# Geospatial R(efresher)

*J. J. Fain*

*1/25/2019*

## Outline

This is designed to be a refresher for data manipulation in R. First we will start with simple tabular data, then we will move on to spatial vector data. If this is your first time using R for spatial applications, you may be pleasantly surprised to discover just how similar tabular and spatial data behave.

---

## Packages & Setup

Start by installing the required packages. We are using *dplyr* with the simple features library, *sf*. *Ggplot2* will be our plotting library for vector data, *ggmap* will be used for rasters. *Rgdal* operates in the background and gives us access to a powerful library of C functions that can significantly speed-up our analyses. *Spdep* has a ton of functions for spatial statistics and point pattern analysis.

Notice that this is just checking if packages are installed, and installing them if they aren't. Behind the scenes, this downloads and compiles binaries of the packages.

```
if(!require(ggplot2)){
  install.packages('ggplot2')
}
```

```
## Loading required package: ggplot2
```

```
if(!require(rgdal)){
  install.packages('rgdal')
}
```

```
## Loading required package: rgdal
```

```
## Loading required package: sp
```

```
## rgdal: version: 1.3-3, (SVN revision 759)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.1.3, released 2017/20/01
##  Path to GDAL shared files: /Library/Frameworks/R.framework/Versions/3.5/Resources/library/rgdal/gdal
##  GDAL binary built with GEOS: FALSE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: /Library/Frameworks/R.framework/Versions/3.5/Resources/library/rgdal/pr
##  Linking to sp version: 1.3-1
```

```
if(!require(gdalUtils)){
  install.packages('gdalUtils')
}
```

```
## Loading required package: gdalUtils
```

```
if(!require(raster)){
  install.packages('raster')
}
```

```
## Loading required package: raster
```

```r
if(!require(sf)){
  install.packages('sf')
}
```

```
## Loading required package: sf
```

```
## Linking to GEOS 3.6.1, GDAL 2.1.3, proj.4 4.9.3
```

```r
if(!require(spdep)){
  install.packages('spdep')
}
```

```
## Loading required package: spdep
```

```
## Loading required package: Matrix
```

```
## Loading required package: spData
```

```
## To access larger datasets in this package, install the spDataLarge
## package with: `install.packages('spDataLarge',
## repos='https://nowosad.github.io/drat/', type='source'))`
```

```r
if(!require(ggmap)){
  install.packages('ggmap')
}
```

```
## Loading required package: ggmap
```

```r
if(!require(dplyr)){
  install.packages('dplyr')
}
```

```
## Loading required package: dplyr
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:raster':
##
##     intersect, select, union
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Now that all packages are installed we will use **require** to attach the packages above.

```r
# raster
require(raster)
# ggplot2
require(ggplot2)
# rgdal
require(rgdal)
# gdalUtils
require(gdalUtils)
```

```r
# sf
require(sf)
# spdep
require(spdep)
# ggmap
require(ggmap)
# dplyr
require(dplyr)
```

### Set Working Directory

The function **setwd** changes the R session's current working directory. This is equivalent to the cd/chdir commands that you may be familiar with. We can double check that **setwd** changed our directory to the place we intended with the command **getwd**, which simply returns the file path of the current directory.

Also note that filepaths in R are somewhat system dependent. You will either need to use double backslashes (\\) if you are more comfortable with the Windows style, or a single forward slash (/) to separate directories like you would on a Mac OS.

```r
# Point this to your workshop file folder
setwd("/Users/fainjj/Documents/Coding/Workshop")

# Make sure this matches what you typed above
getwd()
```

```
## [1] "/Users/fainjj/Documents/Coding/Workshop"
```

### Explore Files

At this point it is a good idea to use the **list.files** function to make sure that the things we need moving forward are in the directory we just moved to.

