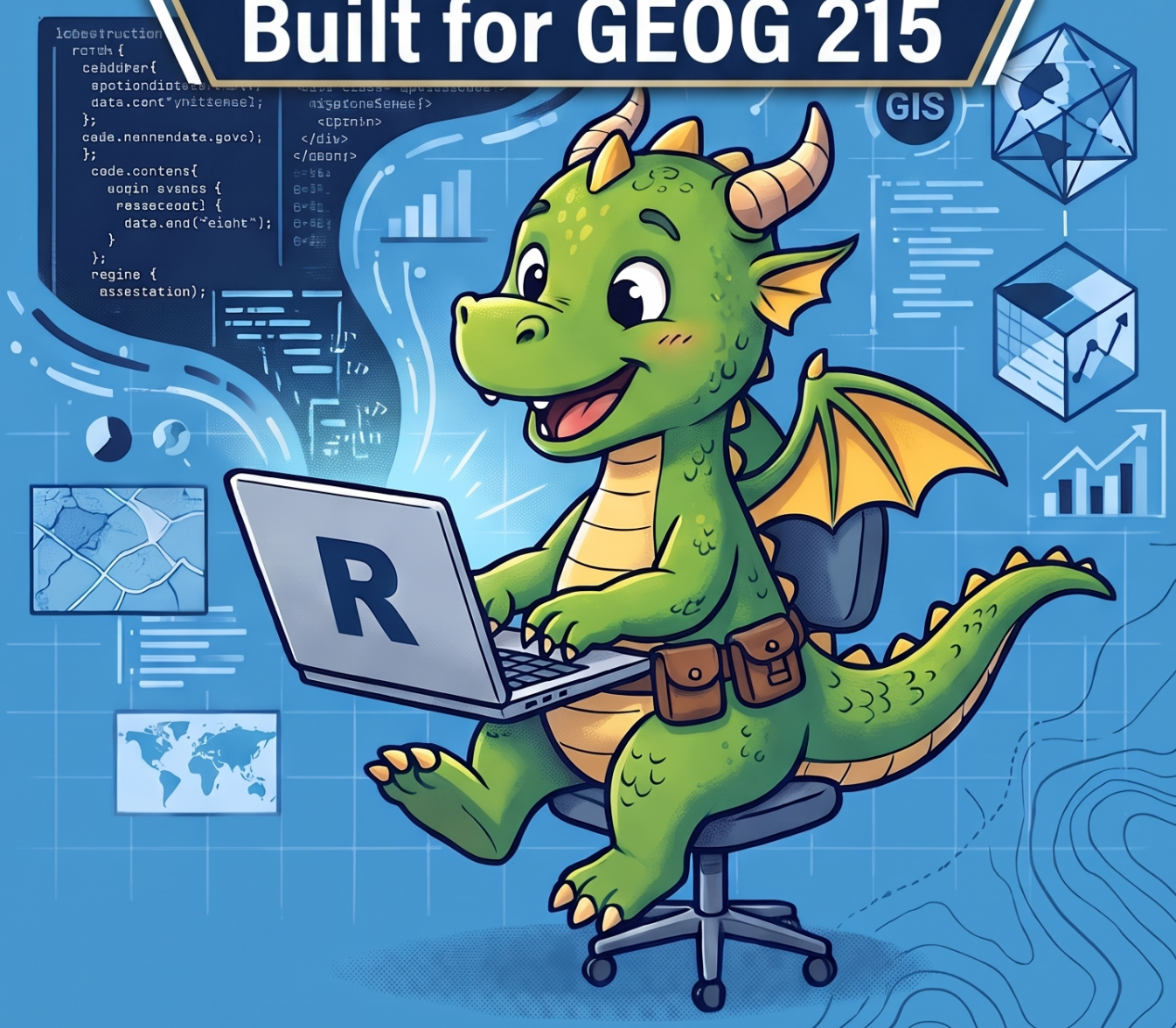
Introduction to SPATIAL DATA SCIENCE

Built for GEOG 215



David Parr

INTRODUCTION TO SPATIAL DATA SCIENCE

The textbook for GEOG 215

David Parr



Copyright © 2026 David Parr
All rights reserved.

First Edition



For Abigail, who is my best companion.

Table of contents

Preface 1
- About this book . . . 1
- How to use this book . . . 1
  - What's in each chapter . . . 1
  - Running the code . . . 1
  - Key terms and the glossary . . . 2
- How this book is organized . . . 2
- Fonts and reading settings . . . 3
- Acknowledgments . . . 3

I Unit 1: R and Spatial Data Science 4
- Chapter 1: What is Spatial Data Science? . . . 5
- Chapter 2: R, RStudio, and R Markdown . . . 5
- Chapter 3: R Basics . . . 5
- Chapter 4: Indexing Vectors and Data Frames . . . 5
- Chapter 5: Blocks, Loops, and Functions . . . 5

1 What is Spatial Data Science? 7
- 1.1 What this book will teach you . . . 7
- 1.2 What is spatial data science? . . . 8
  - 1.2.1 From data to wisdom . . . 8
  - 1.2.2 Spatial, data, science . . . 8
- 1.3 Why does location matter? . . . 9
- 1.4 The spatial data science workflow . . . 9
- 1.5 Spatial thinking . . . 10
  - 1.5.1 Spatial autocorrelation . . . 10
  - 1.5.2 Scale . . . 10
  - 1.5.3 Spatial heterogeneity . . . 11
- 1.6 Our toolbox . . . 11
- 1.7 The road ahead . . . 11
- 1.8 What makes a spatial data science question? . . . 12
- 1.9 Check your understanding . . . 13
- 1.10 Summary . . . 14
- 1.11 Exercises . . . 14
- 1.12 Further reading . . . 14

2 R, RStudio, and R Markdown 15
- 2.1 What is R? . . . 15
  - 2.1.1 Why use R? . . . 15
- 2.2 What is RStudio? . . . 16
  - 2.2.1 The four panes . . . 16
  - 2.2.2 RStudio during a working session . . . 17
- 2.3 R scripts and R Markdown . . . 17
  - 2.3.1 What is R Markdown? . . . 18
  - 2.3.2 Anatomy of an R Markdown file . . . 18
  - 2.3.3 The R Markdown workflow . . . 20

2.3.4 Keep all course files in one folder . . . 21
2.4 Code basics . . . 21
2.4.1 A line of code . . . 22
2.4.2 A block of code . . . 22
2.4.3 Variables . . . 22
2.4.4 Comments . . . 23
2.4.5 Functions . . . 23
2.4.6 Chaining functions with the pipe . . . 24
2.4.7 A program . . . 24
2.5 Some hints for coding . . . 24
2.6 Setting up R and RStudio . . . 25
2.7 Check your understanding . . . 25
3.1 Summary . . . 27
3.2 Exercises . . . 28
3.3 Further reading . . . 28

4 R Basics **29**
4.1 How R runs your code . . . 29
4.1.1 R is an interpreted language . . . 29
4.1.2 Order matters . . . 30
4.1.3 Knitting runs everything again . . . 30
4.1.4 Rule one: think sequentially . . . 30
4.2 Packages and libraries . . . 30
4.2.1 What is a package? . . . 30
4.2.2 Packages versus libraries . . . 31
4.2.3 CRAN . . . 32
4.2.4 Install once, load every session . . . 32
4.3 Data types . . . 33
4.3.1 Variables, again . . . 33
4.3.2 Vectors . . . 33
4.3.3 The four atomic data types . . . 33
4.3.4 One type per vector . . . 34
4.3.5 Naming variables . . . 34
4.4 Debugging and getting help . . . 35
4.4.1 Mistakes are a fact of life . . . 35
4.4.2 The most common mistakes . . . 35
4.4.3 Getting help with a function . . . 36
4.4.4 Getting help with an error . . . 36
4.4.5 Using AI in this course . . . 36
4.4.6 Getting help from people . . . 37
4.5 Check your understanding . . . 37
4.6 Summary . . . 38
4.7 Exercises . . . 39
4.8 Further reading . . . 39

5 Indexing Vectors and Data Frames **40**
5.1 A vector to work with . . . 40
5.2 Indexing by position . . . 41
5.2.1 One element . . . 41
5.2.2 Leaving an element out . . . 41
5.2.3 Several elements . . . 41
5.3 Logical operations . . . 42
5.3.1 Comparisons return TRUE or FALSE . . . 42
5.3.2 A comparison checks every element . . . 42

5.3.3 Indexing with a condition . . . 43
5.3.4 Logical vectors are recycled . . . 43
5.3.5 Combining conditions: AND, OR, and NOT . . . 43
5.4 Indexing by name . . . 44
5.4.1 Naming the elements . . . 44
5.4.2 Selecting by name . . . 45
5.4.3 How long is a vector? . . . 45
5.5 Data frames . . . 45
5.5.1 From one dimension to two . . . 45
5.5.2 Creating a data frame . . . 46
5.5.3 Looking at a data frame . . . 46
5.5.4 Indexing a data frame . . . 47
5.6 Reading tabular data . . . 47
5.7 Check your understanding . . . 48
5.8 Summary . . . 50
5.9 Exercises . . . 50
5.10 Further reading . . . 51

6 Blocks, Loops, and Functions 52
6.1 Blocks group lines of code . . . 52
6.2 if statements run a block only when a test is TRUE . . . 53
6.2.1 Adding else . . . 53
6.2.2 Writing good tests . . . 54
6.3 while loops repeat a block while a test is TRUE . . . 54
6.4 for loops run a block once for each item . . . 55
6.4.1 Saving results from a loop . . . 55
6.4.2 Do you need a loop? . . . 55
6.5 Functions name a block so you can reuse it . . . 56
6.5.1 A preview: the weighted mean center . . . 56
6.5.2 More than one argument, and defaults . . . 57
6.5.3 Putting blocks inside functions . . . 57
6.6 A function's arguments are separate from variables outside it . . . 57
6.7 Your turn . . . 58
6.8 Check your understanding . . . 59
6.9 Summary . . . 60
6.10 Exercises . . . 60
6.11 Further reading . . . 60

II Unit 2: Into the Tidyverse 61
Chapter 6: Into the Tidyverse: glimpse, filter, and the Pipe . . . 62
Chapter 7: Into the Tidyverse: arrange, distinct, select, and mutate . . . 62
Chapter 8: Into the Tidyverse: summarize, group_by, and drop_na . . . 62
Chapter 9: Into the Tidyverse: Graphs with ggplot2 . . . 62

7 Into the Tidyverse: glimpse, filter, and the Pipe 63
7.1 What is the tidyverse? . . . 63
7.1.1 The data science workflow . . . 63
7.1.2 The core packages . . . 63
7.1.3 Loading the tidyverse . . . 64
7.1.4 Tidying data is a set of verbs . . . 65
7.2 Seeing inside a data frame with glimpse() . . . 65
7.2.1 The flights data . . . 65
7.2.2 glimpse() . . . 66

7.3 Keeping rows with filter() . . . 66
7.3.1 One condition . . . 66
7.3.2 Several conditions . . . 67
7.4 The pipe . . . 67
7.4.1 Chaining steps with |> . . . 67
7.4.2 Don't repeat the data frame . . . 68
7.5 Missing values: why the counts don't match . . . 68
7.5.1 The same question, two ways . . . 68
7.5.2 NA: a missing value . . . 69
7.5.3 Deciding what happens to NAs . . . 69
7.6 Practice: counting flights . . . 70
7.7 Check your understanding . . . 71
7.8 Summary . . . 72
7.9 Exercises . . . 73
7.10 Further reading . . . 73

8 Into the Tidyverse: arrange, distinct, select, and mutate 74
8.1 Verbs for rows and verbs for columns . . . 75
8.1.1 Two kinds of verb . . . 75
8.1.2 Where this fits . . . 75
8.2 Finding unique values with distinct() . . . 75
8.2.1 One column . . . 75
8.2.2 Building up a question . . . 76
8.2.3 Several columns . . . 76
8.3 Sorting rows with arrange() . . . 77
8.3.1 Sorting by one column . . . 77
8.3.2 Sorting every row . . . 77
8.3.3 Largest first . . . 77
8.3.4 Sorting by two columns . . . 78
8.4 Keeping columns with select() . . . 78
8.4.1 Naming the columns you want . . . 78
8.4.2 Dropping and finding columns . . . 78
8.5 Making new columns with mutate() . . . 79
8.5.1 Average speed . . . 79
8.5.2 Several columns at once . . . 79
8.5.3 Slowest and fastest . . . 80
8.6 Practice: the longest flight . . . 80
8.7 Check your understanding . . . 81
8.8 Summary . . . 83
8.9 Exercises . . . 83
8.10 Further reading . . . 84

9 Into the Tidyverse: summarize, group_by, and drop_na 85
9.1 Tidy data and the verbs so far . . . 86
9.1.1 Tidy data . . . 86
9.1.2 Three new verbs . . . 86
9.2 Summarizing with summarize() . . . 87
9.2.1 One number from many rows . . . 87
9.2.2 Summary functions . . . 87
9.2.3 Several statistics at once . . . 88
9.3 One result per group with group_by() . . . 88
9.3.1 Grouping, then summarizing . . . 88
9.3.2 group_by() on its own . . . 89

9.4 Removing missing values with drop_na() . . . 90
9.4.1 Two ways to skip NAs . . . 90
9.4.2 They are not the same thing . . . 90
9.4.3 Name your columns . . . 91
9.5 Practice: how fast did planes fly each month? . . . 91
9.5.1 The obvious way . . . 91
9.5.2 The average of averages . . . 91
9.5.3 Sum first, then divide . . . 92
9.6 Practice: the least punctual plane . . . 92
9.7 Check your understanding . . . 93
9.8 Summary . . . 94
9.9 Exercises . . . 95
9.10 Further reading . . . 95

10 Into the Tidyverse: Graphs with ggplot2 96
10.1 The grammar of graphics . . . 97
10.1.1 ggplot2 expects tidy data . . . 97
10.1.2 What is ggplot2? . . . 97
10.1.3 Every graph has the same parts . . . 97
10.1.4 Choosing a geom . . . 98
10.1.5 The template . . . 98
10.2 Scatterplots and aesthetics . . . 98
10.2.1 A first scatterplot . . . 98
10.2.2 Adding a trend line . . . 99
10.2.3 Setting an aesthetic . . . 99
10.2.4 Mapping an aesthetic . . . 100
10.2.5 Too many colors . . . 100
10.3 Bar charts and boxplots . . . 101
10.3.1 Counting with geom_bar() . . . 101
10.3.2 fill and color . . . 101
10.3.3 Stacked bars . . . 101
10.3.4 Boxplots . . . 102
10.4 Small multiples with facets . . . 103
10.4.1 facet_wrap() . . . 103
10.4.2 facet_grid() . . . 103
10.5 Finishing a plot . . . 103
10.5.1 Choosing your own colors . . . 103
10.5.2 Adding labels with labs() . . . 104
10.5.3 Themes . . . 104
10.5.4 Saving a plot with ggsave() . . . 104
10.6 Practice: June flights . . . 105
10.7 Check your understanding . . . 106
10.8 Summary . . . 107
10.9 Exercises . . . 108
10.10 Further reading . . . 108

III Unit 3: Maps and Geographic Data 109
Chapter 10: Thematic Maps with tmap . . . 110
Chapter 11: What Is Geographic Data? . . . 110
Chapter 12: Location and Coordinate Systems . . . 110
Chapter 13: Census Data and APIs . . . 110
Chapter 14: ArcGIS Data with arcpullr . . . 111

11 Thematic Maps with tmap **112**
11.1 Spatial data are tidy data . . . 112
11.1.1 Features and attributes . . . 112
11.1.2 tmap: the thematic mapper . . . 113
11.1.3 Our data: three campus layers . . . 113
11.1.4 tmap and ggplot2 on the same trees . . . 113
11.2 Points: mapping trees with tm_dots() . . . 114
11.2.1 Color trees by height . . . 114
11.2.2 Color trees by species . . . 115
11.2.3 Fix 1: map only the top five species . . . 115
11.2.4 Fix 2: group rare species as "other" . . . 117
11.3 Lines: mapping sidewalks with tm_lines() . . . 117
11.3.1 Draw the sidewalk network . . . 117
11.3.2 Color sidewalks by surface . . . 118
11.4 Polygons: mapping buildings with tm_polygons() . . . 119
11.4.1 Draw the building footprints . . . 119
11.4.2 Shade buildings by year built . . . 119
11.4.3 Control the classes with n and style . . . 121
11.5 Putting a map together . . . 122
11.5.1 Combine points, lines, and polygons . . . 122
11.5.2 Add a scale bar and north arrow . . . 123
11.5.3 Static or interactive: tmap_mode() . . . 123
11.6 Practice: map the world . . . 125
11.7 Check your understanding . . . 127
11.8 Summary . . . 128
11.9 Exercises . . . 128
11.10 Further reading . . . 129

12 What Is Geographic Data? **130**
12.1 From the real world to the computer . . . 130
12.1.1 Entities and features . . . 130
12.1.2 Data abstraction . . . 131
12.1.3 Continuous data and discrete objects . . . 131
12.1.4 Geographic data models . . . 131
12.2 Vector data . . . 132
12.2.1 One object, three representations . . . 132
12.2.2 Generalization and map scale . . . 132
12.2.3 What makes something an entity? . . . 133
12.2.4 Spatial data and attributes . . . 133
12.2.5 The parts of vector data . . . 134
12.2.6 Thematic layers . . . 134
12.3 Raster data . . . 135
12.3.1 Vector and raster compared . . . 135
12.3.2 A raster is a grid of cell values . . . 135
12.3.3 Cell size and resolution . . . 137
12.3.4 Data structures . . . 138
12.4 Geographic file formats . . . 138
12.4.1 Shapefiles . . . 139
12.4.2 GeoJSON . . . 139
12.4.3 KML . . . 140
12.4.4 GeoTIFF . . . 140
12.4.5 Metadata . . . 140
12.4.6 Spatial data in R . . . 141
12.5 Check your understanding . . . 141

12.6 Summary . . . 142
12.7 Exercises . . . 143
12.8 Further reading . . . 143

13 Location and Coordinate Systems 144
13.1 Where are we? . . . 144
13.2 Geodesy: the shape of the Earth . . . 145
13.2.1 Measuring the Earth . . . 145
13.2.2 Sphere or ellipsoid? . . . 145
13.2.3 Datums . . . 146
13.3 Building a coordinate system . . . 146
13.3.1 The graticule . . . 147
13.3.2 A degree isn't a fixed distance . . . 147
13.4 Writing coordinates . . . 147
13.5 Map projections . . . 148
13.5.1 Projecting the globe . . . 148
13.5.2 Distortion . . . 149
13.5.3 Why distance distorts . . . 149
13.5.4 Projection families . . . 150
13.5.5 The State Plane Coordinate System . . . 150
13.6 Choosing a projection . . . 150
13.7 Coordinate reference systems in R . . . 151
13.7.1 Degrees are not meters . . . 151
13.8 Check your understanding . . . 152
13.9 Summary . . . 153
13.10 Exercises . . . 154
13.11 Further reading . . . 154

14 Census Data and APIs 155
14.1 A brief history of the US Census . . . 155
14.1.1 The moving center of population . . . 156
14.2 Census geography . . . 158
14.2.1 FIPS codes and GEOIDs . . . 159
14.2.2 What data are available at each level . . . 160
14.2.3 ACS 1-year and 5-year estimates . . . 161
14.3 Commonly used variables . . . 161
14.4 What is an API? . . . 163
14.4.1 Web APIs: URLs in, JSON out . . . 164
14.4.2 Anatomy of a request . . . 164
14.4.3 Why use an API? . . . 164
14.5 Calling an API from R . . . 165
14.5.1 Step 0: get and store a key . . . 165
14.5.2 By hand, with httr2 . . . 165
14.5.3 With a package: tidycensus . . . 166
14.6 Worked example: mapping young children in North Carolina . . . 167
14.6.1 Find the variables . . . 167
14.6.2 Download data with geometry . . . 168
14.6.3 Calculate and map . . . 169
14.7 Going further with tidycensus . . . 170
14.7.1 Margins of error . . . 170
14.7.2 Calculating percentages . . . 171
14.7.3 Decennial Census data . . . 172
14.7.4 Comparing across years . . . 173
14.7.5 More tidycensus functions . . . 173

14.8 Summary . . . 174
14.9 Exercises . . . 174
14.10 Further reading . . . 175

15 ArcGIS Data with arcpullr **176**
15.1 Recap: APIs . . . 176
15.2 ArcGIS portals and feature services . . . 176
15.3 Finding a layer's REST URL . . . 177
15.3.1 Reading the REST Services Directory page . . . 179
15.4 Anatomy of an arcpullr request . . . 179
15.4.1 Writing a where clause . . . 180
15.5 Two requests, one map . . . 180
15.5.1 Before you analyze . . . 181
15.5.2 More arcpullr functions . . . 181
15.6 Try it . . . 182
15.7 Check your understanding . . . 182
15.8 Summary . . . 183
15.9 Exercises . . . 183
15.10 Further reading . . . 184

IV Unit 4: Spatial Analysis **185**
Chapter 15: Joins: Attribute and Spatial . . . 186
Chapter 16: Exploratory Spatial Data Analysis . . . 186
Chapter 17: Spatial Descriptive Statistics . . . 186

16 Joins: Attribute and Spatial **187**
16.1 What is a join? . . . 187
16.2 Types of joins . . . 188
16.2.1 Inner join . . . 188
16.2.2 Left join . . . 188
16.2.3 Right join . . . 189
16.2.4 Full join . . . 189
16.2.5 Natural joins and cross joins . . . 189
16.2.6 Joining more than two tables . . . 189
16.3 Cardinality . . . 190
16.4 Why join by location? . . . 190
16.5 Spatial joins . . . 191
16.5.1 From attribute joins to spatial joins . . . 191
16.5.2 Spatial joins in R . . . 191
16.5.3 Four kinds of spatial join . . . 192
16.5.4 Summarized joins in R . . . 192
16.6 Spatial joins and the coordinate system . . . 193
16.7 Check your understanding . . . 193
16.8 Summary . . . 195
16.9 Exercises . . . 195
16.10 Further reading . . . 196

17 Exploratory Spatial Data Analysis **197**
17.1 From EDA to ESDA . . . 197
17.1.1 The "understand" loop . . . 197
17.1.2 Two questions guide the investigation . . . 198
17.1.3 ESDA adds a geographic question . . . 198
17.2 Step one: get to know the data table . . . 198

17.3 Describing a variable's center . . . 199
17.3.1 Mean . . . 199
17.3.2 Median . . . 200
17.3.3 Mode . . . 200
17.4 Describing a variable's spread . . . 200
17.4.1 Range and interquartile range . . . 200
17.4.2 Standard deviation . . . 201
17.4.3 All together . . . 201
17.5 Visualizing distributions . . . 202
17.5.1 Histograms and boxplots . . . 202
17.5.2 Choosing a bin width . . . 203
17.5.3 Shape decides whether the mean is typical . . . 203
17.5.4 Outliers . . . 204
17.5.5 Density plots . . . 204
17.5.6 Your turn . . . 205
17.6 Maps add the spatial dimension . . . 206
17.7 Spatial statistics describe location . . . 206
17.8 Check your understanding . . . 207
17.9 Summary . . . 208
17.10 Exercises . . . 208
17.11 Further reading . . . 208

18 Spatial Descriptive Statistics 209
18.1 Many statistics have a spatial counterpart . . . 209
18.2 Mean center . . . 210
18.2.1 The mean center of North Carolina . . . 210
18.3 Weighted mean center . . . 211
18.4 Central feature . . . 212
18.5 Standard distance . . . 212
18.6 Functions in R . . . 213
18.6.1 What is a function? . . . 213
18.6.2 Writing your own . . . 214
18.6.3 A function for the weighted mean center . . . 215
18.7 Check your understanding . . . 216
18.8 Summary . . . 217
18.9 Exercises . . . 217
18.10 Further reading . . . 217

V Unit 5: Statistics and Science 218
Chapter 18: Comparing Groups: p-values and ANOVA . . . 219
Chapter 19: Linear Regression . . . 219
Chapter 20: Writing a Spatial Data Science Project . . . 219
Chapter 21: Distance and Neighborhoods . . . 219
Chapter 22: Spatial Clustering and Spatial Autocorrelation . . . 220

19 Comparing Groups: p-values and ANOVA 221
19.1 What is data analysis? . . . 221
19.2 Statistics and p-values . . . 222
19.2.1 Reading a test result . . . 222
19.2.2 p-values and the replication crisis . . . 222
19.3 Variance: measuring spread . . . 223
19.3.1 Two kinds of variation . . . 223
19.4 ANOVA: are the group means different? . . . 224

19.5 Example: median male age by North Carolina region . . . 225
19.5.1 The ANOVA . . . 227
19.6 After ANOVA: which groups differ? . . . 228
19.6.1 Check the assumptions . . . 228
19.7 What makes it spatial data science? . . . 229
19.8 Check your understanding . . . 229
19.9 Summary . . . 231
19.10 Exercises . . . 231
19.11 Further reading . . . 231

20 Linear Regression **233**
20.1 From data to knowledge . . . 233
20.1.1 Spatial data are special . . . 234
20.2 Regression explains how one variable changes with others . . . 234
20.2.1 Correlation and regression . . . 234
20.3 Writing a regression formula . . . 235
20.3.1 A small example to try . . . 235
20.4 Anatomy of a regression . . . 236
20.5 Reading the model summary . . . 237
20.6 Example: house values in North Carolina . . . 237
20.6.1 The first model . . . 238
20.6.2 Adding variables: a multivariate model . . . 239
20.7 Mapping residuals . . . 240
20.8 Check your understanding . . . 241
20.9 Summary . . . 242
20.10 Exercises . . . 242
20.11 Further reading . . . 243

21 Writing a Spatial Data Science Project **244**
21.1 No information without data . . . 244
21.1.1 The DIKW pyramid . . . 244
21.2 The data science process . . . 245
21.2.1 Big data . . . 245
21.2.2 Where data come from . . . 245
21.3 Science: reproducibility and replicability . . . 246
21.4 Why spatial data are special . . . 246
21.5 Building blocks of a project . . . 247
21.5.1 Introduction: the research question . . . 247
21.5.2 Background: build on earlier research . . . 248
21.5.3 Data and methods: show your work . . . 248
21.5.4 Results: model your question . . . 248
21.5.5 Check the model: map the residuals . . . 248
21.5.6 A second method: comparing regions with ANOVA . . . 249
21.5.7 Conclusion: answer the question . . . 249
21.5.8 References . . . 249
21.6 So, what is spatial data science? . . . 249
21.7 Check your understanding . . . 250
21.8 Summary . . . 251
21.9 Exercises . . . 251
21.10 Further reading . . . 252

