R: Data structures

Roger Bivand

Thursday, 5 September 2019, 09:45-10:15

Required current contributed CRAN packages:

I am running R 3.6.1, with recent `update.packages()`.

```
needed <- c("sf", "stars", "lwgeom")
```

Help, examples and built-in datasets in R

- In RStudio, the Help tab in the lower right pane (default position) gives access to the R manuals and to the installed packages help pages through the Packages link under Reference
- In R itself, help pages are available in HTML (browser) and text form; `help.start()` uses the default browser to display the Manuals, Reference and Miscellaneous Material sections in RStudio's home help tab
- The search engine can be used to locate help pages, but is not great if many packages are installed, as no indices are stored
- The help system needs to be learned in order to provide the user with ways of progressing without wasting too much time
- The base help system does not tell you how to use R as a system, about packages not installed on your machine, or about R as a community
- It does provide information about functions, methods and (some) classes in base R and in contributed packages installed on your machine
- There are different requirements with regard to help systems — in R, the help pages of base R are expected to be accurate although terse

Help pages

- Each help page provides a short description of the functions, methods or classes it covers; some pages cover more than one such
- Help pages are grouped by package, so that the browser-based system is not easy to browse if you do not know which package a function belongs to
- The usage of the function is shown explicitly, including any defaults for arguments to functions or methods
- Each argument is described, showing names and types; in addition details of the description are given, together with the value returned

Interactive use of help pages

- Rather than starting from the packages hierarchy of help pages, users most often use the `help()` function

- The function takes the name of of the function about which we need help, the name may be in quotation marks; class names contain a hyphen and must be quoted
- Instead of using say `help(help)`, we can shorten to the question mark operator: `?help`
- Occasionally, several packages offer different functions with the same name, and we may be offered a choice; we can disambiguate by putting the package name and two colons before the function name

Function arguments

- In the usage section, function arguments are shown by name and order; the `args()` function returns information
- In general, if arguments are given by name, the order is arbitrary, but if names are not used at least sometimes, order matters
- Some arguments do not have default values and are probably required, although some are guessed if missing
- Being explicit about the names of arguments and the values they take is helpful in scripting and reproducible research
- The ellipsis `...` indicates that the function itself examines objects passed to see what to do

Tooltips and completion

- The regular R console does not provide tooltips, that is a bubble first offering alternative function or object names as you type, then lists of argument names
- RStudio, like many IDEs, does provide this, controlled by Tools -> Global options -> Code -> Completion (by default it is operative)
- This may be helpful or not, depending on your style of working; if you find it helpful, fine, if not, you can make it less invasive under Global options
- Other IDE have also provided this facility, which builds directly on the usage sections of help pages of functions in installed packages

Coherence code/documentation

- Base R has a set of checks and tests that ensure coherence between the code itself and the usage sections in help pages
- These mechanisms are used in checking contributed packages before they are released through the the archive network; the description of arguments on help pages must match the function definition
- It is also possible to generate help pages documenting functions automatically, for example using the **roxygen2** package
- It is important to know that we can rely on this coherence

Returned values

- The objects returned by functions are also documented on help pages, but the coherence of the description with reality is harder to check
- This means that use of `str()` or other functions or methods may be helpful when we want to look inside the returned object

- The form taken by returned values will often also vary, depending on the arguments given
- Most help pages address this issue not by writing more about the returned values, but by using the examples section to highlight points of potential importance for the user

Examples

- Reading the examples section on the help page is often enlightening, but we do not need to copy and paste
- The `example()` function runs those parts of the code in the examples section of a function that are not tagged don't run — this can be overridden, but may involve meeting conditions not met on your machine
- This code is run nightly on CRAN servers on multiple operating systems and using released, patched and development versions of R, so checking both packages and the three versions of R
- Some examples use data given verbatim, but many use built-in data sets; most packages also provide data sets to use for running examples

Built-in data sets

- This means that the examples and the built-in data sets are a most significant resource for learning how to solve problems with R
- Very often, one recognizes classic textbook data sets from the history of applied statistics; contemporary text book authors often publish collections of data sets as packages on CRAN
- The built-in data sets also have help pages, describing their representation as R objects, and their licence and copyright status
- These help pages also often include an examples section showing some of the analyses that may be carried out using them
- One approach that typically works well when you have a data set of your own, but are unsure how to proceed, is to find a built-in data set that resembles the real one, and play with that first
- The built-in data sets are often quite small, and if linked to text books, they are well described there as well as in the help pages
- By definition, the built-in data sets do not have to be imported into R, as they are almost always stored as files of R objects
- In some cases, these data sets are stored in external file formats, most often to show how to read those formats
- The built-in data sets in the base **datasets** package are in the search path, but data sets in other packages should be loaded using the `data()` function

Vectors, matrices and `data.frames`

Simple vectors

In R, scalars are vectors of unit length, so most data are vectors or combinations of vectors. The printed results are prepended by a curious `[1]`; all these results are unit length vectors. We can combine several objects with `c()`:

```r
a <- c(2, 3)
a
```

```
## [1] 2 3
```

```r
sum(a)
```

```
## [1] 5
```

```r
str(a)
```

```
##  num [1:2] 2 3
```

```r
aa <- rep(a, 50)
aa
```

```
##   [1] 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2
##  [36] 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3
##  [71] 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3
```

The single square brackets `[]` are used to access or set elements of vectors (the colon `:` gives an integer sequence); negative indices drop elements:

```r
length(aa)
```

```
## [1] 100
```

```r
aa[1:10]
```

```
##  [1] 2 3 2 3 2 3 2 3 2 3
```

```r
sum(aa)
```

```
## [1] 250
```

```r
sum(aa[1:10])
```

```
## [1] 25
```

```r
sum(aa[-(11:length(aa))])
```

```
## [1] 25
```

Arithmetic under the hood

Infix syntax is just a representation of the actual underlying forms

```r
a[1] + a[2]
```

```
## [1] 5
```

```r
sum(a)
```

```
## [1] 5
```

```r
`+`(a[1], a[2])
```

```
## [1] 5
```

```r
Reduce(`+`, a)
```

```
## [1] 5
```

We've done arithmetic on scalars, we can do vector-scalar arithmetic:

```
sum(aa)
```

```
## [1] 250
```

```
sum(aa+2)
```

```
## [1] 450
```

```
sum(aa)+2
```

```
## [1] 252
```

```
sum(aa*2)
```

```
## [1] 500
```

```
sum(aa)*2
```

```
## [1] 500
```

But vector-vector arithmetic poses the question of vector length and recycling (the shorter one gets recycled):

```
v5 <- 1:5
v2 <- c(5, 10)
v5 * v2
```

```
## Warning in v5 * v2: longer object length is not a multiple of shorter
## object length
```

```
## [1]  5 20 15 40 25
```

```
v2_stretch <- rep(v2, length.out=length(v5))
v2_stretch
```

```
## [1]  5 10  5 10  5
```

```
v5 * v2_stretch
```

```
## [1]  5 20 15 40 25
```

In working with real data, we often meet missing values, coded by NA meaning Not Available:

```
anyNA(aa)
```

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

```
is.na(aa) <- 5
aa[1:10]
```

```
##  [1]  2  3  2  3 NA  3  2  3  2  3
```

```
anyNA(aa)
```

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

```
sum(aa)
```

```
## [1] NA
```

```
sum(aa, na.rm=TRUE)
```

```
## [1] 248
```

Checking data

One way to check our input data is to print in the console — this works with small objects as we've seen, but for larger objects we need methods:

```
big <- 1:(10^5)
length(big)
```

```
## [1] 100000
```

```
head(big)
```

```
## [1] 1 2 3 4 5 6
```

```
str(big)
```

```
##  int [1:100000] 1 2 3 4 5 6 7 8 9 10 ...
```

```
summary(big)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       1   25001   50000   50000   75000  100000
```

Basic vector types

There are `length`, `head`, `str` (*str*ucture) and `summary` methods for many types of objects. `str` also gives us a hint of the type of object and its dimensions. We've seen a couple of uses of `str` so far, `str(a)` was `num` and `str(big)` was `int`, what does this signify? They are both numbers, but of different types.