```r
list.files()
```

```
##  [1] "AOI.cpg"                          "AOI.dbf"
##  [3] "AOI.prj"                          "AOI.sbn"
##  [5] "AOI.sbx"                          "AOI.shp"
##  [7] "AOI.shx"                          "country_info.csv"
##  [9] "Geospatial_Refresher.pdf"         "Geospatial_Refresher.Rmd"
## [11] "GeospatialRefresher.R"            "GT_Results_17_18.csv"
## [13] "GT_Results_17_18.xlsx"            "ne_110m_admin_0_countries.cpg"
## [15] "ne_110m_admin_0_countries.dbf" "ne_110m_admin_0_countries.prj"
## [17] "ne_110m_admin_0_countries.shp" "ne_110m_admin_0_countries.shx"
## [19] "ne_110m_admin_0_countries.zip" "R_RS"
## [21] "Untitled.html"                    "Untitled.pdf"
## [23] "Untitled.R"                       "Untitled.Rmd"
## [25] "Workshop_Data"                    "Workshop.nb.html"
## [27] "Workshop.Rmd"                     "Workshop.Rproj"
```

---

# Reading Tabular Data

We should have a shapefile of 0-level administrative boundaries as well as a csv of information about those countries. We can deal with the shapefile later on.

```
dvlp <- read.csv('country_info.csv', stringsAsFactors = FALSE)
```

Now we can use the **head** and **tail** functions to check the data we loaded.

```
head(dvlp)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

```
tail(dvlp)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1699 Zimbabwe 1982  7636524    Africa  60.363  788.8550
## 1700 Zimbabwe 1987  9216418    Africa  62.351  706.1573
## 1701 Zimbabwe 1992 10704340    Africa  60.377  693.4208
## 1702 Zimbabwe 1997 11404948    Africa  46.809  792.4500
## 1703 Zimbabwe 2002 11926563    Africa  39.989  672.0386
## 1704 Zimbabwe 2007 12311143    Africa  43.487  469.7093
```

Looks good! Since we have a large table, we will use **head** a lot to avoid printing every row since that would quickly fill-up the console.

---

# Functions

## Definitions

Defining functions in R is very similar to variable assignment. Here we will make our first function to raise 10 to any arbitrary exponent.

```
pow10 <- function(pwr){
  return(10^pwr)
}
```

Take this function for a spin to make sure it behaves as expected.

```
pow10(3)
```

```
## [1] 1000
```

```
pow10(4)
```

```
## [1] 10000
```

### Closures: Functions really do write themselves

We can also write a *closure* which is a function for creating functions. Think of these as function templates. Below is a *closure* which further generalizes our exponential functions. This allows us to use the same template to quickly make functions for similar tasks such as finding the nth root of a number.

```
nth_rt <- function(pwr){
  function(b){b^(1/pwr)}
}

sqrt <- nth_rt(2)

sqrt(9)
```

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

```
sqrt(256)
```

```
## [1] 16
```

```
sqrt(39601)
```

```
## [1] 199
```

```
cbrt <- nth_rt(3)

cbrt(64)
```

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

```
cbrt(729)
```

```
## [1] 9
```

```
cbrt(1728)
```

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

More about closures: http://adv-r.had.co.nz/Functional-programming.html#closures

---

## Vectors & Data Frames

### Vectors

Vectors are the most basic groupings of information in R. **typeof** tells you the data type stored in the vector. **class** tells you what mode the vector is (logical, numeric, character). **length** does exactly what you would expect and returns the numeric length of your vector.

```
x1 <- c(1,2,3)
c(typeof(x1), class(x1), length(x1))
```

```
## [1] "double"  "numeric" "3"
```

```
x2 <- c('a','b','c')
c(typeof(x2), class(x2), length(x2))
```

```
## [1] "character" "character" "3"
```

```r
x3 <- c(1, TRUE, 3, 'four')
c(typeof(x3), class(x3), length(x3))
```

```
## [1] "character" "character" "4"
```

Notice that in x3, everything became a character. Vectors can not be of mixed type, so they are silently converted to the same type. This is explained in the details of the **c** function's help documentation: *The output type is determined from the highest type of the components in the hierarchy NULL < raw < logical < integer < double < complex < character < list < expression.*

## Data Frames

Now that we have two vectors, we can mash them into a data frame.

```r
df <- data.frame(x1, x2)
View(df)
```

Back to our countries data, let's go ahead and check out the structure and dimensions with **str** and **dim**.

```r
str(dvlp)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: chr  "Asia" "Asia" "Asia" "Asia" ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

```r
dim(dvlp)
```

```
## [1] 1704    6
```

```r
names(dvlp)
```

```
## [1] "country"   "year"      "pop"       "continent" "lifeExp"   "gdpPercap"
```

