Introduction

There are many ways to handle spatial data (e.g. ArcGIS, QGIS, GRASS). In this course, we will do all of our spatial computations with *R*. Why is that a good idea?

- ArcGIS is proprietary, expensive, slow, and buggy. You have to learn both Arc and Python to use it intelligently.
- QGIS is great but also focused on clicking, pointing, and mapping. Most spatial operations are powered by C^{++} libraries.
- *R* is free, open-source, and driven by huge community of developers. There 100+ packages in r-spatial today. Most spatial operations are powered by C^{++} libraries.
- With *R* you can create, modify, and analyze spatial data in the same place!

Spatial data types

There are “only” two types of spatial data:

Vector data

- Points
- Lines
- Polygons

Raster data

- Continuous
- Categorical/ binary

Before we can work with vector data, we have to do a detour.

Introducing the `tidyverse`

Why not base R?

Many base *R* functions were written at least 10 or 20 years ago. It is difficult to change them without breaking code, so most innovation occurs in new packages.

We are going to work with the `tidyverse` which is a collection of packages which follow a new and simple to use paradigm for handling data. Learn more at `www.tidyverse.org`.

The main advantage of using the `tidy`-approach in this course is that it makes handling *Simple Features* — the new paradigm for vector data in *R* from the `sf`-package — considerably easier.

The `tidyverse` is build around `dplyr` and its grammatical approach to data wrangling.

Installing the tidyverse

```
install.packages("tidyverse")
```

Require vs Library

```
require(tidyverse)

## Loading required package: tidyverse

## -- Attaching packages --------- tidyverse 1.2.1 --

## √ ggplot2 3.0.0     √ purrr   0.2.5
## √ tibble  1.4.2     √ dplyr   0.7.6
## √ tidyr   0.8.1     √ stringr 1.3.1
## √ readr   1.1.1     √ forcats 0.3.0

## -- Conflicts ------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

A `tibble` is a `data.frame`

Any tibble is also a data frame, but not all data frames are tibbles.

```
i_df <- iris # built-in data
class(i_df)

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

```
i_tbl <- as_tibble(iris)
class(i_tbl)

## [1] "tbl_df"     "tbl"        "data.frame"
```

Printing

Tibbles print only the first 10 rows, and all the columns that fit on screen.

```
i_tbl  # or print(i_tbl)

## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

You can use tibbles like data frames

```
i_tbl <- i_tbl[1:5, ] # Subset the first 5 rows
i_tbl$Sepal.Length # Extract by name

## [1] 5.1 4.9 4.7 4.6 5.0
```

```
i_tbl[["Sepal.Length"]] # Extract by name

## [1] 5.1 4.9 4.7 4.6 5.0
```

```
i_tbl[[1]] # Extract by position

## [1] 5.1 4.9 4.7 4.6 5.0
```

Why [[ ]] and not [ ]?

Contrary to base *R*, subsetting variables with `[ ]` always returns another tibble:

```
i_tbl["Sepal.Length"] # or i_tbl[1]

## # A tibble: 5 x 1
##   Sepal.Length
##          <dbl>
## 1          5.1
## 2          4.9
## 3          4.7
## 4          4.6
## 5          5
```

Use `[[ ]]` if you want vectors.

Conversion

Not all functions work with tibbles, you can always use `as.data.frame()` to turn a tibble back to a `data.frame`.

The main reason that some older functions do not work with tibbles is the `[` function.

```r
i_tbl <- as_tibble(iris)
i_df <- as.data.frame(i_tbl)
class(i_df)

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

Base *R* functions usually use periods and tidy functions use an underscore.

Reading and writing data with tibbles

The function `read_csv` is much faster and more convenient than base *R*. It does not convert string to factors.

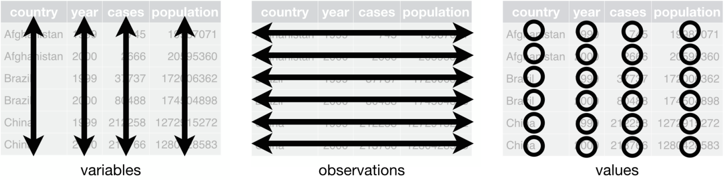
```
cps08 <- read_csv("./data/cps08.csv")
print(cps08, n=2)

## # A tibble: 7,711 x 5
##     ahe  year bachelor female   age
##   <dbl> <int> <chr>    <chr>  <int>
## 1  38.5  2008 yes      no        33
## 2  12.5  2008 yes      no        31
## # ... with 7,709 more rows
```

Use `write_csv(cps08, "newfile.csv")` for saving data.