22 Distance and Neighborhoods **253**
22.1 Distance is a fundamental geographic concept . . . 253
22.1.1 Distance decay . . . 254

22.1.2 Why measure distance? . . . 254
22.2 Measuring distance . . . 254
22.2.1 Euclidean distance . . . 254
22.2.2 Great circle distance . . . 255
22.2.3 Manhattan distance . . . 255
22.2.4 Network distance and travel time . . . 256
22.2.5 Straight lines or networks? . . . 256
22.3 Neighborhoods and neighbors . . . 256
22.3.1 Comparing separation between features . . . 256
22.3.2 Why neighbors matter . . . 257
22.3.3 Binary or continuous neighbors . . . 257
22.4 Three ways to define neighbors . . . 257
22.4.1 Distance band neighbors . . . 258
22.4.2 Inverse distance weights . . . 258
22.4.3 k nearest neighbors . . . 258
22.4.4 Contiguity: rook and queen . . . 260
22.5 The neighborhood weight matrix . . . 260
22.5.1 Row standardization . . . 261
22.5.2 Building neighbors and weights in R . . . 262
22.5.3 Weight matrices can encode social distance . . . 262
22.6 Check your understanding . . . 262
22.7 Summary . . . 264
22.8 Exercises . . . 264
22.9 Further reading . . . 264

23 Spatial Clustering and Spatial Autocorrelation 266
23.1 Tobler's first law . . . 266
23.1.1 Geographic variation . . . 267
23.1.2 Seeing a pattern is not testing it . . . 267
23.2 Spatial cluster analysis: global and local . . . 267
23.3 Spatial pattern: how objects are arranged . . . 267
23.3.1 The Average Nearest Neighbor ratio . . . 268
23.3.2 Limits of the ANN ratio . . . 268
23.4 Spatial autocorrelation . . . 269
23.4.1 Which data suit spatial autocorrelation? . . . 269
23.4.2 It depends on who counts as a neighbor . . . 269
23.5 Moran's I: a global test . . . 269
23.5.1 How Moran's I is calculated . . . 270
23.5.2 Moran's I ranges from –1 to 1 . . . 270
23.5.3 Is it significant? . . . 271
23.6 Interpreting Moran's I . . . 271
23.6.1 Magnitude: the I value . . . 271
23.6.2 Significance: the p-value . . . 272
23.6.3 Importance . . . 272
23.7 Moran's I in R . . . 272
23.8 Local hot-spot analysis: Getis-Ord Gi* . . . 273
23.8.1 Local Getis-Ord Gi* in R . . . 273
23.9 Summary: choosing a spatial statistic . . . 275
23.10 Check your understanding . . . 275
23.11 Summary . . . 276
23.12 Exercises . . . 277
23.13 Further reading . . . 277

VI Unit 6: Next Steps **278**
Chapter 23: Interactive Map Dashboards . . . 279
Chapter 24: What's Next in Spatial Data Science . . . 279

24 Interactive Map Dashboards **280**
24.1 What is a dashboard? . . . 280
24.1.1 Reports and dashboards do different jobs . . . 280
24.1.2 What makes a good dashboard . . . 281
24.2 The R dashboard toolkit . . . 281
24.2.1 Two kinds of interactivity . . . 282
24.2.2 Our example: income and poverty by state . . . 282
24.3 Interactive maps with leaflet . . . 283
24.3.1 Your first leaflet map . . . 283
24.3.2 Coloring a choropleth . . . 283
24.3.3 Labels, popups, and highlighting . . . 284
24.3.4 A shortcut: tmap in view mode . . . 284
24.4 Laying out a flexdashboard . . . 285
24.4.1 A flexdashboard is an R Markdown file . . . 285
24.4.2 Columns, panels, and a sidebar . . . 285
24.4.3 Value boxes, tables, and charts . . . 286
24.4.4 Linking views with crosstalk . . . 286
24.4.5 crosstalk and leaflet: points, not polygons . . . 287
24.5 Adding Shiny . . . 287
24.5.1 When do you need Shiny? . . . 287
24.5.2 Inputs in the sidebar . . . 288
24.5.3 Reactive data and render functions . . . 289
24.6 Sharing your dashboard . . . 290
24.6.1 Common pitfalls and fixes . . . 290
24.6.2 Looking ahead: Quarto dashboards . . . 290
24.7 The complete example files . . . 290
24.8 Summary . . . 295
24.9 Exercises . . . 295
24.10Further reading . . . 295

25 What's Next in Spatial Data Science **296**
25.1 Spatial data science in action . . . 296
25.1.1 Public health: where disease happens . . . 297
25.1.2 Public health: access to care . . . 297
25.1.3 Public health: environmental exposure and justice . . . 297
25.1.4 Environmental science: watching the planet change . . . 297
25.1.5 Environmental science: hazards and resilience . . . 298
25.1.6 Cities, transportation, and planning . . . 298
25.1.7 Social science, history, and politics . . . 298
25.1.8 Business, agriculture, and humanitarian work . . . 298
25.2 Remote sensing: seeing Earth from above . . . 299
25.2.1 Four kinds of resolution . . . 299
25.2.2 From pixels to information . . . 300
25.3 Space-time modeling . . . 300
25.4 GeoAI . . . 301
25.4.1 Foundation models and AI assistants . . . 301
25.4.2 Spatial thinking keeps AI honest . . . 302
25.4.3 Scaling up: cloud, big data, and open science . . . 302
25.5 Keep learning at UNC . . . 302
25.5.1 A path through UNC GIS courses . . . 303

25.5.2 Geography courses by interest 303
25.5.3 GIS courses beyond Geography 304
25.5.4 Turn your interest into a credential 304
25.6 Careers in GIS and spatial data science 304
25.6.1 Where geospatial professionals work 305
25.6.2 Skills employers look for 305
25.6.3 Start building your portfolio now 305
25.7 Summary 306
25.8 Exercises 306
25.9 Further reading 307

Appendices **308**

Glossary **308**

About the author **324**
About this book 324

References **325**

Preface

This book is an open-educational resource textbook for designed for my course, GEOG 215, Introduction to Spatial Data Science, built from the course lectures.

Every chapter is a runnable Quarto document. The R code in each chapter runs when the book is built, so the tables and maps you see come from the code printed beside them.

About this book

This is an interactive textbook specifically designed and written for GEOG 215. It's designed for students new to R and new to spatial data. Students are expected to have some experience with math and/or statistics.

How to use this book

Read the book in order if you can. Each chapter builds on the ones before it, and the R code in later chapters assumes you have met the functions and ideas introduced earlier. Each unit opens with a short page that says what its chapters cover, so you can see where you are going before you start.

What's in each chapter

Most chapters follow the same pattern, so you know where to look:

- **Learning objectives** at the top list what you should be able to do by the end of the chapter. They are a good checklist when you review for an exam.
- **Numbered sections** teach the material, with tables, figures, and R code. Cross-references such as "Section 3.2" or "Figure 5.1" are links you can follow.
- **Think about it** boxes pose a question to stop and consider before reading on. There is no grade attached: they are there to help the idea stick.
- **Check your understanding** questions near the end let you test yourself. You can answer them as many times as you like, and they are not graded.
- A **Summary** recaps the chapter's main points, **Exercises** give you practice to do in RStudio, and **Further reading** points you to sources that go deeper.

Running the code

The code in this printed version is not interactive. To run it, type or paste it into RStudio, which Chapter 2 helps you set up. An online version, where many code boxes run R right in your web browser, is at https://daveparr.com/intro-spatial-data-science/.

In the code, `code` and the names of functions such as `filter()` are printed in a monospaced font. Lines beginning with # are comments that explain the code; R skips them.

Key terms and the glossary

Important terms are printed in **bold** where they are first explained.

All of the key terms and their definitions are collected in the Glossary at the end of the book.

The sources the chapters cite are listed together on the References page at the end of the book.

How this book is organized

The chapters are grouped into six units. Each unit opens with a page that summarizes its chapters.

Unit 1: R and Spatial Data Science (Chapters 1–5) sets up everything else. It explains what spatial data science is and why location matters, then gets you working in R, RStudio, and R Markdown, with R's basic building blocks: vectors, data types, and data frames. It ends with `if` statements, loops, and writing your own functions, so you can make decisions and repeat work in code.

Unit 2: Into the Tidyverse (Chapters 6–9) introduces the tidyverse. Working with a year of flights from New York's airports, you learn the dplyr verbs that filter, sort, select, change, and summarize data, chained together with the pipe, and then turn the results into graphs with ggplot2.

Unit 3: Maps and Geographic Data (Chapters 10–14) moves from graphs to maps. You make thematic maps of the UNC campus with tmap, then step back to see how the real world becomes geographic data: the vector and raster data models, the file formats that store them, and the datums, coordinate systems, and map projections that give every feature its location. It ends by getting geographic data from the places that publish it: the US Census Bureau, through the APIs that let your code download its data directly, and the ArcGIS servers that publish many government GIS layers.

Unit 4: Spatial Analysis (Chapters 15–17) starts to analyze spatial data. You join tables by a shared column and by location, explore variables with descriptive statistics, plots, and maps, and describe where features are with spatial statistics such as the mean center. Along the way you put your own R functions to work.

Unit 5: Statistics and Science (Chapters 18–22) turns data into evidence. You compare groups with p-values and ANOVA, model relationships with linear regression, and put it all together in a reproducible spatial data science project. It ends by asking whether near things are more related than distant things: you measure distance, define neighbors and weight matrices, and test for clustering and hot spots with the Average Nearest Neighbor ratio, Moran's I, and Getis-Ord Gi*.

Unit 6: Next Steps (Chapters 23–24) looks beyond the course. You turn your maps into interactive dashboards with leaflet, flexdashboard, crosstalk, and Shiny, and then look at where spatial data science is used, the advanced topics that build on this book, such as remote sensing, space-time modeling, and GeoAI, and the UNC courses and careers that come next.

Fonts and reading settings

The text of this book is set in **Atkinson Hyperlegible Next**, and the code is set in **Atkinson Hyperlegible Mono**. Both were designed by the Braille Institute of America for readers with low vision, and both are free to use. Many common letters and numbers look alike in ordinary fonts: a capital I, a lowercase l, and the number 1, or a capital O and a zero. Atkinson Hyperlegible gives each of these characters its own distinct shape, so they are hard to mistake for one another. That matters in a book about code, where `l1` and `I1` are different names and a single wrong character breaks a line of R. The zero in the code font has a slash through it, so you can always tell `0` from O.

A font that is easy for one reader can be hard for another. The online version of this book, at https://daveparr.com/intro-spatial-data-science/, lets you change the font and text size with the **Aa** button beside the search box. The choices include Lexend, designed to reduce visual stress, and OpenDyslexic, which some readers with dyslexia find easier to read.

Acknowledgments

I'd like to thank the work of Drs. Julia Cardwell, Paul Delamater, Paul Tailie, and Conghe Song for their support. I want to thank the Teaching Assistants Jacob Tafrate, Braulio Lahuatte, Najila Porno, Ngyuen Tien Anh Quach, Sophia Alhadeff, Andrew Robinson, and Cameron Brousse. I'd like to thank my Undergraduate Learning Assistants, Harvey Chen, Rachel Johnson, and Esther Zhu. Finally, I'd like to acknowledge the hard work of my students, without whom, this would not be possible.



Part I

Unit 1: R and Spatial Data Science



This unit sets up everything else in the book. It starts with what spatial data science is and why location matters. It then gets you working in R and RStudio, writing reproducible R Markdown reports, and handling R's basic building blocks: vectors, data types, and data frames. It ends with the R tools that save you from repeating yourself: making decisions with `if`, repeating work with loops, and writing your own functions. By the end of the unit you can load data into R, pull out exactly the values you need, and automate the steps you would otherwise do by hand.

Chapter 1: What is Spatial Data Science?

Spatial data science uses data with locations to answer questions about the world. This chapter explains how data become information, walks through the steps of a data science project, from a question to communicating results, and shows why spatial data need special care: near things tend to be alike, patterns change with scale, and relationships change from place to place. It ends by introducing the tools used in this course: R, RStudio, the tidyverse, sf, and tmap.

Chapter 2: R, RStudio, and R Markdown

R is the language that does the computing, RStudio is the workspace where you write and run it, and R Markdown is the format that combines your writing, your code, and its results in one report. This chapter tours RStudio's four panes, explains how knitting builds a reproducible report, and introduces variables, comments, functions, and the pipe.

Chapter 3: R Basics

R runs one line at a time, in the order you run it, so this chapter shows how to think sequentially and make sure a file will knit. It covers installing and loading packages, R's four basic data types and how vectors hold them, good names for variables, and how to read error messages and find help.

Chapter 4: Indexing Vectors and Data Frames

Indexing selects part of a vector or a table with square brackets: by position, by name, or by a condition built from comparison and logical operators. The chapter then introduces the data frame, R's table of variables and observations, and shows how to read a CSV file into one and select its rows and columns.

Chapter 5: Blocks, Loops, and Functions

Good programmers are productively lazy. This chapter shows how blocks group lines of code, how `if`, `else if`, and `else` decide whether a block runs, and how `while` and `for` loops repeat one. It compares loops with R's vectorized functions, such as `ifelse()`, then takes



functions apart: arguments, defaults, return values, error checks with `stop()`, and variable scope.

1 What is Spatial Data Science?

Learning objectives

By the end of this chapter you should be able to:

1. Explain what spatial data science is, and how data become information, knowledge, and wisdom.
2. Explain why location matters, with examples from several fields.
3. Describe three ideas that make spatial data different from other data: spatial autocorrelation, scale, and spatial heterogeneity.
4. Recognize a spatial data science question, and write one of your own.
5. Describe the tools and the path this book follows.

This book is about answering questions with data that have a location attached. Where are thefts and burglaries concentrated in a town? Which neighborhoods are far from a hospital? How much forest was lost in the mountains after a hurricane? Questions like these come up in planning, public health, business, environmental science, and nearly every other field, and they share a common set of skills: finding the right data, turning it into useful information, and communicating what you found.

This first chapter steps back from the code to look at the big picture. It explains what spatial data science is, why location deserves special attention, and what kinds of questions you will be able to answer by the end of the book. Every later chapter fills in one part of that picture.

1.1 What this book will teach you

By the end of the book, you will be able to do three things:

1. **Use R to collect and manage spatial data.**
2. **Turn spatial data into information**, such as tables, statistics, and maps.
3. **Answer spatial data science questions.**

Each of these builds on the one before it. You can't make a useful map until you can load and clean the data behind it, and you can't answer a question until you can turn raw data into something that speaks to it.

A word about math

Spatial data science uses statistics, and some chapters include formulas. Don't let that put you off. The emphasis throughout is on what a method does and when to use it, and R does the arithmetic. If you can read a formula slowly and say in words what each part means, you have all the math you need.

1.2 What is spatial data science?

1.2.1 From data to wisdom

It helps to start with the word *data* itself, and with how it differs from the things we actually want from it. A common way to describe this is a ladder with four rungs (Figure 21.1).

- **Data** are facts and figures: a column of numbers, a list of addresses, the pixel values of a satellite image.
- **Information** is data organized so that it is useful: a table of average income by county, or a map of where those addresses fall.
- **Knowledge** is information accumulated and integrated over time and across a broad range of situations: an understanding of *why* income varies the way it does, built from many studies of many places.
- **Wisdom** is the application of principles, reason, and knowledge to discern what is true and right: deciding what a city should actually do about it.

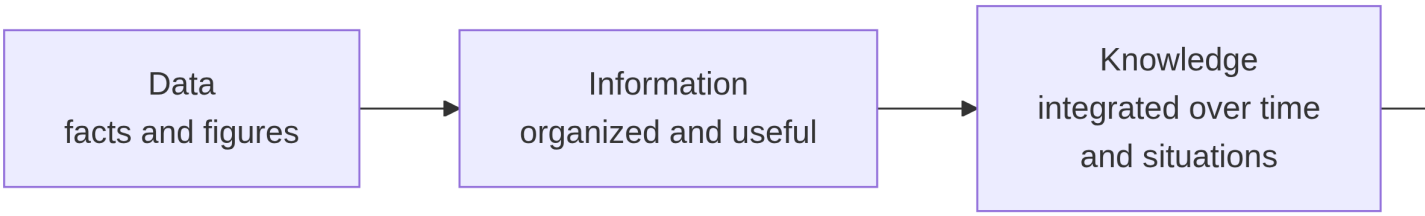


Figure 1.1: The data–information–knowledge–wisdom ladder. Each step adds organization, context, and judgment. This book focuses on the first step: turning data into information.

Most of this book is about the first step, from data to information. That step is where code does the most work: no one can read a million rows of data, but anyone can read a well-made map or a short table that summarizes them. The later steps depend on people, but they can only be as good as the information they rest on.

1.2.2 Spatial, data, science

The name of the field is a good summary of what it involves. Each of its three words covers a different set of skills (Table 1.1).

Table 1.1: The three parts of spatial data science, and some of what each involves.

Spatial	Data	Science
Location	Processing	Research
Environmental data	Organizing	Structure
Satellite imagery	Parsing	Questioning
Demographics and the Census	File formats	Reporting
Transportation	Databases	Critiquing
	Visualizing with maps and graphs	

Spatial is about *what* we study: things that happen in places, measured by sources such as satellites, environmental sensors, the Census, and transportation networks. **Data** is about *how* we handle it: reading files in many formats, organizing and cleaning them, storing them, and visualizing them. **Science** is about *why*: asking a clear question, structuring an analysis to answer it, reporting the results, and critiquing them honestly, including your own.

1.3 Why does location matter?

Everything happens somewhere. From changes in trade policy to the spread of a disease, location is a fundamental attribute of almost everything we might want to study. Knowing *where* something happens often tells you a great deal about *why* it happens, and about who it affects. A few examples show how widely this applies:

- **Urban planning.** Where should a city build new parks or schools so that the most people can reach them?
- **Environmental science.** How does air or water pollution spread across a region from its sources?
- **Public health.** Where are the hotspots of a disease outbreak, and what do those places have in common?
- **Business.** Where should a company put its next store or warehouse, and how should goods move between them?
- **Social sciences.** How are demographic trends, such as aging or migration, distributed across a country?

In each case, a table without locations would miss the point. The question is *about* place.

> **Think about it**
>
> How could you use spatial data science to understand the effects of a change in tariffs, the taxes a country charges on imported goods? What would you map? Which places might be affected most: ports, farming regions that export crops, towns built around one factory? What data would you need to find out?

1.4 The spatial data science workflow

Whatever the question, a spatial data science project moves through the same three stages (Figure 1.2).

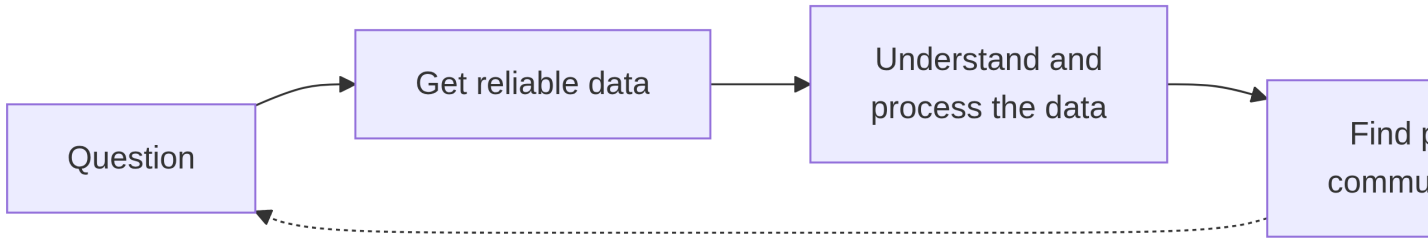


Figure 1.2: The spatial data science workflow. A question leads to data, data must be processed before they can be analyzed, and results must be communicated. What you learn often raises a new question.



1. **Get reliable data.** Whatever question you want to ask, the answer can only be as good as the data behind it. That means knowing where your data came from, who collected it and how, what each column means, and what is missing.
2. **Understand and process the data.** Real data are rarely ready to use. They need to be read into R, cleaned, reshaped, joined to other data, and often given a location or put into the right coordinate system. This stage usually takes the most time.
3. **Find patterns and communicate the results.** Look for patterns, relationships, and trends in the data, then present them in a way that answers your question. A clear map, table, or paragraph is the product of the whole project. Results that no one can understand might as well not exist.

The arrow back to the start of Figure 1.2 matters, too. Answering one question usually raises another, and good projects often loop through the workflow several times.

1.5 Spatial thinking

Much of traditional statistics, and much of data science, treats each observation as **independent** of the others: knowing one tells you nothing about the next. For data about places, that assumption is usually wrong. Spatial thinking means taking seriously how location connects observations. Three ideas come up again and again in this book.

1.5.1 Spatial autocorrelation

Spatial autocorrelation is the tendency for things that are close together to be more alike than things that are far apart. Neighboring homes tend to have similar values. Neighboring counties tend to have similar incomes, voting patterns, and rainfall. The geographer Waldo Tobler put it as what is now called the **first law of geography**: "everything is related to everything else, but near things are more related than distant things" (Tobler 1970).

Spatial autocorrelation is both a problem and an opportunity. It is a problem because standard statistical tests assume independence, and they can be badly overconfident when observations are really echoes of their neighbors. It is an opportunity because it is often exactly what we want to find: a cluster of high values is a hotspot, and hotspots are where to look first. Later chapters show how to measure it.

1.5.2 Scale

Scale is the level of detail you work at: individual addresses, neighborhoods, counties, states. A pattern that is clear at one scale can vanish, or even reverse, at another. A county can have a high average income while containing neighborhoods with very low incomes, and a trend across a whole city might not exist within any single neighborhood.

The way areas are drawn matters as well as their size. Grouping the same individual data into different sets of boundaries, such as different districts of the same size, can change the results of an analysis. This is known as the **modifiable areal unit problem** (Openshaw 1984). The practical lesson is to choose your scale deliberately, to match the question, and to be careful about drawing conclusions at one scale from data collected at another.

1.5.3 Spatial heterogeneity

Spatial heterogeneity means that relationships between variables can change from place to place. The link between, say, distance to downtown and home prices might be strong in one city and weak in another, or even differ between the east and west sides of the same city. A single number that summarizes a relationship for a whole region can hide those differences. Spatial data science often means asking not just "what is the relationship?" but "where is it strongest, and where does it break down?"

> **Three questions to ask of any spatial data**
>
> 1. Are nearby values similar? (spatial autocorrelation)
> 2. Would the pattern look different with larger or smaller areas? (scale)
> 3. Does the relationship I'm studying hold everywhere, or only in some places? (spatial heterogeneity)

1.6 Our toolbox

This book uses the **R** programming language, written and run in **RStudio**. Both are free. On top of R, a few packages do most of the work:

Table 1.2: The main R packages used in this book.

Package	What it does
tidyverse	Reading, cleaning, reshaping, and summarizing data
sf	Storing and working with spatial vector data: points, lines, and polygons
tmap	Making maps quickly, with sensible defaults
Others	Statistics, downloading data, and special tasks as they come up

The next chapter, R, RStudio, and R Markdown, introduces these tools and the R Markdown documents you will write your work in.

1.7 The road ahead

The course is organized in three units, each building on the last (Table 1.3).

Table 1.3: The three units of the course.

Unit	Focus	Topics
1	Getting started with data science	Getting started with R; processing data with the tidyverse; exploring data for information; a map contest with tmap
2	Working with spatial data	Overlaying two spatial layers; joining tables; spatial statistics; a map contest with online data

Unit	Focus	Topics
3	Spatial data science	ANOVA; regression; clustering; a spatial data science project

Unit 1 builds general data skills: you learn to load, clean, summarize, and map data. Unit 2 adds what is special about spatial data: combining layers by location, joining tables, and measuring spatial patterns. Unit 3 brings it together with statistical methods for comparing groups, modeling relationships, and finding clusters, and ends with a project of your own.

1.8 What makes a spatial data science question?

A spatial data science question is one where location is part of the answer, and where data can settle it. Here are some examples from North Carolina. Notice the key word in each, which hints at the kind of analysis it needs.

Table 1.4: Examples of spatial data science questions.

Question	Key idea	Kind of analysis
Where are there higher clusters of theft and burglary in Chapel Hill?	Where	Mapping and cluster detection
Are incomes **different** between Durham and Raleigh?	Difference	Comparing groups (e.g., ANOVA)
What location **factors** contribute to housing prices in Charlotte?	Factors	Modeling relationships (regression)
How has forest cover **changed** in the Blue Ridge Mountains after Hurricane Helene?	Change	Comparing data over time
Which neighborhoods have less **access** to hospitals?	Accessibility	Distance and network analysis

Good questions have a few things in common. They are specific about *what* is being measured and *where*. They can be answered with data that exist, or that you can collect. And the answer would be useful to someone: a planner, a business, a public health department, or the people who live there.

Think about it

What is a spatial data science project you could do in your own field or area of interest? Spatial data science is used in business, computer science, environmental science, geography, geology, law, real estate, public health, the social sciences, urban planning, communications and journalism, government, transportation, and much more. Write your question down. You will come back to it at the end of the course.

1.9 Check your understanding

1. Organizing raw facts and figures into a table of average income by county turns data into... [1]

- (A) wisdom
- (B) information
- (C) knowledge
- (D) metadata

2. The idea that near things are more related than distant things is called... [2]

- (A) spatial heterogeneity
- (B) scale
- (C) spatial autocorrelation
- (D) the data workflow

3. A relationship between two variables that is strong in one city but weak in another is an example of... [3]

- (A) spatial autocorrelation
- (B) spatial heterogeneity
- (C) a file format
- (D) the modifiable areal unit problem

4. True or false: a trend you find at the county level is guaranteed to hold within each neighborhood of that county. [4]

- (A) TRUE
- (B) FALSE

5. Which R package in our toolbox is used to store and work with points, lines, and polygons? _ [5]

Show hint for question 4

Think about scale: a pattern at one level of detail can disappear, or reverse, at another.