```r
summary(dvlp)
```

```
##    country               year           pop               continent
##  Length:1704        Min.   :1952   Min.   :6.001e+04   Length:1704
##  Class :character   1st Qu.:1966   1st Qu.:2.794e+06   Class :character
##  Mode  :character   Median :1980   Median :7.024e+06   Mode  :character
##                     Mean   :1980   Mean   :2.960e+07
##                     3rd Qu.:1993   3rd Qu.:1.959e+07
##                     Max.   :2007   Max.   :1.319e+09
##     lifeExp        gdpPercap
##  Min.   :23.60   Min.   :   241.2
##  1st Qu.:48.20   1st Qu.:  1202.1
##  Median :60.71   Median :  3531.8
##  Mean   :59.47   Mean   :  7215.3
##  3rd Qu.:70.85   3rd Qu.:  9325.5
##  Max.   :82.60   Max.   :113523.1
```

We see that there are 1704 rows (observations), each with 6 columns (variables). This is a 5-year development index data set for a bunch of countries. The **names** function gives us the variable (column) names which should match the csv header row. The **summary** function gives us a bunch of information very quickly, but it isn't particularly useful at this point since there are multiple countries with 5 entries each.

---

# Sequences and Indexing in R

## Sequences

These are all equivalent ways of creating a numeric sequence from 1 to 3.

```
c(1, 2, 3) # too much typing
```

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

```
1:3 # faster, but somewhat hard to read
```

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

```
seq(1, 3) # powerful and versatile
```

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

All of those are perfectly valid but **seq** takes an extra few arguments which tend to make it the most useful in practice. The 'by' argument lets you choose the size of your steps between each item in the sequence. Check out the other arguments on the help page.

```
seq(1, 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

## Indexing

You can use square brackets to subset data frames. This follows the pattern *data[row, col]*. Leaving either of the indices blank will select all of that row or column.

```
dvlp[1,] # First row, all columns
```

```
##       country year     pop continent lifeExp gdpPercap
## 1 Afghanistan 1952 8425333      Asia  28.801  779.4453
```

```
dvlp[,1] # All rows, first column
```

```
##  [1] "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan"
##  [6] "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan"
## [11] "Afghanistan" "Afghanistan" "Albania"     "Albania"     "Albania"
## [16] "Albania"     "Albania"     "Albania"     "Albania"     "Albania"
## [21] "Albania"     "Albania"     "Albania"     "Albania"     "Algeria"
##  [ reached getOption("max.print") -- omitted 1679 entries ]
```

```
dvlp[1,5] # First row, fifth column
```

```
## [1] 28.801
```

You can also subset by sequences.

```
dvlp[1:3, 5]
```

```
## [1] 28.801 30.332 31.997
```

```
# You can mix your sequence-building methods when subsetting
dvlp[1:5, c(1, 4, 5)] # indices don't even have to be continuous
```

```
##       country continent lifeExp
## 1 Afghanistan      Asia  28.801
## 2 Afghanistan      Asia  30.332
## 3 Afghanistan      Asia  31.997
## 4 Afghanistan      Asia  34.020
## 5 Afghanistan      Asia  36.088
```

```
dvlp[1:6, ] # equivalent to the head() function
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

Even better, we can take advantage of the column names we saw earlier.

```
s1 <- dvlp[ , 'year']
head(s1)
```

```
## [1] 1952 1957 1962 1967 1972 1977
```

```
# The dollar sign notation also works for subsetting columns
s1 <- dvlp$year
head(s1)
```

```
## [1] 1952 1957 1962 1967 1972 1977
```

Use **nrow** and **seq** to create a new data frame containing every 100th row of the dvlp df, starting at row 100.