What is tidy data?

R is so flexible, you can hold many data sets in memory and even make a `data.frame` or `list` of data sets.



Tidy data imposes structure, just like Stata or Excel:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Subsetting dplyr-style I

Subsetting variables is easy using dplyr's select() function:

```
select(cps08, bachelor)

## # A tibble: 7,711 x 1
##   bachelor
##   <chr>
## 1 yes
## 2 yes
## # ... with 7,709 more rows
```

Note that dplyr auto “quotes” function inputs for you. However, select() has lots of convenience functions where you want to quote, e.g. starts_with(), ends_with(), or contains().

Subsetting dplyr-style II

Subsetting columns is just as easy using the filter() function:

```
filter(cps08, age==33)

## # A tibble: 720 x 5
##     ahe  year bachelor female   age
##   <dbl> <int> <chr>    <chr>  <int>
## 1  38.5  2008 yes      no        33
## 2  29.8  2008 yes      no        33
## # ... with 718 more rows
```

You can use all standard operators ==, >, >=, &, |, ! or functions is.na(), but also stuff like between(x, left, right).

Sorting dplyr-style

The `arrange()` function is used to reorder rows according to one or more variable(s):

```
arrange(cps08, year, bachelor, female, age)

## # A tibble: 7,711 x 5
##      ahe  year bachelor female   age
##    <dbl> <int> <chr>    <chr>  <int>
## 1 11.8   2008 no       no        25
## 2  8.65  2008 no       no        25
## # ... with 7,709 more rows
```

Use `desc(varname)` to sort a variable in descending order.

Renaming variables dplyr-style

Renaming variables is horribly complicated in base *R* but super simple using `rename()`:

```
rename(cps08, lohn=ahe, jahr=year, uni=bachelor,
       frau=female, alter=age)

## # A tibble: 7,711 x 5
##    lohn  jahr uni   frau  alter
##   <dbl> <int> <chr> <chr> <int>
## 1  38.5  2008 yes   no       33
## 2  12.5  2008 yes   no       31
## # ... with 7,709 more rows
```

You can also just use `newname = oldname` when calling `select()`.

Changing variables dplyr-style

Changing variables is annoying in base *R* but easy using `mutate()`:

```
mutate(cps08, ahe = ahe - mean(ahe, na.rm=TRUE) )

## # A tibble: 7,711 x 5
##      ahe  year bachelor female   age
##    <dbl> <int> <chr>    <chr>  <int>
## 1  19.5   2008 yes      no        33
## 2  -6.48  2008 yes      no        31
## # ... with 7,709 more rows
```

This returns the entire data, `transmute()` only the transformed variable(s). Useful functions are `log()`, `lead()` or `lag()`.

Grouping and summarizing dplyr-style

Getting grouped statistics is super cumbersome in base *R*. dplyr provides us with easy-to-use verbs:

```
cps08_grpd <- group_by(cps08, female, bachelor)
summarize(cps08_grpd, mean(ahe))

## # A tibble: 4 x 3
## # Groups:   female [?]
##   female bachelor `mean(ahe)`
##   <chr>  <chr>          <dbl>
## 1 no     no              16.6
## 2 no     yes             25.0
## 3 yes    no              13.2
## 4 yes    yes             20.9
```

summarize() and mutate() play well with other

... dplyr functions

- first(), last() and nth()
- n() and n_distinct()

... base *R* functions

- mean(), median(), min(), and max()
- sd(), var(), cor(), and cov()
- and many more.

You can also get summary stats for all columns using `summarise_all(cps08, funs(mean))`.

`mutate()` supports useful window functions, see `?mutate`.

The %>% pipe operator

Pipes are a powerful tool for clearly expressing a sequence of multiple operations. Let's group again:

```
cps08 %>% group_by(female, bachelor) %>%
  summarize(mean(ahe))

## # A tibble: 4 x 3
## # Groups:   female [?]
##   female bachelor `mean(ahe)`
##   <chr>  <chr>          <dbl>
## 1 no     no              16.6
## 2 no     yes             25.0
## 3 yes    no              13.2
## 4 yes    yes             20.9
```

New variables by groups

Using `group_by()` with `mutate()` also summarizes data and simultaneously creates new variables:

```
cps08 <- cps08 %>% group_by(female, bachelor) %>%
  mutate(ahe_group = mean(ahe)) %>% ungroup()
cps08

## # A tibble: 7,711 x 6
##      ahe  year bachelor female   age ahe_group
##    <dbl> <int> <chr>    <chr>  <int>     <dbl>
## 1  38.5  2008 yes      no          33      25.0
## 2  12.5  2008 yes      no          31      25.0
## # ... with 7,709 more rows
```

The index in `group_by()` is sticky until you call `ungroup()`.