There are six basic vector types: list, integer, double, logical, character and complex. The derived type factor (to which we return shortly) is integer with extra information. `str` reports these as int, num, logi, chr and cplx, and lists are enumerated recursively. In RStudio you see more or less the `str` output in the environment pane as Values in the list view; the grid view adds the object size in memory. From early S, we have `typeof` and `storage.mode` (including single precision, not used in R) — these are important for interfacing C, C++, Fortran and other languages. Beyond this is `class`, but then the different class systems (S3 and formal S4) complicate things. Objects such as vectors may also have attributes in which their class and other information may be placed. Typically, a lot of use is made of attributes to squirrel away strings and short vectors.

`is` methods are used to test types of objects; note that integers are also seen as numeric:

```
set.seed(1)
x <- runif(50, 1, 10)
is.numeric(x)
```

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

```
y <- rpois(50, lambda=6)
is.numeric(y)
```

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

```
is.integer(y)
```

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

```
xy <- x < y
is.logical(xy)
```

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

`as` methods try to convert between object types and are widely used:

```
str(as.integer(xy))
```

```
##  int [1:50] 1 1 0 0 1 0 0 0 1 1 ...
```

```
str(as.numeric(y))
```

```
##  num [1:50] 6 9 5 4 3 3 5 6 7 5 ...
```

```
str(as.character(y))
```

```
##  chr [1:50] "6" "9" "5" "4" "3" "3" "5" "6" "7" "5" "9" "5" "6" "5" ...
```

```
str(as.integer(x))
```

```
##  int [1:50] 3 4 6 9 2 9 9 6 6 1 ...
```

The data frame object

First, let us see that is behind the `data.frame` object: the `list` object. `list` objects are vectors that contain other objects, which can be addressed by name or by 1-based indices. Like the vectors we have already met, lists can be accessed and manipulated using square brackets `[]`. Single list elements can be accessed and manipulated using double square brackets `[[]]`.

List objects

Starting with four vectors of differing types, we can assemble a list object; as we see, its structure is quite simple. The vectors in the list may vary in length, and lists can (and do often) include lists

```
V1 <- 1:3
V2 <- letters[1:3]
V3 <- sqrt(V1)
V4 <- sqrt(as.complex(-V1))
L <- list(v1=V1, v2=V2, v3=V3, v4=V4)
str(L)
```

```
## List of 4
##  $ v1: int [1:3] 1 2 3
##  $ v2: chr [1:3] "a" "b" "c"
##  $ v3: num [1:3] 1 1.41 1.73
##  $ v4: cplx [1:3] 0+1i 0+1.41i 0+1.73i
```

```
L$v3[2]
```

```
## [1] 1.414214
```

```
L[[3]][2]
```

```
## [1] 1.414214
```

Data Frames

Our `list` object contains four vectors of different types but of the same length; conversion to a `data.frame` is convenient. Note that by default strings are converted into factors:

```
DF <- as.data.frame(L)
str(DF)
```

```
## 'data.frame':	3 obs. of  4 variables:
##  $ v1: int  1 2 3
##  $ v2: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ v3: num  1 1.41 1.73
##  $ v4: cplx  0+1i 0+1.41i 0+1.73i
```

```
DF <- as.data.frame(L, stringsAsFactors=FALSE)
str(DF)
```

```
## 'data.frame':	3 obs. of  4 variables:
##  $ v1: int  1 2 3
##  $ v2: chr  "a" "b" "c"
##  $ v3: num  1 1.41 1.73
##  $ v4: cplx  0+1i 0+1.41i 0+1.73i
```

We can also provoke an error in conversion from a valid `list` made up of vectors of different length to a `data.frame`:

```
V2a <- letters[1:4]
V4a <- factor(V2a)
La <- list(v1=V1, v2=V2a, v3=V3, v4=V4a)
DFa <- try(as.data.frame(La, stringsAsFactors=FALSE), silent=TRUE)
message(DFa)
```

```
## Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,  :
##   arguments imply differing number of rows: 3, 4
```

We can access `data.frame` elements as `list` elements, where the `$` is effectively the same as `[[]]` with the list component name as a string:

```
DF$v3[2]
```

```
## [1] 1.414214
```

```
DF[[3]][2]
```

```
## [1] 1.414214
```

```
DF[["v3"]][2]
```

```
## [1] 1.414214
```

Since a `data.frame` is a rectangular object with named columns with equal numbers of rows, it can also be indexed like a matrix, where the rows are the first index and the columns (variables) the second:

```
DF[2, 3]
```

```
## [1] 1.414214
```

```
DF[2, "v3"]
```

```
## [1] 1.414214
```

```
str(DF[2, 3])
```

```
##  num 1.41
```

```
str(DF[2, 3, drop=FALSE])
```

```
## 'data.frame':	1 obs. of  1 variable:
##  $ v3: num 1.41
```

If we coerce a `data.frame` containing a character vector or factor into a matrix, we get a character matrix; if we extract an integer and a numeric column, we get a numeric matrix.

```
as.matrix(DF)
```

```
##      v1  v2  v3         v4
## [1,] "1" "a" "1.000000" "0+1.000000i"
## [2,] "2" "b" "1.414214" "0+1.414214i"
## [3,] "3" "c" "1.732051" "0+1.732051i"
```

```
as.matrix(DF[,c(1,3)])
```

```
##      v1       v3
## [1,]  1 1.000000
## [2,]  2 1.414214
## [3,]  3 1.732051
```

The fact that `data.frame` objects descend from `list` objects is shown by looking at their lengths; the length of a matrix is not its number of columns, but its element count:

```
length(L)
```

```
## [1] 4
```

```
length(DF)
```

```
## [1] 4
```

```
length(as.matrix(DF))
```

```
## [1] 12
```

There are `dim` methods for `data.frame` objects and matrices (and arrays with more than two dimensions); matrices and arrays are seen as vectors with dimensions; `list` objects have no dimensions:

```
dim(L)
```

```
## NULL
```

```
dim(DF)
```

```
## [1] 3 4
```

```
dim(as.matrix(DF))
```

```
## [1] 3 4
```

```
str(as.matrix(DF))
```

```
##  chr [1:3, 1:4] "1" "2" "3" "a" "b" "c" "1.000000" "1.414214" ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:4] "v1" "v2" "v3" "v4"
```

`data.frame` objects have `names` and `row.names`, matrices have `dimnames`, `colnames` and `rownames`; all can be used for setting new values:

```
row.names(DF)
```

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

```
names(DF)
```

```
## [1] "v1" "v2" "v3" "v4"
```

```
names(DF) <- LETTERS[1:4]
names(DF)
```

```
## [1] "A" "B" "C" "D"
```

```
str(dimnames(as.matrix(DF)))
```

```
## List of 2
##  $ : NULL
##  $ : chr [1:4] "A" "B" "C" "D"
```

R objects have attributes that are not normally displayed, but which show their structure and class (if any); we can see that `data.frame` objects are quite different internally from matrices:

```
str(attributes(DF))
```

```
## List of 3
##  $ names    : chr [1:4] "A" "B" "C" "D"
##  $ class    : chr "data.frame"
##  $ row.names: int [1:3] 1 2 3
```

```
str(attributes(as.matrix(DF)))
```

```
## List of 2
##  $ dim     : int [1:2] 3 4
##  $ dimnames:List of 2
##   ..$ : NULL
##   ..$ : chr [1:4] "A" "B" "C" "D"
```