[1] information
[2] spatial autocorrelation
[3] spatial heterogeneity
[4] FALSE
[5] sf

1.10 Summary

- **Spatial data science** uses data with locations to answer questions about the world. It combines the *spatial* (what we study), *data* skills (how we handle it), and *science* (how we ask and answer questions).
- Data become **information** when they are organized to be useful. Information, accumulated and integrated, becomes **knowledge**, and knowledge applied with judgment becomes **wisdom**. This book focuses on turning data into information.
- **Everything happens somewhere**, and location matters in planning, environmental science, public health, business, the social sciences, and beyond.
- A project moves from a **question** to **reliable data**, to **processing**, to **finding and communicating results**, and often loops back to a new question.
- Spatial data are rarely independent. Watch for **spatial autocorrelation** (near things are alike), **scale** (patterns change with the size and shape of areas), and **spatial heterogeneity** (relationships change from place to place).
- Our tools are **R** and **RStudio**, with the **tidyverse**, **sf**, and **tmap** packages.

1.11 Exercises

1. **Data to information.** Pick a dataset you use or have heard of, such as weather records, sports statistics, or campus enrollment. Describe one example each of data, information, knowledge, and wisdom based on it.
2. **Everything happens somewhere.** Find a recent news story whose central question involves location. What is the question? What spatial data would you need to answer it, and where might you get them?
3. **Spatial thinking.** For each of the three ideas in Section 1.5, give your own example of how it could affect an analysis of a topic you care about.
4. **Classify questions.** For each question in Table 1.4, name one dataset you would need. Which questions need data from more than one time period?
5. **Your question.** Write your own spatial data science question from the last *Think about it* box as one clear sentence. Underline its key word (where, different, factors, change, access, or another), and list the data you would need to answer it.

1.12 Further reading

- *Geocomputation with R* by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, free at https://r.geocompx.org, is an excellent companion to this book for working with spatial data in R.
- *Spatial Data Science: With Applications in R* by Edzer Pebesma and Roger Bivand, free at https://r-spatial.org/book/, is a deeper treatment by the authors of the **sf** package.
- Waldo Tobler's short paper that introduced the first law of geography (Tobler 1970) is still worth reading for its clear statement of why location matters.

2 R, RStudio, and R Markdown

> **Learning objectives**
>
> By the end of this chapter you should be able to:
>
> 1. Explain how R, RStudio, and R Markdown differ, and how they fit together.
> 2. Find your way around the four panes of RStudio.
> 3. Create, run, and knit an R Markdown document.
> 4. Read basic R code: lines, blocks, variables, comments, and functions.

Every chapter after this one asks you to *do* spatial data science: download data, clean it, analyze it, and map it. You will do all of that by writing code. This chapter introduces the three tools you will use for every lab and assignment, and the small vocabulary you need to read and write your first lines of R.

The three tools are easy to mix up, so it helps to keep one sentence in mind from the start: **R is the language, RStudio is the workspace, and R Markdown is the document format.** The first half of the chapter takes each in turn. The second half covers the building blocks of code that appear in every script you will write.

2.1 What is R?

R is a programming language and free software environment for statistical computing and graphics. It was created by statisticians, and its strengths still reflect that: cleaning, analyzing, modeling, and visualizing data. Statisticians, data scientists, and geographers all use it for the same reason. A few lines of R can do what would take dozens of clicks in a spreadsheet or a desktop GIS, and those lines can be run again tomorrow.

On its own, R knows how to do arithmetic, statistics, and basic plots. What makes it useful for geography is its **packages** (also called libraries): bundles of code, written by other users, that add new abilities. There are thousands of them. In this course you will use **sf** for vector data such as points, lines, and polygons, **terra** for rasters, **tidyverse** for cleaning and reshaping data, and **tmap** and **ggplot2** for maps.

2.1.1 Why use R?

R has four strengths that make it the tool for this course.

- **Power.** R has a huge range of tools for statistics, machine learning, and spatial analysis, and new methods often appear as R packages before they appear anywhere else.
- **Visualization.** R makes high-quality, fully customizable plots, charts, and maps. Every figure in this book was made in R.
- **Reproducibility.** A script records every step of an analysis, so you, a classmate, or a reviewer can rerun it and check the result.



- **Community.** A large, global community of users builds packages, answers questions, and writes tutorials. When you get stuck, someone has almost always asked the same question online.

Reproducibility matters most for us. A point-and-click GIS workflow is hard to repeat exactly: which tool did you run, with which settings, in which order? An R script answers all of those questions in writing. When the data are updated, or when you find a mistake in step two of ten, you fix it and run the script again, and every later step updates with it.

There are practical reasons, too. R is free and open source, so everything you learn here keeps working after you graduate, with no license fees. It is updated frequently, it is popular in both research and industry, and it is consistently ranked among the most widely used languages for statistics and data analysis. The skills transfer to jobs and to graduate research.

2.2 What is RStudio?

You can type R commands into a plain window, but almost nobody works that way. **RStudio** is an **integrated development environment (IDE)** for R: one window where you write code, run it, see the results, and check your data. A good way to think about the relationship is that R is the engine and RStudio is the dashboard. R does the computing underneath; RStudio is where you work.

Because RStudio drives R, you need both, and you must **install R first, then RStudio**. RStudio cannot run any code until R is installed. Both are free. Installation steps are in Section 2.6 at the end of this chapter.

2.2.1 The four panes

When you open RStudio, the window is divided into four panes (Figure 2.1).

Table 2.1: The four panes of RStudio.

Pane	Default position	What it is for
Editor	Top left	Writing and saving scripts and R Markdown files
Console	Bottom left	Running commands and reading results, warnings, and errors
Environment	Top right	Listing the variables and data currently in memory
Files, Plots, Help	Bottom right	Browsing folders, viewing graphs, and reading documentation

The difference between the Editor and the Console is the most important one in Table 2.1. The **Editor** is where your work is saved. Anything you type there can be kept in a file and run again. The **Console** is for quick tests: R runs a command the moment you press Enter, but nothing you type there is saved. A useful habit is to experiment in the Console and keep what works in the Editor.

The **Environment** pane lists every variable you have created, with a short description of what it holds. It is the first place to look when you are debugging: if a variable you expected is missing, or holds something unexpected, the Environment will show it. The bottom-right pane holds several tabs, including **Files**, **Plots**, **Packages**, and **Help**.

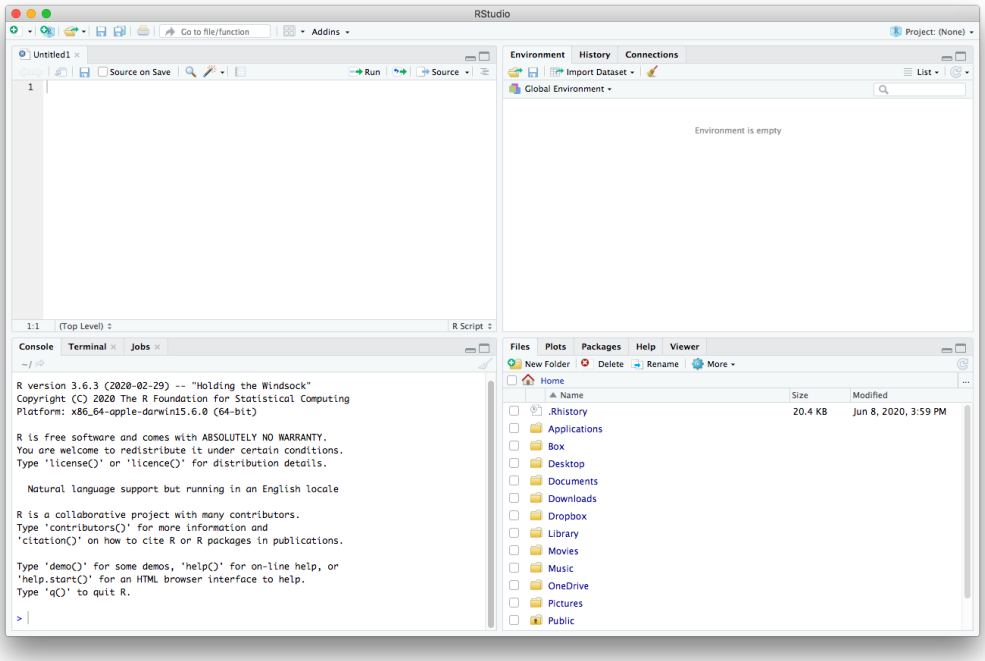


Figure 2.1: RStudio's default four-pane layout, just after opening. The Editor is at the top left, the Console at the bottom left, the Environment at the top right, and Files, Plots, and Help at the bottom right.

2.2.2 RStudio during a working session

Figure 2.2 shows RStudio partway through an analysis, with every pane in use.

The screenshot captures the loop you will repeat constantly:

1. **Write** a line of code in the Editor.
2. **Run** it. Click **Run**, or press Ctrl+Enter (Cmd+Return on a Mac), to send the current line to the Console.
3. **Check the Environment.** If the line created a variable, it appears there.
4. **Read the Console.** Output, warnings, and errors print here.
5. **Look at the Plots pane** if the line drew a graph.

Notice the red error message in the Console of Figure 2.2. Errors are a normal part of writing code, and they are not something to skip past. Read them. They usually say what went wrong, and often where. (This one, "figure margins too large," just means the Plots pane was too small to draw the graph, and it went away when the pane was enlarged.)

2.3 R scripts and R Markdown

There are two kinds of files you will write R in, and it is worth being clear about the difference (Table 2.2).

Table 2.2: R scripts compared with R Markdown documents.

	R script	R Markdown
File extension	`.R`	`.Rmd`

	R script	R Markdown
Contains	Code only	Formatted text with code chunks
Notes and explanation	Comments that start with #	Ordinary text, headings, and lists
Produces	Results in the Console and Plots pane	A finished document in HTML, PDF, or Word
Best for	Short tasks and reusable code	Reports, labs, and assignments

An **R script** is only code. Anything in it that is not code has to be written as a comment. An **R Markdown** file mixes formatted text and code, so the explanation of an analysis lives right next to the analysis itself. It is easier to read and easier to share. All assignments in this course are R Markdown files, so you will write both kinds of content: prose in Markdown and code inside chunks.

2.3.1 What is R Markdown?

R Markdown is a file format for dynamic, reproducible documents. An `.Rmd` file combines text written in **Markdown**, a simple way of formatting plain text, with chunks of R code. When you **knit** the file, R runs every code chunk and places the results (numbers, tables, and plots) into a finished document, as HTML, PDF, or Word (Figure 2.3).

The key idea is that the document and the analysis are the same file. If the data change, you knit again, and every number and figure in the report updates with them. Nothing has to be copied and pasted by hand, so nothing can fall out of date. In this course we mostly knit to HTML, because it is fastest and needs no extra software.

> **R Markdown and Quarto**
>
> This book is written in **Quarto**, the newer successor to R Markdown, made by the same developers. Quarto files end in `.qmd` instead of `.Rmd`, but the YAML header, Markdown text, and code chunks work the same way. Everything in this chapter applies to both.

2.3.2 Anatomy of an R Markdown file

Every R Markdown file has three kinds of content: a YAML header, Markdown text, and code chunks.

The YAML header sits at the very top of the file, between two lines of exactly three dashes. It holds **metadata**, which is data about data. Here it describes the document rather than the analysis: its title, author, date, and output format.

```
---
title: "My First R Markdown Document"
author: "Jane Doe"
date: "August 14, 2025"
output: html_document
---
```

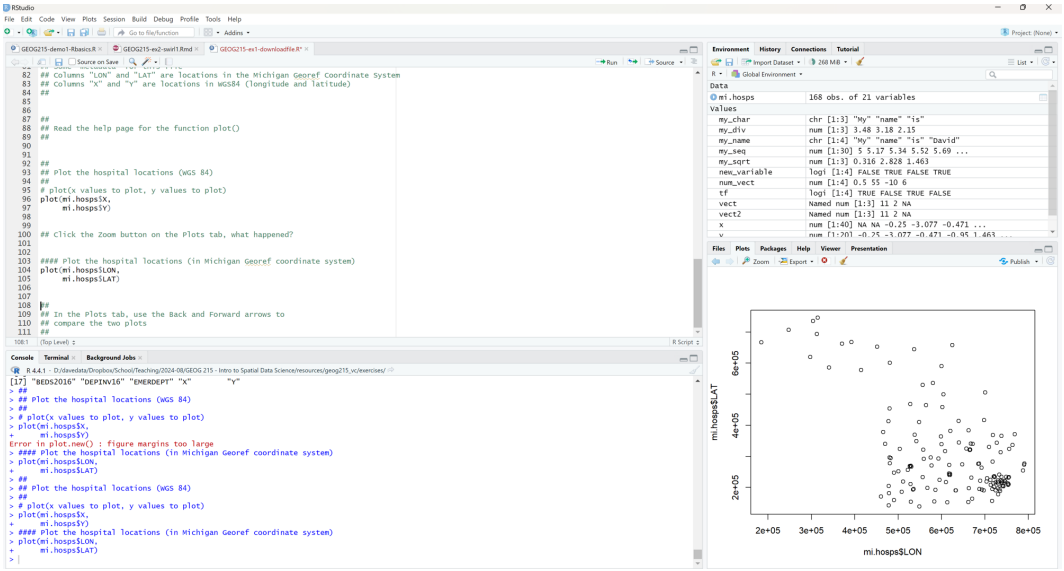


Figure 2.2: RStudio during a working session. Code in the Editor (top left) has been run, the variables it created are listed in the Environment (top right), output and an error message appear in the Console (bottom left), and the resulting scatterplot is in the Plots pane (bottom right).

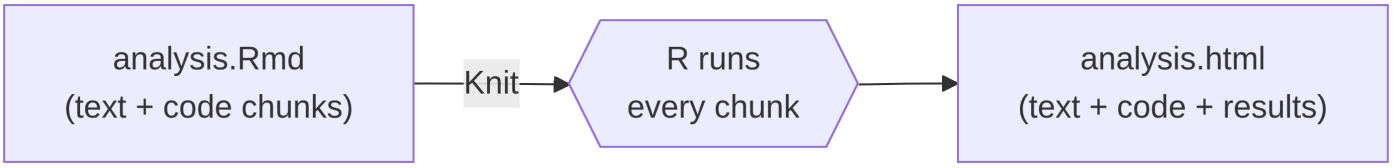


Figure 2.3: Knitting an R Markdown file. R runs every code chunk from top to bottom and combines your text with the code's results in a finished report.



R reads the header to know what to put at the top of the report and what kind of document to produce. YAML is picky: the dashes must be exactly three, each line needs a space after the colon, and indentation matters.

Markdown text is plain text with a few simple formatting marks. A line that starts with # becomes a heading (## a smaller heading, ### smaller still). Wrapping a word in single asterisks makes it *italic*, and double asterisks make it **bold**. A line that starts with - becomes an item in a bulleted list.

Code chunks hold R code. A chunk starts with three backticks followed by `{r}`, and ends with three backticks:

````
```{r}
1 + 2 + 3
```
````

The `{r}` tells R Markdown that the chunk contains R code. Click the small green "play" arrow at the top right of a chunk to run just that chunk. When the file is knit, the result prints directly below the chunk in the report:

```
1 + 2 + 3
#> [1] 6
```

The `[1]` in front of the answer is R's way of saying "this is the first element of the result." You will see it in front of almost every result R prints, and you can ignore it for now.

Putting the three pieces together, a complete (very short) R Markdown file looks like this:

````
---
title: "Lab 1"
author: "Jane Doe"
output: html_document
---

## How many months are in a century?

There are 12 months in a year and 100 years in a century.

```{r}
# Months in a century
months <- 12
years <- 100
months * years
```

So a century has **1,200** months.
````

2.3.3 The R Markdown workflow

You will follow the same four steps for every assignment.

1. **Create a file.** In RStudio, choose **File > New File > R Markdown...**, give the document a title, and click OK. RStudio opens a new file that already contains a YAML header and some example content.

2. **Edit the document.** Write your narrative in Markdown, and insert code chunks where you need them (**Code > Insert Chunk**, or Ctrl+Alt+I; Cmd+Option+I on a Mac).
3. **Run code chunks** as you go, by clicking each chunk's green play arrow, to check that each one does what you expect.
4. **Knit the document.** Click **Knit** in the Editor toolbar. R builds an HTML report with all of your text, code, output, and plots, and opens it in a preview window.

Knitting does something important that is easy to miss: it reruns *all* of your code, from the top, in a fresh R session that knows nothing about what you have already run. A file that knits successfully is therefore a file that anyone can reproduce from scratch. If it fails to knit, the error message names the chunk that broke.

2.3.4 Keep all course files in one folder

Most of the errors students run into early in the semester are not really coding errors. They are file path errors: R cannot find a data file because the `.Rmd` file and the data are saved in different places. When you knit, R looks for files **relative to the folder where the `.Rmd` file is saved**, so a tidy folder structure prevents a whole category of "file not found" errors.

Create one folder for the class, called `geog215`, and save all of your work there, with subfolders for data, labs, and notes:

```
geog215/
├── data/
├── labs/
│   └── lab01.Rmd
└── notes/
```

Use short file names with no spaces, such as `lab01.Rmd`. Spaces and special characters in file names cause trouble in code, and a consistent naming pattern makes files easy to find.

2.4 Code basics

The rest of this chapter introduces five ideas that appear in every R script: lines, blocks, variables, comments, and functions. A cooking analogy runs through all of them (Table 2.3).

Table 2.3: Code ideas and their cooking analogies.

Code	Cooking analogy
A line of code	One step in a recipe
A block of code	A recipe: several steps that make one dish
A program	A full meal: several recipes combined
A function	An appliance: ingredients go in, a dish comes out

The code boxes in the rest of this chapter are live. Click **Run Code** to run the R inside them, right here in your browser. You can also edit the code and run it again, which is the best way to learn what each piece does. The first time you run a box, R takes a few seconds to load.



2.4.1 A line of code

Every program starts with one line of code, and every line does something. When you run a line, R **evaluates** it and returns the result. Try it:

```
5 + 6
```

R returns `11`, again with `[1]` in front to mark the first element of the result. A single line is like a single step in a recipe: "chop the onion," or "add 5 and 6."

Sometimes a line produces no visible result. Storing a value in a variable, which you will see below, prints nothing at all. The line still did something, though. It changed what R is holding in memory.

2.4.2 A block of code

A **block** of code is a sequence of lines that work together on one related task, the way the steps of a recipe work together to make one dish. In an R Markdown file, a block usually lives in one code chunk. Here is a block that answers a question: how many months are in a century?

```
# How many months are in a century?
months <- 12
years <- 100
months * years
```

The block stores 12 in `months`, stores `100` in `years`, multiplies them, and prints the answer, `1200`. Its first line starts with #, which makes it a comment, covered below.

2.4.3 Variables

A **variable** is a named piece of memory that holds data. Once a value is stored in a variable, you can reuse it by name instead of typing it again.

You store a value with the **assignment operator**, `<-`, which looks like an arrow pointing left. The value on the right is stored in the name on the left. Read it out loud as "gets": `months <- 12` is "months gets 12." In RStudio, the shortcut Alt+- (Option+- on a Mac) types the arrow for you.

Variable names are **case-sensitive**: `Months` and `months` are two different variables. Run this box, which asks for `Months` with a capital M:

```
months <- 12
Months
```

R replies with an error, `object 'Months' not found`, because no variable has exactly that name. This is one of the most common errors you will see, and now you know how to read it. Change `Months` to `months` and run the box again.

When you run code in RStudio, each new variable appears in the Environment pane. Checking that pane after you run a line is a good way to confirm that the line did what you meant.

2.4.4 Comments

A **comment** is a note for people, not for R. A comment begins with #, and R ignores everything after the # on that line. In the century example above, R skipped `# How many months are in a century?`, but a reader knows straight away what the block is for.

Use comments to explain *why* your code does what it does, not just *what* it does. The code itself already says that `months * years` multiplies two numbers. A good comment says why you needed that number. When you return to an assignment weeks later, comments are what let you understand your own reasoning, and in this course they are also how your instructor sees it.

2.4.5 Functions

A **function** takes inputs, does a job, and returns an output. The inputs are called **arguments**, and they go inside parentheses after the function's name. If variables are the nouns of R, functions are the verbs.

In the cooking analogy, a function is an appliance, such as an oven. To make a pizza, you put bread, cheese, and tomato sauce into the oven, and the oven returns a pizza (Figure 2.4).

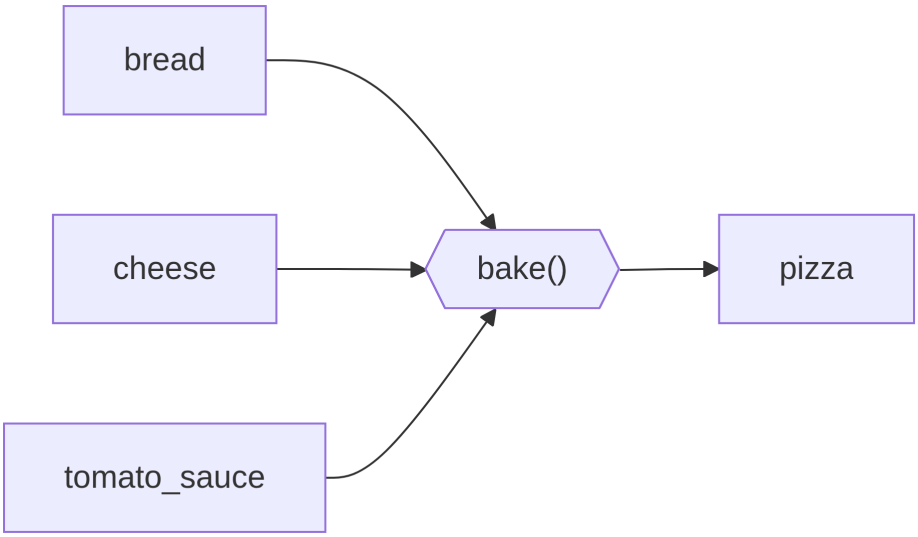


Figure 2.4: A function takes inputs (arguments), does a job, and returns an output. In R this would be written pizza <- bake(bread, cheese, tomato_sauce).

In R, that would be written:

```
pizza <- bake(bread, cheese, tomato_sauce)
```

Here `bake()` is the function; `bread`, `cheese`, and `tomato_sauce` are the arguments; and the result is stored in a new variable, `pizza`. (`bake()` isn't a real R function, but the pattern is exactly the one every real function follows.)

R comes with hundreds of real functions, and packages add thousands more. Run this box to try a few:

```
# Five daily high temperatures, in degrees F
temps <- c(71, 75, 68, 80, 77)
```



```
mean(temps)   # the average
max(temps)    # the highest value
round((mean(temps) - 32) * 5 / 9, digits = 1)   # the average in degrees C
```

`c()` is itself a function: it **combines** values into a **vector**, a sequence of values that R stores together. `mean()` and `max()` each take a vector and return one number. The last line converts the average to degrees Celsius. It **nests** one function inside another, so `mean()` runs first and `round()` rounds its result, and it shows a **named argument**, `digits = 1`, which tells `round()` how many decimal places to keep. Every function has a help page that lists its arguments; type `?round` in the Console to see one in the Help pane.

2.4.6 Chaining functions with the pipe

Nested functions like the last line above are read from the inside out, which gets hard to follow as they grow. The **pipe** operator, `|>`, lets you write the same steps from left to right instead. The pipe takes the result on its left and passes it as the first argument to the function on its right:

```
temps |>
  max() |>
  sqrt() |>
  round(digits = 1)
```

This gives the same answer as `round(sqrt(max(temps)), digits = 1)`, but reads in the order the steps happen. Read the pipe out loud as "and then": take `temps`, *and then* find the maximum, *and then* take its square root, *and then* round it to one decimal place. You will use the pipe constantly in later chapters to string together steps that clean, analyze, and map data, and it is required in the course's assignments.

2.4.7 A program

A **program** is a set of lines and blocks of code that work together to do a task, the way several recipes combine into a full meal. Every assignment in this course is a program: an R Markdown file that loads data, analyzes it, and makes a map.

R runs a program from top to bottom, so **order matters**. A variable must be created before it is used. When you work interactively, it is easy to run chunks out of order, or to delete a line that created a variable that is still sitting in your Environment. Everything seems to work until you knit, and then the fresh session that knitting starts in reports `object not found`. That error is doing you a favor: it means your file, as written, would not have worked for anyone else either.

2.5 Some hints for coding

Two habits make the biggest difference when you start writing code.

Think about the syntax. R is exact. Spelling, capital letters, commas, quotation marks, and brackets all have to be right, and every opening parenthesis needs a closing one. A single missing comma is enough to stop a program. When something fails, check the syntax of the line the error points to first.



Think about the purpose. Before you type a line, say in plain words what it should do: "store the number of years," "find the average temperature." If you can't say it, you aren't ready to write it yet. If you can, the words make a good comment.

And leave lots of comments. They are notes to **Future You**, who will not remember, three weeks from now, why you wrote that line.

2.6 Setting up R and RStudio

Before the first lab, set up your computer:

1. **Install R** from CRAN, https://cran.r-project.org. Choose the download for your operating system (Windows, macOS, or Linux) and run the installer with its default settings.
2. **Install RStudio Desktop** from Posit, https://posit.co/download/rstudio-desktop/. Remember the order: R first, then RStudio.
3. **Create your geog215 folder**, with `data`, `labs`, and `notes` subfolders, somewhere you will find it easily.
4. **Test everything.** Open RStudio, create a new R Markdown file (**File > New File > R Markdown...**), save it in `geog215/labs`, and click **Knit**. If an HTML report appears, you are ready.

2.7 Check your understanding

1. Which program does the actual computing when you run code: R or RStudio? [1]

- (A) R
- (B) RStudio

2. In which RStudio pane would you look to see the variables currently in memory? [2]

- (A) Editor
- (B) Console
- (C) Environment
- (D) Files

3. What file extension does an R Markdown document have? ___ [3]
4. After running `months <- 12` and `years <- 100`, what does `months * years / 10` return? __ [4]
5. Which character starts a comment in R? [5]

- (A) //

[1] R
[2] Environment
[3] .Rmd, Rmd
[4] 120
[5] #

- (B)



3

- (C) -
- (D) %

6. In `round(3.14159, digits = 2)`, what is `digits = 2`? [1]

- (A) A variable
- (B) A comment
- (C) A named argument
- (D) A function

7. True or false: knitting runs your code in a fresh R session, so a variable you created only in the Console will not exist. TRUE / FALSE[2]

Show hint for question 4

`months * years` is 1200. Dividing by 10 gives the answer.