When not to use the pipe

The pipe is a powerful tool, but it does not solve every problem!

Pipes are most useful for rewriting a fairly short linear sequence of operations.

Do not use when:

- Your pipes are longer than (say) ten steps. Create intermediate objects instead.
- You have multiple inputs or outputs. If there is no primary object, do not use pipes.
- You have some complex dependency structure. Pipes are fundamentally linear.

Other useful dplyr-verbs

Reshaping data:

- gather() columns into rows
- spread() rows into columns
- separate() one column into several
- unite() several columns into one

Combining data:

- bind_rows(y, z) append z to y as new rows
- bind_cols(y, z) append z to y as new columns
- left_join(a, b, by = "ID_1") matching rows from a to b
- right_join(a, b, by = "ID_1") matching rows from b to a
- inner_join(a, b, by = "ID_1") retain only rows in both sets
- full_join(a, b, by = "ID_1") retain all matching rows/ values

Saving and loading data in *R*'s native binary format

R has its own native binary format: RDS. You can store the entire work space (not recommended) or single objects (recommended).

Saving an object:

```r
# from tidyverse, or use base R saveRDS()
write_rds(cps08, path = "./data/cps08.rds")
```

Loading an object:

```r
# from tidyverse, or use base R readRDS()
cps08 <- read_rds(path = "./data/cps08.rds")
```

Use RDS if you will continue using the file in R and do not want to lose auxiliary information about the data (e.g. `group_by` keys).

Using data in other binary formats

haven is part of the tidyverse and reads/writes Stata, SPSS or SAS files.

```
require(haven)
path <- system.file("examples", "iris.dta",
                    package = "haven")
my_tbl <- read_dta(path)
```

readxl is also part of the tidyverse and reads/writes Excel files

```
require(readxl)
xlsx_example <- readxl_example("datasets.xlsx")
my_tbl <- read_excel(xlsx_example, sheet = "mtcars")
```

Simple features and vector data

What are simple features?

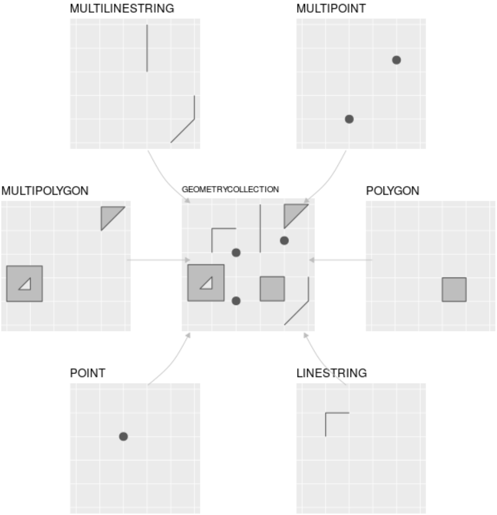
Simple features are an ISO standard (ISO 19125-1:2004) that describes how the spatial geometry of real world objects can be represented by computers.

The standard is implemented in PostGIS or ESRI ArcGIS and forms the vector data basis for libraries such as GDAL. A subset of simple features make up GeoJSON.

The old `sp`-package did not implement this standard and was horribly slow. The `sf` package solves this, is super fast, and plays well with the `tidyverse`.

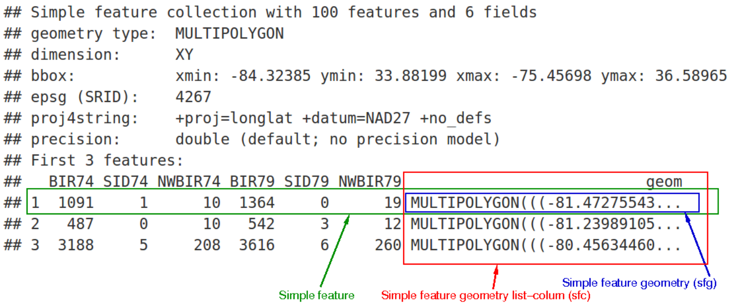
Simple features exist in their basic form (points, lines and polygons) and as simple feature collections. Collections are tibbles or data frames with an added geometry column.