If the reason for different vector lengths was that one or more observations are missing on that variable, `NA` should be used; the lengths are then equal, and a rectangular table can be created:

```
V1a <- c(V1, NA)
V3a <- sqrt(V1a)
La <- list(v1=V1a, v2=V2a, v3=V3a, v4=V4a)
DFa <- as.data.frame(La, stringsAsFactors=FALSE)
str(DFa)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ v1: int  1 2 3 NA
##  $ v2: chr  "a" "b" "c" "d"
##  $ v3: num  1 1.41 1.73 NA
##  $ v4: Factor w/ 4 levels "a","b","c","d": 1 2 3 4
```

New style spatial vector representation

The sf package

The recent **sf** package bundles GDAL and GEOS (**sp** just defined the classes and methods, leaving I/O and computational geometry to other packages **rgdal** and **rgeos**). **sf** uses `data.frame` objects with one (or more) geometry column for vector data. The representation follows ISO 19125 (*Simple Features*), and has WKT (text) and WKB (binary) representations (used by GDAL and GEOS internally). The drivers include PostGIS and other database constructions permitting selection, and WFS for server APIs. These are the key references for **sf**: (Lovelace, Nowosad, and Muenchow 2019), (Pebesma and Bivand, n.d.), (Pebesma 2018), package vignettes and blog posts on (https://www.r-spatial.org/).

```
library(sf)
```

```
## Linking to GEOS 3.7.2, GDAL 3.0.1, PROJ 6.2.0
```

The `st_read()` method, here for a `"character"` first object giving the file name and path, uses GDAL through **Rcpp** to identify the driver required, and to use it to read the feature geometries and fields. The character string fields are not converted to `"factor"` representation, as they are not categorical variables:

```r
lux <- st_read("../data/lux_regions.gpkg", stringsAsFactors=FALSE)
```

```
## Reading layer `lux_regions' from data source `/home/rsb/presentations/ectqg19-workshop/data/lux_regi
## Simple feature collection with 102 features and 10 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 5.735708 ymin: 49.44786 xmax: 6.530898 ymax: 50.18277
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

The vector drivers available to me with my GDAL build are:

```r
st_drivers(what="vector")[,c(2:4, 7)]
```

```
##                                                                    long_name
## PCIDSK                                                    PCIDSK Database File
## netCDF                                                  Network Common Data Format
## PDS4                                                  NASA Planetary Data System 4
## JP2OpenJPEG                                JPEG-2000 driver based on OpenJPEG library
## PDF                                                             Geospatial PDF
## MBTiles                                                               MBTiles
## EEDA                                                     Earth Engine Data API
## ESRI Shapefile                                                  ESRI Shapefile
## MapInfo File                                                      MapInfo File
## UK .NTF                                                                UK .NTF
## OGR_SDTS                                                                  SDTS
## S57                                                             IHO S-57 (ENC)
## DGN                                                           Microstation DGN
## OGR_VRT                                                 VRT - Virtual Datasource
## REC                                                                EPIInfo .REC
## Memory                                                                  Memory
## BNA                                                                  Atlas BNA
## CSV                                                   Comma Separated Value (.csv)
## NAS                                                               NAS - ALKIS
## GML                                                 Geography Markup Language (GML)
## GPX                                                                        GPX
## KML                                                   Keyhole Markup Language (KML)
## GeoJSON                                                                GeoJSON
## GeoJSONSeq                                                      GeoJSON Sequence
## ESRIJSON                                                                ESRIJSON
## TopoJSON                                                                TopoJSON
## Interlis 1                                                            Interlis 1
## Interlis 2                                                            Interlis 2
## OGR_GMT                                                   GMT ASCII Vectors (.gmt)
## GPKG                                                                  GeoPackage
## SQLite                                                     SQLite / Spatialite
## ODBC                                                                        ODBC
## WAsP                                                            WAsP .map format
## PGeo                                                     ESRI Personal GeoDatabase
## MSSQLSpatial                                      Microsoft SQL Server Spatial Database
```

```
## PostgreSQL                                                 PostgreSQL/PostGIS
## OpenFileGDB                                                      ESRI FileGDB
## XPlane                                 X-Plane/Flightgear aeronautical data
## DXF                                                               AutoCAD DXF
## CAD                                                            AutoCAD Driver
## Geoconcept                                                         Geoconcept
## GeoRSS                                                                 GeoRSS
## GPSTrackMaker                                                   GPSTrackMaker
## VFK                                    Czech Cadastral Exchange Data Format
## PGDUMP                                                    PostgreSQL SQL dump
## OSM                                                 OpenStreetMap XML and PBF
## GPSBabel                                                             GPSBabel
## SUA                         Tim Newport-Peace's Special Use Airspace Format
## OpenAir                                                               OpenAir
## OGR_PDS                                           Planetary Data Systems TABLE
## WFS                                               OGC WFS (Web Feature Service)
## WFS3                                     OGC WFS 3 client (Web Feature Service)
## HTF                                               Hydrographic Transfer Vector
## AeronavFAA                                                        Aeronav FAA
## Geomedia                                                         Geomedia .mdb
## EDIGEO                                            French EDIGEO exchange format
## GFT                                                      Google Fusion Tables
## SVG                                                   Scalable Vector Graphics
## CouchDB                                                     CouchDB / GeoCouch
## Cloudant                                                    Cloudant / CouchDB
## Idrisi                                                     Idrisi Vector (.vct)
## ARCGEN                                                        Arc/Info Generate
## SEGUKOOA                                                   SEG-P1 / UKOOA P1/90
## SEGY                                                                     SEG-Y
## XLS                                                           MS Excel format
## ODS                     Open Document/ LibreOffice / OpenOffice Spreadsheet 
## XLSX                                             MS Office Open XML spreadsheet
## ElasticSearch                                                    Elastic Search
## Walk                                                                       Walk
## Carto                                                                     Carto
## AmigoCloud                                                           AmigoCloud
## SXF                                                    Storage and eXchange Format
## Selafin                                                                 Selafin
## JML                                                                 OpenJUMP JML
## PLSCENES                                                 Planet Labs Scenes API
## CSW                                      OGC CSW (Catalog  Service for the Web)
## VDV                                         VDV-451/VDV-452/INTREST Data Format
## GMLAS          Geography Markup Language (GML) driven by application schemas
## MVT                                                          Mapbox Vector Tiles
## TIGER                                                     U.S. Census TIGER/Line
## AVCBin                                                   Arc/Info Binary Coverage
## AVCE00                                              Arc/Info E00 (ASCII) Coverage
## NGW                                                                 NextGIS Web
## HTTP                                                       HTTP Fetching Wrapper
##               write  copy   vsi
## PCIDSK         TRUE FALSE  TRUE
## netCDF         TRUE  TRUE  TRUE
## PDS4           TRUE  TRUE  TRUE
## JP2OpenJPEG   FALSE  TRUE  TRUE
```

