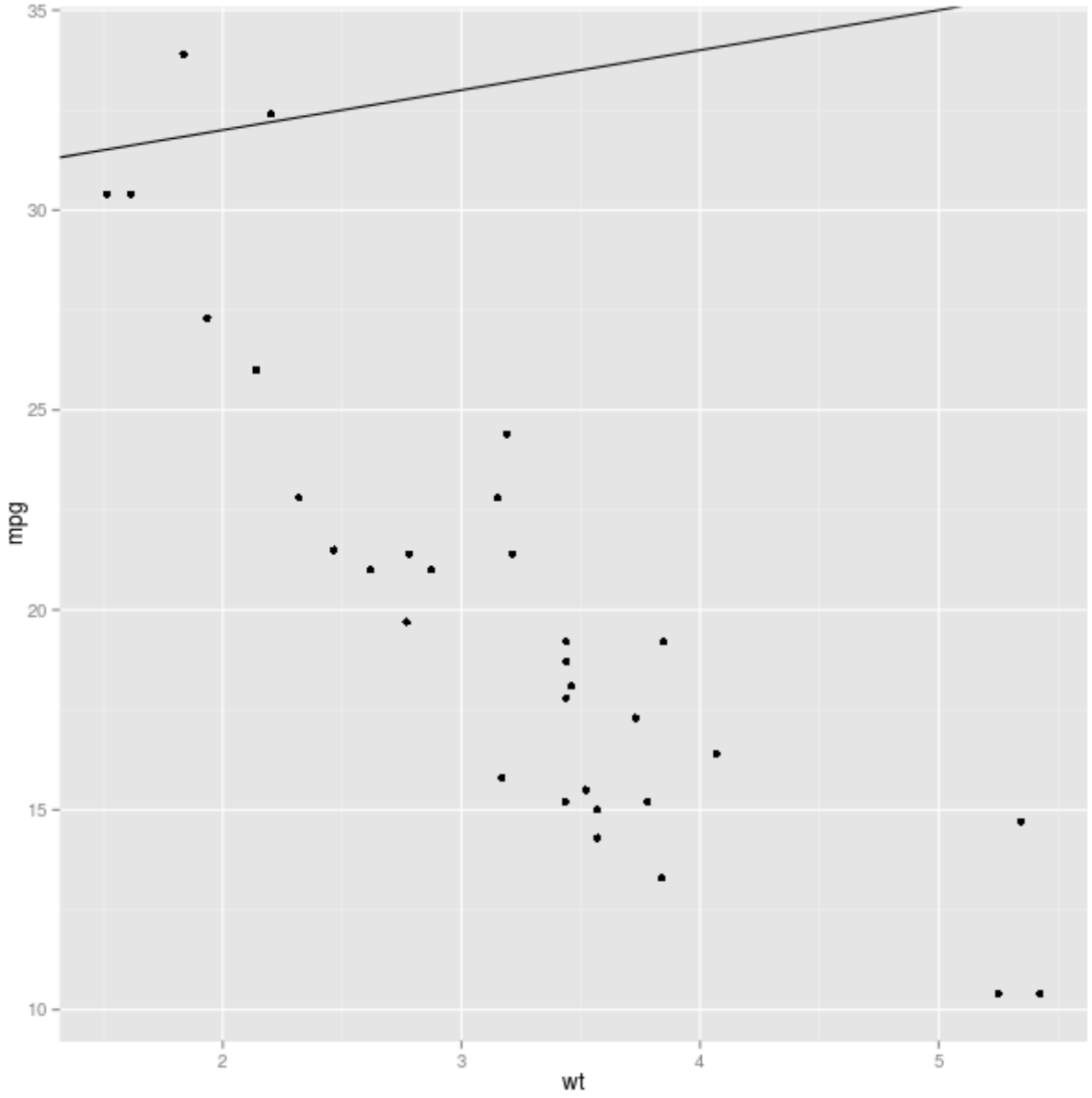



The way data are represented (the *geometry* in the terminology used the grammar of graphics) are still specified the usual way.