3.1 Summary

- **R** is the language that does the computing. **RStudio** is the workspace where you write and run R. **R Markdown** is the document format that combines your writing with your code and its results.
- RStudio has four panes: the **Editor** for writing and saving code, the **Console** for running commands and reading output, the **Environment** for the variables in memory, and **Files, Plots, and Help**.
- An R Markdown file has a **YAML header**, **Markdown text**, and **code chunks**. **Knitting** reruns every chunk from the top in a fresh session and builds a finished report, which is what makes the work reproducible.
- A **line** of code does one thing; a **block** of lines does one task; a **program** combines blocks to do a whole job, and runs from top to bottom.
- **Variables** store values with `<-` and are case-sensitive. **Comments** start with # and are notes for people. **Functions** take arguments and return an output, and the pipe `|>` chains them from left to right.
- Keep all course files in one folder, mind your syntax, and comment generously.

[1] A named argument
[2] TRUE

3.2 Exercises

1. **Your first R Markdown file.** Create a new R Markdown file titled "Exercise 1" with your name as the author. Delete the example content below the YAML header, add a heading, and write one sentence about why you are taking this course. Knit it to HTML.
2. **Arithmetic.** In a code chunk, calculate how many hours are in a year, and how many seconds. Store each intermediate value (days, hours per day, and so on) in a variable with a clear name, and add a comment to each line.
3. **Read the error.** Run each line below, one at a time, in the Console, and explain in your own words what is wrong. Two of them give an error message. The first gives a + prompt instead: what is R waiting for? (Press Esc to get back to the normal > prompt.)

```
mean(c(1, 2, 3)
Mean(c(1, 2, 3))
mean(c(1, 2 3))
```

4. **Functions and help.** Look up the help page for `seq()` with `?seq`. Use it to create the sequence 0, 5, 10, ..., 100, and store it in a variable. Then use `length()`, `sum()`, and `mean()` on your variable.
5. **Nesting versus piping.** Rewrite `round(sqrt(sum(c(4, 9, 16))), 2)` using the pipe `|>`. Check that both versions give the same result.
6. **Order matters.** Write a chunk that uses a variable before it is created, and knit the file. What happens? Fix the chunk so that the file knits.

3.3 Further reading

- *R for Data Science* by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund is free online at https://r4ds.hadley.nz. Its early chapters cover the RStudio workflow, the basics of R code, and the pipe in more depth.
- *R Markdown: The Definitive Guide* by Yihui Xie, J. J. Allaire, and Garrett Grolemund, free at https://bookdown.org/yihui/rmarkdown/, is the full reference for R Markdown.
- In RStudio, **Help > Cheat Sheets** links to one-page summaries of the RStudio IDE and R Markdown that are worth printing and keeping by your computer.

4 R Basics

Learning objectives

By the end of this chapter you should be able to:

1. Explain how R runs code one line at a time, and why the order you run code in matters.
2. Install a package once and load it in every session.
3. Use R's four basic data types: logical, integer, numeric, and character.
4. Name variables clearly and consistently.
5. Find help when your code breaks.

The last chapter introduced R, RStudio, and R Markdown, and the building blocks of a line of code. This chapter moves from reading code to writing it. It covers four things you will rely on in every assignment: how R actually runs your code, how to add new abilities to R with packages, how R stores different kinds of values, and what to do when something goes wrong.

None of it is spatial yet. But every map and every analysis later in the book is built from these pieces, and most of the problems students hit in the first weeks of the course come from one of them: chunks run out of order, a package that was never loaded, a number stored as text, or an error message nobody read.

4.1 How R runs your code

4.1.1 R is an interpreted language

R is an **interpreted language**. An **interpreter** is a program that executes instructions directly, one line at a time: it reads a line of code, translates it into something the computer can act on, runs it, and then moves on to the next line.

Other languages, such as C or Java, use a **compiler** instead. A compiler translates the whole program into machine code before any of it runs. A good analogy is translation at a conference. An interpreter is like a live translator who translates the speaker sentence by sentence, as the speech happens. A compiler is like a book translator, who translates the whole book before anyone reads a page.

Being interpreted is what makes R so good for exploring data. You can run a single line, look at the result, and decide what to do next, without building and running a whole program. That is exactly what you do in RStudio every time you press Ctrl+Enter.

4.1.2 Order matters

The flip side is that R only knows what you have run so far, *in the order you ran it*. Nothing stops you from running the chunks of an R Markdown file out of order, and when you do, the same code can leave a different result.

Here are three chunks. Each one stores a different number in a variable called `day`.

```
day <- 1
```

```
day <- 2
```

```
day <- 3
```

And here is a chunk that prints whatever `day` holds right now:

```
day
```

Run the three chunks top to bottom, then run the last one: `day` is 3. Now run the second chunk again, and then the last one: `day` is 2. The code in the file hasn't changed, but the value in memory has, because R keeps whichever value it was given most recently. Before you click, try to predict what `day` will be.

In RStudio, the **Environment** pane always shows the current value of every variable, so it is the place to check what R is holding at any moment.

4.1.3 Knitting runs everything again

Out-of-order code is a quiet problem while you work, and a loud one when you knit. As the last chapter explained, knitting starts a fresh R session and runs every chunk from the top of the file to the bottom. If your file only worked because you ran chunk 5 before chunk 3, or because a variable you made in the Console is still sitting in the Environment, the knit fails. This is the most common reason an assignment won't knit.

4.1.4 Rule one: think sequentially

The fix is a habit, and it is the first rule of this course: **think sequentially.**

- Run your code in order, top to bottom, at least while you are learning.
- Check the Environment pane to see which variables exist and what they hold.
- Every so often, choose **Run > Restart R and Run All Chunks** (Figure 4.1). This clears memory and reruns the whole file from the top, the same way knitting does. If it works, your file will knit.

4.2 Packages and libraries

4.2.1 What is a package?

Base R, the R you get when you install it, can do arithmetic, statistics, and basic plots. Almost everything else reaches you through **packages**. A package bundles functions, data, and

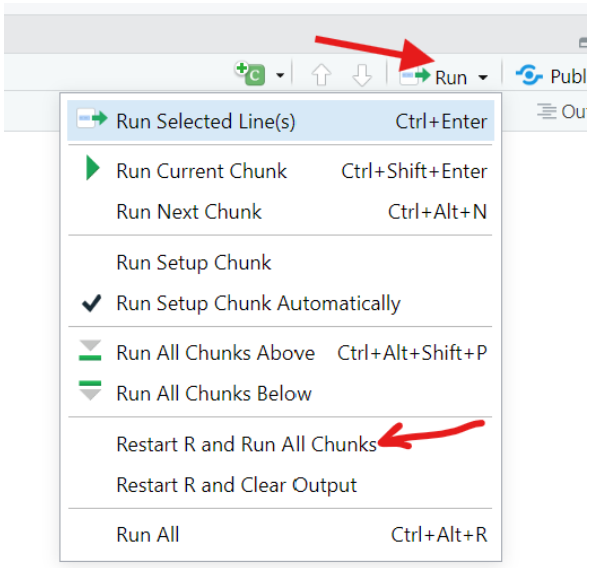


Figure 4.1: The Run menu at the top of the RStudio Editor. Restart R and Run All Chunks reruns the whole file in a fresh session, as knitting does.

documentation that someone else has written and tested, and that you can add to your own copy of R. A useful comparison is a car: base R is the engine, and packages are the add-ons, like GPS or a sunroof. In this course you will use the **tidyverse** for cleaning data, **sf** for spatial data, and **tmap** for maps, all of them packages.

Packages are worth using for four reasons:

- **They save you from reinventing the wheel.** Why write your own code for a complex task when an expert has already written, tested, and documented a function for it?
- **They give you specialized tools** for particular fields, from spatial analysis to genomics to finance.
- **They are often faster** than code you would write yourself, because they are optimized for speed and memory.
- **They make your work reproducible.** Anyone who installs the same packages can rerun your analysis.

4.2.2 Packages versus libraries

The words *package* and *library* are often used as if they meant the same thing, but there is a difference:

- A **package** is the bundle itself: the collection of code, data, and documentation.
- A **library** is the folder on your computer where installed packages are stored. You can have more than one.

You *install* a package *into* a library. Then, confusingly, you load it with a function called `library()`. In casual conversation people say "the tidyverse library," and that's fine, but it helps to know which is which.

4.2.3 CRAN

Most packages come from **CRAN**, the Comprehensive R Archive Network. CRAN is the official repository for R packages: think of it as R's app store. It hosts thousands of packages, and each one passes a set of quality checks before it is published. You may later meet other sources, such as Bioconductor for genomics packages or GitHub for development versions, but everything in this course comes from CRAN.

4.2.4 Install once, load every session

Using a package takes two steps, and the difference between them matters (Table 4.1).

Table 4.1: Installing a package compared with loading it.

	Install	Load
What it does	Downloads the package into your library	Makes the package's functions available right now
How often	**Once** per computer (and again to update)	**Every** R session
How	RStudio's Packages tab, or `install.packages("tidyverse")`	`library(tidyverse)`
Quotes?	Yes, around the package name	No
Belongs in your `.Rmd` file?	**No**	**Yes**, at the top

Installing downloads the package from CRAN and saves it in your library. You can do it with code, `install.packages("tidyverse")`, but in this course please use RStudio instead: open the **Packages** tab in the bottom-right pane, click **Install**, type the package name, and click **Install** again (Figure 4.2). Either way, you only need to do it once on each computer.

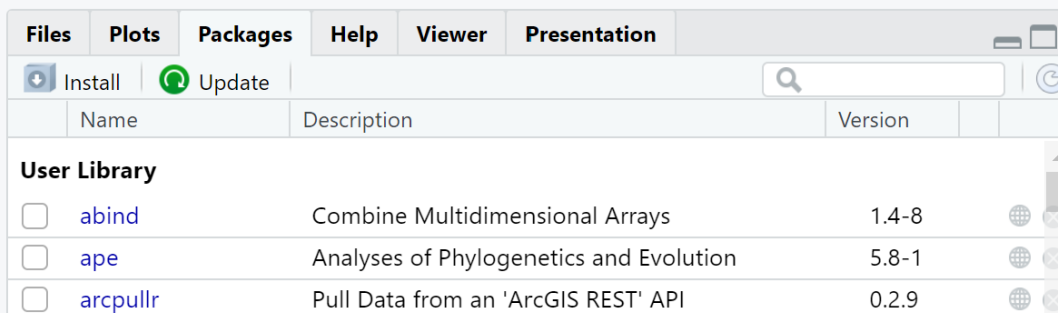


Figure 4.2: The Packages tab in RStudio's bottom-right pane. Click Install to download a new package from CRAN. The list below shows the packages already installed in your library.

Never leave `install.packages()` in an R Markdown file. Knitting runs every line, so the file would download and reinstall the package every single time you knit.

Loading makes an installed package's functions available in the current session. Put a `library()` call for each package you need at the top of every R Markdown file:

```
library(tidyverse)
```



You need to load packages every time, because every R session, including the fresh one that knitting starts, begins with no packages loaded. Notice that `library()` doesn't need quotation marks around the package name, while `install.packages()` does. That small difference trips up a lot of people.

If a file calls `library()` for a package that isn't installed, R stops with an error like `there is no package called 'tidyverse'`. RStudio also tries to help: when you open a file that uses a package you don't have, a yellow bar at the top of the Editor offers to install it for you.

4.3 Data types

Before you can analyze data, you need to know how R stores it. This section covers variables, vectors, and the four basic kinds of values R works with.

4.3.1 Variables, again

Recall from the last chapter that a **variable** is a named piece of data in memory, and that the **assignment operator** `<-` puts the value on its right into the name on its left:

```
# a character (text) variable
myName <- "Harry Potter"
```

Read it out loud as "`myName` gets `"Harry Potter"`." In R we (usually) assign from right to left. The name always goes on the left of the arrow and the value on the right. `"Harry Potter" -> myName` also works, but it is easy to misread, so don't use it.

4.3.2 Vectors

The most basic data structure in R is the **vector**: a sequence of values that all have the **same data type**. You make one with `c()`, short for *combine*:

```
temps <- c(71, 75, 68, 80, 77)          # numbers
cities <- c("Chapel Hill", "Durham")    # text
isRaining <- c(TRUE, FALSE, FALSE)      # TRUE or FALSE

temps
cities
isRaining
```

Even a single value is a vector, with a length of one. That is why R prints `[1]` in front of `5 + 6`: the answer is the first (and only) element of a vector. It is also why so many R functions work on whole vectors at once. `temps * 2` doubles every temperature, with no loop required.

4.3.3 The four atomic data types

R has four main **atomic data types**, the basic kinds of values a vector can hold (Table 4.2).



Table 4.2: R's four main atomic data types.

Type	What it stores	Example
logical	TRUE or FALSE	`isHot <- TRUE`
integer	Whole numbers	`myAge <- 21L`
numeric	Numbers that can have decimals	`myAccount <- 103.21`
character	Text, always in quotation marks	`myName <- "Harry Potter"`

- **Logical** values are TRUE and FALSE, written in capitals with no quotation marks. They are the answers to yes-or-no questions such as "is it raining?" or "is this county's income above the median?" The next chapter uses them to pick out data.
- **Integer** values are whole numbers. One detail surprises people: R stores a number you type, such as `21`, as *numeric* unless you add an `L` to the end, as in `21L`. In practice you rarely need to care; R converts between integers and numerics for you.
- **Numeric** values (also called *doubles*) are numbers that can have a decimal part, such as temperatures, incomes, or coordinates. Most numbers you work with will be numeric.
- **Character** values are text, also called **strings**. They always go inside quotation marks, which is how R tells the text `"Durham"` apart from a variable named `Durham`.

The function `class()` tells you what type a value is. Run this box, then try changing the values:

```
class(TRUE)
class(21L)
class(21)
class(103.21)
class("Harry Potter")
```

4.3.4 One type per vector

Because a vector holds values of a single type, mixing types forces R to convert some of them. This conversion is called **coercion**, and R picks the type that can hold everything without losing information. Text can hold anything, so if a vector contains even one character value, the whole vector becomes character:

```
mixed <- c(71, 75, "68", 80)
mixed
class(mixed)
mean(mixed)
```

The numbers have all turned into text (notice the quotation marks), and `mean()` can no longer average them. It returns NA with a warning. This is a very common source of trouble with real data: one stray letter or symbol in a column of numbers, such as `"N/A"` or `"12,000"`, turns the entire column into text. When a calculation gives an odd result, `class()` is one of the first things to check.

4.3.5 Naming variables

Variable names can't contain spaces, so names made of several words need a convention. Two are common (Table 4.3).

Table 4.3: Two conventions for naming variables.

Convention	How it works	Examples
camelCase	Join the words; capitalize every word after the first	`whatIsTheTemp`, `popChapelHill`
snake_case	Lowercase words joined by underscores	`what_is_the_temp`, `pop_chapel_hill`

The tidyverse's style guide uses snake_case, and so do most of the packages you will use. Either is fine in this course, as long as you **pick one and use it consistently**. Remember that names are case-sensitive, so `popChapelHill` and `popchapelhill` are different variables, and a file that mixes conventions is an easy place for that kind of typo to hide.

Whichever convention you use, choose names that say what the variable holds. `popChapelHill` tells a reader something; `x` or `data2` does not. Short abbreviations, such as `popCH`, are fine if they are clear in context.

4.4 Debugging and getting help

4.4.1 Mistakes are a fact of life

Everyone's code breaks, including the code of people who have programmed for decades. A helpful way to think about errors is as puzzles to solve, not failures. Each one is a chance to understand the code, the language, and the problem a little better, so the time you spend **debugging** is an investment in your skills, not time wasted. If you get stuck, that is part of learning to program, and it gets easier with practice.

When something breaks, work through it in order:

1. **Read the error message**, word by word. It usually says what went wrong, and often names the object or line involved.
2. **Find the line** that caused it. In a knit, the error names the chunk.
3. **Make a guess** about the cause, change one thing, and run the code again.

4.4.2 The most common mistakes

Three mistakes account for most of the problems students run into early in the course. Check for these first.

1. **Not programming sequentially**, and then being surprised when the file won't knit (Section 4.1).
2. **Not storing files consistently**, for example working straight out of the Downloads folder, so that R can't find the data.
3. **Missing parentheses, commas, and quotation marks.** One missing `)` or `"` is enough to stop a program.

The second one has a simple fix. For now, save all of the files for an assignment in the same folder, inside your `geog215` folder. When you knit, the HTML file appears in that folder; when you download data, save it there too. Then R will find everything. Don't use the Desktop or Downloads folders as a place to work.

4.4.3 Getting help with a function

If you aren't sure what a function does, or what arguments it takes, open its **help page**. Either of these works:

```
help(mean)
?mean
```

The help page opens in the Help tab of the bottom-right pane (Figure 4.3). Every help page has the same layout: a **Description** of what the function does, its **Usage** (how to call it, with each argument's default value), a list of **Arguments**, and, at the bottom, **Examples** you can copy and run. The examples are often the quickest way to understand a function.



Figure 4.3: A help page in RStudio's Help tab, for the function download.file(). Every help page has the same sections: Description, Usage, Arguments, and Examples further down.

4.4.4 Getting help with an error

If you don't understand an error message, copy it into a search engine. Someone has almost always run into the same error before, and the answer is often on Stack Overflow or the Posit Community forum. Leave out the parts that are specific to you, such as your variable names, so the search matches other people's posts.

4.4.5 Using AI in this course

You may use AI tools such as Copilot to help *explain* an error message. For example, given the line `my_age -> 39`, an AI tool can point out that the assignment arrow is backwards and that you meant `my_age <- 39`. Use the explanation to understand the fix, then make the change yourself.

Don't use AI to *write* your code. Generated code is often buggy, it often doesn't follow the course's coding standards (code that doesn't won't receive credit), and, most importantly, the point is for you to learn to write R yourself.

4.4.6 Getting help from people

In GEOG 215 there are three ways to get help from the teaching team:

- **Post in the Help! channel** in the course's Microsoft Teams team. The TAs and the instructor check it daily, and your classmates can learn from the answer too.
- **Come to office hours**, in person or online, with the instructor or a TA.
- **Email the instructor** for anything private, such as grades or accommodations.

When you ask, include the code you ran and the full error message. "My code doesn't work" is hard to answer; "`library(tidyverse)` gives `there is no package called 'tidyverse'`" can be answered in one line.

4.5 Check your understanding

1. An R Markdown file has three chunks: `day <- 1`, `day <- 2`, and `day <- 3`. You run the third chunk, then the first. What does `day` hold now? _[1]
2. How often do you need to install a package on your computer? [2]

- (A) Every R session
- (B) Every time you knit
- (C) Once
- (D) Every time you use one of its functions

3. Which line belongs at the top of an R Markdown file? [3]

- (A) install.packages("tidyverse")
- (B) library(tidyverse)

4. A *library* is the folder on your computer where installed packages are stored. TRUE / FALSE[4]
5. What does `class("103.21")` return? [5]

- (A) numeric
- (B) integer
- (C) character
- (D) logical

6. What type is the vector `c(1, 2, "three")`? _____[6]

[1] 1
[2] Once
[3] library(tidyverse)
[4] TRUE
[5] character
[6] character

7. Which variable name follows the snake_case convention? [7]

- (A) popChapelHill
- (B) pop_chapel_hill
- (C) Pop Chapel Hill
- (D) PopChapelHill

8. Which command opens the help page for the function `round()`? [8]

- (A) help.round()
- (B) round(help)
- (C) ?round
- (D) #round

Show hint for question 1

R keeps whichever value it was given most recently. The order you *ran* the chunks matters, not their order in the file.

Show hint for question 6

A vector holds one type. Text can hold anything, so one character value turns the whole vector into character.

4.6 Summary

- R is an **interpreted language**: it runs one line at a time, in the order you run it. The same chunks run in a different order can leave different values in memory.
- **Knitting** reruns every chunk from the top in a fresh session. Think sequentially, check the Environment pane, and use **Restart R and Run All Chunks** to make sure your file will knit.
- A **package** is a bundle of code written by others; a **library** is the folder where installed packages live. Most packages come from **CRAN**.
- **Install** a package once per computer, with the Packages tab. **Load** it in every session with `library()`, at the top of every R Markdown file. Never put `install.packages()` in an `.Rmd` file.
- A **vector** is a sequence of values of one type, made with `c()`. R's four atomic types are **logical**, **integer**, **numeric**, and **character**; `class()` tells you which one you have. Mixing types **coerces** the whole vector, usually to character.
- Name variables in **camelCase** or **snake_case**, consistently, with names that say what they hold.
- Errors are normal. Read the message, find the line, and check the common mistakes first. For help, use `?function`, search the error, and ask on Teams or in office hours. Use AI to explain errors, not to write your code.

[7] pop_chapel_hill
[8] ?round

4.7 Exercises

1. **Order matters.** Make a new R Markdown file with three chunks: `x <- 10`, `x <- x * 2`, and `x`. Run them in order and note the result. Now run the second chunk two more times, then the third. What is `x`, and why? What will `x` be after you knit?
2. **Install and load.** Using the Packages tab, install the `nycflights13` package, which you will use in a later chapter. Then load it in a chunk at the top of your file and knit. What happens if you remove the `library()` line and run `head(flights)`?
3. **Data types.** Create one variable of each atomic type, describing yourself or your hometown (for example, the town's name, its population, its area in square miles, and whether you live there now). Use `class()` on each one. Did the population come out as integer or numeric? Why?
4. **Coercion.** Predict the type of each vector below, then check with `class()`. Which result surprised you?

```
c(1, 2, 3.5)
c(TRUE, FALSE, 1)
c(1, "2", 3)
c(TRUE, "FALSE")
```

5. **Naming.** Rewrite these names in snake_case and in camelCase: `median household income`, `Number of Counties`, `pop 2020`. Which convention will you use this semester?
6. **Read a help page.** Open the help page for `seq()`. What do its `from`, `to`, and `by` arguments do? Use the Examples section to make the sequence 2, 4, 6, ..., 20.
7. **Debug.** Each line below has one mistake. Find and fix it, and say which of the "most common mistakes" it is.

```
my_age -> 39
myCities <- c("Raleigh", "Durham, "Chapel Hill")
round(mean(c(4, 8, 15), digits = 1)
library("tidyverse
```

4.8 Further reading

- *R for Data Science* (2nd ed.) by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund, free at https://r4ds.hadley.nz. The chapters "Workflow: basics" and "Workflow: getting help" cover the same ground as this chapter.
- The *tidyverse style guide*, https://style.tidyverse.org, explains snake_case and many other conventions for writing readable R code.
- *What They Forgot to Teach You About R* by Jenny Bryan and Jim Hester, https://rstats.wtf, has practical advice on project folders and working sequentially.



5 Indexing Vectors and Data Frames

Learning objectives

By the end of this chapter you should be able to:

1. Index a vector by position, by a logical condition, and by name.
2. Write comparisons and combine them with the logical operators &, |, and !.
3. Create a data frame and pull out its rows, columns, and values.
4. Read a table of data from a CSV file into R.

So far you have created values and stored them in variables. Most of the work of data analysis, though, is not creating data but *picking out* the part of it you need: the counties with incomes above the state median, the flights that left LaGuardia in June, the census tracts inside one city. Selecting part of a vector or table is called **indexing**, and it is the subject of this chapter.

We start with vectors, which you met in the last chapter, and three ways to index them: by position, by a logical condition, and by name. Then we stack vectors side by side into a data frame, R's table, and finish by reading a table from a file.

Warm-up: what's wrong with this code?

```
18 <- myAge
```

Find the bug before you read on. (The answer is at the end of Section 5.2.)

5.1 A vector to work with

Most of the examples in this chapter use one vector, called `vect`, which holds ten numbers between 1 and 1000:

```
vect <- c(1, 2, 5, 10, 25, 50, 100, 200, 500, 1000)
vect
```

Each element of a vector has a position, called its **index**. Table 5.1 shows each value above its index. Notice that R starts counting at **1**, not 0 as some other languages do: the first element is at index 1.

Table 5.1: The values in `vect` and their index positions.

Value	1	2	5	10	25	50	100	200	500	1000
Index	1	2	3	4	5	6	7	8	9	10

To index a vector, put **square brackets** after its name. What you put inside the brackets decides which elements come back, and there are three choices:



Table 5.2: Three ways to index a vector.

Index by	Example	Returns
Position (number)	`vect[c(1, 3)]`	The elements at positions 1 and 3
Logical condition	`vect[vect > 10]`	The elements where the condition is `TRUE`
Name	`vect["Fozzie"]`	The element labeled `"Fozzie"`

The live code boxes in this chapter share one R session, so run the box above first to create `vect`. As you read, try to predict each answer before you run the code.

5.2 Indexing by position

5.2.1 One element

Put a single number in the brackets to get the value at that position:

```
x <- vect[1]
x
```

`vect[1]` is 1, the value in position 1. What is `vect[3]`? Check Table 5.1, then run it:

```
vect[3]
```

The answer is 5, the value *at* position 3. A common mistake is to answer 3, which is the index rather than the value. Keep the two apart: the index says *where*, and the value is *what is there*.

An indexed value behaves like any other number, so you can do arithmetic with it. `vect[3]` is 5 and `vect[4]` is 10, so:

```
vect[3] * vect[4]
```

5.2.2 Leaving an element out

A **negative index** drops an element and returns all the others:

```
vect[-3]
```

This returns nine values: everything except the 5 in position 3. It does not change `vect` itself. Indexing always makes a new vector and leaves the original alone, unless you store the result back in the same name with `<-`.

5.2.3 Several elements

To get several elements at once, index with a vector of positions:

```
vect[c(1, 3, 5, 7, 9)]
```

This returns 1, 5, 25, 100, and 500, the values in positions 1, 3, 5, 7, and 9. The values come back in the order you ask for them, so `vect[c(9, 1)]` returns 500 and then 1.

For a run of positions in a row, the **colon operator** `:` makes the sequence for you. `6:8` is shorthand for `c(6, 7, 8)`:

```
6:8
vect[6:8]
```

> **Warm-up answer**
>
> `18 <- myAge` has the assignment arrow pointing the wrong way. It asks R to store `myAge` *in* the number 18, which isn't allowed, so R stops with the error `invalid (do_set) left-hand side to assignment`. The name goes on the left of `<-` and the value on the right: `myAge <- 18`.

5.3 Logical operations

5.3.1 Comparisons return TRUE or FALSE

TRUE and FALSE are special values in R, called **logical** or **Boolean** values (after the mathematician George Boole). Every **comparison operator** returns one of them (Table 5.3).

Table 5.3: Comparison operators in R.