```
## PDF             TRUE  TRUE FALSE
## MBTiles         TRUE  TRUE  TRUE
## EEDA           FALSE FALSE FALSE
## ESRI Shapefile  TRUE FALSE  TRUE
## MapInfo File    TRUE FALSE  TRUE
## UK .NTF        FALSE FALSE  TRUE
## OGR_SDTS       FALSE FALSE  TRUE
## S57             TRUE FALSE  TRUE
## DGN             TRUE FALSE  TRUE
## OGR_VRT        FALSE FALSE  TRUE
## REC            FALSE FALSE FALSE
## Memory          TRUE FALSE FALSE
## BNA             TRUE FALSE  TRUE
## CSV             TRUE FALSE  TRUE
## NAS            FALSE FALSE  TRUE
## GML             TRUE FALSE  TRUE
## GPX             TRUE FALSE  TRUE
## KML             TRUE FALSE  TRUE
## GeoJSON         TRUE FALSE  TRUE
## GeoJSONSeq      TRUE FALSE  TRUE
## ESRIJSON       FALSE FALSE  TRUE
## TopoJSON       FALSE FALSE  TRUE
## Interlis 1      TRUE FALSE  TRUE
## Interlis 2      TRUE FALSE  TRUE
## OGR_GMT         TRUE FALSE  TRUE
## GPKG            TRUE  TRUE  TRUE
## SQLite          TRUE FALSE  TRUE
## ODBC            TRUE FALSE FALSE
## WAsP            TRUE FALSE  TRUE
## PGeo           FALSE FALSE FALSE
## MSSQLSpatial    TRUE FALSE FALSE
## PostgreSQL      TRUE FALSE FALSE
## OpenFileGDB    FALSE FALSE  TRUE
## XPlane         FALSE FALSE  TRUE
## DXF             TRUE FALSE  TRUE
## CAD            FALSE FALSE  TRUE
## Geoconcept      TRUE FALSE  TRUE
## GeoRSS          TRUE FALSE  TRUE
## GPSTrackMaker   TRUE FALSE  TRUE
## VFK            FALSE FALSE FALSE
## PGDUMP          TRUE FALSE  TRUE
## OSM            FALSE FALSE  TRUE
## GPSBabel        TRUE FALSE FALSE
## SUA            FALSE FALSE  TRUE
## OpenAir        FALSE FALSE  TRUE
## OGR_PDS        FALSE FALSE  TRUE
## WFS            FALSE FALSE  TRUE
## WFS3           FALSE FALSE FALSE
## HTF            FALSE FALSE  TRUE
## AeronavFAA     FALSE FALSE  TRUE
## Geomedia       FALSE FALSE FALSE
## EDIGEO         FALSE FALSE  TRUE
## GFT             TRUE FALSE FALSE
## SVG            FALSE FALSE  TRUE
```

```
## CouchDB          TRUE FALSE FALSE
## Cloudant         TRUE FALSE FALSE
## Idrisi          FALSE FALSE  TRUE
## ARCGEN          FALSE FALSE  TRUE
## SEGUKOOA        FALSE FALSE  TRUE
## SEGY            FALSE FALSE  TRUE
## XLS             FALSE FALSE FALSE
## ODS              TRUE FALSE  TRUE
## XLSX             TRUE FALSE  TRUE
## ElasticSearch    TRUE FALSE FALSE
## Walk            FALSE FALSE FALSE
## Carto            TRUE FALSE FALSE
## AmigoCloud       TRUE FALSE FALSE
## SXF             FALSE FALSE  TRUE
## Selafin          TRUE FALSE  TRUE
## JML              TRUE FALSE  TRUE
## PLSCENES        FALSE FALSE FALSE
## CSW             FALSE FALSE FALSE
## VDV              TRUE FALSE  TRUE
## GMLAS           FALSE  TRUE  TRUE
## MVT              TRUE FALSE  TRUE
## TIGER            TRUE FALSE  TRUE
## AVCBin          FALSE FALSE  TRUE
## AVCE00          FALSE FALSE  TRUE
## NGW              TRUE  TRUE FALSE
## HTTP            FALSE FALSE FALSE
```

Package **sf** provides handling of feature data, where feature geometries are points, lines, polygons or combinations of those. It implements the full set of geometric functions described in the *simple feature access* standard, and some. The basic storage is very simple, and uses only base R types (list, matrix).

- feature sets are held as records (rows) in `"sf"` objects, inheriting from `"data.frame"`
- `"sf"` objects have at least one simple feature geometry list-column of class `"sfc"`
- geometry list-columns are *sticky*, that is they stay stuck to the object when subsetting columns, for example using `[`
- `"sfc"` geometry list-columns have a bounding box and a coordinate reference system as attribute, and a class attribute pointing out the common type (or `"GEOMETRY"` in case of a mix)
- a single simple feature geometry is of class `"sfg"`, and further classes pointing out dimension and type

Storage of simple feature geometry:

- `"POINT"` is a numeric vector
- `"LINESTRING"` and `"MULTIPOINT"` are numeric matrix, points/vertices in rows
- `"POLYGON"` and `"MULTILINESTRING"` are lists of matrices
- `"MULTIPOLYGON"` is a lists of those
- `"GEOMETRYCOLLECTION"` is a list of typed geometries

```
class(lux)
```

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

The columns of the `"data.frame"` object have these names:

```
names(lux)
```

```
##  [1] "POPULATION"  "COMMUNE_1"   "LAU2"        "X_subtype"   "COMMUNE"
##  [6] "DISTRICT"    "CANTON"      "tree_count"  "ghsl_pop"    "light_level"
## [11] "geom"
```

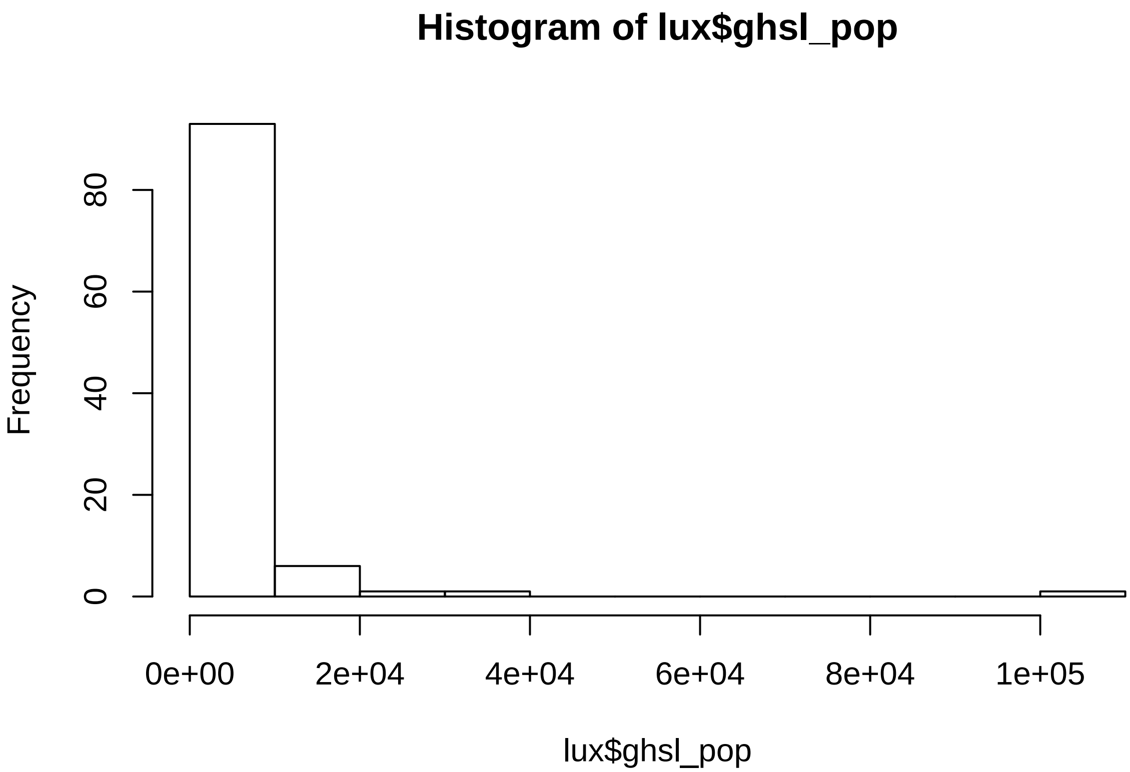
Two of the attributes of the object are those all "data.frame" objects possess: names shown above and row.names. The fourth, sf_column gives the name of the active geometry column.

```
names(attributes(lux))
```

```
## [1] "names"     "row.names" "class"     "sf_column" "agr"
```

The $ access operator lets us operate on a single column of the object as with any other "data.frame" object:

```
hist(lux$ghsl_pop)
```



Using the attribute value to extract the name of the geometry column, and the [[access operator to give programmatic access to a column by name, we can see that the "sfc" object is composed of POLYGON objects:

```
class(lux[[attr(lux, "sf_column")]])
```

```
## [1] "sfc_MULTIPOLYGON" "sfc"
```

The geometry column is a list column, of the same length as the other columns in the "data.frame" object.