Basic simple features



Vector data with simple features

Although geometries are native R objects, they are printed as well-known text (WKT), we see



```
## Simple feature collection with 100 features and 6 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
## epsg (SRID):    4267
## proj4string:    +proj=longlat +datum=NAD27 +no_defs
## precision:      double (default; no precision model)
## First 3 features:
##   BIR74 SID74 NWBIR74 BIR79 SID79 NWBIR79                           geom
## 1  1091     1      10  1364     0      19 MULTIPOLYGON(((-81.47275543...
## 2   487     0      10   542     3      12 MULTIPOLYGON(((-81.23989105...
## 3  3188     5     208  3616     6     260 MULTIPOLYGON(((-80.45634460...
```

Simple feature
Simple feature geometry list–colum (sfc)
Simple feature geometry (sfg)

- in green a simple feature: a data.frame or tibble row
- in blue a single simple feature geometry (class sfg)
- in red a simple feature list-column (class sfc)

Installing and loading the sf-package

```
install.packages(sf)
```

Put this in the preamble of each script:

```
require(sf)

## Loading required package: sf

## Linking to GEOS 3.5.0, GDAL 2.2.2, PROJ 4.8.0
```

sf automatically installs and links to the Geometry Engine Open Source (GEOS), Geospatial Data Abstraction Library (GDAL), and proj.4 projection libraries.

Loading vector data

`st_read()` and `st_write()` can import and export most common vector data types: ESRI shapefile, ESRI geodatabase geoJSON, KMZ/KML, and many more, see `st_drivers()`.

Let's start a new script and set the working directory

```r
rm(list=ls()) # clear space
require(tidyverse); require(sf) # get pkgs
setwd("~/UHannover/000000_Glad_Geospatial/")
```

and then load a shapefile of Africa (get `africa.zip` from my website and put it in `./data/`)

```r
africa_sf <- st_read("./data/africa_scale.shp")
```

Learning about this data I

```r
class(africa_sf) # What is it?

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

```r
st_crs(africa_sf) # What's the CRS?

## Coordinate Reference System:
##   EPSG: 4326
##   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
```

```r
st_bbox(africa_sf) # And the extent/bounding box?

##      xmin      ymin      xmax      ymax
## -17.62504 -34.81917  51.13387  37.34999
```

Learning about this data II

```r
# Get dimensions
dim(africa_sf)

## [1] 51 64
```

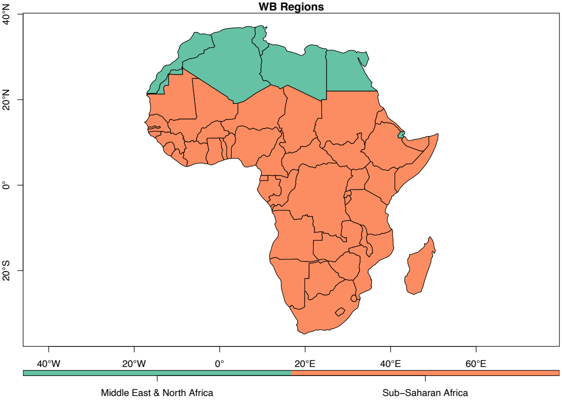
```r
# Subset, way too many variables
africa_sf <- africa_sf %>%
  select(admin, type, iso_a3, region_wb, pop_est)

# A magical sixth column?
names(africa_sf)[6]

## [1] "geometry"
```

Plotting African countries

```r
plot(africa_sf["region_wb"], axes = T,
     main = "WB Regions", key.pos = 1,
     key.width = lcm(1.3), key.length = 1.0)
```



Reading points from CSV files

Point data often come in CSV files with coordinates and need to be made spatially aware first:

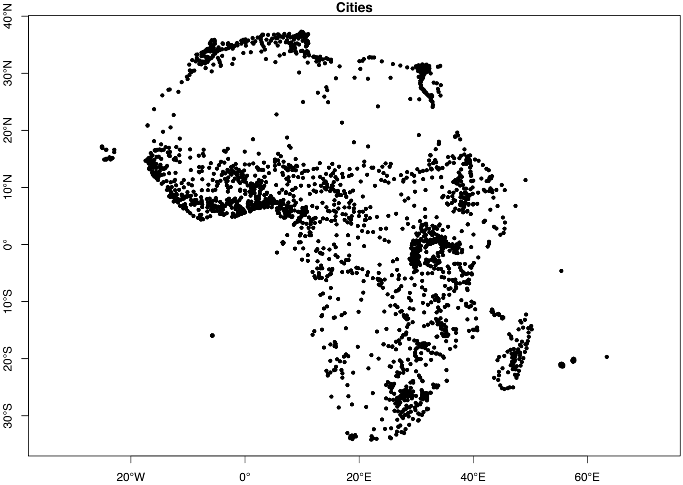
```r
# read the csv of african cities
cities_csv <- read_csv("./data/africa_cities.csv")