```
dvlp[seq(100, nrow(dvlp), 100), ]
```

```
##                     country year        pop continent lifeExp  gdpPercap
## 100              Bangladesh 1967   62821884      Asia  43.453   721.1861
## 200            Burkina Faso 1987    7586551    Africa  49.557   912.0631
## 300                   China 2007 1318683096      Asia  72.961  4959.1149
## 400          Czech Republic 1967    9835109    Europe  70.380 11399.4449
## 500                 Eritrea 1987    2915959    Africa  46.453   521.1341
## 600                  Greece 2007   10706290    Europe  79.483 27538.4119
## 700                   India 1967  506000000      Asia  47.193   700.7706
## 800                   Japan 1987  122091325      Asia  78.670 22375.9419
## 900                 Liberia 2007    3193942    Africa  45.678   414.5073
## 1000               Mongolia 1967    1149500      Asia  51.253  1226.0411
## 1100            New Zealand 1987    3317166   Oceania  74.320 19007.1913
## 1200               Paraguay 2007    6667147  Americas  71.752  4172.8385
## 1300 Sao Tome and Principe 1967      70787    Africa  54.425  1384.8406
## 1400                Somalia 1987    6921858    Africa  44.501  1093.2450
## 1500                  Syria 2007   19314747      Asia  74.143  4184.5481
## 1600         United Kingdom 1967   54959000    Europe  71.360 14142.8509
## 1700               Zimbabwe 1987    9216418    Africa  62.351   706.1573
```

**Seq** gives us a list of row indices from 100 to the max number of rows in dvlp, counting by 100.

```
seq(from = 100, to = nrow(dvlp), by = 100)
```

```
##  [1]  100  200  300  400  500  600  700  800  900 1000 1100 1200 1300 1400
## [15] 1500 1600 1700
```

All of this is helpful to know, but numeric indices aren't really all that useful in practice. We will look at a few other ways to subset things using conditional expressions

```
(dvlp$lifeExp < 30)[1:6] # this is a logical vector
```

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

```
typeof(dvlp$lifeExp < 30)
```

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

```
length(dvlp$lifeExp < 30) # should be a logical vector of the same length as our input
```

```
## [1] 1704
```

```
# The logical vector returned by dvlp$lifeExp < 30 can be used to subset
dvlp[dvlp$lifeExp < 30, ]
```

```
##          country year     pop continent lifeExp gdpPercap
## 1    Afghanistan 1952 8425333      Asia  28.801  779.4453
## 1293      Rwanda 1992 7290203    Africa  23.599  737.0686
```

---

# Introducing dplyr

Dplyr is a powerful data manipulation package. It is also incredibly fast since most of the functions are really just convenience wrappers for underlying C functions.

If you just want to move a column to the front of a data frame, you can use some of *dplyr*'s super handy tools. Let's move the continents column to the front using **select** and **everything**.

```
dvlp2 <- dplyr::select(dvlp, 'continent', everything())
head(dvlp2)
```

```
##   continent     country year      pop lifeExp gdpPercap
## 1      Asia Afghanistan 1952  8425333  28.801  779.4453
## 2      Asia Afghanistan 1957  9240934  30.332  820.8530
## 3      Asia Afghanistan 1962 10267083  31.997  853.1007
## 4      Asia Afghanistan 1967 11537966  34.020  836.1971
## 5      Asia Afghanistan 1972 13079460  36.088  739.9811
## 6      Asia Afghanistan 1977 14880372  38.438  786.1134
```

Dplyr's **select** and **filter** functions allow you to subset data intuitively.

```
output <- dplyr::select(dvlp, country, year, lifeExp)
head(output)
```

```
##       country year lifeExp
## 1 Afghanistan 1952  28.801
## 2 Afghanistan 1957  30.332
## 3 Afghanistan 1962  31.997
## 4 Afghanistan 1967  34.020
## 5 Afghanistan 1972  36.088
## 6 Afghanistan 1977  38.438
```

```
filter(dvlp, lifeExp < 30)
```

```
##       country year     pop continent lifeExp gdpPercap
```

```
## 1 Afghanistan 1952 8425333      Asia  28.801  779.4453
## 2      Rwanda 1992 7290203    Africa  23.599  737.0686
```

### More filtering and selecting using the pipe!

This is the pipe: %>% It passes the left-hand side as the first argument to the function on the right-hand side. This lets you chain a bunch of operations together without nesting your functions. It is far more readable but can sometimes be a pain to debug.

Let's look at how it can be used to make our code more human-readable.

