Alexander Abuabara
DAEN 489 - Spring 2026
College Station 22-Jan-2026

Week 2 💬

Contents

1 Vector Data / sf (simple features)

- R package **sf** https://cran.r-project.org/web/packages/sf/index.html provides a table format for simple features, where feature geometries are stored in a list-column.
- R package **stars** https://cran.r-project.org/web/packages/stars/index.html was written to support raster and vector data cubes (we will see that later), supporting raster layers, raster stacks and feature time series as special cases.
- **sf** first appeared on CRAN https://cran.rstudio.com in 2016, **stars** in 2018. Development of both packages received support from the R Consortium as well as strong community engagement.
- The packages were designed to work together. Functions or methods operating on **sf** or **stars** objects start with **st_**, making it easy to recognize them or to search for them when using command line completion.

```
1 library(sf)
2
3 methods(class = "sf")
```

Creation

An sf object can be created from scratch by

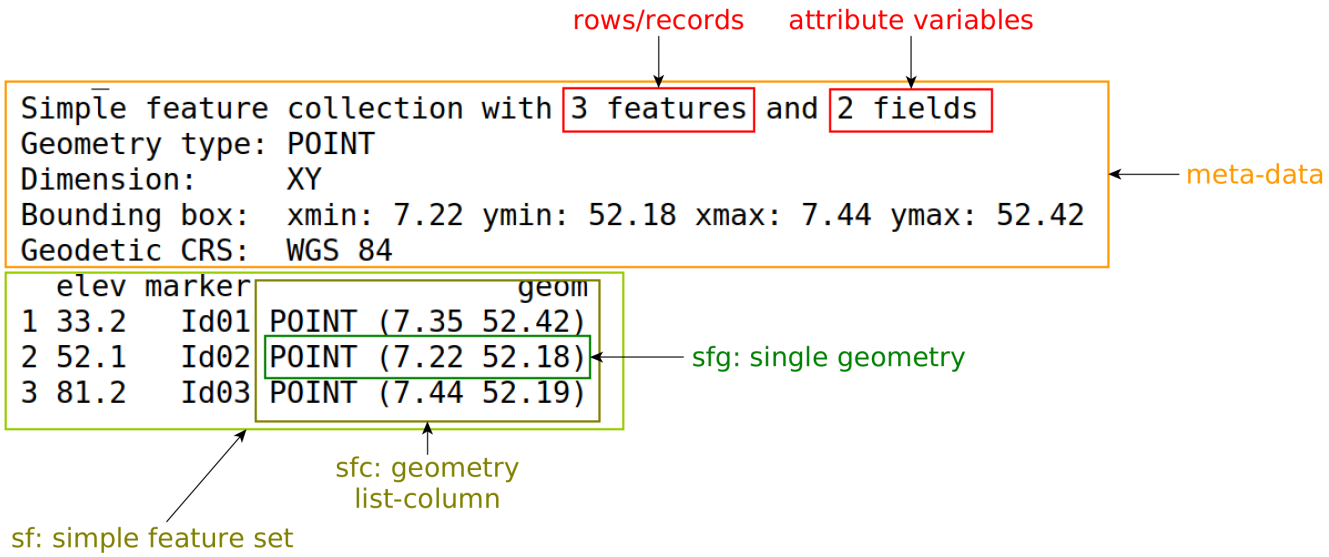
```r
library(sf)

p1 <- st_point(c(7.35, 52.42))
p2 <- st_point(c(7.22, 52.18))
p3 <- st_point(c(7.44, 52.19))

sfc <- st_sfc(list(p1, p2, p3), crs = 'OGC:CRS84')

st_sf(elev = c(33.2, 52.1, 81.2),
      marker = c("Id01", "Id02", "Id03"), geom = sfc)

plot(sfc)
```