Operator	Meaning	Example	Result
`>`	greater than	`5 > 5`	FALSE
`<`	less than	`5 < 5`	FALSE
`>=`	greater than or equal to	`5 >= 5`	TRUE
`<=`	less than or equal to	`5 <= 5`	TRUE
`==`	equal to	`5 == 5`	TRUE
`!=`	not equal to	`5 != 5`	FALSE

Notice that "equal to" is written with **two** equals signs, ==. A single = does something different: it sets an argument inside a function call, as in `round(x, digits = 1)`. Mixing them up is one of the most common errors in R, and you will see it again when you start filtering data.

```
5 >= 5
5 == 5
5 < 5
5 > 5
```

5.3.2 A comparison checks every element

Compare a whole vector with a number, and R checks every element, returning one TRUE or FALSE for each. The result is a **logical vector** the same length as the original:

```
vect > 10
```

The first four values (1, 2, 5, and 10) are not greater than 10, so they are FALSE; the other six are TRUE (Table 5.4).

Table 5.4: The result of `vect > 10` for each element (T is TRUE, F is FALSE).

Value	1	2	5	10	25	50	100	200	500	1000
vect > 10	F	F	F	F	T	T	T	T	T	T

5.3.3 Indexing with a condition

Now put that logical vector inside the brackets. R keeps the elements in the TRUE positions and drops the rest:

```
vect[vect > 10]
```

The result is 25, 50, 100, 200, 500, and 1000: every value greater than 10. Read `vect[vect > 10]` out loud as "the elements of `vect` where `vect` is greater than 10." This is the single most useful idea in the chapter. Selecting the counties whose income is above the median, or the flights that left late, works exactly the same way.

5.3.4 Logical vectors are recycled

What if the logical vector is shorter than the vector you are indexing? R **recycles** it, repeating it until it is long enough. Here `c(TRUE, FALSE)` becomes `TRUE, FALSE, TRUE, FALSE, ...`, ten values in all:

```
vect[c(TRUE, FALSE)]
```

The result is every other value, starting with the first: 1, 5, 25, 100, and 500. Recycling is handy now and then, but it can also hide mistakes, so make sure a logical index is the same length as the vector unless you mean it to be shorter.

5.3.5 Combining conditions: AND, OR, and NOT

Often you need two conditions at once: values greater than 10 *and* less than 500, say. R has three **logical operators** for combining TRUE and FALSE values.

AND (&) is TRUE only when **both** sides are TRUE. Is the statement "the sun rises in the west AND humans breathe oxygen" true? No: the first part is false, so the whole statement is false.

OR (|) is TRUE when **either** side is TRUE (or both). "The sun rises in the west OR humans breathe oxygen" is true, because the second part is. On most keyboards the vertical bar | is on the backslash key, above Enter.

Table 5.5 gives every combination. These are called *truth tables*.



Table 5.5: Truth tables for AND, written a & b in R, and OR, written with a vertical bar.

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

With AND, only TRUE & TRUE gives TRUE. With OR, only FALSE | FALSE gives FALSE.

Now back to the question: which values of `vect` are greater than 10 and less than 500?

```
vect[vect > 10 & vect < 500]
```

The answer is 25, 50, 100, and 200. 500 is left out because the test is *strictly* less than 500; use <= to include it. Now try the same conditions joined by OR:

```
vect[vect > 10 | vect < 500]
```

This returns all ten values, because every number is *either* greater than 10 *or* less than 500. It's a useful warning: a careless OR can end up filtering out nothing at all. When a filter returns more than you expected, check whether you meant &.

NOT (!) flips a logical value: !TRUE is FALSE, and !FALSE is TRUE. Combined with = it gives the "not equal to" operator !=, which is handy for removing particular values:

```
!TRUE
vect[vect != 2]
```

Only the element equal to 2 is dropped. You can also put ! in front of a whole condition in parentheses: `vect[!(vect > 10)]` returns the values that are *not* greater than 10.

5.4 Indexing by name

5.4.1 Naming the elements

The third way to index uses labels. The function `names()` attaches a name to each element of a vector. The vector of names must be the same length as the vector of data:

```
names(vect) <- c("Kermit", "Piggy", "Gonzo", "Fozzie", "Animal",
                 "DrTeeth", "Scooter", "Rowlf", "Chef", "Jim")
vect
```

The values haven't changed; each one now just has a label printed above it (Table 5.6).

Table 5.6: The values in `vect`, with their index positions and names.

Value	1	2	5	10	25	50	100	200	500	1000
Index	1	2	3	4	5	6	7	8	9	10
Name	Kermit	Piggy	Gonzo	Fozzie	Animal	DrTeeth	Scooter	Rowlf	Chef	Jim



5.4.2 Selecting by name

Put a name, in quotation marks, inside the brackets:

```
vect["Fozzie"]
vect[c("Kermit", "Scooter")]
```

`vect["Fozzie"]` returns 10, because Fozzie is the fourth name and position 4 holds 10. `vect[c("Kermit", "Scooter")]` returns 1 and 100, each with its name attached. Use straight quotation marks (`"`), not the curly ones a word processor inserts, or R won't recognize them.

Names make code easier to read, and they keep working if the order of the elements changes. `vect["Fozzie"]` still finds Fozzie's value even if someone sorts the vector; `vect[4]` would quietly return whatever has moved into position 4. This is the same reason you will refer to the columns of a data frame by name.

5.4.3 How long is a vector?

The function `length()` returns the number of elements in a vector:

```
length(vect)
```

`vect` has 10 elements. How would you get the *last* element of a vector, whatever its length? You could type `vect[10]`, but that only works while `vect` has exactly ten elements. Since `length()` gives the position of the last element, use it as the index:

```
vect[length(vect)]
```

This returns 1000 (with its name, Jim) for a vector of any length.

5.5 Data frames

5.5.1 From one dimension to two

A vector is one-dimensional: a single row of values. Stand several vectors of the same length side by side, and you get a two-dimensional table. In R, that table is a **data frame**, and it is how R stores almost every data set you will work with, including, later in the course, spatial data.

A data frame is organized into rows and columns (Table 5.7):

- Each **column** is a variable: something measured or recorded, such as a name or an exam score.
- Each **row** is an **observation**: one thing that was measured, such as a student, a county, or a flight.
- **All the values in a column must have the same data type.** Each column is a vector, and, as you saw in the last chapter, a vector holds just one type. Different columns can have different types.

Table 5.7: A small data frame. Each row is a student and each column is a variable.

name	exam1	exam2	bonus
Eliza	“A”	98.1	TRUE
Shonda	“A-”	85.4	TRUE
Me’shelle	“A”	74.3	FALSE
Jack	“A-”	92.2	TRUE

Here `name` and `exam1` are character columns, `exam2` is numeric, and `bonus` is logical. Suppose a number, such as 75.4, crept into the `exam1` column of letter grades. R would not keep a number in a character column: it would coerce the 75.4 into the text `"75.4"`. The same thing happens in reverse, and causes more trouble: a single letter or symbol in a column of numbers, such as `"N/A"` in `exam2`, turns the whole column into text, and then you can no longer average it.

5.5.2 Creating a data frame

The function `data.frame()` builds a data frame from vectors. Each argument becomes a column: the name before = is the column name, and the vector after = holds its values. Every vector must be the same length, here four:

```
# Exam results for four students in a small seminar
grades <- data.frame(
  name  = c("Eliza", "Shonda", "Me'shelle", "Jack"),
  exam1 = c("A", "A-", "A", "A-"),
  exam2 = c(98.1, 85.4, 74.3, 92.2),
  bonus = c(TRUE, TRUE, FALSE, TRUE)
)
grades
```

Think about it

Here is another data frame:

```
df <- data.frame(
  x = 1:5,
  y = c("a", "b", "c", "d", "e")
)
```

What are its column names? Is `df` a good name for a data frame? Are `x` and `y` good names for its columns? Neither tells a reader anything about the data. Prefer descriptive names, such as `grades` or `nc_counties`, and add a comment saying what the data frame holds and where it came from.

5.5.3 Looking at a data frame

A few functions tell you what is in a data frame without printing all of it, which matters once a data frame has thousands of rows:



```
nrow(grades)    # number of rows (observations)
ncol(grades)    # number of columns (variables)
names(grades)   # the column names
str(grades)     # the structure: each column, its type, and its first values
```

In RStudio, `View(grades)` also opens the data frame in a spreadsheet-like tab, and clicking a data frame's name in the Environment pane does the same.

5.5.4 Indexing a data frame

The same bracket indexing works on data frames, with one change: a data frame has two dimensions, so the brackets take **two** indexes, separated by a comma. The row comes first, then the column: `grades[row, column]`.

```
grades[2, 3]              # row 2, column 3: Shonda's exam2 score
grades[2, ]               # all of row 2 (leave the column blank)
grades[, "exam2"]         # all of the exam2 column (leave the row blank)
grades[1:2, c("name", "bonus")]
```

Leaving a position blank means "all of them," so `grades[2, ]` is the whole second row and `grades[, "exam2"]` is the whole `exam2` column. Don't forget the comma: without it, R treats the number as a column index, not a row.

Because pulling out one column is so common, R has a shortcut, the dollar sign. `grades$exam2` returns the `exam2` column as a vector, which you can then use like any other vector:

```
grades$exam2
mean(grades$exam2)
```

Everything from earlier in the chapter now applies to rows. Put a condition in the *row* position, and you get the rows where it is TRUE:

```
grades[grades$exam2 > 90, ]
grades[grades$bonus & grades$exam1 == "A", "name"]
```

The first line returns the two students who scored over 90 on exam 2. The second returns the names of the students who earned the bonus *and* an A on exam 1. In Chapter 6 you will meet `filter()`, the tidyverse's way to do the same thing with less typing.

5.6 Reading tabular data

You will rarely type a data set in by hand. Usually it arrives as a file, and the most common format is a **CSV** file (comma-separated values): plain text in which each line is a row and commas separate the values. The first line usually holds the column names, called the **header**. Here is the grades table as a CSV file:

```
name,exam1,exam2,bonus
Eliza,A,98.1,TRUE
Shonda,A-,85.4,TRUE
```



```
Me'shelle,A,74.3,FALSE
Jack,A-,92.2,TRUE
```

Base R has two functions for reading text files like this into a data frame.

`read.table()` is the general-purpose workhorse. It reads any table of text in which the values are separated by some character, called the **delimiter**, and it has many options, including:

- `header`: is the first row the column names (TRUE or FALSE)?
- `sep`: the delimiter, such as `","` for commas or `"\t"` for tabs.
- `row.names` and `col.names`: labels for the rows and columns.

```
scores <- read.table("scores.txt", header = TRUE, sep = "\t")
```

`read.csv()` is `read.table()` with its defaults set for CSV files, `sep = ","` and `header = TRUE`, so usually all it needs is the file name:

```
scores <- read.csv("scores.csv")
```

Its sibling `read.csv2()` reads the kind of CSV file common in much of Europe, which uses a comma as the decimal mark (3,14) and a semicolon to separate values.

Remember from Section 4.4 that R looks for the file **relative to the folder your `.Rmd` file is saved in**. If the CSV file is in the same folder, the file name alone is enough; if it is in a `data` subfolder, write `read.csv("data/scores.csv")`. Most "cannot open file" errors are file path errors.

The box below can't read a file from your computer, because it runs inside your web browser. Instead, it writes the CSV file above to a temporary file first, and then reads it back:

```
# Write the grades CSV to a temporary file, then read it back in
csv_file <- tempfile(fileext = ".csv")
writeLines(c("name,exam1,exam2,bonus",
             "Eliza,A,98.1,TRUE",
             "Shonda,A-,85.4,TRUE",
             "Me'shelle,A,74.3,FALSE",
             "Jack,A-,92.2,TRUE"), csv_file)

scores <- read.csv(csv_file)
str(scores)
```

`read.csv()` worked out each column's type by itself: character for `name` and `exam1`, numeric for `exam2`, and logical for `bonus`. Always check the types after reading a file. If a column you expected to be numeric comes in as character, something in it, such as a comma in `"12,000"` or a `"N/A"`, isn't a number.

The tidyverse has its own, faster function for reading CSV files, `read_csv()` (with an underscore), from the readr package. It works the same way, and you will see it in later chapters.

5.7 Check your understanding

For questions 1–5, `vect <- c(1, 2, 5, 10, 25, 50, 100, 200, 500, 1000)`.

1. What does `vect[4]` return? __[1]
2. What does `vect[2] + vect[5]` return? __[2]
3. How many values does `vect[-1]` return? _[3]
4. What does `vect[vect >= 100 & vect < 1000]` return? [4]

- (A) 100 200 500 1000
- (B) 200 500
- (C) 100 200 500
- (D) 1 2 5 10 25 50

5. Which line returns the last element of `vect`, whatever its length? [5]

- (A) vect[10]
- (B) vect[-1]
- (C) vect[length(vect)]
- (D) vect[last]

6. 5 = 5 tests whether 5 is equal to 5. TRUE / FALSE[6]
7. What does TRUE | FALSE return? [7]

- (A) TRUE
- (B) FALSE

8. In a data frame, each row is an [8]

- (A) variable
- (B) observation
- (C) vector
- (D) index

and each column is a variable.

9. Which function reads a comma-separated file into a data frame? _____[9]

---

[1] 10
[2] 27
[3] 9
[4] 100 200 500
[5] vect[length(vect)]
[6] FALSE
[7] TRUE
[8] observation
[9] read.csv, read.csv(), read_csv, read_csv()



Show hint for question 4

Work out `vect >= 100` and `vect < 1000` separately, then keep only the values where both are TRUE. Is `1000` less than `1000`?

Show hint for question 6

In R, a single `=` sets an argument, as in `round(x, digits = 1)`. Testing for equality takes two.

5.8 Summary

- **Indexing** selects part of a vector or table, with square brackets. R counts positions from 1.
- **By position:** `vect[3]` is the value at position 3; `vect[-3]` drops it; `vect[c(1, 3)]` and `vect[6:8]` return several values. The colon operator `:` makes a sequence.
- **Comparison operators** (>, <, >=, <=, ==, !=) return TRUE or FALSE for every element. Equality is ==; a single = sets an argument.
- **By condition:** `vect[vect > 10]` keeps the elements where the condition is TRUE. Combine conditions with & (AND: both must be true), | (OR: either can be true), and ! (NOT). Short logical vectors are **recycled**.
- **By name:** `names()` labels the elements, and `vect["Fozzie"]` selects by label. `length()` counts elements, and `vect[length(vect)]` is the last one.
- A **data frame** is a table of equal-length vectors: columns are variables, rows are observations, and each column holds one data type. Make one with `data.frame()`, and index it with `df[row, column]` or `df$column`.
- Read CSV files with `read.csv()` (or `read.table()` for other delimiters), and check the column types after you read.

5.9 Exercises

1. **Predict, then check.** For `vect <- c(1, 2, 5, 10, 25, 50, 100, 200, 500, 1000)`, write down what each line returns *before* you run it. Then run them and check.

```
vect[c(2, 4, 6)]
vect[-(1:3)]
vect[vect < 10 | vect > 200]
vect[!(vect > 5)]
vect[c(FALSE, FALSE, TRUE)]
```

2. **Temperatures.** Store a week of daily high temperatures in a vector, and name each element with a day of the week. Then return (a) Wednesday's temperature, (b) the temperatures above 80, (c) the days on which it was between 70 and 80, and (d) the last day's temperature, using `length()`.

3. **Count with logicals.** `sum(vect > 10)` returns 6. Why? (Hint: R counts TRUE as 1 and FALSE as 0.) Use the same idea to count how many values in `vect` are between 10 and 500, inclusive.



4. **Build a data frame.** Make a data frame of five North Carolina cities with three columns: the city's name, its approximate population, and whether it is a county seat. Use descriptive names and add a comment. Then return (a) the population column with `$`, (b) the third row, and (c) the names of the cities with more than 100,000 people.

5. **Type trouble.** Create the data frame below and run `str()` on it. Why can't you take the mean of `pop`? Fix the data so that you can.

```
towns <- data.frame(
  town = c("Carrboro", "Hillsborough", "Pittsboro"),
  pop  = c("21,295", "9,660", "4,537")
)
```

6. **Read a CSV file.** In a spreadsheet program, make a small table with a header row and at least three columns, and save it as a CSV file in your `geog215` folder. Read it into R with `read.csv()`, and use `str()` to check that each column has the type you expected.

5.10 Further reading

- *R for Data Science* (2nd ed.), https://r4ds.hadley.nz. The chapter "Logical vectors" covers comparisons, `&`, `|`, and `!` in more depth, and "Data import" covers reading CSV files with readr.
- *Hands-On Programming with R* by Garrett Grolemund, free at https://rstudio-education.github.io/hopr/. Its chapter "R Notation" is a gentle, thorough guide to indexing vectors and data frames.
- *Advanced R* (2nd ed.) by Hadley Wickham, https://adv-r.hadley.nz. The chapter "Subsetting" is the complete reference, for when you are ready for it.



6 Blocks, Loops, and Functions

> **Learning objectives**
>
> By the end of this chapter you should be able to:
>
> 1. Explain what a block of code is and how curly braces mark one.
> 2. Use `if` and `else` to run a block only when a test is `TRUE`.
> 3. Repeat a block with a `while` loop or a `for` loop, and avoid an infinite loop.
> 4. Decide when you need a loop and when a vectorized function already does the job.
> 5. Write a function with one or more arguments, a default value, and a return value.
> 6. Explain why a function's arguments don't change the variables outside it.

What is the first virtue of a programmer? Ingenuity? Grit? Devotion to R? Larry Wall, who created the Perl programming language, gave a surprising answer: **laziness**. He meant a particular kind of laziness: refusing to do the same work twice. A lazy programmer writes a piece of code once and reuses it, instead of typing (or copying and pasting) the same steps again and again.

This chapter covers the R tools that let you be productively lazy. **Blocks** group lines of code together. `if` statements decide *whether* a block runs, and loops decide *how many times* it runs. **Functions**, which you have been calling since Chapter 2, give a block a name so you can run it again with new inputs.

6.1 Blocks group lines of code

A **block** is the code between a pair of curly braces, `{ }`. R reads everything from the opening brace to the matching closing brace and treats it as one unit:

```
{
  x <- 5
  y <- x * 2
  print(y)
}
```

All three lines run together, and R prints 10. On their own, braces don't change much. Their power comes when you attach them to an `if`, a loop, or a function, which decide whether, or how many times, the block runs. A block either runs completely or is skipped completely.

Two habits make blocks easy to read:

- **Indent** the lines inside a block by two spaces. RStudio does this for you when you press Enter after a `{`. The indentation shows at a glance where a block starts and ends.



- **Check your braces.** Every `{` needs a matching `}`. A missing closing brace is one of the most common errors in R. When it happens, the Console shows a + prompt, because R is waiting for the rest of the block. Press Esc to start again.

The most common blocks are `if` statements, `while` loops, `for` loops, and functions. We'll take them in that order.

6.2 if statements run a block only when a test is TRUE

An **if statement** has two parts: a **test** in parentheses and a block in braces. R works out the test. If it is TRUE, the block runs. If it is FALSE, R skips the whole block.

```
if (test) {
  # runs only when test is TRUE
}
```

Before you run the code below, predict the value of `x` at the end:

```
x <- 10
if (x < 100) {
  x <- x + 1
}

if (x < 5) {
  x <- x - 1
}

x
```

`x` starts at `10`. Is `10` less than `100`? Yes, so the first block runs and `x` becomes 11. Is 11 less than 5? No, so the second block is skipped. `x` ends at 11.

6.2.1 Adding else

Add `else` to give R a second block for when the test is FALSE. Exactly one of the two blocks runs:

```
x <- 3
if (x < 5) {
  "small"
} else {
  "big"
}
```

You can chain tests with `else if`. R checks them from the top and runs the first block whose test is TRUE:

```
births <- 2500
if (births < 1000) {
```



```
  "few"
} else if (births < 5000) {
  "some"
} else {
  "many"
}
```

6.2.2 Writing good tests

The test uses the comparison and logical operators from Chapter 4: <, >, <=, >=, ==, !=, &, and |. Two mistakes come up again and again:

- **= is not ==.** `x = 5` assigns 5 to x. To ask "is x equal to 5?", write `x == 5`.
- **The test must be one TRUE or FALSE.** `if (c(1, 8) > 5)` is an error, because the test gives two answers, FALSE TRUE, and R doesn't know which one to use. To decide something for every element of a vector, use the vectorized `ifelse()` instead:

```
births <- c(480, 2500, 21588)
ifelse(births < 1000, "few", "not few")
```

6.3 while loops repeat a block while a test is TRUE

A **while loop** looks like an `if` statement, but instead of running its block once, it goes back to the test after every run. As long as the test is TRUE, the block runs again:

```
x <- 11
while (x < 100) {
  x <- x + 1
}

x
```

R checks the test (is 11 less than 100?), runs the block (x becomes 12), checks the test again, and so on. When x reaches 100 the test is FALSE and the loop stops, so x is 100.

> ⚠ **Watch out for infinite loops**
>
> The block must change something in the test, or the test never becomes FALSE and the loop never ends. If R seems stuck, click the red stop sign at the top of RStudio's Console, or press Esc.

In practice, R users rarely write `while` loops. A `for` loop, or a function that works on a whole vector at once, is almost always simpler.



6.4 for loops run a block once for each item

A **for loop** doesn't have a TRUE or FALSE test. Instead, it walks through a vector one item at a time. On each pass, the **loop variable** takes the next item:

```
for (county in c("Orange", "Durham", "Wake")) {
  print(paste(county, "County"))
}
```

On the first pass `county` is `"Orange"`, on the second `"Durham"`, and on the third `"Wake"`. After the last item the loop stops on its own, so a `for` loop can't run forever. That makes it safer than a `while` loop.

Use a `for` loop when you know what you want to repeat over: a set of files, counties, or years. This loop reads each CSV file in a folder and prints how many rows it has:

```
files <- list.files("data", pattern = "\\.csv$", full.names = TRUE)

for (f in files) {
  d <- read.csv(f)
  print(nrow(d))
}
```

6.4.1 Saving results from a loop

`print()` only shows a result. To keep the results, make an empty vector before the loop and fill in one element on each pass. `seq_along()` gives the positions 1, 2, 3, ... of a vector, which makes a good loop variable:

```
miles <- c(5, 12, 26.2)
km <- numeric(length(miles))   # an empty numeric vector: 0 0 0

for (i in seq_along(miles)) {
  km[i] <- miles[i] * 1.609
}

km
```

6.4.2 Do you need a loop?

Many R functions are **vectorized**: they already work on every element of a vector at once. The loop above does the same job as one line:

```
miles <- c(5, 12, 26.2)
miles * 1.609
```

Before you write a loop, check for a vectorized way to do the job. `x + 1`, `sqrt(x)`, `round(x)`, `ifelse()`, and the dplyr verbs you'll meet in Unit 2 all work on whole vectors or columns. Reach for a loop when there isn't one, such as reading many files or downloading data for many years.



> **Think about it**
>
> Later in the book, Chapter 13 downloads American Community Survey data with tidycensus. How could a `for` loop help you download the same table for 2019, 2020, 2021, 2022, and 2023, and save each year to its own file?

6.5 Functions name a block so you can reuse it

A **function** is a block with a name, inputs, and an output. You have already called many functions, such as `mean()`, `round()`, and `read.csv()`. Now that you know what a block is, you can write your own. Every function has the same parts:

```
name <- function(argument1, argument2) {
  # the body: a block of code
  result <- ...
  result          # the last value is the output
}
```

- **Name.** You create a function the same way you create any object, with `<-`.
- **Arguments.** The inputs, in parentheses after the word `function`. Inside the body they are placeholders for whatever you pass in.
- **Body.** The block between the braces.
- **Return value.** The value of the last line in the block is what the function returns. You can also write `return(result)` to make it explicit.

Here is the smallest useful function, `addSeven()`. Running the first three lines only *creates* the function; nothing is printed. The function runs when you call it:

```
addSeven <- function(numbers) {
  numbers + 7
}

addSeven(1)
addSeven(c(1, 2, 3))
```

Because `+` is vectorized, `addSeven(c(1, 2, 3))` returns 8, 9, 10 without a loop.

6.5.1 A preview: the weighted mean center

In Chapter 17 you'll use a function, `meanCenter()`, that finds the weighted mean center of a map layer. You don't need to follow its spatial steps yet, but you can already pick out its parts. Its name is `meanCenter`, its arguments are `layer` and `weights`, its body turns polygons into centroids and takes the weighted mean of their coordinates, and its last line, `mean_center`, is the return value:

```
meanCenter <- function(layer, weights) {
  # convert the polygons into points
  points <- st_centroid(layer)
  # weighted mean of the x and y coordinates
  mc_x <- weighted.mean(st_coordinates(points)[, 1], weights)
```



```
  mc_y <- weighted.mean(st_coordinates(points)[, 2], weights)
  # create a point with the same CRS as the input layer
  mean_center <- st_sfc(st_point(c(mc_x, mc_y)))
  mean_center <- st_set_crs(mean_center, st_crs(layer))
  mean_center
}

mc_pop <- meanCenter(nc_counties, nc_counties$popE)
```

6.5.2 More than one argument, and defaults

Separate several arguments with commas. You can give an argument a **default value** with =. The function uses the default unless you pass something else:

```
toKm <- function(distance, factor = 1.609) {
  distance * factor
}

toKm(10)                    # uses the default: miles to km
toKm(10, factor = 0.3048)   # feet to meters instead
```

6.5.3 Putting blocks inside functions

A function's body can hold any block, including `if` statements and loops. A common use is to check the input and stop with a helpful message if it's wrong. `stop()` ends the function with an error:

```
addSeven <- function(numbers) {
  if (!is.numeric(numbers)) {
    stop("addSeven() needs numbers, not ", class(numbers))
  }
  numbers + 7
}

addSeven(c(1, 2, 3))
addSeven("seven")
```

6.6 A function's arguments are separate from variables outside it

What do these lines print? Predict, then run them:

```
numbers <- 10

addSeven <- function(numbers) {
  numbers + 7
}
```

```
addSeven(1)
numbers
```

`addSeven(1)` returns 8, and `numbers` is still 10. Inside the function, `numbers` means whatever you passed in, here 1. Each time a function runs, R gives it a fresh, private workspace, called its **local environment**, and throws that workspace away when the function finishes. The `numbers` in your own workspace, the **global environment**, is never touched.