```r
# Nested functions make it unclear what is our data, and what are variable names. They must be read fro
ind.dvlp <- dplyr::select(filter(dvlp, country == 'India'), year, lifeExp)

# The pipe streamlines this process, allowing you to read from top to bottom through the workflow.
ind.dvlp <- dvlp %>%
  filter(country == 'India') %>%
  select(year, lifeExp)

ind.dvlp
```

```
##    year lifeExp
## 1  1952  37.373
## 2  1957  40.249
## 3  1962  43.605
## 4  1967  47.193
## 5  1972  50.651
## 6  1977  54.208
## 7  1982  56.596
## 8  1987  58.553
## 9  1992  60.223
## 10 1997  61.765
## 11 2002  62.879
## 12 2007  64.698
```

#### Note:

The package *magrittr* adds a few new types of pipe that I *love* to use. My favorite is the reverse assignment pipe %<>% which passes the variable on the left-hand side into a pipeline, then reassigns the result to the variable on the left-hand side.

It looks like this: x %<>% f1() %>% f2() But works like this: x <- x %>% f1() %>% f2()

### Negative indices

We can also use **select** to drop variables from the table. If we only select entries from India, having the country variable becomes redundant. We could do something like this instead:

```r
dvlp %>%
  filter(country == 'India') %>%
  filter(year > 1992) %>%
  select(-country)
```

```
##   year       pop continent lifeExp gdpPercap
## 1 1997 959000000      Asia  61.765  1458.817
```

```
## 2 2002 1034172547      Asia  62.879  1746.769
## 3 2007 1110396331      Asia  64.698  2452.210
```

Rather than dropping variables, we can use **mutate** to add new columns. This calculates new values row-by-row.

```
dvlp %>%
  mutate(gdpTotal = pop*gdpPercap) %>%
  head()
```

```
##       country year      pop continent lifeExp gdpPercap    gdpTotal
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453  6567086330
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530  7585448670
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007  8758855797
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971  9648014150
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811  9678553274
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134 11697659231
```

We can also use the **group_by** function to gather similar observations into distinct bundles that we can then perform operations on.

```
dvlp %>%
  group_by(year) %>%
  summarise(avgLifeExp = median(lifeExp)) %>%
  head()
```

```
## # A tibble: 6 x 2
##    year avgLifeExp
##   <int>      <dbl>
## 1  1952       45.1
## 2  1957       48.4
## 3  1962       50.9
## 4  1967       53.8
## 5  1972       56.5
## 6  1977       59.7
```

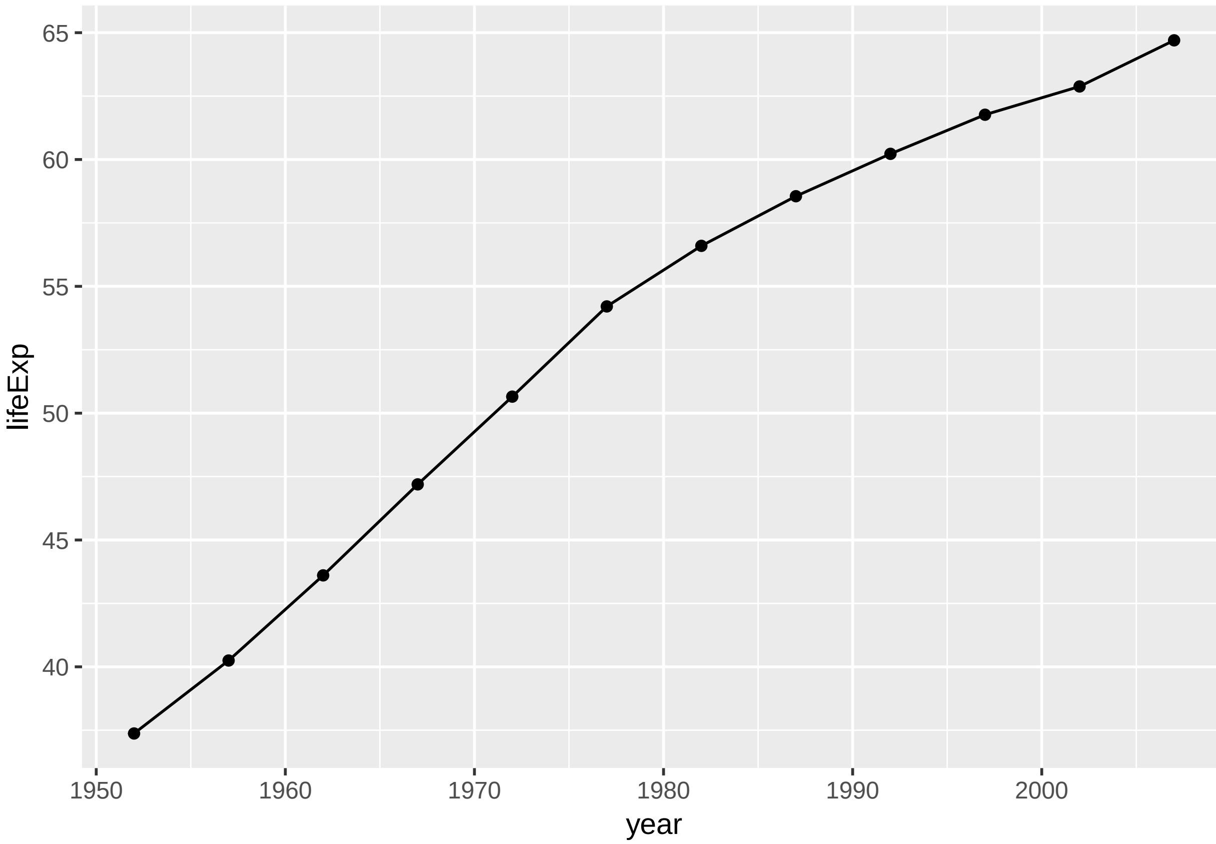
```
dvlp %>%
  filter(year %in% c(2002, 2007)) %>%
  group_by(country) %>%
  summarise(meanGDP = mean(gdpPercap)) %>%
  head()
```