```
is.list(lux[[attr(lux, "sf_column")]])
```

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

"sf" objects may be subsetted by row and column in the same way as regular "data.frame" objects, with the implicit understanding that the geometry column is *sticky*; here we choose only the first column, but the geometry column follows along, *stuck* to the subsetted object, and obviously subsetted by row too.

```
class(lux[1:5, 1])
```

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

Geometry columns have their own list of attributes, the count of empty geometries, the coordinate reference system, the precision and the bounding box (subsetting will refresh the bounding box; transformation will update the coordinate reference system and the bounding box):

```r
attributes(lux[[attr(lux, "sf_column")]])
```

```
## $n_empty
## [1] 0
##
## $crs
## Coordinate Reference System:
##   EPSG: 4326
##   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
##
## $class
## [1] "sfc_MULTIPOLYGON" "sfc"
##
## $precision
## [1] 0
##
## $bbox
##      xmin      ymin      xmax      ymax
##  5.735708 49.447859  6.530898 50.182772
```

The coordinate reference system is an object of class `"crs"`:

```r
class(attr(lux[[attr(lux, "sf_column")]], "crs"))
```

```
## [1] "crs"
```

It contains an integer EPSG code (so far not compound codes), and a PROJ string:

```r
str(attr(lux[[attr(lux, "sf_column")]], "crs"))
```

```
## List of 2
##  $ epsg       : int 4326
##  $ proj4string: chr "+proj=longlat +datum=WGS84 +no_defs"
##  - attr(*, "class")= chr "crs"
```

Objects of this class can be instantiated for example by giving the relevant EPSG code:

```r
st_crs(4674)
```

```
## Coordinate Reference System:
##   EPSG: 4674
##   proj4string: "+proj=longlat +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +no_defs"
```

```r
st_crs(31983)
```

```
## Coordinate Reference System:
##   EPSG: 31983
##   proj4string: "+proj=utm +zone=23 +south +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs"
```

We can drill down to the first feature geometry `"sfg"` object, which is a matrix with a class attribute - a vector of three elements, `"XY"` for two dimensions, `"POLYGON"` for the simple features definition, and `"sfg"` as the container class:

```r
str(lux[[attr(lux, "sf_column")]][[1]])
```

```
## List of 1
##  $ :List of 1
##   ..$ : num [1:159, 1:2] 6 6 6 6 6 ...
##  - attr(*, "class")= chr [1:3] "XY" "MULTIPOLYGON" "sfg"
```

Checking the data

Stepping back, we can try to check where the `POPULATION` field/column came from; the Luxembourg public data source provides a file, here as `"geojson"` https://data.public.lu/en/datasets/population-per-municipality/:

```
pop <- st_read("../data/statec_population_by_municipality.geojson")
```

```
## Reading layer `Inspire_10072018' from data source `/home/rsb/presentations/ectqg19-workshop/data/sta
## Simple feature collection with 102 features and 9 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 5.735708 ymin: 49.44786 xmax: 6.530898 ymax: 50.18277
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

The difference is in the ordering of the features in the two objects:

```
all.equal(pop$POPULATION, lux$POPULATION)
```

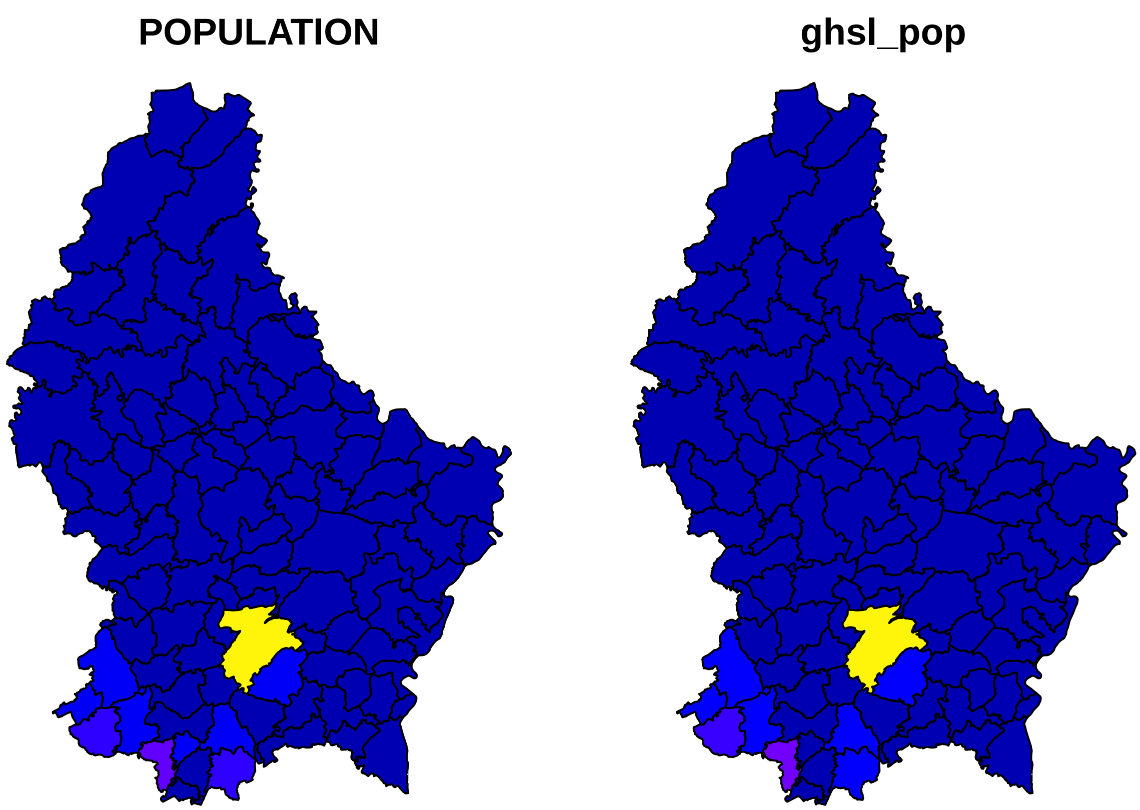
```
## [1] "Mean relative difference: 1.045941"
```

```
o <- match(as.character(pop$LAU2), as.character(lux$LAU2))
all.equal(pop$POPULATION, lux$POPULATION[o])
```

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

We'll also be able to look at the differences between population counts from this data source and from GHSL https://ghsl.jrc.ec.europa.eu/; the data source is probably the 2011 census:

```
plot(lux[, c("POPULATION", "ghsl_pop")])
```



The trees are from https://data.public.lu/en/datasets/remarkable-trees/, but in projected, not geographical coordinates:

```
trees <- st_read("../data/trees/anf_remarkable_trees_0.shp")
```

```
## Reading layer `anf_remarkable_trees_0' from data source `/home/rsb/presentations/ectqg19-workshop/dat
## Simple feature collection with 535 features and 5 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 51180 ymin: 57820 xmax: 104902 ymax: 136714
## epsg (SRID):    2169
## proj4string:    +proj=tmerc +lat_0=49.8333333333333 +lon_0=6.16666666666667 +k=1 +x_0=80000 +y_0=1000
```

If we would like to find the areas of the administrative units, we could use spherical areas from **lwgeom**, or take them by transforming to for example the projection of the trees:

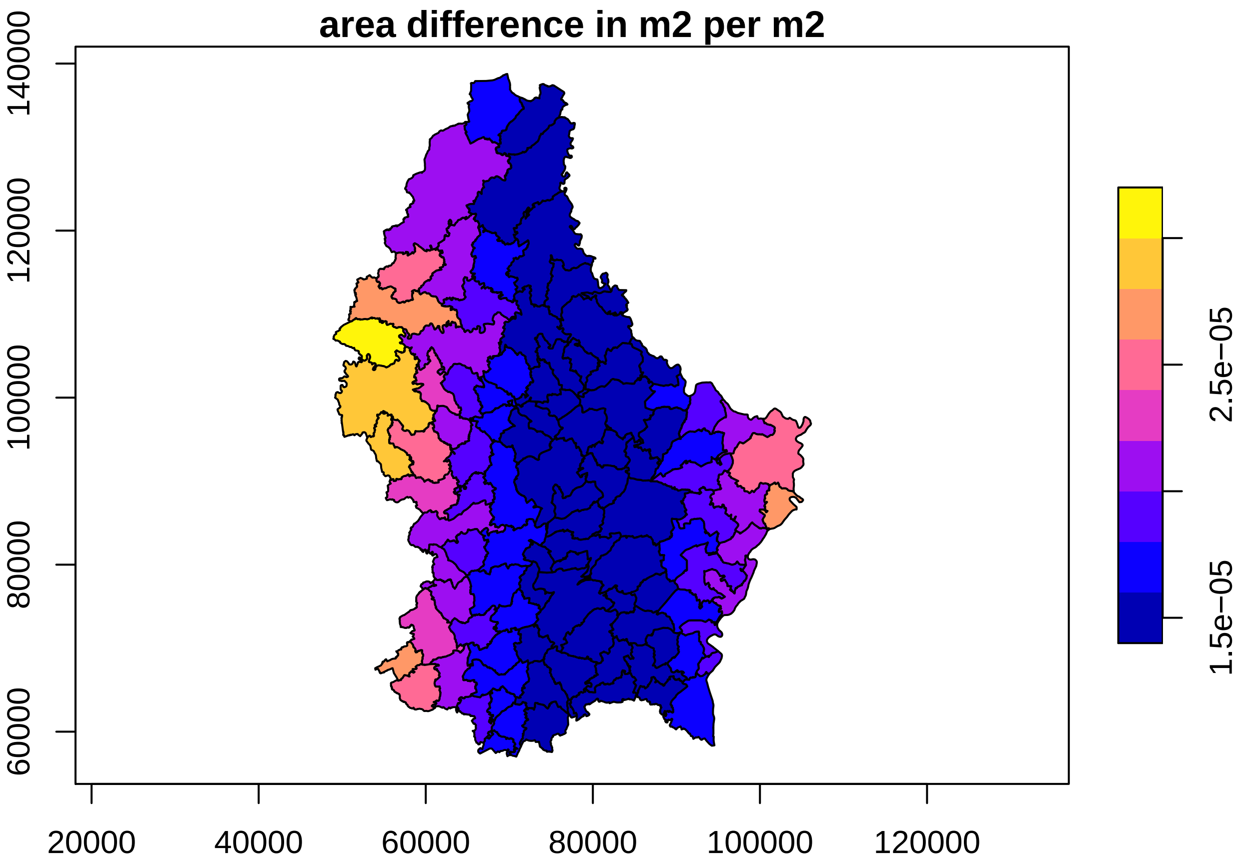
```
area_sph <- lwgeom::st_geod_area(lux)
lux_tmerc <- st_transform(lux, 2169)
area_tmerc <- st_area(lux_tmerc)
```

There are small differences between these area outputs:

```
lux_tmerc$area <- area_tmerc
lux_tmerc$area_err <- (lux_tmerc$area - area_sph)/lux_tmerc$area
summary(lux_tmerc$area_err)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 1.412e-05 1.493e-05 1.710e-05 1.800e-05 1.984e-05 3.116e-05
```

```
plot(lux_tmerc[, "area_err"], axes=TRUE, main="area difference in m2 per m2")
```



The area is in square meters, so we can use facilities in **units** to change to square kilometers to calculate population densities:

```r
units(lux_tmerc$area)
```

```
## $numerator
## [1] "m" "m"
##
## $denominator
## character(0)
##
## attr(,"class")
## [1] "symbolic_units"
```

```r
units(lux_tmerc$area) <- "km^2"
units(lux_tmerc$area)
```

```
## $numerator
## [1] "km" "km"
##
## $denominator
## character(0)
##
## attr(,"class")
## [1] "symbolic_units"
```

```r
lux_tmerc$pop_den <- lux_tmerc$POPULATION/lux_tmerc$area
summary(lux_tmerc$pop_den)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   35.60   76.89  139.36  290.12  293.68 2433.16
```

```r
lux_tmerc$ghsl_den <- lux_tmerc$ghsl_pop/lux_tmerc$area
summary(lux_tmerc$ghsl_den)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   32.91   71.10  136.68  268.31  284.85 2088.24
```

We can do the same kinds of sourcing checks with the tree counts:

```r
trees_sgbp <- st_intersects(lux_tmerc, trees)
trees_cnt <- sapply(trees_sgbp, length)
all.equal(trees_cnt, lux_tmerc$tree_count)
```

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

But we are not ready to move on yet.

PROJ

Because so much open source (and other) software uses the PROJ library and framework, many are affected when PROJ upgrades. Until very recently, PROJ has been seen as very reliable, and the changes taking place now are intended to confirm and reinforce this reliability. Before PROJ 5 (PROJ 6 is out now, PROJ 7 is coming early in 2020), the `+datum=` tag was used, perhaps with `+towgs84=` with three or seven coefficients, and possibly `+nadgrids=` where datum transformation grids were available. However, transformations from one projection to another first inversed to longitude-latitude in WGS84, then projected on to the target projection.

Big bump coming:

'Fast-forward 35 years and PROJ.4 is everywhere: It provides coordinate handling for almost every geospatial program, open or closed source. Today, we see a drastical increase in the need for high accuracy GNSS coordinate handling, especially in the agricultural and construction engineering sectors. This need for geodetic-accuracy transformations is not satisfied by "classic PROJ.4". But with the ubiquity of PROJ.4, we can provide these transformations "everywhere", just by implementing them as part of PROJ.4' (Evers and Knudsen 2017).

Escaping the WGS84 hub/pivot: PROJ and OGC WKT2

Following the introduction of geodetic modules and pipelines in PROJ 5 (Knudsen and Evers 2017; Evers and Knudsen 2017), PROJ 6 moves further. Changes in the legacy PROJ representation and WGS84 transformation hub have been coordinated through the GDAL barn raising initiative. Crucially WGS84 often ceases to be the pivot for moving between datums. A new OGC WKT is coming, and an SQLite EPSG file database has replaced CSV files. SRS will begin to support 3D by default, adding time too as SRS change. See also PROJ migration notes.

There are very useful postings on the PROJ mailing list from Martin Desruisseaux, first proposing clarifications and a follow-up including a summary:

> - "Early binding": hub transformation technique.
> - "Late binding": hub transformation technique NOT used, replaced by a more complex technique consisting in searching parameters in the EPSG database after the transformation context (source, target, epoch, area of interest) is known.
> - The problem of hub transformation technique is independent of WGS84. It is caused by the fact that transformations to/from the hub are approximate. Any other hub we could invent in replacement of WGS84 will have the same problem, unless we can invent a hub for which transformations are exact (I think that if such hub existed, we would have already heard about it).
>
> The solution proposed by ISO 19111 (in my understanding) is:
>
> - Forget about hub (WGS84 or other), unless the simplicity of early-binding is considered more important than accuracy.
> - Associating a CRS to a coordinate set (geometry or raster) is no longer sufficient. A {CRS, epoch} tuple must be associated. ISO 19111 calls this tuple "Coordinate metadata". From a programmatic API point of view, this means that getCoordinateReferenceSystem() method in Geometry objects (for instance) needs to be replaced by a getCoordinateMetadata() method.