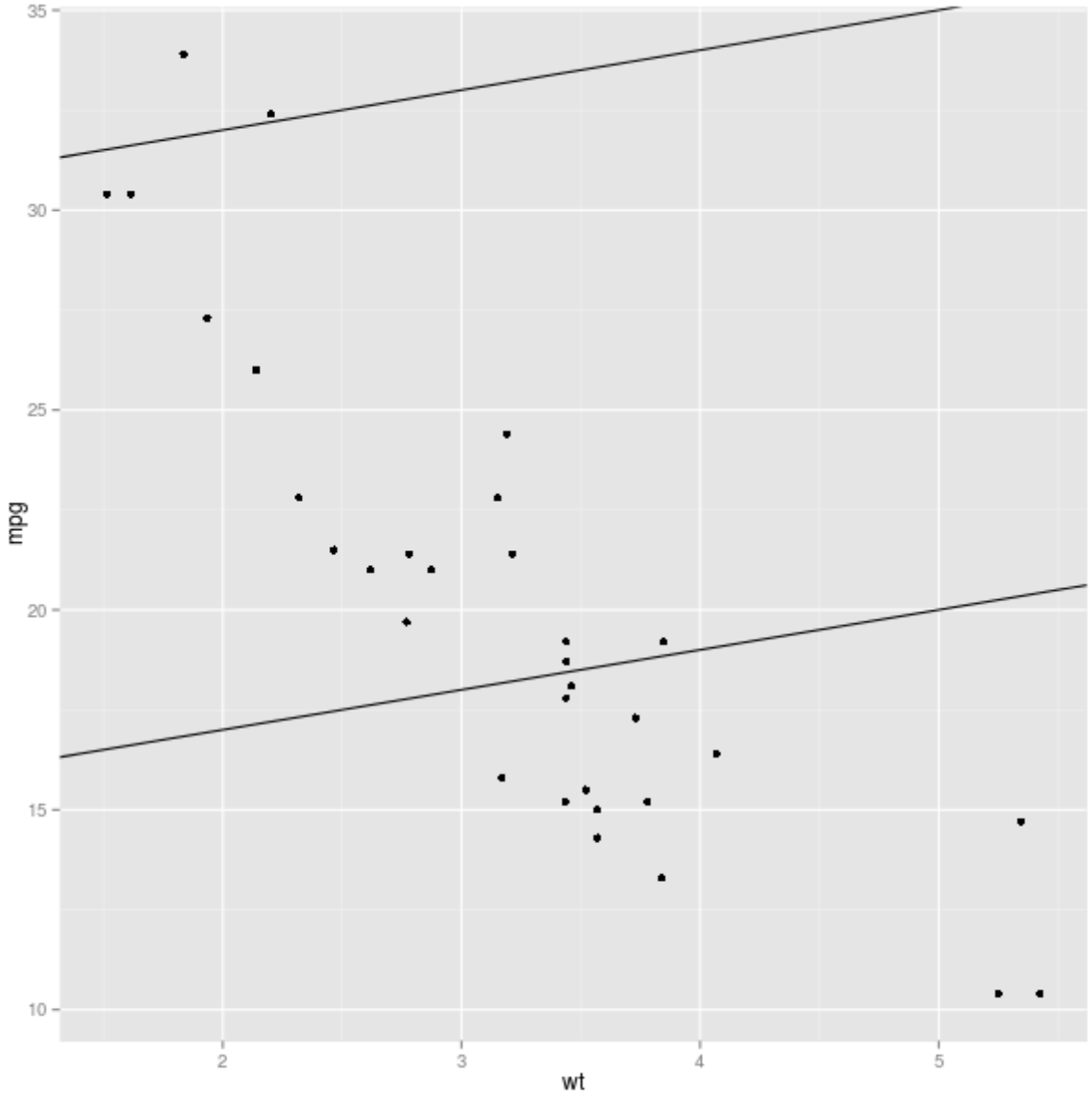
```
pp = gp + \
     ggplot2.aes_string(x='wt') + \
     ggplot2.geom_histogram(binwidth=2) + \
     ggplot2.facet_grid(ro.Formula('. ~ cyl'))

pp.plot()
```