```
## # A tibble: 6 x 2
##   country     meanGDP
##   <chr>         <dbl>
## 1 Afghanistan    851.
## 2 Albania       5271.
## 3 Algeria       5756.
## 4 Angola        3785.
## 5 Argentina    10789.
## 6 Australia    32562.
```

# Ggplot2 and a new syntax

The plotting library *ggplot2* allows you to build visualizations in an intuitive layer-by layer fashion. To represent this process, *ggplot2* uses a '+' to add new layers. This symbol also acts somewhat like the pipe, insomuch as it passes the first argument given to **ggplot** to all subsequent layers.

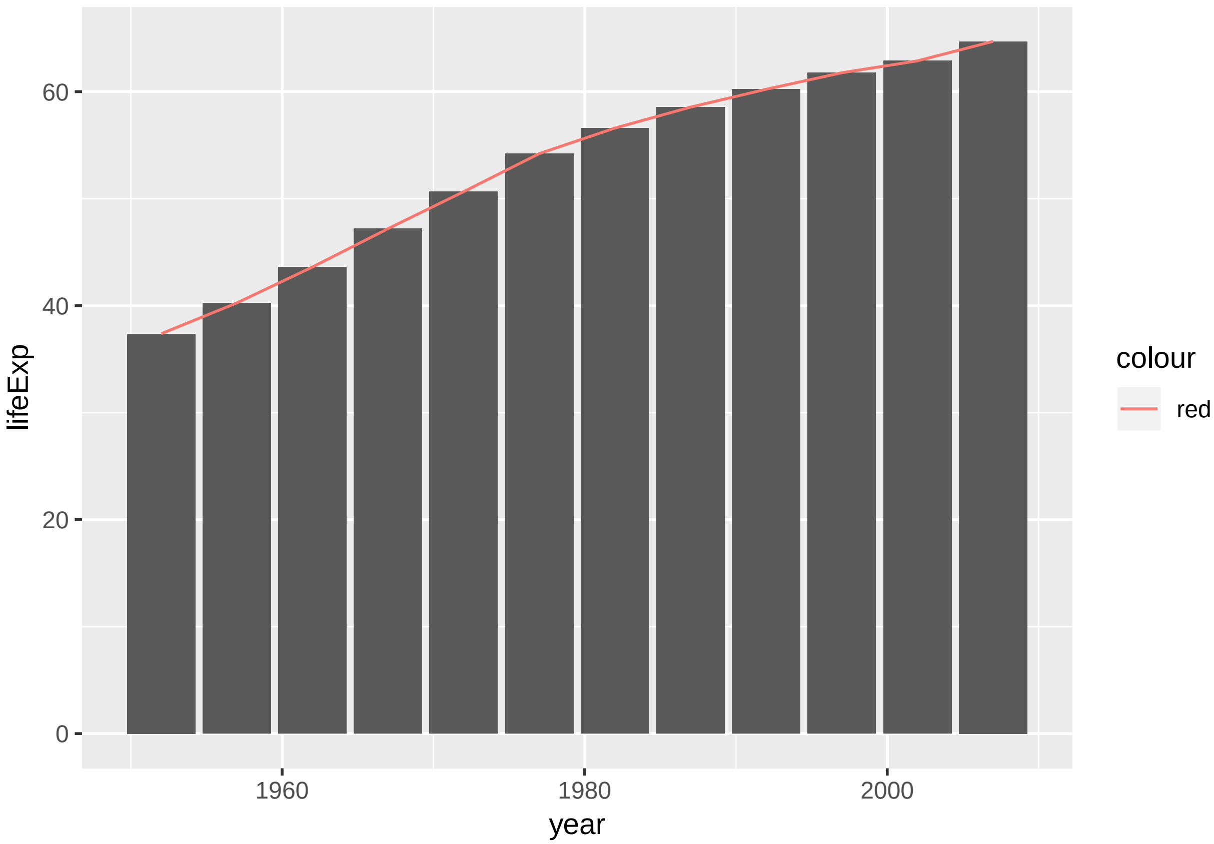
```
ggplot(ind.dvlp, mapping = aes(x = year, y = lifeExp)) +
  geom_line() +
  geom_point()
```




Because *ggplot2* is an additive process, we can define a basemap and store it to a variable for use later.