Maybe watch Even Roualt's recent FOSS4G talk: https://media.ccc.de/v/bucharest-198-revamp-of-coordinate-reference-system

The `projinfo` utility is available with the external PROJ library, so most likely not in general; this is for PROJ 6.1.1 (current as of writing):

```
cat(system("projinfo EPSG:4326 -o PROJ", intern=TRUE), sep="\n")
```

```
## PROJ.4 string:
## +proj=longlat +datum=WGS84 +no_defs +type=crs
```

```
cat(system("projinfo EPSG:4326 -o WKT1_GDAL", intern=TRUE), sep="\n")
```

```
## WKT1_GDAL:
## GEOGCS["WGS 84",
##     DATUM["WGS_1984",
##         SPHEROID["WGS 84",6378137,298.257223563,
```

```
##             AUTHORITY["EPSG","7030"]],
##         AUTHORITY["EPSG","6326"]],
##     PRIMEM["Greenwich",0,
##         AUTHORITY["EPSG","8901"]],
##     UNIT["degree",0.0174532925199433,
##         AUTHORITY["EPSG","9122"]],
##     AUTHORITY["EPSG","4326"]]
```

```
cat(system("projinfo EPSG:4326 -o WKT2_2018", intern=TRUE), sep="\n")
```

```
## WKT2_2018 string:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["geodetic longitude (Lon)",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     USAGE[
##         SCOPE["unknown"],
##         AREA["World"],
##         BBOX[-90,-180,90,180]],
##     ID["EPSG",4326]]
```

```
cat(system("projinfo EPSG:2169 -o PROJ", intern=TRUE), sep="\n")
```

```
## PROJ.4 string:
## +proj=tmerc +lat_0=49.8333333333333 +lon_0=6.16666666666667 +k=1 +x_0=80000 +y_0=100000 +ellps=intl +
```

```
cat(system("projinfo EPSG:2169 -o WKT1_GDAL", intern=TRUE), sep="\n")
```

```
## WKT1_GDAL:
## PROJCS["Luxembourg 1930 / Gauss",
##     GEOGCS["Luxembourg 1930",
##         DATUM["Luxembourg_1930",
##             SPHEROID["International 1924",6378388,297,
##                 AUTHORITY["EPSG","7022"]],
##             TOWGS84[-189.6806,18.3463,-42.7695,-0.33746,-3.09264,2.53861,0.4598],
##             AUTHORITY["EPSG","6181"]],
##         PRIMEM["Greenwich",0,
##             AUTHORITY["EPSG","8901"]],
##         UNIT["degree",0.0174532925199433,
##             AUTHORITY["EPSG","9122"]],
##         AUTHORITY["EPSG","4181"]],
##     PROJECTION["Transverse_Mercator"],
##     PARAMETER["latitude_of_origin",49.8333333333333],
##     PARAMETER["central_meridian",6.16666666666667],
##     PARAMETER["scale_factor",1],
##     PARAMETER["false_easting",80000],
##     PARAMETER["false_northing",100000],
```

```
##     UNIT["metre",1,
##         AUTHORITY["EPSG","9001"]],
##     AUTHORITY["EPSG","2169"]]
```

```r
cat(system("projinfo EPSG:2169 -o WKT2_2018", intern=TRUE), sep="\n")
```

```
## WKT2_2018 string:
## PROJCRS["Luxembourg 1930 / Gauss",
##     BASEGEOGCRS["Luxembourg 1930",
##         DATUM["Luxembourg 1930",
##             ELLIPSOID["International 1924",6378388,297,
##                 LENGTHUNIT["metre",1]]],
##         PRIMEM["Greenwich",0,
##             ANGLEUNIT["degree",0.0174532925199433]],
##         ID["EPSG",4181]],
##     CONVERSION["Luxembourg Gauss",
##         METHOD["Transverse Mercator",
##             ID["EPSG",9807]],
##         PARAMETER["Latitude of natural origin",49.8333333333333,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8801]],
##         PARAMETER["Longitude of natural origin",6.16666666666667,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8802]],
##         PARAMETER["Scale factor at natural origin",1,
##             SCALEUNIT["unity",1],
##             ID["EPSG",8805]],
##         PARAMETER["False easting",80000,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8806]],
##         PARAMETER["False northing",100000,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8807]]],
##     CS[Cartesian,2],
##         AXIS["northing (X)",north,
##             ORDER[1],
##             LENGTHUNIT["metre",1]],
##         AXIS["easting (Y)",east,
##             ORDER[2],
##             LENGTHUNIT["metre",1]],
##     USAGE[
##         SCOPE["unknown"],
##         AREA["Luxembourg"],
##         BBOX[49.44,5.73,50.19,6.53]],
##     ID["EPSG",2169]]
```

```r
cat(system("projinfo -s EPSG:4326 -t EPSG:2169 -o PROJ", intern=TRUE), sep="\n")
```

```
## Candidate operations found: 2
## -------------------------------------
## Operation n°1:
##
## unknown id, Inverse of Luxembourg 1930 to WGS 84 (3) + Luxembourg Gauss, 1 m, Luxembourg
##
## PROJ string:
```

```
## +proj=pipeline +step +proj=axisswap +order=2,1 +step +proj=unitconvert +xy_in=deg +xy_out=rad +step +
##
## ------------------------------------
## Operation n°2:
##
## unknown id, Inverse of Luxembourg 1930 to WGS 84 (4) + Luxembourg Gauss, 1 m, Luxembourg
##
## PROJ string:
## +proj=pipeline +step +proj=axisswap +order=2,1 +step +proj=unitconvert +xy_in=deg +xy_out=rad +step +
```

Raster

The GHSL data from https://ghsl.jrc.ec.europa.eu/ and provided in the `ghsl_pop` column is not documented here (yet). Using the GeoTIFF file (which turns out to be `+proj=moll`), we can read using a GDAL driver, first into memory, then proxy:

```r
library(stars)
```

```
## Loading required package: abind
```

```r
system.time(ghsl0 <- read_stars("../data/ghsl.tiff", proxy=FALSE))
```

```
##    user  system elapsed
##   0.005   0.000   0.005
```

```r
ghsl0
```

```
## stars object with 2 dimensions and 1 attribute
## attribute(s):
##    ghsl.tiff
##  Min.   :  0.00
##  1st Qu.:  0.00
##  Median :  0.00
##  Mean   : 10.68
##  3rd Qu.:  0.00
##  Max.   :588.57
##  NA's   :554
## dimension(s):
##   from  to  offset delta                        refsys point values
## x    1 246  436750   250 +proj=moll +lon_0=0 +x_0=... FALSE   NULL [x]
## y    1 309 5892750  -250 +proj=moll +lon_0=0 +x_0=... FALSE   NULL [y]
```