This idea, which variables a piece of code can see, is called **scope**. It is why functions are safe to reuse: they don't overwrite your variables unless you save their result with `<-`. If you want to keep what a function returns, assign it:

```
numbers <- 10
numbers <- addSeven(numbers)
numbers
```

6.7 Your turn

Each of these only needs the pattern `name <- function(input) { body }`.

```
# 1. timesTwo(): double its input. Try it on c(1, 2, 3).

# 2. addThis(): two inputs, numbers and amount; add amount to numbers.

# 3. countAbove(): count how many values in a vector are above a cutoff,
#    with a default cutoff of 0.
```

Show answers

```
timesTwo <- function(x) {
  x * 2
}
timesTwo(c(1, 2, 3))          # 2 4 6

addThis <- function(numbers, amount) {
  numbers + amount
}
addThis(c(1, 2, 3), 10)       # 11 12 13

countAbove <- function(values, cutoff = 0) {
  sum(values > cutoff)
}
countAbove(c(-2, 5, 0, 8))    # 2
countAbove(c(-2, 5, 0, 8), 6) # 1
```

6.8 Check your understanding

1. In R, a block of code is marked by: [1]

- (A) Parentheses ( )
- (B) Curly braces { }
- (C) Square brackets [ ]
- (D) Quotation marks

2. What is `x` after this code runs? `x <- 10; if (x < 100) { x <- x + 1 }; if (x < 5) { x <- x - 1 }` _[2]

3. A `while` loop whose block never changes anything in its test will: [3]

- (A) Run once
- (B) Never run
- (C) Run forever, until you stop it
- (D) Give an error right away

4. A `for` loop stops on its own after the last item in its vector. TRUE / FALSE[4]

5. To test whether `x` equals 5 inside an `if`, you write `if (x ____ 5)`. Fill in the blank. _[5]

6. Which is the simplest way to add 1 to every element of a numeric vector `x`? [6]

- (A) A for loop
- (B) A while loop
- (C) `x + 1`
- (D) An if statement

7. A function returns the value of the last line in its body, unless you use `return()` earlier. TRUE / FALSE[7]

8. After `numbers <- 10`, `addSeven <- function(numbers) { numbers + 7 }`, and `addSeven(1)`, what is `numbers`? _[8]

---

[1] Curly braces { }
[2] 11
[3] Run forever, until you stop it
[4] TRUE
[5] ==
[6] x + 1
[7] TRUE
[8] 10

6.9 Summary

- A **block** is code inside curly braces `{ }` that R runs as one unit. Indent it, and match every `{` with a `}`.
- An **if statement** runs its block only when its test is `TRUE`. `else` gives a second block for `FALSE`, and `else if` chains more tests. The test must be one `TRUE` or `FALSE`; use `ifelse()` for a whole vector.
- A **while loop** repeats its block while its test is `TRUE`. Something in the block must move the test toward `FALSE`, or the loop never ends.
- A **for loop** runs its block once for each item in a vector, then stops. Fill a vector made before the loop to keep the results.
- Many R functions are **vectorized**, so check for one before you write a loop.
- A **function** is a named block: `name <- function(arguments) { body }`. The last line is the **return value**. Arguments can have **default values**.
- A function's arguments live in its own **local environment**, so a function never changes your variables unless you assign its result.

6.10 Exercises

1. **Predict, then run.** Without running it, predict what this prints, then check: `x <- 1; while (x < 50) { x <- x * 2 }; x`.
2. **Grades.** Write a function, `letterGrade(score)`, that uses `if`, `else if`, and `else` to return `"A"` for 90 or more, `"B"` for 80 to 89, `"C"` for 70 to 79, and `"F"` below 70. Why does `letterGrade(c(95, 72))` give an error? Rewrite it with nested `ifelse()` so that it works on a vector.
3. **Loop or not?** Write a `for` loop that converts `c(32, 50, 68, 86, 104)` from degrees Fahrenheit to Celsius, `(f - 32) * 5 / 9`, and stores the results. Then do the same thing in one line without a loop.
4. **Running total.** Use a `for` loop to find the running total of births in `c(1091, 487, 3188, 508, 1421)`: the first value, then the first two added up, and so on. Check your answer with `cumsum()`.
5. **Checking inputs.** Add an `if` statement and `stop()` to `toKm()` so that it gives a helpful error when `distance` is negative.
6. **Standard distance again.** (Come back to this one after Chapter 17.) Chapter 17's `standardDistance(x, y)` function returns the standard distance of a set of points. Add a check that `x` and `y` are the same length, and stop with a message if they aren't.

6.11 Further reading

- *R for Data Science* (2nd ed.), by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund, free at https://r4ds.hadley.nz. The "Functions" and "Iteration" chapters show more ways to write functions and to avoid loops.
- *Hands-On Programming with R*, by Garrett Grolemund, free at https://rstudio-education.github.io/hopr/. Its chapters on `if` and `else`, loops, and environments go further into the ideas in this chapter.
- *Programming Perl* (3rd ed.), by Larry Wall, Tom Christiansen, and Jon Orwant (O'Reilly, 2000), where Wall describes the three virtues of a programmer: laziness, impatience, and hubris.



Part II

Unit 2: Into the Tidyverse



This unit introduces the tidyverse, a family of R packages that share one way of working with data. Using a year of flights from New York's three airports (the nycflights13 data), you learn the dplyr verbs that filter, sort, select, change, and summarize a data frame, chained together with the pipe. The unit ends by turning those tables into graphs with ggplot2.

Chapter 6: Into the Tidyverse: glimpse, filter, and the Pipe

This chapter introduces the tidyverse and the data science workflow it is built around. You look at a data frame with `glimpse()`, keep the rows you want with `filter()`, and chain steps together with the pipe, `|>`. It also explains missing values (NA) and how they affect filtering.

Chapter 7: Into the Tidyverse: arrange, distinct, select, and mutate

dplyr verbs work on either rows or columns. This chapter sorts rows with `arrange()`, finds unique values with `distinct()`, keeps the columns you need with `select()`, and calculates new columns with `mutate()`. Sorting a new column turns out to be a quick way to spot outliers worth checking.

Chapter 8: Into the Tidyverse: summarize, group_by, and drop_na

`summarize()` collapses many rows into summary statistics, and `group_by()` makes it give one answer for each group, such as each month or each airline. The chapter compares two ways of handling missing values, `na.rm = TRUE` and `drop_na()`, and shows why the average of averages can mislead and how a weighted average fixes it.

Chapter 9: Into the Tidyverse: Graphs with ggplot2

ggplot2 builds graphs from the grammar of graphics: data, aesthetics that map columns to visual properties, and geoms that draw the marks, added in layers with +. This chapter makes scatterplots, bar charts, and boxplots, explains the difference between setting and mapping an aesthetic, splits plots into small multiples with facets, and finishes plots with colors, labels, themes, and `ggsave()`.



7 Into the Tidyverse: glimpse, filter, and the Pipe

> **Learning objectives**
>
> By the end of this chapter you should be able to:
>
> 1. Explain what the tidyverse is, and load it.
> 2. Look inside a data frame with `glimpse()`.
> 3. Keep the rows you need with `filter()`, using one or more conditions.
> 4. Chain steps together with the pipe, `|>`.
> 5. Explain how `filter()` and base R brackets treat missing values differently.

In Chapter 4 you selected rows of a data frame with square brackets, as in `grades[grades$exam2 > 90, ]`. That works, but it gets long and hard to read quickly: the data frame's name appears again for every condition, and every step needs a new variable. This chapter introduces the **tidyverse**, a set of packages designed to make that kind of work shorter and easier to read, and three tools from it that you will use in almost every assignment from here on: `glimpse()` to see what is in a data frame, `filter()` to keep the rows you need, and the pipe, `|>`, to chain steps together.

The examples use real data: every flight that left New York City's three airports in 2013, from the **nycflights13** package. By the end of the chapter you will be able to answer questions such as "how many flights left LaGuardia in June?" in four short lines.

7.1 What is the tidyverse?

7.1.1 The data science workflow

The **tidyverse** is a family of R packages built for data science. Its packages share a design philosophy, a grammar, and a way of storing data, so they fit together. They are organized around the workflow in Figure 7.1, from the book *R for Data Science* (Wickham et al. 2023):

First you **import** data into R. Then you **tidy** it into a consistent shape, with one variable per column and one observation per row. Then you go round a loop of **transforming** it (keeping the rows you need, making new columns, summarizing), **visualizing** it, and **modeling** it, until you understand it. Finally you **communicate** what you found, usually in an R Markdown report. This chapter lives in the Tidy and Transform boxes: getting data into shape and pulling out the rows you need.

7.1.2 The core packages

One command, `install.packages("tidyverse")` (or one trip to the Packages tab), installs the core tidyverse packages (Table 7.1).

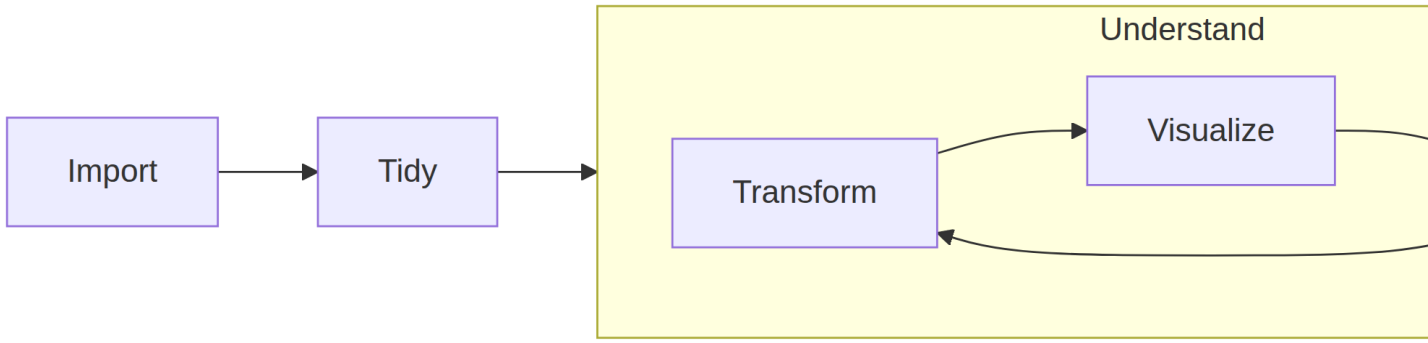


Figure 7.1: The data science workflow, after Wickham, Çetinkaya-Rundel, and Grolemund (2023). Data are imported and tidied, then transformed, visualized, and modeled, often several times over, before the results are communicated. This chapter is about tidying and transforming.

Table 7.1: The core tidyverse packages.

Package	What it is for
dplyr	Transforming data frames: keeping rows, choosing columns, making new columns, summarizing
ggplot2	Plots and charts
tidyr	Reshaping data into a tidy form
readr	Reading files such as CSVs, with `read_csv()`
tibble	The tidyverse's version of the data frame
stringr	Working with text
forcats	Working with categories (factors)
lubridate	Working with dates and times
purrr	Repeating a task over many inputs

This chapter uses **dplyr**, which supplies `glimpse()` and `filter()`. Posit publishes a one-page cheatsheet for dplyr, *Data transformation with dplyr*, linked on the course Canvas page and at https://posit.co/resources/cheatsheets/. `filter()` is on the front page under **Manipulate Cases** (Extract Cases). It is worth keeping open while you work.

7.1.3 Loading the tidyverse

Load the tidyverse at the top of every R Markdown file, the same way as any other package:

```
library(tidyverse)
```

The first time you do, R prints a block of messages like this (your version numbers may differ):

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.1     ✔ tibble    3.2.1
✔ lubridate 1.9.3     ✔ tidyr     1.3.1
✔ purrr     1.0.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
```



```
✖ dplyr::lag()    masks stats::lag()
```

These messages are normal. They are not errors, even though some are printed in red. The first block lists the packages that were loaded. The second, **Conflicts**, says that two packages contain a function with the same name, and which one R will use. Here, dplyr's `filter()` **masks** the `filter()` in base R's stats package (a function for time series). From now on, when you type `filter()`, you get dplyr's version, which is the one you want. If you ever need the other one, you can name its package explicitly with two colons: `stats::filter()`.

7.1.4 Tidying data is a set of verbs

What do we actually do when we tidy data? We select, extract, arrange, sort, combine, group, join, and summarize. dplyr gives each of these jobs its own function, named with a verb (Table 7.2).

Table 7.2: The main dplyr verbs.

dplyr verb	What it does	Covered
`filter()`	Keeps the rows that meet a condition	This chapter
`select()`	Picks columns	Next chapter
`arrange()`	Sorts rows	Next chapter
`mutate()`	Makes new columns, or changes existing ones	Next chapter
`group_by()` and `summarize()`	Collapse groups of rows into summary values	Later
`left_join()` and friends	Combine two tables	Later in the course

Every dplyr verb works the same way: its first argument is a data frame, the arguments after it say what to do, and it returns a new data frame. That consistency is what makes the pipe, later in this chapter, work so well.

7.2 Seeing inside a data frame with glimpse()

7.2.1 The flights data

The **nycflights13** package contains a data frame called `flights`: one row for every flight that departed from New York City's three airports (Newark, EWR; John F. Kennedy, JFK; and LaGuardia, LGA) in 2013. You might use it to compare flights by day, by airport, or by how late they left. Install it once from the Packages tab, then load it along with the tidyverse:

```
library(tidyverse)
library(nycflights13)
```

The live code boxes in this chapter load **dplyr**, the part of the tidyverse this chapter uses, rather than the whole tidyverse, because it loads faster in a web browser. Everything works the same way in RStudio with `library(tidyverse)`. The first time you run a box, it takes a little while to download the flights data.



7.2.2 glimpse()

`flights` has more than 300,000 rows, far too many to print. Typing `flights` shows only the first few rows and as many columns as fit across the screen. `glimpse()` is the tidyverse's way to see everything that is in a data frame at once. It turns the data on its side: each **column** becomes one line of output, showing the column's name, its type, and its first few values. Run it:

```
library(dplyr)
library(nycflights13)

glimpse(flights)
```

The first two lines give the size: 336,776 rows and 19 columns. Below them is one line per column. The type of each column is in angle brackets, abbreviated (Table 7.3).

Table 7.3: Column types as `glimpse()` shows them.

In `glimpse()`	Type	Example columns in `flights`
`<int>`	integer (whole numbers)	`year`, `month`, `day`, `dep_time`
`<dbl>`	numeric ("double"), may have decimals	`dep_delay`, `arr_delay`, `distance`
`<chr>`	character (text)	`carrier`, `origin`, `dest`, `tailnum`
`<dttm>`	a date and time	`time_hour`

The columns this chapter uses are `month` and `day` (numbers, so June is 6, not `"June"`), `origin` (the three-letter code of the airport the flight left from), and `dep_delay`, the departure delay in minutes. A negative `dep_delay` means the flight left early.

`glimpse()` is one of several ways to look at a data frame. `head(flights)` prints the first six rows; `str(flights)`, from Chapter 4, is base R's version of `glimpse()`; and in RStudio, `View(flights)` opens the whole table in a spreadsheet-like tab. Get in the habit of looking at every data frame before you work with it. Most surprises (a number stored as text, a column you didn't expect, far fewer rows than you thought) show up in a glimpse.

7.3 Keeping rows with filter()

7.3.1 One condition

`filter()` keeps the rows of a data frame that meet a condition. Its first argument is the data frame, and the arguments after that are one or more logical conditions:

```
filter(data, condition)
```

It returns only the rows where the condition is TRUE. Here are the flights that left on the first day of a month:

```
filter(flights, day == 1)
```



`flights` is the data frame, and `day == 1` is the logical condition. Notice that, inside `filter()`, you refer to the columns by name alone, `day` rather than `flights$day`: `filter()` knows to look for them in the data frame. The result has 11,036 rows, the flights on the 1st of each of the twelve months.

The condition uses **==**, which tests whether two things are equal. A single = sets an argument, and writing `filter(flights, day = 1)` is one of the most common mistakes in R. dplyr catches it and tells you, with an error that says you have used = instead of == and asks "Did you mean `day == 1`?"

`filter()` doesn't change `flights`; it returns a new data frame. To keep the result, store it with `<-`, and give it a new name, so that you still have the original:

```
first_of_month <- filter(flights, day == 1)
nrow(first_of_month)
nrow(flights)
```

7.3.2 Several conditions

Give `filter()` more than one condition, separated by commas, and it keeps the rows where **all** of them are TRUE. A comma inside `filter()` means AND, exactly like &. These two lines both find the flights on January 1:

```
filter(flights, month == 1, day == 1)
filter(flights, month == 1 & day == 1)
```

Both return 842 rows. For OR, use `|`. This finds the flights in June *or* July:

```
filter(flights, month == 6 | month == 7)
```

Conditions on text work the same way, but the text needs quotation marks, as always: `origin == "LGA"`, not `origin == LGA`. Without the quotes, R looks for a column or variable named LGA and doesn't find one.

7.4 The pipe

7.4.1 Chaining steps with |>

Most questions take several steps. Suppose you want to know how many flights left LaGuardia early, or on time, on January 1, 2013. That takes three filters and a count. You could save a new data frame after every step, or nest the functions inside one another, but both get hard to read.

You met the **pipe**, `|>`, in an earlier chapter. It takes the output of one function and feeds it in as the first argument of the next. Because every dplyr verb takes a data frame as its first argument and returns one, verbs can be chained with pipes as long as you like:

```
flights |>
  filter(month == 1, day == 1) |>
  filter(origin == "LGA") |>
  filter(dep_delay <= 0) |>
```



```
  nrow()
```

Read the pipe out loud as "**and then**": take `flights`, *and then* keep January 1, *and then* keep LaGuardia, *and then* keep the flights that left on time or early (a delay of zero or less), *and then* count the rows. The answer is 172 flights. Each step narrows the rows further, and the code reads in the same order as the question.

7.4.2 Don't repeat the data frame

With the pipe, the data frame goes *in front of* the pipe, not inside `filter()`. These two lines do exactly the same thing:

```
filter(flights, day == 1)
flights |> filter(day == 1)
```

In the second, `|>` passes `flights` into `filter()` as its first argument. So inside a pipe chain you never write `filter(flights, ...)`. If you do, `flights` is passed in twice, once by the pipe and once by you, and R reports an error.

> **This is not a pipe**
>
> René Magritte's 1929 painting *The Treachery of Images* shows a pipe above the words *Ceci n'est pas une pipe*, "This is not a pipe." It isn't: it's a picture of one. In the same spirit, `|>` isn't really a pipe either. It's an operator, a piece of R grammar that rewrites `x |> f(y)` as `f(x, y)` before the code runs. The plumbing metaphor is just a good way to picture it: data flow in at one end and out at the other.
> You may also see %>% in older code and online answers. It is the original pipe, from the magrittr package (named after Magritte), and does almost the same job. The native pipe `|>` was added to R itself in version 4.1, and it is the one this course requires.

7.5 Missing values: why the counts don't match

7.5.1 The same question, two ways

Here is the January 1 LaGuardia question again, answered first with base R brackets, as in Chapter 4, and then the tidy way with `filter()` and the pipe:

```
# The base R way: repeat the data frame's name, and save every step
jan1 <- flights[flights$month == 1 & flights$day == 1, ]
jan1_lga <- jan1[jan1$origin == "LGA", ]
jan1_lga_early <- jan1_lga[jan1_lga$dep_delay <= 0, ]
nrow(jan1_lga_early)

# The tidy way: less typing, read top to bottom
flights |>
  filter(month == 1, day == 1, origin == "LGA", dep_delay <= 0) |>
  nrow()
```



The base R way needs more typing, and more places for a typo: `flights$` for every condition, and a new object at every step. The tidy way reads top to bottom. But look at the answers. Base R finds **174** flights, and `filter()` finds **172**. They don't match. Why?

7.5.2 NA: a missing value

The answer is **missing values**. In R, a value that is unknown or missing is written **NA** ("not available"). Flights that were cancelled never departed, so their departure delay is unknown: `dep_delay` is NA. Two of the January 1 LaGuardia flights were cancelled.

What is `NA <= 0`? R can't say whether an unknown number is less than zero, so the answer is itself unknown: NA, not TRUE or FALSE. Each approach handles that differently (Table 7.4):

- **Base R brackets** don't drop a row when the condition is NA. They return a row in which every value is NA, so the result has two extra rows of missing values, 174 in all.
- **`filter()`** keeps only the rows where the condition is TRUE. It quietly drops rows where the condition is NA, so it finds 172.

Table 7.4: How base R brackets and `filter()` treat each result of a condition.

	Condition is TRUE	Condition is FALSE	Condition is NA
Base R `df[condition, ]`	Row kept	Row dropped	A row of NAs is returned
`filter(df, condition)`	Row kept	Row dropped	Row dropped

You can see the two extra rows by looking at the last rows of the base R result:

```
tail(jan1_lga_early[, c("month", "day", "origin", "dep_delay")], 3)
```

7.5.3 Deciding what happens to NAs

Neither approach is wrong, but they give different answers, and *nothing warns you*. That is annoying, and it is exactly the kind of quiet difference that leads to wrong results in a report. The lesson is to check the number of rows after every filter, and to decide on purpose what should happen to missing values.

If you *do* want to keep the rows with a missing delay, say so in the condition, with the function `is.na()`, which returns TRUE when a value is NA:

```
flights |>
  filter(month == 1, day == 1, origin == "LGA") |>
  filter(is.na(dep_delay) | dep_delay <= 0) |>
  nrow()
```

Read the second filter as "keep the flight if its delay is missing *or* it left on time or early." Now `filter()` also returns 174. To count the missing values themselves, filter for them: `filter(flights, is.na(dep_delay))` returns all 8,255 flights in 2013 with no departure delay recorded.



Note that you can't test for NA with ==. `dep_delay == NA` is always NA, whatever `dep_delay` is, because R can't tell whether something equals an unknown value. Always use `is.na()`.

7.6 Practice: counting flights

Put it all together. These steps follow the in-class exercise.

Step 1: set up. In RStudio, use the Packages tab to install **nycflights13** and **tidyverse**, if you haven't yet. Start a new R Markdown file, and load both in the first chunk:

```
library(tidyverse)
library(nycflights13)
```

You will see the tidyverse's startup messages. Remember that they are not errors.

Step 2: count all the flights. The dplyr function `count()` counts the rows of a data frame, so pipe `flights` into it:

```
flights |>
  count()
```

There were 336,776 flights from New York City's airports in 2013.

Step 3: flights in June. Add a `filter()` before `count()` that keeps month 6. Remember that `month` is stored as a number, so write 6, not `"June"`. Try it before you look at the answer:

```
flights |>
  # add a filter() here
  count()
```

Show the answer to step 3

```
flights |>
  filter(month == 6) |>
  count()
```

There were 28,243 flights in June.

Step 4: June flights from LaGuardia. Now keep only the June flights that left from LaGuardia. LaGuardia's airport code is `"LGA"`, and it is text, so it needs quotation marks. You can add a second `filter()`, or put both conditions in one, joined with a comma or &:

```
flights |>
  filter(month == 6) |>
  # add a second condition
  count()
```

Show the answer to step 4

All three of these return the same count, 8,596 flights. Pick the one you find easiest to read.

```
# Two filters, one after the other
flights |>
  filter(month == 6) |>
  filter(origin == "LGA") |>
  count()

# One filter, with a comma (AND)
flights |>
  filter(month == 6, origin == "LGA") |>
  count()

# One filter, with & (AND)
flights |>
  filter(month == 6 & origin == "LGA") |>
  count()
```

Read the first version as "take `flights`, *and then* keep June, *and then* keep LGA, *and then* count." If you got a much bigger number, check for missing quotation marks around `"LGA"`, or a single = where you need ==.

7.7 Check your understanding

1. Which tidyverse package provides `filter()` and `glimpse()`? [1]

- (A) ggplot2
- (B) readr
- (C) dplyr
- (D) tidyr

2. When you load the tidyverse, the message `dplyr::filter() masks stats::filter()` means something has gone wrong. TRUE / FALSE[2]

3. In `glimpse()` output, what type is a column marked `<chr>`? [3]

- (A) integer
- (B) numeric
- (C) character
- (D) logical

[1] dplyr
[2] FALSE
[3] character

4. Complete the code to keep the flights that left from JFK: `filter(flights, origin ____ "JFK")` _[4]

5. Inside `filter()`, what does a comma between two conditions mean? [5]

- (A) OR: either can be TRUE
- (B) AND: both must be TRUE
- (C) NOT
- (D) Start a new filter

6. Read |> out loud as: [6]

- (A) or
- (B) and then
- (C) is not
- (D) gets

7. `filter()` keeps the rows where the condition is NA. TRUE / FALSE[7]

8. Which function tells you whether a value is missing? _____[8]

Show hint for question 7

`filter()` keeps only the rows where the condition is TRUE. To keep the missing ones, you have to ask for them with `is.na()`.

7.8 Summary

- The **tidyverse** is a family of packages for data science that work together, organized around the workflow import, tidy, transform, visualize, model, and communicate. Load it with `library(tidyverse)`; the startup and conflict messages are normal.
- **dplyr** gives each data-tidying job a verb. Every verb takes a data frame first and returns a data frame.
- **`glimpse()`** shows a data frame's size and every column's name, type (`<int>`, `<dbl>`, `<chr>`, `<dttm>`), and first values. Look at every data frame before you work with it.
- **`filter()`** keeps the rows where a condition is TRUE. Use == to test equality, quotation marks around text, and commas or & for AND, | for OR.
- The **pipe** |> passes the result on its left into the first argument of the function on its right. Read it as "and then." Inside a pipe chain, don't repeat the data frame.
- **NA** marks a missing value. Base R brackets return a row of NAs when a condition is NA; `filter()` drops it. Check row counts, and use `is.na()` when you want to keep or find missing values.

[4] ==
[5] AND: both must be TRUE
[6] and then
[7] FALSE
[8] is.na, is.na()



7.9 Exercises

Use the `flights` data from nycflights13 for these exercises, and answer each with a pipe chain.

1. **Glimpse.** Run `glimpse(flights)`. Which columns hold times? Which hold distances? What does a negative value in `arr_delay` mean?
2. **One airport.** How many flights left Newark (`"EWR"`) in 2013? How many left JFK? Which of the three airports was busiest?
3. **Christmas Day.** How many flights left New York City on December 25, 2013? How many of them left more than an hour late?
4. **Raleigh–Durham.** The `dest` column holds each flight's destination airport. How many flights flew to Raleigh–Durham (`"RDU"`) in 2013? How many of them have a missing `dep_delay`? What do you think happened to those flights?
5. **AND or OR?** Predict which of these returns more rows, then check. Explain the difference in words.