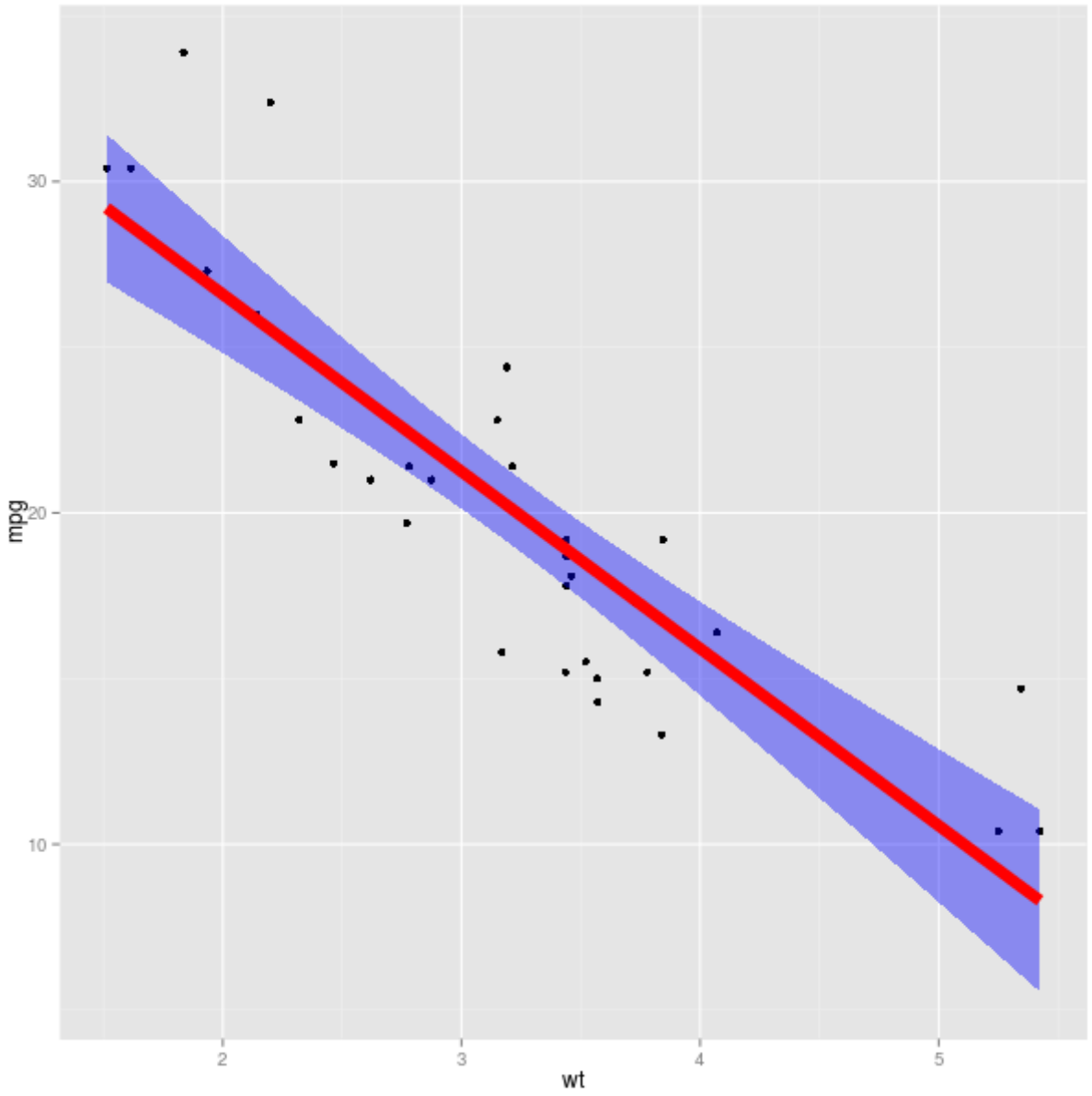

```
pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point() + \
     ggplot2.geom_abline(intercept = 30)
pp.plot()
```




```
pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point() + \
     ggplot2.geom_abline(intercept = 30) + \
     ggplot2.geom_abline(intercept = 15)
pp.plot()
```




```
pp = gp + \
     ggplot2.aes_string(x='wt', y='mpg') + \
     ggplot2.geom_point() + \
     ggplot2.stat_smooth(method = 'lm', fill = 'blue',
                         color = 'red', size = 3)
pp.plot()
```