Reading and writing

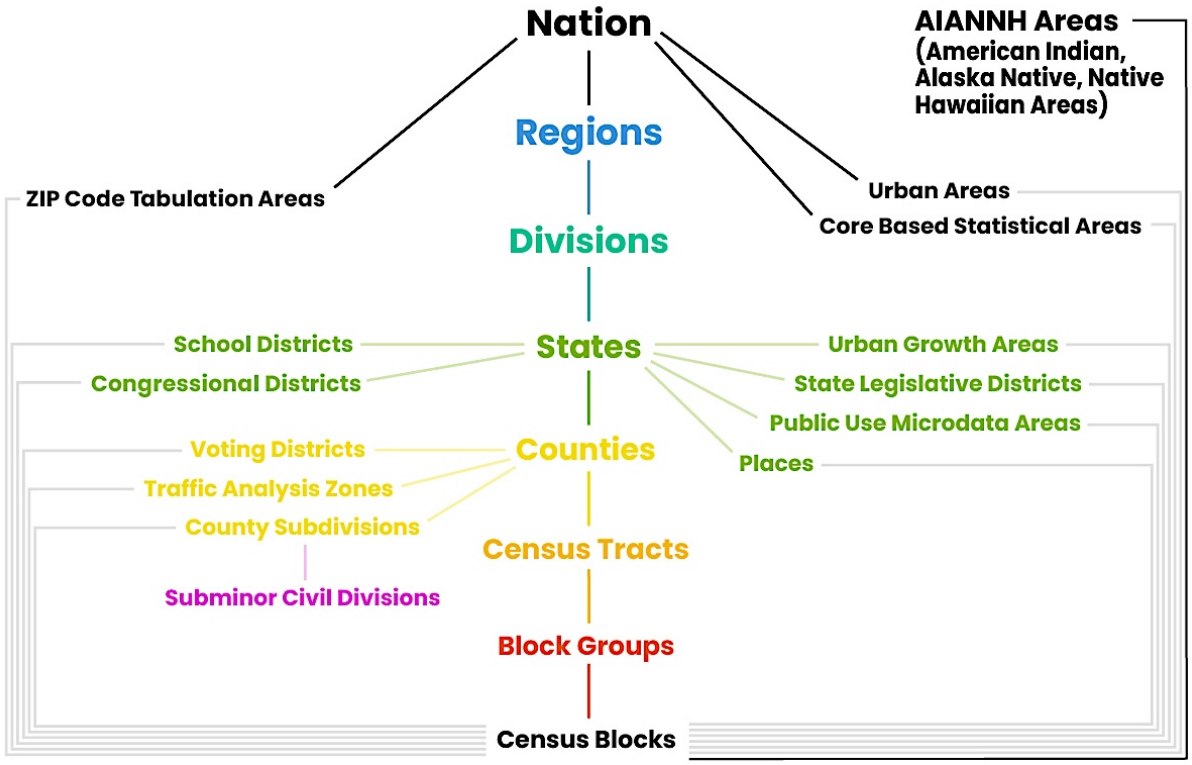
```r
library(sf)

(file <- system.file("gpkg/nc.gpkg", package = "sf"))

nc <- st_read(file)

st_layers(file)
```

US Census TIGER/Line Shapefiles



TIGER=Topologically Integrated Geographic Encoding & Referencing

- Nationwide spatial database maintained by the U.S. Census Bureau
- Administrative boundaries: states, counties, tracts, block groups, blocks
- Legal and statistical areas: places, ZCTAs
- Features: roads, rail, hydrography, landmarks
  - Official geometry for Decennial Census and ACS products
  - Ensures consistent geographies across datasets and time
  - Free, open, and nationally comprehensive

Typical uses

- Joining tabular census data to spatial boundaries
- Choropleth mapping and spatial aggregation
- Defining analysis units for demographic and policy studies

Key characteristics

- Updated annually, standardized schema, nationwide coverage
- Designed for analysis, not high-precision surveying