# filter out missing coords
cities_csv <- cities_csv %>%
  filter( !( is.na(lon) & is.na(lat) ) )

# turn into sf points, specify coords and crs
cities_sf <- st_as_sf(cities_csv,
                      coords = c("lon", "lat"),
                      crs = 4326)
```

Plotting African cities

```
plot(cities_sf[1], axes = T, pch = 20,
     col = "black",  main = "Cities")
```



Sources for vector data

R packages:

- `rnaturalearth` provides you with direct access to its database in `sf` format (countries, rivers, roads, etc.)
- `tidycensus` and `tigris` give you full access to the US census API and delivers the data in `sf` format
- `osmdata` puts OpenStreetMaps at your fingertips in `sf` format
- `eurostat` gives you Eurostat data and maps in `sf` format

A few links:

- A huge link list: `freegisdata.rtwilson.com`
- Humanitarian data exchange: `data.humdata.org`
- Google data search: `toolbox.google.com/datasetsearch`

Projections redux

What if you get data without a CRS an need to assign one?

```
# delete the existing proj.4 and epsg code
africa_sf <- africa_sf %>% st_set_crs(NA)
st_crs(africa_sf)

## Coordinate Reference System: NA
```

```
# assign a CRS
africa_sf <- africa_sf %>% st_set_crs("+proj=longlat")
st_crs(africa_sf)

## Coordinate Reference System:
##   EPSG: 4326
##   proj4string: "+proj=longlat +ellps=WGS84 +no_defs"
```

A tale of two cities

```
# Nairobi in longlat
cty1 <- tibble(name= "Nairobi",
               lon = 36.82241, lat = -1.287822) %>%
  st_as_sf(coords = c("lon", "lat"), crs = 4326)
st_is_longlat(cty1)
st_coordinates(cty1)

# Kinshasa in mollweide
cty2 <- tibble(name = "Kinshasa",
               x = 1531775, y = -531896.9) %>%
  st_as_sf(coords = c("x", "y"), crs = "+proj=moll")
st_is_longlat(cty2)
st_coordinates(cty2)
```

```
# try to row bind both together
ctys <-  rbind(cty1, cty2)
```

Always transform data in different CRSs

Always make sure your data are in the same CRS. `st_transform()` interfaces with GDAL for all geographic transformations. You can use EPSG codes or specify a (partial) Proj.4 string.

```r
# transform to longlat
cty2 <- cty2 %>% st_transform(4326)
st_is_longlat(cty2)
st_coordinates(cty2)

# row bind and take a look
ctys <- rbind(cty1, cty2) %>% glimpse()
```

Want to know all EPSG codes in GDAL? Install `rgdal` and then run `View(rgdal::make_EPSG())`.

Distances redux

Distances between points are easily calculated using `st_distance()` but read the help file carefully:

- Geographic coordinates: the function uses Vincenty distances if proj.4 $<$ v4.8.0 and great-circle distances otherwise.
- Projected coordinates: the function always uses Euclidean distances in *native* map units.

We can calculate pairwise distance or a distance matrix:

```
st_distance(cty1,cty2) # st_distance(ctys) for matrix

## Units: [m]
##         [,1]
## [1,] 2414704
```

Areas and length

Calculating areas of irregular polygons is mathematically difficult (you cycle through the vertices or think hard about geodesics).

Don't worry about it, use `st_area()`:

- Geographic coordinates: the function uses geodetic areas.
- Projected coordinates: the function always computes areas in *native* map units.

```
st_area(ctys) # points have no area

## Units: [m^2]
## [1] 0 0
```

Use `st_length()` for line feature, works like distances.

An aside on units

Simple features always report distances, lengths, and areas with their actual units (meters, miles, etc). This is a revolution.

You can convert units without math, even use math on units, and units will *fail* gracefully if you ask for something stupid:

```
require(units)
a_to_b <- st_distance(cty1,cty2)
set_units(a_to_b, km)

## Units: [km]
##           [,1]
## [1,] 2414.704
```

```
# try: set_units(a_to_b, km^2)
# strip it of its units: as.numeric(a_to_b)
```