Wrapper for the popular R library ggplot2.

With rpy2, the most convenient general way to import packages is to use *importr()*, for example with ggplot2:

```
from rpy2.robjects.packages import importr
ggplot2 = importr('ggplot2')
```

This module is an supplementary layer in which an attempt at modelling the package as it was really developed as Python package is made. Behind the scene, *importr()* is used and can be accessed with:

from robjects.robjects.lib import ggplot2 ggplot2.ggplot2

GGplot2 is designed using a prototype-based approach to Object-Oriented Programming, and this module is trying to define class-hierachies so the nature of a given instance can be identified more easily.

The main families of classes are:

- GGplot
- Aes and AesString
- Layer
- Stat

A downside of the approach is that the code in the module is 'hand-made'. In hindsight, this can be tedious to maintain and document but this is a good showcase of "manual" mapping of R code into Python classes.

The codebase in R for ggplot2 has evolved since this was initially written, and many functions have signature-defined parameters (used to be ellipsis about everywhere). Metaprogramming will hopefully be added to shorten the Python code in the module, and provide a more dynamic mapping.

*class* rpy2.robjects.lib.ggplot2.Aes(*o*)

Aesthetics mapping, using expressions rather than string (this is the most common form when using the package in R - it might be easier to use AesString when working in Python using rpy2 - see class AesString in this Python module).

*classmethod* new(***kwargs*)

Constructor for the class Aes.

*class* rpy2.robjects.lib.ggplot2.AesString(*o*)

Aesthetics mapping, using strings rather than expressions (the later being most common form when using the package in R - see class Aes in this Python module).

This associates dimensions in the data sets (columns in the DataFrame), possibly with a transformation applied on-the-fly (e.g., "log(value)", or "cost / benefits") to graphical "dimensions" in a chosen graphical representation (e.g., x-axis, color of points, size, etc...).

Not all graphical representations have all dimensions. Refer to the documentation of ggplot2, online tutorials, or Hadley's book for more details.

*classmethod* **new**(***kwargs*)

Constructor for the class AesString.

*class* rpy2.robjects.lib.ggplot2.**Coord**

Coordinates

*class* rpy2.robjects.lib.ggplot2.**CoordCartesian**

Cartesian coordinates.

*class* rpy2.robjects.lib.ggplot2.**CoordEqual**

This class seems to be identical to CoordFixed.

*class* rpy2.robjects.lib.ggplot2.**CoordFixed**

Cartesian coordinates with fixed relationship (that is fixed ratio between units in axes). CoordEquel seems to be identical to this class.

*class* rpy2.robjects.lib.ggplot2.**CoordFlip**

Flip horizontal and vertical coordinates.

*class* rpy2.robjects.lib.ggplot2.**CoordMap**

Map projections.

*class* rpy2.robjects.lib.ggplot2.**CoordPolar**

Polar coordinates.

*class* rpy2.robjects.lib.ggplot2.**CoordTrans**

Apply transformations (functions) to a cartesian coordinate system.

*class* rpy2.robjects.lib.ggplot2.**Facet**

Panels

*class* rpy2.robjects.lib.ggplot2.**FacetGrid**

Panels in a grid.