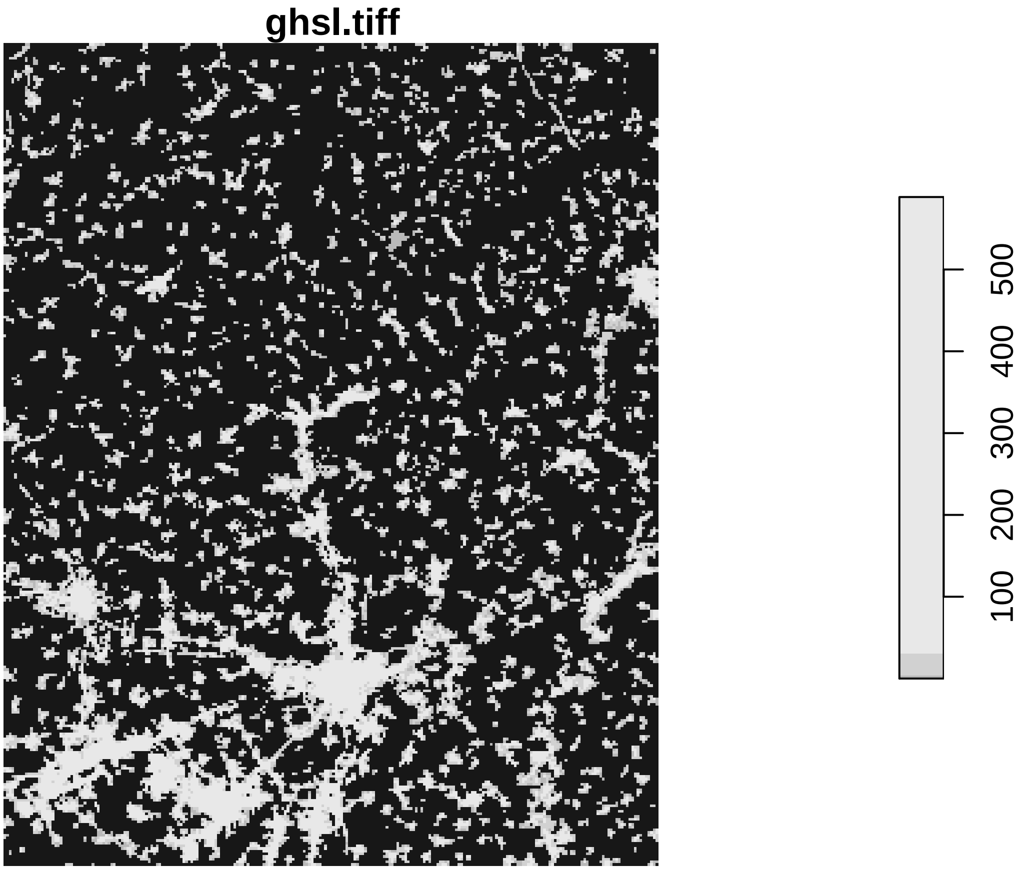
```r
system.time(ghsl1 <- read_stars("../data/ghsl.tiff", proxy=TRUE))
```

```
##    user  system elapsed
##   0.003   0.000   0.003
```

```r
ghsl1
```

```
## stars_proxy object with 1 attribute in file:
## $ghsl.tiff
## [1] "../data/ghsl.tiff"
##
## dimension(s):
##   from  to  offset delta                        refsys point values
## x    1 246  436750   250 +proj=moll +lon_0=0 +x_0=... FALSE   NULL [x]
## y    1 309 5892750  -250 +proj=moll +lon_0=0 +x_0=... FALSE   NULL [y]
```

```r
plot(ghsl0)
```



Using the aggregate method on the input raster in memory warped to the Luxembourg transverse Mercator projection, by the municipality boundaries in `lux_tmerc`, we can recover the population counts.

```r
system.time(ghsl_sum0 <- aggregate(st_warp(ghsl0, crs=2169, cellsize=250, use_gdal=FALSE), lux_tmerc, su
```

```
##    user  system elapsed
##   0.782   0.028   0.821
```

Using the proxy in this case takes about the same time:

```r
system.time(ghsl_sum1 <- aggregate(st_warp(ghsl1, crs=2169, cellsize=250, use_gdal=FALSE), lux_tmerc, su
```

```
##    user  system elapsed
##   0.656   0.012   0.678
```

```r
system.time(ghsl_sum2 <- aggregate(ghsl0, st_transform(lux, crs=st_crs(ghsl0)$proj4string), sum))
```

```
##    user  system elapsed
##   0.513   0.003   0.526
```

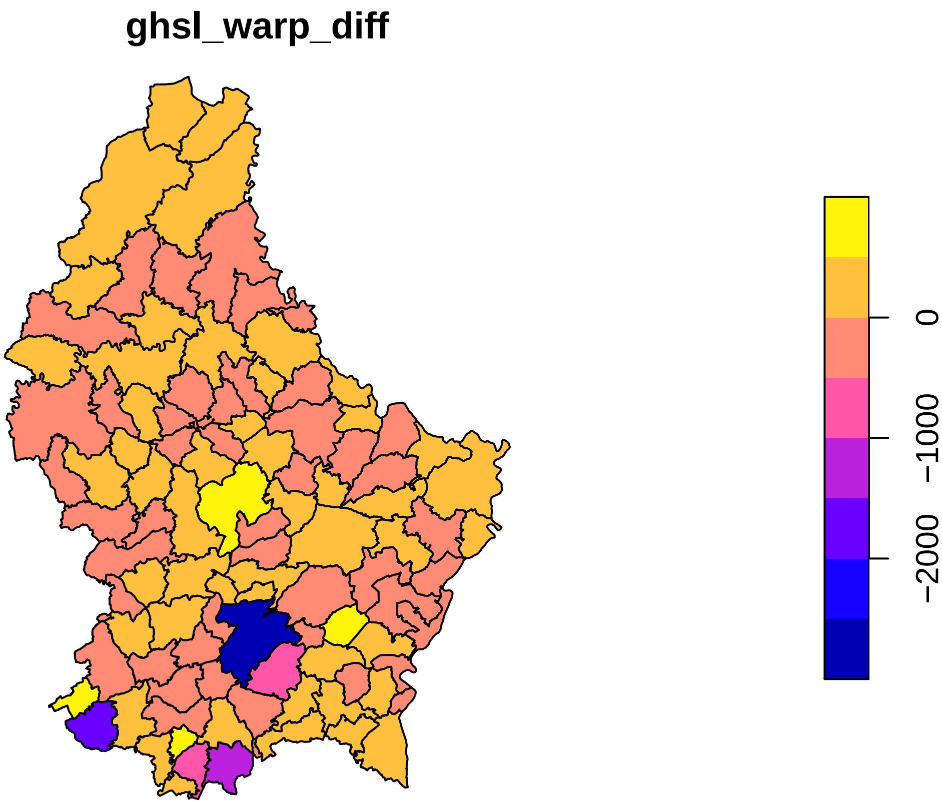
The output values following warping are closely aligned with, but differ a little from those included in the vector object read to begin with; it looks as though the vector object was transformed to match the raster before aggregation:

```r
summary(cbind(orig=lux_tmerc$ghsl_pop, warp=ghsl_sum0$ghsl.tiff, warp_proxy=ghsl_sum1$ghsl.tiff, moll=gh
```

```
##       orig                 warp           warp_proxy           moll
##  Min.   :   815.1   Min.   :   806   Min.   :   806   Min.   :   815.1
```

```
##  1st Qu.:  1726.2   1st Qu.:  1679   1st Qu.:  1679   1st Qu.:  1726.2
##  Median :  3072.4   Median :  2953   Median :  2953   Median :  3072.4
##  Mean   :  5542.2   Mean   :  5524   Mean   :  5524   Mean   :  5542.2
##  3rd Qu.:  5190.2   3rd Qu.:  5519   3rd Qu.:  5519   3rd Qu.:  5190.2
##  Max.   :106144.0   Max.   :103598   Max.   :103598   Max.   :106144.0
```

```
lux_tmerc$ghsl_tiff <- ghsl_sum0$ghsl.tiff
lux_tmerc$ghsl_warp_diff <- lux_tmerc$ghsl_tiff - lux_tmerc$ghsl_pop
plot(lux_tmerc[,"ghsl_warp_diff"])
```




```
st_write(lux_tmerc, "../data/lux_tmerc.gpkg", delete_dsn=TRUE)
```

```
## Deleting source `../data/lux_tmerc.gpkg' using driver `GPKG'
## Updating layer `lux_tmerc' to data source `../data/lux_tmerc.gpkg' using driver `GPKG'
## Writing 102 features with 16 fields and geometry type Multi Polygon.
```


Evers, Kristian, and Thomas Knudsen. 2017. *Transformation Pipelines for Proj.4*. https://www.fig.net/resources/proceedings/fig_proceedings/fig2017/papers/iss6b/ISS6B_evers_knudsen_9156.pdf.

Knudsen, Thomas, and Kristian Evers. 2017. *Transformation Pipelines for Proj.4*. https://meetingorganizer.copernicus.org/EGU2017/EGU2017-8050.pdf.

Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. 2019. *Geocomputation with R*. Boca Raton, FL: Chapman and Hall/CRC. https://geocompr.robinlovelace.net/.

Pebesma, Edzer. 2018. "Simple Features for R: Standardized Support for Spatial Vector Data." *The R Journal* 10 (1): 439–46. https://doi.org/10.32614/RJ-2018-009.

Pebesma, Edzer, and Roger S. Bivand. n.d. *Spatial Data Science; Uses Cases in R*. CRC. https://r-spatial.org/book/.