Common Pitfalls When Using TIGER/Line Data

- **Geographic units are not stable over time**
  - Tracts, block groups, and boundaries change between censuses
  - Always align year of geometry with year of attributes
- **ZCTAs are not ZIP codes**
  - ZCTAs approximate USPS ZIPs, but are not identical
  - Not appropriate for all mailing or service-area analyses
- **Projection and coordinate system issues**
  - Shapefiles are typically in geographic coordinates (NAD83)
  - Reproject before distance, area, or buffering operations
- **Geometry quality limitations**
  - Boundaries are generalized for national coverage
  - Not suitable for parcel-level or engineering-grade analysis
- **Joining errors are common**
  - Mismatched GEOIDs, leading zeros, and type coercion
  - Always validate join completeness and key formats
- **Large file sizes and performance**
  - National layers can be very large
  - Subset by state or county for efficient workflows

Hands-on

- Download: **Counties (and equivalent)**
  https://www.census.gov/geographies/mapping-files/2025/geo/tiger-line-file.html
- Unzip and copy path to "*tl_2025_us_county.shp*"

```
us_counties <-
 ↪  st_read("/Users/abuabara/.../tl_2025_us_county/tl_2025_us_county.shp")

class(us_counties)

dim(us_counties)

glimpse(us_counties)

library(dplyr) # tidyverse
```

```r

glimpse(us_counties)

filter(us_counties, STATEFP == 48)

tx_counties <- filter(us_counties, STATEFP == 48)

# plot(tx_counties)
# plot(st_geometry(tx_counties))
```

Very Basic Plotting

Ref.:https://r-spatial.github.io/sf/articles/sf5.html

More details about the map's composition will be explored later (Week 9).

ggplot2

- `ggplot2` is a powerful R package for creating **statistical graphics** using a consistent, structured grammar.
- It is based on the **Grammar of Graphics**, which decomposes every plot into a small set of independent components.
- Instead of choosing a plot type first, you:
  - Specify the data,
  - Map variables to visual features, and
  - Add layers that build the final figure.
- The basic template is:

```
ggplot(data = ..., mapping = aes(...)) + geom_...
```

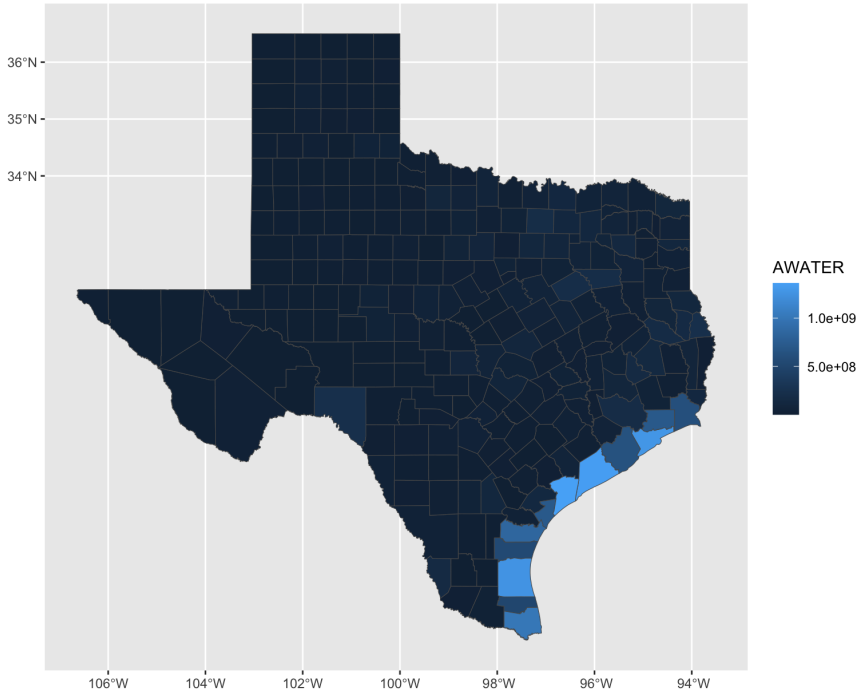
- Every ggplot has three core parts:
  - **Data**: the data frame being plotted,
  - **Aesthetics** (`aes`): how variables map to x, y, color, size, etc.,
  - **Geoms**: the geometric objects that draw the plot (points, lines, …).
- Plots are built **incrementally** using the `+` operator:
  - Start with a base plot,
  - Then add layers, scales, labels, and themes.