*class* rpy2.robjects.lib.ggplot2.**FacetWrap**

Sequence of panels in a 2D layout

*class* rpy2.robjects.lib.ggplot2.**GGPlot**

A Grammar of Graphics Plot.

GGPlot instances can be added to one an other in order to construct the final plot (the method *__add__()* is implemented).

*classmethod* **new**(*data*)

Constructor for the class GGplot.

*class* rpy2.robjects.lib.ggplot2.**Layer**

At this level, aesthetics mapping can (should ?) be specified (see Aes and AesString).

*classmethod* **new**(**args*, ***kwargs*)

Constructor for the class Layer.

*class* rpy2.robjects.lib.ggplot2.**Stat**

A “statistical” processing of the data in order to make a plot, or a plot element.

This is an abstract class; material classes are called Stat* (e.g., StatAbline, StatBin, etc...).

*class* rpy2.robjects.lib.ggplot2.**StatAbline**

Line from slope and intercept.

*class* rpy2.robjects.lib.ggplot2.**StatBin**

Bin data.

*class* rpy2.robjects.lib.ggplot2.**StatBin2D**

2D binning of data into squares/rectangles.

*class* rpy2.robjects.lib.ggplot2.**StatBinhex**

2D binning of data into hexagons.

*class* rpy2.robjects.lib.ggplot2.**StatBoxplot**

Components of box and whisker plot.

*class* rpy2.robjects.lib.ggplot2.**StatContour**

Contours of 3D data.

*class* rpy2.robjects.lib.ggplot2.**StatDensity**

1D density estimate

*class* rpy2.robjects.lib.ggplot2.**StatDensity2D**

2D density estimate

*class* rpy2.robjects.lib.ggplot2.**StatFunction**

Superimpose a function

*class* rpy2.robjects.lib.ggplot2.**StatHline**

Horizontal line

*class* rpy2.robjects.lib.ggplot2.**StatIdentity**

Identity function

*class* rpy2.robjects.lib.ggplot2.**StatQQ**

Calculation for quantile-quantile plot.

*class* rpy2.robjects.lib.ggplot2.**StatQuantile**

Continuous quantiles

*class* rpy2.robjects.lib.ggplot2.**StatSmooth**

Smoothing function

*class* rpy2.robjects.lib.ggplot2.**StatSpoke**

Convert angle and radius to xend and yend

*class* rpy2.robjects.lib.ggplot2.**StatSum**

Sum unique values. Useful when overplotting.

*class* rpy2.robjects.lib.ggplot2.**StatSummary**

Summarize values for y at every unique value for x

*class* rpy2.robjects.lib.ggplot2.**StatUnique**

Remove duplicates.

*class* rpy2.robjects.lib.ggplot2.**StatVline**

Vertival line.

## Package *grid*

The *grid* package is the underlying plotting environment for*lattice* and *ggplot2* figures. In few words, it consists in pushing and poping systems of coordinates (*viewports*) into a stack, and plotting graphical elements into them. The system can be thought of as a scene graph, with each *viewport* a node in the graph.

```
>>> from rpy2.robjects.lib import grid
```

Getting a new page is achieved by calling the function`grid.newpage()`.

Calling `layout()` will create a layout, e.g. create a layout with one row and 3 columns:

```
>>> lt = grid.layout(1, 3)
```

That layout can be used to construct a viewport:

```
>>> vp = grid.viewport(layout = lt)
```

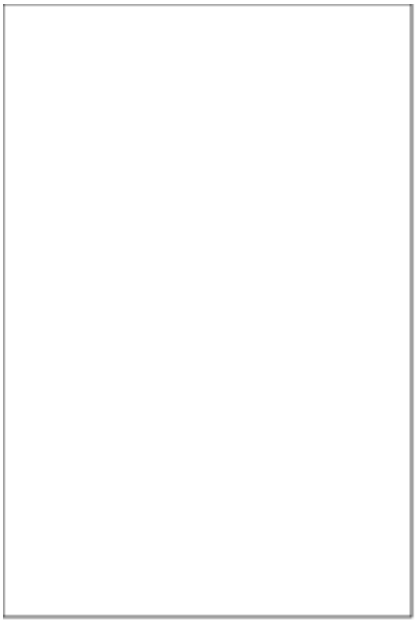
The created viewport corresponds to a graphical entity. Pushing into the current viewport, can be done by using the class method `grid.Viewport.push()`:

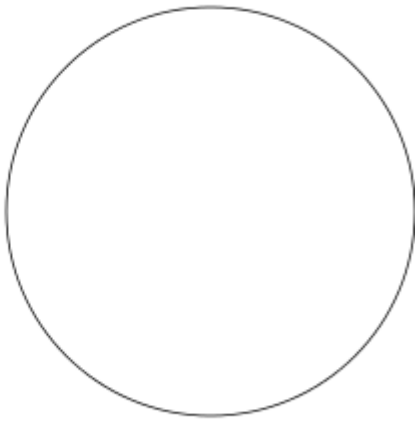
```
>>> vp.push()
```

Example:

```
grid.newpage()
# create a rows/columns layout
lt = grid.layout(2, 3)
vp = grid.viewport(layout = lt)
# push it the plotting stack
vp.push()

# create a viewport located at (1,1) in the layout
vp = grid.viewport(**{'layout.pos.col':1, 'layout.pos.row': 1})
# create a (unit) rectangle in that viewport
```

```
grid.rect(vp = vp).draw()

vp = grid.viewport(**{'layout.pos.col':2, 'layout.pos.row': 2})
# create text in the viewport at (1,2)
grid.text("foo", vp = vp).draw()

vp = grid.viewport(**{'layout.pos.col':3, 'layout.pos.row': 1})
# create a (unit) circle in the viewport (1,3)
grid.circle(vp = vp).draw()
```