```
flights |> filter(month == 6 & month == 7) |> count()
flights |> filter(month == 6 | month == 7) |> count()
```

6. **Long delays.** How many flights left more than two hours late? Write the same filter in base R with square brackets, as in Chapter 4. Do the two methods agree? Why or why not?
7. **Fix the code.** Each chain below has one mistake. Find and fix it.

```
flights |> filter(origin = "LGA") |> count()
flights |> filter(flights, month == 6) |> count()
flights |> filter(origin == LGA) |> count()
flights |> filter(dep_delay == NA) |> count()
```

7.10 Further reading

- *R for Data Science* (2nd ed.) (Wickham et al. 2023), free at https://r4ds.hadley.nz, is the course's main R reference. The chapters "Data transformation" and "Workflow: code style" cover `filter()`, the pipe, and the other dplyr verbs, using the same flights data.
- The dplyr website, https://dplyr.tidyverse.org, has a reference page for every function, with examples.
- Posit's cheatsheets, https://posit.co/resources/cheatsheets/, include *Data transformation with dplyr*. Print it and keep it next to your computer.

8 Into the Tidyverse: arrange, distinct, select, and mutate

Learning objectives

By the end of this chapter you should be able to:

1. Sort the rows of a data frame with `arrange()`, in either direction and by more than one column.
2. Find the unique values, or unique combinations of values, in a data frame with `distinct()`.
3. Keep only the columns you need with `select()`.
4. Make new columns from existing ones with `mutate()`.
5. Chain these verbs and `filter()` together with the pipe to answer a question step by step.

In the last chapter you met the tidyverse, looked inside the `flights` data with `glimpse()`, and kept the rows you needed with `filter()`. This chapter adds four more dplyr verbs. Two of them work on rows: `distinct()` finds the unique values in a column, and `arrange()` sorts the rows. The other two work on columns: `select()` keeps the columns you name, and `mutate()` makes new ones. With `filter()`, these five verbs cover most of the day-to-day work of getting data into shape.

The examples keep using the **nycflights13** data, so you can build each answer one step at a time. By the end of the chapter you will be able to answer questions such as "where can I fly from JFK in under three hours?" and "what was the fastest flight out of New York in 2013?"

Warm-up: why doesn't this code work?

```
flights |> filter(month = 1)
```

Run it in the box below and read the error before you look at the answer.

```
library(dplyr)
library(nycflights13)

flights |> filter(month = 1)
```

Show the answer

R says it "detected a named input" and asks "Did you mean `month == 1`?" A single `=` names an argument; a comparison needs a double `==`. The fix is `flights |> filter(month == 1)`. dplyr's error messages often tell you exactly what went wrong, so read them before you change anything.



8.1 Verbs for rows and verbs for columns

8.1.1 Two kinds of verb

Every dplyr **verb** takes a data frame as its first argument and returns a new data frame, which is why they chain together so well with the pipe. But they change different things. Some verbs change *which rows* you have, or the order they are in. Others change *which columns* you have (Table 8.1).

Table 8.1: The five dplyr verbs in this chapter and the last, grouped by what they work on. Every example is code you will run in this chapter.

Works on	Verb	What it does	Example
Rows	`filter()`	Keeps the rows that meet a condition	`filter(origin == "JFK")`
Rows	`distinct()`	Keeps one row for each unique value	`distinct(dest)`
Rows	`arrange()`	Sorts the rows	`arrange(dest, origin)`
Columns	`select()`	Keeps the columns you name	`select(origin, dest)`
Columns	`mutate()`	Adds new columns	`mutate(avg_speed = distance / air_time * 60)`

Keeping the two groups in mind helps you plan a pipe chain. If the question is about *which flights*, you need a row verb. If it is about *which information about each flight*, you need a column verb.

8.1.2 Where this fits

These verbs all belong to the **Transform** step of the data science workflow you met in the last chapter. None of them changes what a row means: before and after, each row of `flights` is one flight. They only change which flights you keep, the order they are in, and which facts about each flight you can see. That relies on the data being **tidy**: each variable is a column, each observation is a row, and each value has its own cell.

8.2 Finding unique values with distinct()

8.2.1 One column

`flights` has 336,776 rows, but most of the values in a column like `origin` repeat thousands of times. `distinct()` keeps one row for each **unique value**. Give it the data frame and the column to check:

```
flights |>
  distinct(origin)
```



Read it as "take `flights`, *and then* keep one row for each different `origin`." The answer is three rows: the three New York City airports, Newark (`EWR`), LaGuardia (`LGA`), and John F. Kennedy (`JFK`). Notice that the result has only the column you named. `distinct()` answers the question "which values are there?", so it drops the other columns.

8.2.2 Building up a question

`distinct()` is most useful at the end of a pipe chain, after `filter()` has kept the rows you care about. Here is one question, asked four ways, each a little narrower than the last.

Where can I fly from New York? Every row of `flights` is one departure, so the same destination appears thousands of times. `distinct(dest)` keeps each destination once:

```
flights |>
  distinct(dest)
```

The result has 105 rows: 105 destination airports. They are all domestic. Most are in the 50 states, and a few are in US territories, such as San Juan (SJU) in Puerto Rico and St. Thomas (STT) in the US Virgin Islands.

Where can I fly from JFK? Put a `filter()` in front, so that `distinct()` only sees the JFK flights:

```
flights |>
  filter(origin == "JFK") |>
  distinct(dest)
```

JFK serves 70 of the 105 destinations.

Where can I fly from JFK in 3 hours or less? `air_time` is the time in the air, in minutes, so 3 hours is 180 minutes. Add a second filter:

```
flights |>
  filter(origin == "JFK") |>
  filter(air_time <= 180) |>
  distinct(dest)
```

That leaves 50 destinations. <= means "less than or equal to," so this keeps flights of exactly 180 minutes too. For "less than 3 hours," strictly, use <. In this case both give the same 50 destinations, but they won't always, so write the one that matches your question.

And sorted by airport code? The result above is in the order the destinations first appear in the data, which is hard to scan. The next section shows how to sort it.

8.2.3 Several columns

Give `distinct()` more than one column and it keeps each unique *combination* of their values. Which airports are linked by a direct flight from New York?

```
flights |>
  distinct(origin, dest)
```



There are 224 origin and destination pairs. That is more than 105, because many destinations are served from more than one New York airport: Atlanta, for example, appears three times, once with each origin.

To count unique values, pipe the result into `nrow()`, as in `flights |> distinct(dest) |> nrow()`, which returns 105. If you want to keep the other columns, `distinct()` has an argument for that, `.keep_all = TRUE`, which keeps the first row it finds for each unique value.

8.3 Sorting rows with arrange()

8.3.1 Sorting by one column

`arrange()` sorts the rows of a data frame by the column you name. Add it to the end of the JFK question to list the destinations alphabetically:

```
flights |>
  filter(origin == "JFK") |>
  filter(air_time <= 180) |>
  distinct(dest) |>
  arrange(dest)
```

The same 50 rows now start with ACK (Nantucket), ATL (Atlanta), and AUS (Austin). `arrange()` changes the *order* of the rows, never how many there are. It sorts in **ascending order** by default: A to Z for text, smallest to largest for numbers.

8.3.2 Sorting every row

`arrange()` works on any data frame, not only on the result of `distinct()`. Take out `distinct()`, and `arrange()` sorts every one of the 64,248 short JFK flights by destination:

```
flights |>
  filter(origin == "JFK") |>
  filter(air_time <= 180) |>
  arrange(dest)
```

All 19 columns are still there, and the rows with the same `dest` now sit together, starting with the flights to Nantucket. Comparing this result with the last one shows that it was `distinct()`, not `arrange()`, that collapsed 64,248 flights into 50 destinations.

8.3.3 Largest first

To sort in **descending order**, largest first or Z to A, wrap the column in `desc()`. Which flights left latest?

```
flights |>
  arrange(desc(dep_delay)) |>
  select(month, day, carrier, origin, dest, dep_delay)
```



The longest departure delay in 2013 was 1,301 minutes, nearly 22 hours, on a flight from JFK to Honolulu on January 9. (The `select()` at the end keeps a few columns so the result fits on the screen. It is the subject of the next section.)

For a numeric column, a minus sign does the same job: `arrange(-dep_delay)` sorts from largest to smallest. The minus sign only works on numbers, though, while `desc()` also works on text, so `desc()` is the safer habit.

Missing values always go to the end, whichever direction you sort. The flights with no `dep_delay` (the cancelled flights) are at the bottom of both `arrange(dep_delay)` and `arrange(desc(dep_delay))`.

8.3.4 Sorting by two columns

What does this code do? Read it one step at a time, and predict the order of the Atlanta (`ATL`) rows before you run it.

```
flights |>
  distinct(origin, dest) |>
  arrange(dest, origin)
```

`distinct(origin, dest)` keeps the 224 unique pairs. `arrange(dest, origin)` sorts them by destination first. When two rows have the same destination, which is called a **tie**, it uses the second column, `origin`, to decide their order. All three New York airports fly to Atlanta, so `ATL` appears three times, and those three rows are sorted by origin: `EWR`, `JFK`, `LGA`. You can name as many columns as you like; each one breaks the ties left by the one before.

8.4 Keeping columns with select()

8.4.1 Naming the columns you want

`flights` has 19 columns, too many to see at once. `select()` keeps the columns you name, in the order you name them, and every row:

```
flights_dates <- flights |>
  select(year, month, day, dep_time)

glimpse(flights_dates)
```

The result has all 336,776 rows but only 4 columns. Storing it with `<-` under a new name, `flights_dates`, leaves the original `flights` unchanged.

`select()` and `filter()` are easy to mix up, because in everyday English you "select" the rows you want. Remember that **`filter()` picks rows** and **`select()` picks columns**.

8.4.2 Dropping and finding columns

Typing out every column you want gets tedious with wide data. `select()` has shortcuts (Table 8.2).



Table 8.2: Shortcuts for choosing columns with `select()`.

Code	What it keeps
`select(-tailnum)`	Every column except `tailnum`
`select(year:day)`	Every column from `year` to `day`, in the order they sit in the data frame
`select(starts_with("dep"))`	Columns whose names start with `dep`: `dep_time` and `dep_delay`
`select(ends_with("delay"))`	Columns whose names end with `delay`: `dep_delay` and `arr_delay`
`select(contains("time"))`	Columns whose names contain `time` anywhere

`starts_with()`, `ends_with()`, and `contains()` are **selection helpers**: functions that only work inside `select()` (and a few other tidyverse functions), and that pick columns by their names. You can combine any of these, separated by commas:

```
flights |>
  select(year:day, origin, dest, ends_with("delay"))
```

8.5 Making new columns with mutate()

8.5.1 Average speed

`mutate()` adds new columns to a data frame, calculated from the columns it already has. Its arguments take the form `new_name = calculation`. How fast was each flight? `distance` is in miles and `air_time` is in minutes, so distance divided by air time is miles per minute, and multiplying by 60 gives miles per hour:

```
flights |>
  mutate(avg_speed = distance / air_time * 60) |>
  select(origin, dest, distance, air_time, avg_speed)
```

`mutate()` did the calculation for every one of the 336,776 flights at once. That is because arithmetic in R is **vectorized**: dividing one column by another divides each value by the value in the same row, without a loop. The new column goes at the end of the data frame. Here, `select()` keeps it and the columns it was made from, so you can check the arithmetic: the first flight flew 1,400 miles in 227 minutes, or about 370 miles per hour.

The 9,430 flights with no `air_time` (most of them cancelled) get an `avg_speed` of NA, because any arithmetic with a missing value gives a missing value.

8.5.2 Several columns at once

Separate several new columns with commas. A later column can use one made earlier in the same `mutate()`:

```
flights |>
  mutate(
    hours = air_time / 60,
```



```
    avg_speed = distance / hours
  ) |>
  select(origin, dest, distance, hours, avg_speed)
```

This gives the same speeds as before, in two steps. If you give a new column the name of one that already exists, `mutate()` replaces the old column with the new values, so pick new names unless you mean to overwrite.

8.5.3 Slowest and fastest

Put `mutate()`, `arrange()`, and `select()` together to find the slowest flights:

```
flights |>
  mutate(avg_speed = distance / air_time * 60) |>
  arrange(avg_speed) |>
  select(origin, dest, avg_speed)
```

The slowest flight, from LaGuardia to Philadelphia, averaged about 77 miles per hour. Philadelphia is less than 100 miles from New York, so a flight there spends most of its time taking off, circling, and landing.

Change `arrange(avg_speed)` to `arrange(desc(avg_speed))` to find the fastest:

```
flights |>
  mutate(avg_speed = distance / air_time * 60) |>
  arrange(desc(avg_speed)) |>
  select(month, day, carrier, origin, dest, distance, air_time, avg_speed)
```

> **Think about it**
>
> The fastest flight averaged 703 miles per hour: a Delta flight from LaGuardia to Atlanta on May 25, which flew 762 miles in 65 minutes. A passenger jet usually cruises at about 550 to 580 miles per hour. The typical LaGuardia to Atlanta flight in 2013 took 112 minutes, at about 408 miles per hour.
> Do you believe it? A strong tailwind can push a plane's speed over the ground well above its cruising speed, but a 65-minute flight to Atlanta is still extraordinary. It could also be an error in the recorded air time. Either way, a value that is far from all the others, an **outlier**, deserves a second look before you report it. Sorting with `arrange()` is one of the quickest ways to find them.

8.6 Practice: the longest flight

Put it all together. These steps follow the in-class exercise.

Step 1: set up. Start a new R Markdown file, and load the packages in the first chunk:

```
library(tidyverse)
library(nycflights13)
```

Then take a quick look at the data with `glimpse(flights)`.



Step 2: the longest flight. What is the longest domestic flight from New York? Sort by `distance`, largest first, and keep the columns that answer the question. Try it before you look at the answer:

```
flights |>
  # sort, then select a few columns
  head()
```

Show the answer to step 2

```
flights |>
  arrange(desc(distance)) |>
  select(origin, dest, distance, air_time)
```

The longest flight is from JFK to Honolulu (HNL), 4,983 miles. Every one of the top rows is that same route, because it was flown every day. Add `distinct(origin, dest, distance)` after the `arrange()` to see the longest routes rather than the longest flights: the second is Newark to Honolulu, 4,963 miles.

Step 3: how fast was it? Add a `mutate()` that calculates `avg_speed`, and include it in the `select()`.

```
flights |>
  arrange(desc(distance)) |>
  # add mutate() here
  select(origin, dest, distance, air_time)
```

Show the answer to step 3

```
flights |>
  arrange(desc(distance)) |>
  mutate(avg_speed = distance / air_time * 60) |>
  select(origin, dest, distance, air_time, avg_speed)
```

The first flight in the list took 659 minutes, an average of about 454 miles per hour. But the next took 638 minutes and the one after 616, so the speeds differ. Over the year, JFK to Honolulu flights averaged between about 433 and 516 miles per hour. The distance is the same every day, so the difference is all in `air_time`. Why might the same route take an hour longer on one day than another? Think about the wind over the Pacific, and the route each flight was cleared to fly.

8.7 Check your understanding

1. Which verb sorts the rows of a data frame? [1]

- (A) filter()
- (B) arrange()

[1] arrange()

- (C) distinct()
- (D) select()

2. `arrange()` can remove rows from a data frame. TRUE / FALSE[2]
3. Complete the code to sort `flights` from the longest departure delay to the shortest: `flights |> arrange(___(dep_delay)) ___`[3]
4. `flights |> distinct(origin, dest)` returns: [4]

- (A) Every flight, sorted by origin and destination
- (B) One row for each unique pair of origin and destination
- (C) The three origin airports
- (D) The number of destinations

5. Which verb keeps some of the **columns** of a data frame? [5]

- (A) filter()
- (B) select()
- (C) mutate()
- (D) arrange()

6. What does `select(-tailnum)` do? [6]

- (A) Keeps only tailnum
- (B) Keeps every column except tailnum
- (C) Sorts by tailnum, largest first
- (D) Removes the rows where tailnum is missing

7. `mutate()` puts a new column at the end of the data frame. TRUE / FALSE[7]
8. `air_time` is in minutes. Complete the code to make a column of air time in hours: `mutate(hours = air_time ___ 60)`[8]

Show hint for question 4

With more than one column, `distinct()` keeps each unique *combination* of their values. It keeps only the columns you name.

[2] FALSE
[3] desc
[4] One row for each unique pair of origin and destination
[5] select()
[6] Keeps every column except tailnum
[7] TRUE
[8] /



8.8 Summary

- dplyr verbs work either on **rows** (`filter()`, `distinct()`, `arrange()`) or on **columns** (`select()`, `mutate()`). Each takes a data frame first and returns a data frame, so they chain with the pipe.
- **`distinct()`** keeps one row for each unique value of the columns you name, or each unique combination if you name several. It keeps only those columns.
- **`arrange()`** sorts rows, in ascending order by default. Wrap a column in `desc()` for descending order. With several columns, each one breaks the ties left by the one before. Sorting never adds or removes rows, and missing values go last.
- **`select()`** keeps the columns you name, in that order. `-` drops a column, `:` keeps a range, and `starts_with()`, `ends_with()`, and `contains()` pick columns by name. `filter()` picks rows; `select()` picks columns.
- **`mutate()`** adds new columns calculated from existing ones, with `new_name = calculation`. The calculation is done for every row at once. Arithmetic with NA gives NA.
- Sorting a new column with `arrange()` is a quick way to find **outliers**. Check them before you trust them.

8.9 Exercises

Use the `flights` data from nycflights13 for these exercises, and answer each with a pipe chain.

1. **Airlines.** How many different airlines (`carrier`) flew out of New York in 2013? Which airlines flew from LaGuardia to Raleigh–Durham (`"RDU"`)?
2. **Early birds.** Which flight left earliest compared with its scheduled time? (Hint: sort by `dep_delay`.) Which airport did it leave from?
3. **Two-column sort.** Sort `flights` by `month`, and then by `dep_delay` from largest to smallest within each month. What was the longest delay in June?
4. **Pick your columns.** Write three different `select()` calls that each return exactly the columns `dep_time`, `dep_delay`, `arr_time`, and `arr_delay`. (Hint: try `starts_with()`, `ends_with()`, and `contains()`, or a range with `:`.)
5. **Delay in hours.** Use `mutate()` to add a column `dep_delay_hours` that gives the departure delay in hours. Sort by it and keep only `origin`, `dest`, and `dep_delay_hours`. How many hours late was the latest flight?
6. **Making up time.** Make a column `gain = dep_delay - arr_delay`. What does a positive `gain` mean? Which flight made up the most time in the air?
7. **Fix the code.** Each chain below has one mistake. Find and fix it.

```
flights |> arrange(dest) |> distinct()        # wanted: one row per destination
flights |> select(origin == "JFK")
flights |> arrange(-dest)
flights |> mutate(avg_speed == distance / air_time * 60)
```



8.10 Further reading

- *R for Data Science* (2nd ed.) (Wickham et al. 2023), free at https://r4ds.hadley.nz, covers `arrange()`, `distinct()`, `select()`, and `mutate()` in its "Data transformation" chapter, with the same flights data.
- The dplyr website, https://dplyr.tidyverse.org, has a reference page for each verb. The page for `select()` lists all of its selection helpers.
- Posit's *Data transformation with dplyr* cheatsheet, https://posit.co/resources/cheatsheets/, shows `arrange()` and `distinct()` under **Manipulate Cases** and `select()` and `mutate()` under **Manipulate Variables**.



9 Into the Tidyverse: summarize, group_by, and drop_na

Learning objectives

By the end of this chapter you should be able to:

1. Collapse a data frame into summary statistics with `summarize()`.
2. Choose a summary function, such as `mean()`, `median()`, `max()`, or `n()`, for the question you are asking.
3. Calculate a statistic for each group of rows with `group_by()` and `summarize()`.
4. Handle missing values with `na.rm = TRUE` or `drop_na()`, and explain how the two differ.
5. Explain why the average of averages can mislead, and calculate a weighted average instead.

So far, every verb you have used gives back one row for each flight, or fewer flights, or the same flights in a new order. But many questions are about *many* flights at once. What was the average departure delay in 2013? Which month had the worst delays? Which plane was late most often? This chapter introduces the verbs that answer them: `summarize()`, which collapses many rows into a few numbers, and `group_by()`, which makes `summarize()` give one answer for each group. Along the way you will meet `drop_na()`, a third way to deal with the missing values you met two chapters ago.

The examples still use the **nycflights13** data, and the chapter ends with a question that looks simple but has a trap in it: how fast did planes fly each month?



Warm-up: why doesn't this code work?

This code is meant to find the fastest flights in October. Read the error message, then trace the code one line at a time.

```
library(dplyr)
library(tidyr)
library(nycflights13)

flights_new <- flights |>
  filter(month == 10)
  mutate(avg_speed = distance / air_time * 60) |>
  arrange(desc(avg_speed))
```

Show the answer

R says `object 'distance' not found`. The `filter()` line has no pipe at the end, so the chain stops there: `flights_new` is just the October flights. R then runs `mutate()` as a separate command, with no data frame, so it can't find a column called `distance`. The fix is to add `|>` at the end of the `filter()` line. The error message names the *symptom* (a missing object), not the *cause* (a missing pipe), so when a column "can't be found," check that every line but the last ends with `|>`.

9.1 Tidy data and the verbs so far

9.1.1 Tidy data

dplyr's verbs expect **tidy data**, which follows three rules (Wickham et al. 2023):

1. Each variable is a column.
2. Each observation is a row.
3. Each value is a cell.

`flights` is tidy. Each row is one flight, each column is one thing recorded about it, such as `dep_delay` or `carrier`, and each cell holds one value. Because every month is in the same column, and every delay in the same column, a single line of code can work on all of them at once. That is what makes this chapter's verbs possible.

9.1.2 Three new verbs

Table 9.1 adds this chapter's verbs to the ones you already know.

Table 9.1: The dplyr verbs so far, and this chapter's three new ones.

Works on	Verb	What it does
Rows	`filter()`	Keeps the rows that meet a condition
Rows	`arrange()`	Sorts the rows
Rows	`distinct()`	Keeps one row for each unique value
Rows	`group_by()`	Groups the rows that share a value (*new*)



Works on	Verb	What it does
Rows	`drop_na()`	Removes the rows with missing values (*new*)
Columns	`select()`	Keeps the columns you name
Columns	`mutate()`	Adds new columns
Columns	`summarize()`	Collapses many rows into summary columns (*new*)

`drop_na()` comes from the **tidyr** package rather than dplyr, but `library(tidyverse)` loads both, so you don't need to do anything extra. The live code boxes in this chapter load dplyr and tidyr separately, because that is faster in a web browser.

9.2 Summarizing with summarize()

9.2.1 One number from many rows

How late, on average, did flights leave New York in 2013? `summarize()` collapses a whole data frame into a single row of **summary statistics**. Inside it, name the new column and say how to calculate it:

```
flights |>
  summarize(delay = mean(dep_delay, na.rm = TRUE))
```

The answer is one row and one column: across all 336,776 flights, the average departure delay was about 12.6 minutes. `delay` is the name we chose for the result; you can call it anything. `mean(dep_delay)` is the calculation.

`na.rm = TRUE` ("remove NAs") tells `mean()` to skip the missing values. Without it, the answer is NA, because the 8,255 cancelled flights have no departure delay, and any calculation that includes an unknown number gives an unknown answer. Try it: delete `, na.rm = TRUE` and run the box again.

`summarize()` is also spelled `summarise()`. The two are the same function; use whichever spelling you like.

9.2.2 Summary functions

The calculation inside `summarize()` can be any function that takes a column and returns one number (Table 9.2).

Table 9.2: Functions to use inside `summarize()`, with their results for departure delays in 2013. All but `n()` need `na.rm = TRUE` for `dep_delay`.

Function	What it returns	`dep_delay` in 2013
`mean()`	The average	12.6 minutes
`median()`	The middle value: half are above, half below	–2 minutes
`min()`, `max()`	The smallest and largest values	–43 and 1,301 minutes
`quantile(x, 0.25)`	The value below which a given share falls; here the 25th percentile	–5 minutes



Function	What it returns	`dep_delay` in 2013
`sd()`	The standard deviation: how spread out the values are	40.2 minutes
`var()`	The variance, the standard deviation squared	1,617
`sum()`	The total	
`n()`	The number of rows	336,776

`n()` is different from the others: it takes no arguments, and it counts rows rather than looking at a column. It only works inside dplyr verbs such as `summarize()`.

Look at the first two rows of the table. The **mean** delay is 12.6 minutes, but the **median** is –2 minutes: more than half of all flights left *early*. How can both be true? Delays are **skewed**. Most flights leave a few minutes early or late, but a few leave hours late, and those few pull the mean up a long way. The median only cares about the middle flight, so it ignores them. When a variable has a long tail like this, report the median as well as the mean, or instead of it.

9.2.3 Several statistics at once

Separate several statistics with commas. Each becomes its own column, still in a single row:

```
flights |>
  summarize(
    max_delay = max(dep_delay, na.rm = TRUE),
    min_delay = min(dep_delay, na.rm = TRUE),
    n_flights = n()
  )
```

The longest delay was 1,301 minutes, almost 22 hours. The shortest is negative: one flight left 43 minutes before its scheduled time. One maximum for the whole year is a start, but it would be more useful to know the longest delay in each month. That is the job of `group_by()`.

9.3 One result per group with group_by()

9.3.1 Grouping, then summarizing

`group_by()` sorts the rows into groups that share a value. On its own it changes nothing you can see. But every verb after it then works on each group separately. Put `group_by(month)` in front of the `summarize()` from the last section:

```
flights |>
  group_by(month) |>
  summarize(
    max_delay = max(dep_delay, na.rm = TRUE),
    min_delay = min(dep_delay, na.rm = TRUE)
  )
```

The `summarize()` code is exactly the same, but now there are 12 rows, one per month, instead of one. `group_by(month)` split the 336,776 flights into 12 groups, and `summarize()` ran



once for each. The grouping column, `month`, stays in the result so you can tell the rows apart. The longest delay of the year was in January, and the longest in June was 1,137 minutes.

Which month had the worst delays on average? Group by month, take the mean, and sort:

```
flights |>
  group_by(month) |>
  summarize(
    mean_delay = mean(dep_delay, na.rm = TRUE),
    n_flights = n()
  ) |>
  arrange(desc(mean_delay))
```

July (21.7 minutes) and June (20.8) were the worst; October and November, at around 6 minutes, were the best. Summer thunderstorms are a likely suspect. `n()` counts the flights in each group, here between about 25,000 and 29,000 a month.

You can group by any column, and by more than one: `group_by(origin)` gives one row per airport, and `group_by(origin, month)` gives one row for each airport in each month, 36 rows in all.

9.3.2 group_by() on its own

What does `group_by()` do without `summarize()`?