- The same data can produce many different plots by changing only the **geom**.
- `ggplot2` encourages:
  - Reproducibility,
  - Clear separation between data and presentation, and
  - Code that reads like a description of the graphic.
- Common first geoms:
  - `geom_point()` for scatterplots,
  - `geom_line()` for time series,
  - `geom_bar()` for counts and categories.
- By default, `ggplot2` produces **publication-quality** figures with sensible styling.

```
install.packages("ggplot2")
```

```
library(ggplot2)

ggplot() +
  geom_sf(data = tx_counties, aes(fill = AWATER)) +
  scale_y_continuous(breaks = 34:36)
```



tmap

tmap is powerful for creating thematic maps with "elegant", flexible syntax.

- Layer-based approach similar to ggplot2
- Supports both static and interactive maps
- Works seamlessly with sf (simple features) objects
- Publication-ready cartographic output
- Easy to customize colors, legends, and layouts

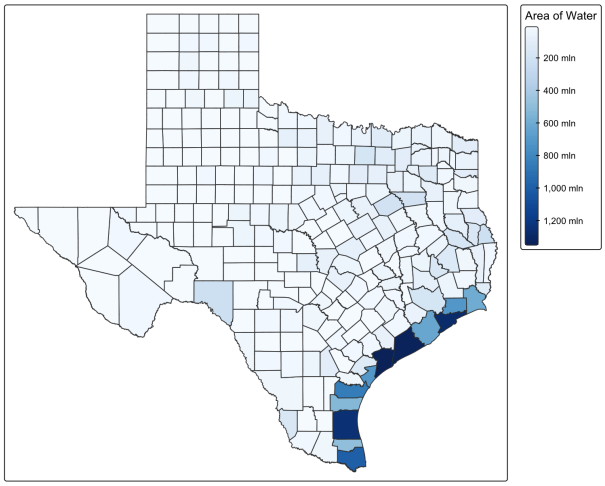
```
install.packages("tmap")
```

The tmap syntax follows a layered approach:

```
library(tmap)

# version 3
tm_shape(tx_counties) +
  tm_polygons("AWATER",
              palette = "Blues",
              title = "Area of Water")

# version 4
tm_shape(tx_counties) +
  tm_polygons(fill = "AWATER",
              fill.scale = tm_scale_continuous(values = "brewer.blues"),
              fill.legend = tm_legend(title = "Area of Water"))
```



Common functions:

- `tm_shape()`: Define the spatial data
- `tm_polygons()`: Draw filled polygons
- `tm_borders()`: Draw polygon boundaries
- `tm_dots()`: Add point symbols
- `tm_text()`: Add labels

Some interesting options to explore:

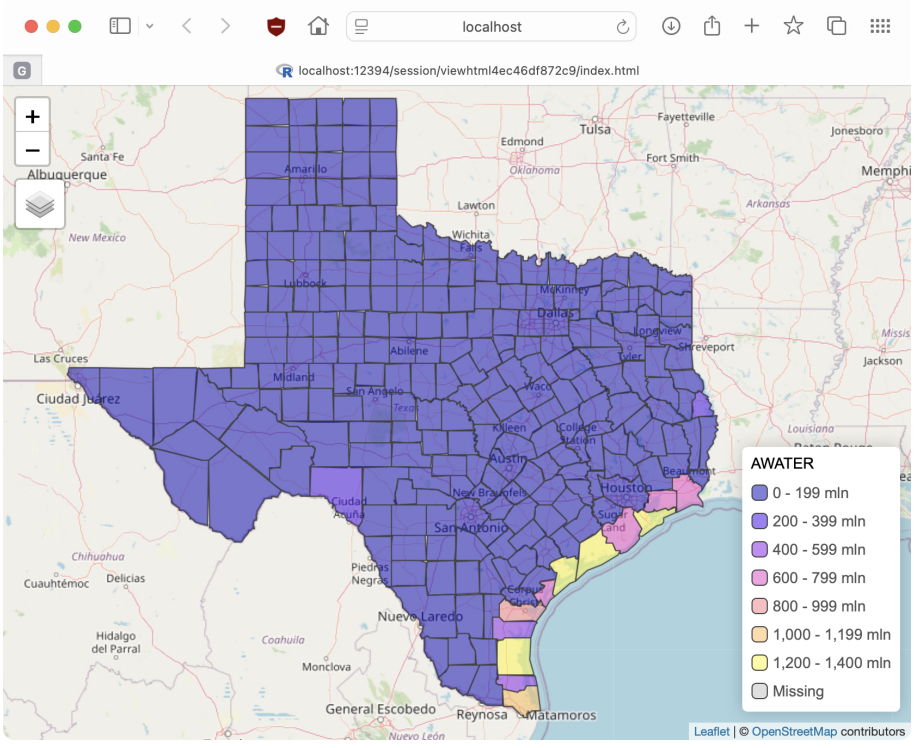
- Draw a thematic map quickly (quick thematic map plot)

```
qtm(tx_counties)
```

- This function is a convenient wrapper of the main plotting method of stacking tmap-elements.
- Without arguments or with a search term, this functions draws an interactive map.
- Set tmap mode to static plotting or interactive viewing:

```
tmap_mode("view")

tm_shape(tx_counties) + tm_polygons("AWATER", palette = sf.colors(5),
↪  alpha = .5)
```



2 Attribute Data Operations

- Attribute data operations focus on **selecting, creating, transforming, and summarizing columns** in a data frame.
- In R, attributes (columns) are commonly accessed using:
  - Column names: `df$variable`
  - Bracket indexing: `df[, "variable"]` or `df[, j]`
- Common goals of attribute operations:
  - Select a subset of variables,
  - Create new variables from existing ones,
  - Modify or recode variables,
  - Compute summaries by column.
- **Selecting and Subsetting Attributes**
  - Extract a single column: `df$age`, `df[, "age"]`
  - Extract multiple columns: `df[, c("age", "income", "gender")]`
  - Remove columns by index or name: `df[, -3]`, `df[, !names(df) %in% c("id", "timestamp")]`
- **Creating and Modifying Attributes**
  - Create a new attribute: `df$income_thousands <- df$income / 1000`
  - Modify an existing attribute: `df$age <- df$age + 1`
  - Vectorized operations apply to the entire column at once.
- **Recoding and Transforming Attributes**
  - Apply transformations: `log(df$income)`, `scale(df$score)`
  - Recode categorical variables: `df$gender <- factor(df$gender, levels = c("M", "F"))`
  - Create indicator variables: `df$is_adult <- df$age >= 18`
- **Summarizing Attributes**
  - Column-wise summaries: `mean(df$age, na.rm = TRUE)`, `summary(df)`
  - Apply a function to columns: `sapply(df, mean, na.rm = TRUE)`
  - Grouped summaries with `aggregate()` or `dplyr::summarise()`
- **Good Practices**: Inspect structure and names: `names(df)`, `str(df)`, `glimpse(df)`. Be mindful of missing values, data types, and recycling rules. Prefer readable code and avoid hard-coded column indices.

```r
us_counties_df <- st_drop_geometry(us_counties)

class(us_counties_df)

glimpse(us_counties_df)

dim(us_counties_df)
ncol(us_counties_df)
nrow(us_counties_df)
```