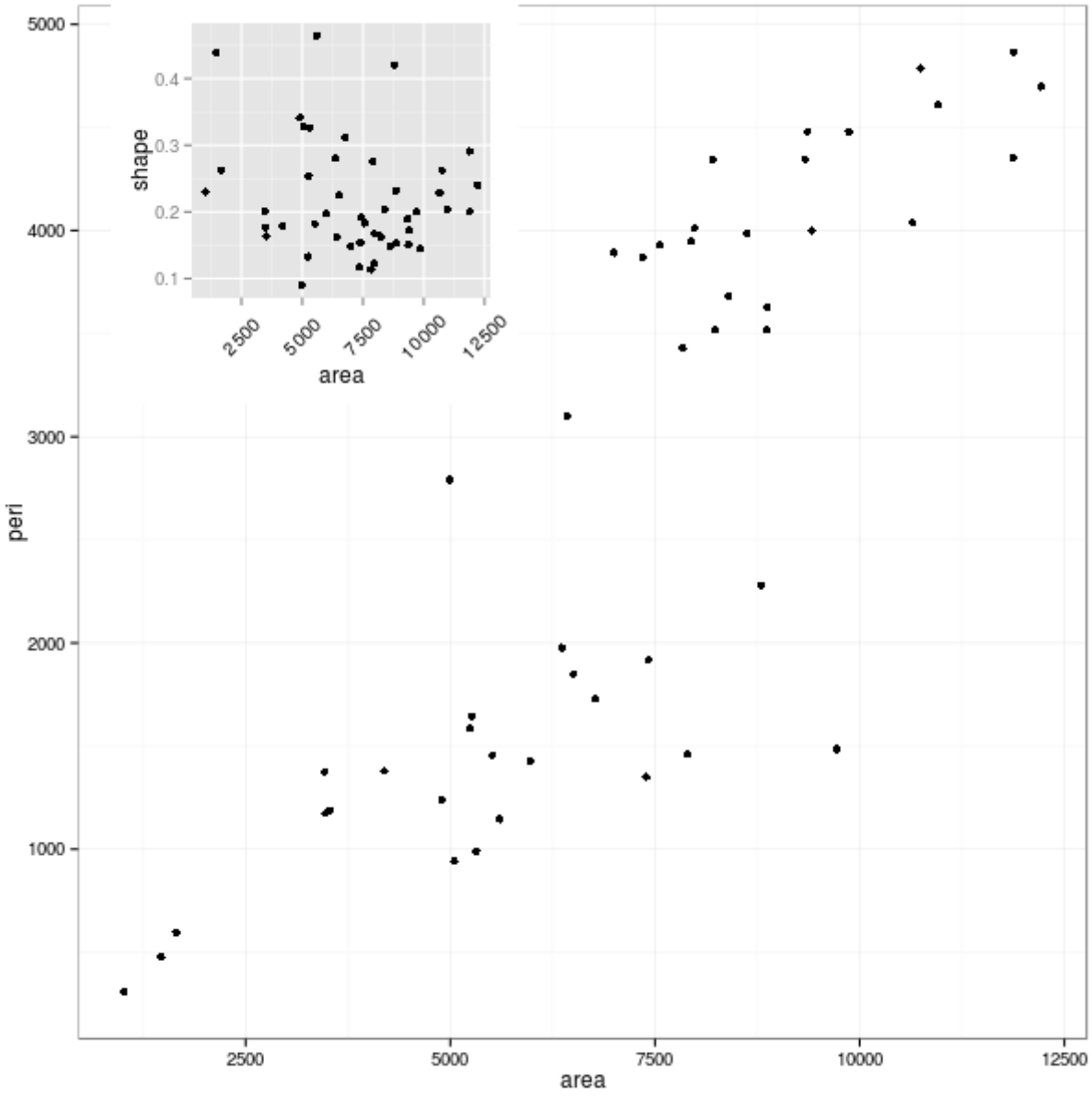





## Custom ggplot2 layout with grid

```
grid.newpage()
```

```
# create a viewport as the main plot
vp = grid.viewport(width = 1, height = 1)
vp.push()

tmpenv = datasets.__rdata__.fetch("rock")
rock = tmpenv["rock"]

p = ggplot2.ggplot(rock) + \
    ggplot2.geom_point(ggplot2.aes_string(x = 'area', y = 'peri')) + \
    ggplot2.theme_bw()
p.plot(vp = vp)

vp = grid.viewport(width = 0.6, height = 0.6, x = 0.37, y=0.69)
vp.push()
p = ggplot2.ggplot(rock) + \
    ggplot2.geom_point(ggplot2.aes_string(x = 'area', y = 'shape')) + \
    ggplot2.theme(**{'axis.text.x': ggplot2.element_text(angle = 45)})

p.plot(vp = vp)
```




## Classes¶

*class* rpy2.robjects.lib.grid.Viewport(*o*)

Bases: rpy2.robjects.robject.RObject

Drawing context. Viewports can be thought of as nodes in a scene graph.

*classmethod* current()

Return the current viewport in the stack.

*classmethod* default(***kwargs*)

*classmethod* down(*name*, *strict=False*, *recording=True*)

Return the number of Viewports it went down

*classmethod* pop(*n*)

Pop n viewports from the stack.

push(*recording=True*)

*classmethod* seek(*name*, *recording=True*)

Seek and return a Viewport given its name

*classmethod* up(*n*, *recording=True*)

Go up n viewports

*classmethod* viewport(***kwargs*)

Constructor: create a Viewport

*class* rpy2.robjects.lib.grid.Grob(*o*)

Bases: rpy2.robjects.robject.RObject

Graphical object

draw(*recording=True*)

Draw a graphical object (calling the R function grid::grid.raw())

*classmethod* grob(***kwargs*)

Constructor (uses the R function grid::grob())

*class* rpy2.robjects.lib.grid.GTree(*o*)

Bases: rpy2.robjects.lib.grid.Grob

gTree

*classmethod* grobtree(***kwargs*)

Constructor (uses the R function grid::grobTree())

*classmethod* `gtree`(***kwargs*)

Constructor (uses the R function grid::gTree())

## Class diagram