```
base <- ggplot(data = ind.dvlp, mapping = aes(x = year, y = lifeExp))

base + geom_bar(stat = 'identity') + geom_line(aes(color = 'red'))
```

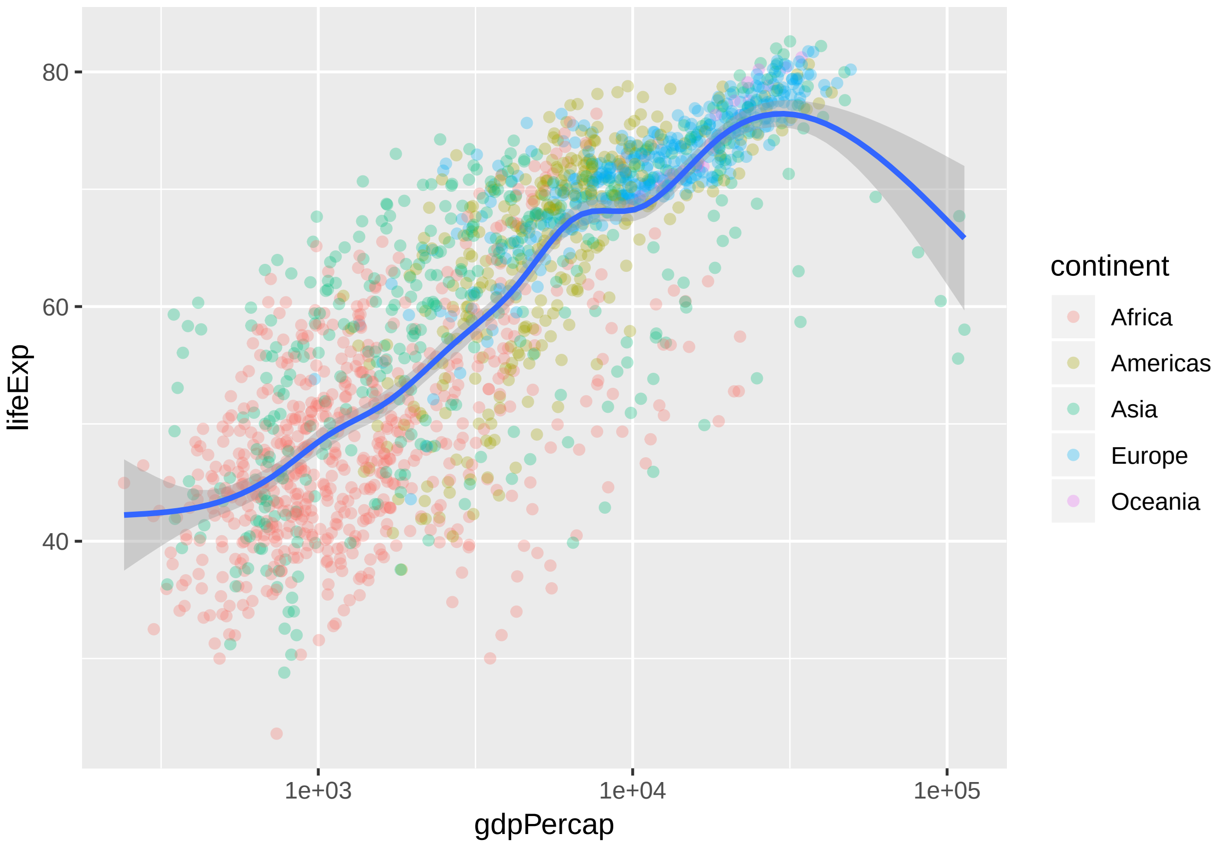



Now we can combine all of these techniques to create some very impressive plots.