Vector Attribute Manipulation

```r
tx_counties$AREA = tx_counties$ALAND + tx_counties$AWATER

st_crs(tx_counties)

tx_counties$AREA_km2 = tx_counties$AREA / 1e+6

tx_counties = mutate(tx_counties, Area2 = (ALAND + AWATER) / 1e+6)
```

Vector Attribute Subsetting

A demonstration of the utility of using logical vectors for subsetting is shown in the code chunk below.

```r
tx_counties[3:4, ]        # subset rows by position
tx_counties[, 3:4]        # subset columns by position
tx_counties[, c(1,4,7)]   # subset columns by position
tx_counties[4:6, 3:6]     # subset rows and columns by position
tx_counties[, c("GEOID", "NAME")] # columns by name
tx_counties[, c(F, T)]    # by logical indices
tx_counties[, 888]        # an index representing a non-existent column
```

This creates a new object, small_counties, containing counties whose surface area is smaller than 1,000 km2.

```
i_small = tx_counties$AREA_km2 < 1000
summary(i_small) # a logical vector
small_tx_counties = tx_counties[i_small, ]
```

The intermediary i_small (short for index representing small counties) is a logical vector that can be used to subset the nine smallest counties in Texas by surface area. A more concise command, which omits the intermediary object, generates the same result:

```
small_tx_counties = tx_counties[tx_counties$AREA_km2 < 1000, ]
```

The base R function `subset()` provides another way to achieve the same result:

```
small_tx_counties = subset(tx_counties, AREA_km2 < 1000, )
```

Base R functions are mature, stable and widely used, making them a rock solid choice, especially in contexts where reproducibility and reliability are key.

`dplyr` functions enable 'tidy' workflows which a lot of people find intuitive and productive for interactive data analysis, especially when combined with code editors such as RStudio that enable auto-completion of column names.

Key functions for subsetting data frames (including `sf` data frames) with `dplyr` functions are demonstrated below.

```
small_tx_counties = filter(tx_counties, AREA_km2 < 1000)
```

Chaining Commands with Pipes

Pipes let you write a sequence of operations in a clear, linear fashion.

In `R`, the pipe operator:

- is written as `%>%` (magrittr / tidyverse)
  or `|>` (native R pipe since R 4.1.0)
- takes the output of the previous expression
  and feeds it into the next function as the first argument. :contentReference[oaicite:1]index=1

This enables expressive and readable workflows, especially when combined with `dplyr` verbs such as `filter()`, `mutate()`, `select()`, and `slice()`.

Example (using native pipe):

```
small_tx_counties = us_counties |>
  filter(STATEFP == 48) |>
  mutate(AreaKm2 = (ALAND + AWATER) / 1e+6) |>
  select(GEOID, Name = NAME, AreaKm2) |>
  # slice(1:5)
  filter(AreaKm2 < 1000)
```

Alternative 1: Nested calls (harder to read)

```
small_tx_counties <-
  slice(
  select(
    filter(us_counties, STATEFP == 48),
    Name = NAME, GEOID),
  1:5)
```

NOTE:

In R, both `<-` and `=` can be used for assignment, but they are **not fully interchangeable** and have different roles and precedence.

The traditional operator: `<-`

- Canonical assignment operator in R.
- Works in all contexts: scripts, functions, loops, and interactive use.

- Has lower precedence, reducing ambiguity in expressions.
- Recommended by most style guides and in package development.

The dual-purpose operator: `=`

- Used for assignment in some contexts.
- Used to name function arguments.
- Has higher precedence, which can lead to subtle parsing differences.

Example

```
x <- 10
y = 10
mean(x = c(1, 2, 3))   % argument naming, not assignment
```

Precedence matters

```
x <- y = 5   % valid: assigns 5 to y, then to x
x = y <- 5   % error or unexpected behavior
```

Because `<-` binds more loosely, it is safer in chained or nested expressions.

Recommended style

- Use `<-` for all assignments.
- Use `=` only for naming function arguments.

Feature	`<-`	`=`
Primary role	Assignment	Assignment & argument naming
Works in all contexts	Yes	Limited / ambiguous
Operator precedence	Lower (safer)	Higher
Recommended for assignment	Yes	No
Common in function calls	No	Yes (for arguments)

Alternative 2: Intermediate variables

- An intermediate variable is a temporary object created to store the result of a step in a multi-stage computation.
- It contrasts with a piped workflow.

```
tx_counties_filtered <- filter(us_counties, STATEFP == 48)

tx_counties_areaed <- mutate(tx_counties_filtered, AreaKm2 = (ALAND + AWATER) /
↪  1e+6)

tx_counties_selected <- select(tx_counties_areaed, Name = NAME, GEOID, AreaKm2)

small_tx_counties <- filter(tx_counties_selected, AreaKm2 < 1000)
```

Piped operations streamline code and reduce the need for temporary names when interactively exploring and cleaning data.

An **intermediate variable** stores the result of a step in a multi-stage computation.

Advantages

- **Improved "debugability"**: Each step can be inspected independently using `summary()`, `head()`, or `str()`.
- **Reusability of partial results**: Intermediate objects can be reused in multiple downstream computations, avoiding "recomputation" of expensive steps.
- **Better integration with control flow**: Works naturally with `if`/`else`, loops, and conditional branching.
- **Pedagogical clarity**: Makes step-by-step transformations explicit, which is helpful for teaching and for understanding complex workflows.

Disadvantages

- **Verbosity and workspace clutter**: Many temporary objects increase the number of symbols in the environment and make scripts longer.
- **Reduced high-level readability**: The logical data flow is fragmented across multiple assignments, making the overall pipeline harder to see at a glance.
- **Risk of inconsistency bugs**: Intermediate objects can be accidentally reused, overwritten, or left stale when code is modified.

- **Higher maintenance burden**: More variable names must be tracked and kept consistent as code evolves.

Aspect	Advantage	Disadvantage
Debugging	Easy inspection of each step	More objects to manage
Reusability	Partial results can be reused	Risk of stale values
Readability	Clear step-by-step logic	Breaks high-level flow
Code length	Explicit and modular	More verbose
Workflow style	Works with control flow	Less expressive than pipelines

Vector Attribute Aggregation

Aggregation summarizes attribute data by one or more grouping variables, reducing a dataset to a smaller set of summary values. For example, summing country populations by continent produces one population total per continent rather than one per country.

Base R:

`aggregate()`

```
library(sf)
library(spData)
library(dplyr)

glimpse(world)

world_agg1 = aggregate(pop ~ continent,
                       FUN = sum,
                       data = world,
                       na.rm = TRUE)

world_agg1
```

This computes the sum of `pop` for each `continent`. The result is a (non-spatial) `data.frame`.

sf method

`aggregate()` behaves as a generic. When used on an `sf` object and a `by` argument is provided, the result retains spatial features:

```
world_agg2 = aggregate(world["pop"],
                       by = list(world$continent),
                       FUN = sum,
                       na.rm = TRUE)

world_agg2
```

This returns an `sf` object with aggregated geometries.

The tidyverse approach uses `group_by()` and `summarize()`.

Simple aggregation

```
world_agg3 = world |>
  group_by(continent) |>
  summarize(pop = sum(pop, na.rm = TRUE))
```

This produces population totals per continent.

Multiple summaries

You can compute multiple summary statistics in one pipeline:

```
world_agg4 = world |>
  group_by(continent) |>
  summarize(
    Pop  = sum(pop, na.rm = TRUE),
    Area = sum(area_km2),
    N    = n()
  )
```

Here, `Pop` and `Area` are sums; `N` gives the count of features per group.