```
gdp_exp <- ggplot(dvlp, mapping = aes(gdpPercap, lifeExp))

gdp_exp +
  geom_point(alpha = 0.3, mapping = aes(color = continent)) +
  scale_x_log10() +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```




## Simple Features with the SF package

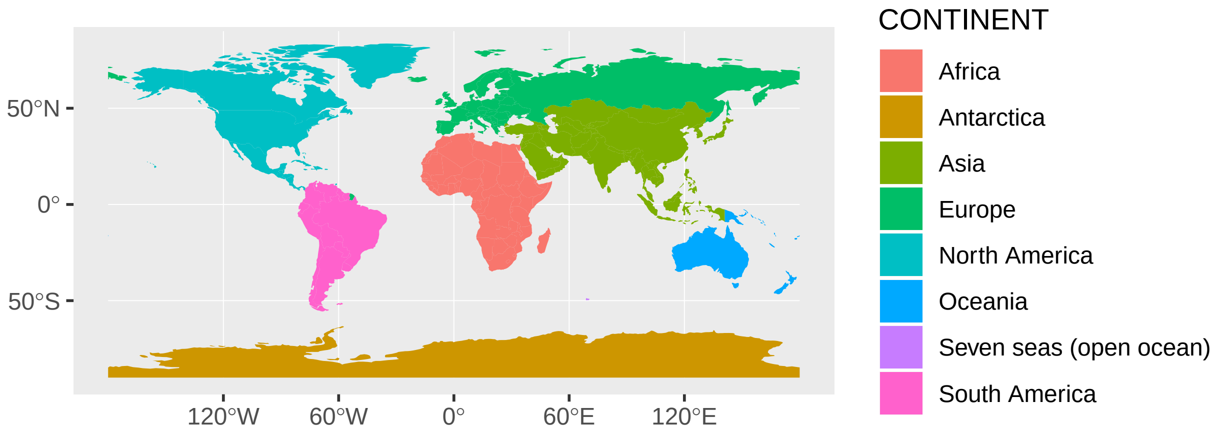
The simple features library is a fast and easy way to store geometries. Under the hood, it is very similar to the way Post handles spatial data.

```
unzip('ne_110m_admin_0_countries.zip')
countries <- st_read('ne_110m_admin_0_countries.shp')
```

```
## Reading layer `ne_110m_admin_0_countries' from data source `/Users/fainjj/Documents/Coding/Workshop/r
## Simple feature collection with 177 features and 94 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -180 ymin: -90 xmax: 180 ymax: 83.64513
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

*Sf* also plays nicely with *ggplot2*.

```
ggplot() +
  geom_sf(data = countries, color = NA, mapping = aes(fill =CONTINENT)) +
  coord_sf()
```




---

## Acknowledgements

- Adapted in part from *Data Visualization in R* Workshop by K. Arthur Endsley
    - http://karthur.org/
    - https://github.com/arthur-e
- *Advanced R* by Hadley Wickham.
    - Available at http://adv-r.had.co.nz/ or in print

---

This handout was written in R Markdown