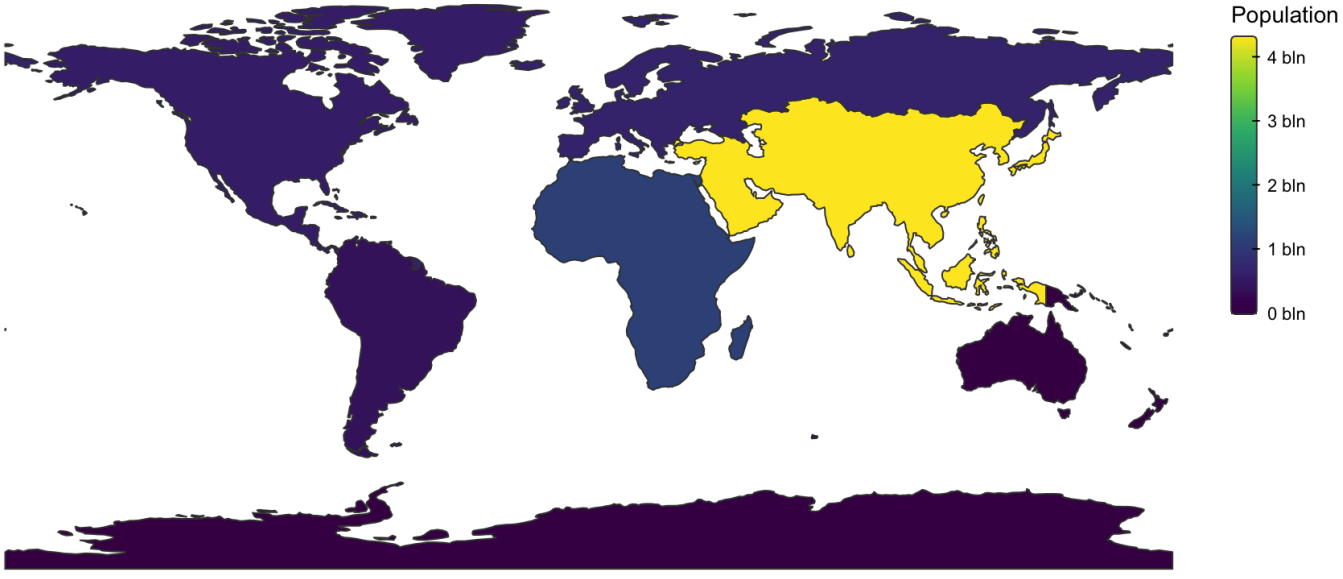
As a simple visualization check, we can map the aggregated populations:

```
library(tmap)
qtm(world_agg4)

tm_shape(world_agg4) +
  tm_polygons("Pop", style = "cont")
```

or

```
tm_shape(world_agg4) +
  tm_polygons("Pop",
              style   = "cont",
              palette = "viridis",
              title   = "Population",
              textNA  = "",
              legend.show = TRUE,
              legend.reverse = TRUE) +
  tm_layout(
    frame = FALSE,
    legend.outside = TRUE,
    legend.outside.position = "bottom",
    legend.outside.size = 0.15,
    legend.frame.lwd = 0
  )
```



Extended aggregation with derived variables

By combining with other verbs, e.g., `mutate()`, `arrange()`, and `slice_max()`, you can compute densities, rank results, and subset summaries:

```
world_agg5 = world |>
  st_drop_geometry() |>
  select(pop, continent, area_km2) |>
  group_by(Continent = continent) |>
  summarize(
    Pop  = sum(pop, na.rm = TRUE),
    Area = sum(area_km2),
    N    = n()
  ) |>
  mutate(Density = round(Pop / Area)) |>
  slice_max(Pop, n = 3) |>
  arrange(desc(N))
```

This chain calculates population density, selects the top 3 groups by population, and orders by number of features.

NEXT WEEK:

Vector Attribute Joining

Creating Attributes and Removing Spatial Information

Raster Subsetting

Summarizing Raster Objects

Pre-processing and Workflows