```
flights |>
  group_by(month) |>
  glimpse()
```

Nothing seems to change: there are still 336,776 rows and 19 columns, with the same values. The only difference is a new line at the top, `Groups: month [12]`. The data frame is now a **grouped data frame**: it carries a hidden note saying that its rows belong to 12 groups by month.

That note matters, because every verb that comes later works *within* each group, until you remove it with `ungroup()`. That is what you want right before `summarize()`, but it can surprise you later in a chain. For example, `mutate(mean_delay = mean(dep_delay, na.rm = TRUE))` on a grouped data frame gives each flight the mean delay *for its month*, not for the year. When you have finished working by group, add `ungroup()`:

```
flights |>
  group_by(month) |>
  mutate(month_mean = mean(dep_delay, na.rm = TRUE)) |>
  ungroup()
```

`summarize()` removes one level of grouping for you, so after `group_by(month) |> summarize(...)` the result is no longer grouped. After `group_by(origin, month)`, it is still grouped by `origin`, and dplyr prints a message to say so.



9.4 Removing missing values with drop_na()

9.4.1 Two ways to skip NAs

You have now seen `na.rm = TRUE` many times. `drop_na()`, from tidyr, is another way to deal with missing values: it removes every row where a column you name is NA. These two chains give exactly the same 12 rows:

```
# Skip the NAs inside each calculation
flights |>
  group_by(month) |>
  summarize(
    max_delay = max(dep_delay, na.rm = TRUE),
    min_delay = min(dep_delay, na.rm = TRUE)
  )

# Remove the rows with NAs first
flights |>
  group_by(month) |>
  drop_na(dep_delay) |>
  summarize(
    max_delay = max(dep_delay),
    min_delay = min(dep_delay)
  )
```

In the second, `drop_na(dep_delay)` removes the 8,255 flights with no departure delay before `summarize()` sees them, so `max()` and `min()` never meet an NA and don't need `na.rm`.

9.4.2 They are not the same thing

Here the results match, but the two approaches do different things (Table 9.3).

Table 9.3: How `na.rm = TRUE` and `drop_na()` differ.

	`na.rm = TRUE`	`drop_na()`
Where it goes	Inside one summary function	As its own step in the pipe chain
What it changes	Only that one calculation	The data: whole rows are removed for every later step
Columns affected	The one being summarized	All columns, because whole rows go

The difference shows up when you count. `n()` counts rows, whether or not they have missing values. With `na.rm = TRUE`, `n()` still counts the cancelled flights; after `drop_na(dep_delay)`, it doesn't. Neither is wrong, but they answer different questions ("how many flights were scheduled?" against "how many flights have a delay recorded?"), so choose on purpose.



9.4.3 Name your columns

`drop_na()` with no columns named removes every row that has an NA in *any* column. That can remove more than you meant to:

```
flights |> drop_na(dep_delay) |> nrow()
flights |> drop_na() |> nrow()
```

`drop_na(dep_delay)` keeps 328,521 flights. `drop_na()` keeps only 327,346, because it also removes flights that departed but have no arrival delay or air time recorded. Always name the columns your question needs.

9.5 Practice: how fast did planes fly each month?

9.5.1 The obvious way

Here is a question that combines everything so far: what was the average air speed of flights from New York in each month? A first attempt might calculate the speed of every flight with `mutate()`, then take the mean for each month:

```
flights |>
  group_by(month) |>
  drop_na(distance, air_time) |>
  mutate(speed = distance / air_time * 60) |>
  summarize(avg_speed = mean(speed))
```

That runs without errors and gives one speed per month: about 370 miles per hour in January. But is it the right answer?

9.5.2 The average of averages

The problem isn't the code; it is the statistics. `mean(speed)` gives every flight an equal say. A half-hour hop to Philadelphia counts as much as a 6-hour flight to San Francisco. Short flights spend most of their time climbing and descending, so they are slow, and there are a lot of them, so they drag the average down.

What does "the average speed in January" really ask for? A good definition is the total distance flown in January, divided by the total time spent in the air:

$$\text{speed}_{\text{month}} = \frac{\sum \text{distance}_{\text{month}}}{\sum \text{air_time}_{\text{month}}} \times 60$$

This is a **weighted average** of the flights' speeds. Each flight counts in proportion to its air time, so a 6-hour flight counts twelve times as much as a 30-minute one.



9.5.3 Sum first, then divide

To calculate it, add up the distances and air times in each month with `summarize()`, and *then* divide the totals with `mutate()`:

```
flights |>
  group_by(month) |>
  drop_na(distance, air_time) |>
  summarize(
    distance = sum(distance),
    air_time = sum(air_time)
  ) |>
  mutate(speed = distance / air_time * 60)
```

The order matters: `summarize()` runs first to get one total per month, and `mutate()` then works on those 12 rows. In January the speed is 394 miles per hour, against 370 from averaging the flights' speeds, and the weighted speed is higher in every month. Long flights, which spend most of their time at cruising speed, now count in proportion to the time they spend in the air.

Here `drop_na(distance, air_time)` does more than clean up. Every cancelled flight still has a `distance`, because the route is known, but no `air_time`. If you used `sum(air_time, na.rm = TRUE)` without dropping those rows, the total distance would include miles that were never flown, and January's speed would come out at about 401 miles per hour, too fast. Dropping the rows keeps the distances and the times about the same flights.

9.6 Practice: the least punctual plane

These steps follow the in-class exercise. Every plane has a tail number, in the `tailnum` column. Which plane had the worst on-time record, meaning it left late the most times?

Step 1: set up. Start a new R Markdown file, and load the packages and look at the data:

```
library(tidyverse)
library(nycflights13)

glimpse(flights)
```

Step 2: decide what "late" means. This is a choice, not a fact in the data. Is a flight late if it leaves one minute after its scheduled time (`dep_delay > 0`)? Or if it arrives more than 15 minutes late (`arr_delay > 15`)? The US Department of Transportation counts a flight as late if it arrives 15 minutes or more after its scheduled time (`arr_delay >= 15`). Pick one and write it down.

Step 3: count the late flights for each plane. Keep the late flights with `filter()`, group them by plane, count each group with `n()`, and sort. Try it first:

```
flights |>
  filter(dep_delay > 0)
  # group, summarize, and sort
```



Show the answer to step 3

```
flights |>
  filter(dep_delay > 0) |>
  group_by(tailnum) |>
  summarize(n_late = n()) |>
  arrange(desc(n_late))
```

N258JB, a JetBlue plane, left late 186 times in 2013. If you define late as `arr_delay > 15` instead, N258JB still comes first, with 121 late arrivals. But with the Department of Transportation's rule, `arr_delay >= 15`, a different plane wins: N725MQ, with 126 late arrivals to N258JB's 123. Changing > to >= changed the answer, which is why step 2 matters.

Step 4: be fair to busy planes. N258JB flew 427 times in 2013, so of course it was late a lot: it flew a lot. A fairer measure is the *share* of each plane's flights that left late. Count all the flights and the late ones for each plane, then divide:

```
flights |>
  group_by(tailnum) |>
  summarize(
    n_flights = n(),
    n_late = sum(dep_delay > 0, na.rm = TRUE)
  ) |>
  mutate(share_late = n_late / n_flights) |>
  arrange(desc(share_late))
```

`sum()` of a logical vector counts the TRUEs, because R treats TRUE as 1 and FALSE as 0. Look at the top of the result. Planes that flew once or twice, and were late once, have a share of 1, which tells you nothing. Add `filter(n_flights >= 100)` before the `arrange()` to keep the planes with a decent record. Also look for a row where `tailnum` is NA: 2,512 flights, all of them cancelled, have no tail number, and `group_by()` puts them together in one group. `drop_na(tailnum)` removes it.

9.7 Check your understanding

1. `flights |> summarize(delay = mean(dep_delay, na.rm = TRUE))` returns how many rows? [1]

- (A) 1
- (B) 12
- (C) 105
- (D) 336,776

2. Without `na.rm = TRUE`, `mean(dep_delay)` returns NA for the `flights` data. TRUE / FALSE[2]

[1] 1
[2] TRUE

3. Which function counts the rows in each group, and takes no arguments? ___[3]
4. The mean departure delay is 12.6 minutes but the median is –2 minutes. Why? [4]

- (A) One of them was calculated wrongly
- (B) A few very long delays pull the mean up, but not the median
- (C) The median ignores negative values
- (D) The mean ignores missing values

5. `flights |> group_by(month)` on its own returns 12 rows. TRUE / FALSE[5]
6. `flights |> group_by(origin) |> summarize(n = n())` returns how many rows? [6]

- (A) 1
- (B) 3
- (C) 12
- (D) 105

7. `drop_na()` with no columns named removes the rows that have an NA in: [7]

- (A) The first column
- (B) The column you last used
- (C) Any column
- (D) No columns

8. To find the average speed for a month, it is better to divide the total distance by the total air time than to take the mean of the flights' speeds. TRUE / FALSE[8]

Show hint for question 5

`group_by()` doesn't change the rows. It only notes which group each row is in. It takes `summarize()` to collapse each group into one row.

9.8 Summary

- dplyr's verbs work on **tidy data**: each variable a column, each observation a row, each value a cell.
- **`summarize()`** collapses a data frame into one row of summary statistics, with `name = function(column)`. Separate several statistics with commas.

[3] n, n()
[4] A few very long delays pull the mean up, but not the median
[5] FALSE
[6] 3
[7] Any column
[8] TRUE



- Useful summary functions include `mean()`, `median()`, `min()`, `max()`, `quantile()`, `sd()`, `var()`, `sum()`, and `n()`. Skewed data, like flight delays, can have a mean far from the median.
- **`group_by()`** splits the rows into groups. On its own it changes nothing you can see, but `summarize()` after it gives one row per group. Use `ungroup()` when you have finished working by group.
- **`na.rm = TRUE`** skips missing values in one calculation. **`drop_na()`** removes whole rows with missing values from the data. Always name the columns in `drop_na()`.
- The average of averages gives every row an equal say. To find a rate such as speed for a group, sum first, then divide: that is a **weighted average**.

9.9 Exercises

Use the `flights` data from nycflights13 for these exercises, and answer each with a pipe chain.

1. **Busiest airport.** Use `group_by()` and `summarize()` with `n()` to count the flights from each of the three New York airports. Then try `flights |> count(origin)`. What does `count()` do?
2. **Most popular destinations.** Which three destinations had the most flights from New York in 2013?
3. **Delays by airport.** For each `origin`, calculate the mean and median departure delay. Which airport had the worst delays? Do the mean and the median agree?
4. **Arrivals.** Calculate the mean arrival delay (`arr_delay`) for each airline (`carrier`), and sort from worst to best. How many flights does each airline have? Would you trust the average for an airline with very few flights?
5. **Two groups.** Group by `origin` and `month` and count the flights. Which airport was busiest in which month? Is the result still grouped?
6. **Counting NAs.** Without using `drop_na()`, count how many flights in each month have a missing `dep_delay`. (Hint: `sum(is.na(dep_delay))`.) Which month had the most cancelled flights?
7. **Fix the code.** Each chain below has one mistake. Find and fix it.

```
flights |> summarize(mean(dep_delay))          # result is NA
flights |> group_by(month) summarize(n = n())
flights |> summarize(n = n(month))
flights |> group_by(month) |> summarize(delay = mean(dep_delay, na.rm = TRUE) |>
```

9.10 Further reading

- *R for Data Science* (2nd ed.) (Wickham et al. 2023), free at https://r4ds.hadley.nz. The "Data transformation" chapter covers `group_by()` and `summarize()` with the flights data, and the "Data tidying" chapter explains tidy data.
- The tidyr website, https://tidyr.tidyverse.org, has the reference page for `drop_na()`.
- Posit's *Data transformation with dplyr* cheatsheet, https://posit.co/resources/cheatsheets/, shows `summarize()`, `group_by()`, and the summary functions under **Summarize Cases** and **Group Cases**.

10 Into the Tidyverse: Graphs with ggplot2

Learning objectives

By the end of this chapter you should be able to:

1. Explain the grammar of graphics: data, aesthetic mappings, and geoms.
2. Build scatterplots, bar charts, and boxplots with `ggplot()`.
3. Tell *setting* an aesthetic from *mapping* one.
4. Split a plot into small multiples with `facet_wrap()` and `facet_grid()`.
5. Add labels and your own colors, and save a plot with `ggsave()`.

In the last three chapters you used dplyr to filter, sort, and summarize the **nycflights13** data. Tables of numbers answer precise questions, but they are a poor way to *see* a pattern. Do flights that leave late also arrive late? Which airline flies the longest routes? Is there a weekly rhythm to the number of flights? A graph answers each of these at a glance.

This chapter introduces **ggplot2**, the tidyverse package for graphs. Everything in it builds on one short template, and the same ideas carry straight over to the maps in the next chapter.

Warm-up: does this code run?

Predict what happens, then run it.

```
library(ggplot2)
library(dplyr)
library(tidyr)
library(nycflights13)

flights -> flights.new
glimpse(flights.new)
```

Show the answer

Yes, it runs. `->` is the assignment arrow pointing the other way: it stores `flights` in a new object called `flights.new`. It is legal R, but uncommon. The tidyverse style guide recommends `<-`, so that the name of the object you are making is always on the left, where readers look for it.
`glimpse()` prints one line per column, with its type (`<int>`, `<dbl>`, `<chr>`) and the first few values. It is a quick check, before you plot anything, that the data loaded and that the columns have the types you expect.



10.1 The grammar of graphics

10.1.1 ggplot2 expects tidy data

ggplot2 is built for **tidy data**, the same shape dplyr produces: each variable is a column, each observation is a row, and each value is a cell. Every variable you want on an axis, or as a color, or to split a plot by, must be its own column. `flights` is already tidy: one row per flight and one column per thing recorded about it. If a variable is spread over several columns (say, one column per year), you would reshape it first with `pivot_longer()` from tidyr.

10.1.2 What is ggplot2?

ggplot2 is part of the tidyverse, so `library(tidyverse)` loads it. The package is called ggplot2, but its main function is `ggplot()`. It was written by Hadley Wickham, based on Leland Wilkinson's book *The Grammar of Graphics*, and it has three things going for it:

- A readable graph takes very little code, because the defaults are good.
- You can still change every part of the graph when you need to.
- Graphs are built in **layers**, joined with +, so you can start simple and add one piece at a time.

10.1.3 Every graph has the same parts

The **grammar of graphics** is the idea that any chart, from a scatterplot to a map, can be described with the same small set of parts (Table 10.1).

Table 10.1: The parts of the grammar of graphics.

Part	What it is	In ggplot2
Data	The tidy table you plot	`ggplot(data = flights)`
Aesthetics	Visual properties, such as x position, y position, color, or size, that a column is shown with	`aes(x = dep_delay, y = arr_delay)`
Geoms	The marks that show the data: points, bars, lines, boxes	`geom_point()`, `geom_bar()`
Scales	How data values turn into visual values: which colors, which axis breaks	`scale_fill_manual()`, `scale_x_log10()`
Facets	Splitting the data into small plots, one per group	`facet_wrap()`
Coordinates	Where the marks land on the page	`coord_flip()`, or a map projection

The key word is **aesthetic**: a visual property of a mark. `aes()` is where you connect the columns in your data to those properties. A bar chart and a scatterplot can share the same data and the same coordinates; only the **geom** changes.



10.1.4 Choosing a geom

ggplot2 has more than 40 geoms. To pick one, first ask what kinds of variables you are plotting (Table 10.2). The answer narrows it down to two or three choices.

Table 10.2: Choosing a geom from the kinds of variables.

You are plotting	Try
One continuous variable	`geom_histogram()`, `geom_density()`
One categorical variable	`geom_bar()`
Two continuous variables	`geom_point()`, `geom_smooth()`
A categorical x and a continuous y	`geom_boxplot()`, `geom_col()`
Change over time	`geom_line()`

Posit's ggplot2 cheatsheet groups all the geoms this way. In RStudio, find it under **Help > Cheat Sheets**.

10.1.5 The template

Every ggplot follows the same template:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

Swap in a data frame, a geom function, and the mappings, and you have a plot. Building one takes five steps, but only the first three are required:

1. Start with tidy data: `ggplot(data = flights)`.
2. Map variables to aesthetics: `aes(x = dep_delay, y = arr_delay)`.
3. Add a geom to draw them: `+ geom_point()`.
4. Adjust the scales and coordinates, for example `+ scale_x_log10()`.
5. Add labels: `+ labs(title = "...")`.

Two rules trip people up. First, the + must go at the *end* of a line, not the start of the next one; otherwise R thinks the command has ended. Second, ggplot2 uses + to add layers, while dplyr uses |> to pass data along. You can pipe data *into* `ggplot()`, but after that, every layer is joined with +.

The argument names `data =` and `mapping =` are optional, so `ggplot(flights) + geom_point(aes(x = dep_delay, y = arr_delay))` works too. Most code you find online leaves them out.

10.2 Scatterplots and aesthetics

10.2.1 A first scatterplot

Do flights that leave late arrive late? To keep the plot readable, start with the October flights, and drop the cancelled ones, which have no delay values:



```
# Filter for October flights and remove missing values
october_flights <- flights |>
  filter(month == 10) |>
  drop_na(dep_delay, arr_delay)

# Plot departure vs. arrival delays
ggplot(data = october_flights) +
  geom_point(mapping = aes(x = dep_delay, y = arr_delay))
```

Each point is one October flight. The points form a rising band: flights that leave late arrive late, by about the same amount. Without `drop_na()`, the plot would look the same, but ggplot2 would warn that it had removed rows with missing values.

What does a point far *above* the band mean? A flight that arrived much later than its departure delay predicts, perhaps because it was held in the air or waited a long time on the runway.

10.2.2 Adding a trend line

`geom_smooth()` draws a trend line. Layer it on top of the points with another +:

```
ggplot(data = october_flights,
       mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point() +
  geom_smooth()
```

Two things changed. The mapping moved into `ggplot()`. Mappings set there apply to *every* layer, so `geom_point()` and `geom_smooth()` both use the same x and y, and you don't have to repeat them. And the layers draw in order, so the line sits on top of the points.

By default, `geom_smooth()` fits a smooth curve, with a gray band showing its 95% **confidence interval**. R prints a message saying which method it used. For a straight regression line, use `geom_smooth(method = "lm")`, and to hide the band, add `se = FALSE`.

10.2.3 Setting an aesthetic

To make the points blue and the line red, give each geom a color *outside* `aes()`:

```
ggplot(data = october_flights,
       mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point(color = "blue") +
  geom_smooth(color = "red", linewidth = 2)
```

This is **setting** an aesthetic: every mark gets the same fixed value, whatever the data say. It is a design choice, not a data choice. Each geom takes its own settings, so here the points are blue, and the line is red and thicker (`linewidth = 2`).

Note the quotes around `"blue"` and `"red"`. R knows 657 color names (run `colors()` to see them all), and it also takes **hex codes** such as `"#2C5080"`.

10.2.4 Mapping an aesthetic

Now give each origin airport its own color. Move `color` *inside* `aes()`, and give it a column instead of a color:

```
ggplot(data = october_flights,
       mapping = aes(x = dep_delay, y = arr_delay, color = origin)) +
  geom_point() +
  geom_smooth(linewidth = 0.5)
```

This is **mapping** an aesthetic: the color now comes from the data. ggplot2 picks one color for each value of `origin`, and builds the **legend** for you. Because `color` is mapped in `ggplot()`, `geom_smooth()` also splits by airport, and draws three trend lines.

Table 10.3 sums up the difference. It is the most important idea in this chapter.

Table 10.3: Setting an aesthetic against mapping one.

	Setting	Mapping
Where	In the geom, outside `aes()`	Inside `aes()`
Value	A fixed value: `color = "blue"`	A column: `color = origin`
Result	Every mark looks the same	Marks differ by the data, with a legend

A common mistake

What happens if you write `aes(color = "blue")`? Inside `aes()`, ggplot2 treats `"blue"` as a data value, as if every row had a column whose value was the word "blue". It makes a one-item legend called *blue*, and colors the points with its first default color, which is a salmon pink. Try it in the box above. If your points come out the wrong color with a strange legend, check whether a setting has slipped inside `aes()`.

Think about it

Move `color = origin` out of `ggplot(aes())` and into `geom_point(aes())` instead. Before you run it, predict: how many trend lines will you get now? Why?

10.2.5 Too many colors

Try mapping color to the airline, `carrier`. With all 16 airlines, the colors become hard to tell apart, so this code keeps six large ones first:

```
# Compare delays across major carriers
october_flights |>
  filter(carrier %in% c("AA", "US", "UA", "DL", "B6", "WN")) |>
  ggplot() +
  geom_point(mapping = aes(x = dep_delay, y = arr_delay, color = carrier))
```



The data flow into `ggplot()` through the pipe, and from there on the layers are joined with `+`. Even with six carriers, the points pile on top of each other. Color works best for about five groups or fewer. Facets, later in this chapter, are a better fix.

10.3 Bar charts and boxplots

10.3.1 Counting with geom_bar()

How many October flights did each airline fly? A **bar chart** compares counts across categories:

```
flights |>
  filter(month == 10) |>
  ggplot() +
  geom_bar(mapping = aes(x = carrier))
```

You only map `x`. `geom_bar()` counts the rows for each carrier itself, the same as `count(carrier)` would, and plots the counts as the bar heights. If you already have a table of counts, for example from `group_by()` and `summarize()`, use `geom_col(aes(x = carrier, y = n))` instead, which draws the heights you give it.

United (UA), ExpressJet (EV), and JetBlue (B6) flew the most, with about 5,000 flights each. Hawaiian (HA) flew just 21. The carrier codes are hard to read; the `airlines` table in nycflights13 has the full names, which you could join in.

10.3.2 fill and color

For bars, boxes, and polygons, **`fill`** colors the inside, and `color` only the outline. (For points and lines, `color` is the one to use.) Map `fill` to the carrier:

```
# Count of flights per carrier in October
october_flights |>
  ggplot() +
  geom_bar(mapping = aes(x = carrier, fill = carrier))
```

Each bar now has its own color and a legend entry. But the legend only repeats the x axis, so it adds nothing. Hide it with `show.legend = FALSE` inside `geom_bar()`. Color earns its place when it shows a *different* variable from x.

10.3.3 Stacked bars

Map `fill` to a different variable from `x`. Here x is the day of the month, and the colors show which of five carriers flew that day:

```
october_flights |>
  filter(carrier %in% c("AA", "B6", "DL", "UA", "WN")) |>
  ggplot() +
  geom_bar(mapping = aes(x = day, fill = carrier))
```



Each bar splits into colored segments, one per carrier, and the bar's height is still the total number of flights that day. This is a **stacked bar chart**. The regular dips are Saturdays, when airlines schedule fewer flights.

Stacking makes the totals easy to read, but the middle segments are hard to compare, because they don't sit on a shared baseline. The `position` argument gives two alternatives:

```
five_carriers <- october_flights |>
  filter(carrier %in% c("AA", "B6", "DL", "UA", "WN"))

# Every bar is 100%: compare shares
ggplot(five_carriers) +
  geom_bar(aes(x = day, fill = carrier), position = "fill") +
  labs(y = "Share of flights")

# Bars side by side: compare counts
ggplot(five_carriers) +
  geom_bar(aes(x = day, fill = carrier), position = "dodge")
```

`position = "fill"` stretches every bar to the same height, so you compare *proportions*. The y axis would still say "count", although it now shows a share, which is why the code relabels it with `labs()`. `position = "dodge"` puts each day's bars next to each other on a shared baseline, so you can compare counts directly. With 31 days and five carriers it gets crowded; dodging works best with only a few categories on x.

10.3.4 Boxplots

Which airlines fly the longest routes? A **boxplot** compares a continuous variable across categories. Map a categorical x and a continuous y:

```
# Distribute flight distances across major carriers
october_flights |>
  filter(carrier %in% c("AA", "B6", "DL", "UA", "WN")) |>
  ggplot(mapping = aes(x = carrier, y = distance)) +
  geom_boxplot()
```

A boxplot summarizes each group in five numbers. The box runs from the 25th to the 75th **percentile**, so it holds the middle half of the flights. The thick line is the **median**. The whiskers reach the furthest points within 1.5 times the height of the box, and anything beyond them is drawn as a dot: an **outlier**.

United (UA) flies the longest routes, with a median of about 1,400 miles, and its outliers near 5,000 miles are its flights from Newark to Honolulu. Southwest (WN) flies the shortest, with a median of about 750 miles. A boxplot hides the shape of the data inside the box; `geom_violin()` is an alternative that shows it.



10.4 Small multiples with facets

10.4.1 facet_wrap()

Sixteen colors on one chart are impossible to tell apart. A better idea is **small multiples**: one small chart per group, all with the same axes. In ggplot2 this is called **faceting**:

```
flights |>
  ggplot(aes(x = dep_delay, y = arr_delay)) +
  geom_point(alpha = 0.3) +
  facet_wrap(~ carrier)
```

`facet_wrap(~ carrier)` draws one panel per carrier. The tilde, `~`, makes a *formula*: read `~ carrier` as "split by carrier". The panels share their axes, so you can compare them directly. `nrow` or `ncol` sets the shape of the grid, as in `facet_wrap(~ carrier, ncol = 4)`.

`alpha = 0.3` makes each point 70% transparent, so places where many points overlap look darker, and you can see where most flights fall. Look at the panel for HA (Hawaiian Airlines): it is nearly empty, because Hawaiian flew only one flight a day from New York. You would never have spotted that in a single chart with 16 colors.

10.4.2 facet_grid()

To split by *two* categorical variables, use `facet_grid(rows ~ columns)`:

```
flights |>
  ggplot(aes(x = distance, y = air_time)) +
  geom_point(alpha = 0.3) +
  facet_grid(origin ~ month)
```

This makes 3 rows (one per airport) by 12 columns (one per month): 36 panels. Read across a row to follow one airport through the year, or down a column to compare the airports in one month. At this size the axis labels overlap; fixes include filtering to a few months, or setting fewer breaks with `scale_x_continuous(breaks = c(0, 1000))`. `facet_grid(~ month)` makes columns only.

Use `facet_wrap()` for one variable with many levels, and `facet_grid()` when you want to cross two variables.

10.5 Finishing a plot

10.5.1 Choosing your own colors

Every mapped aesthetic has a **scale**, which controls how data values turn into visual values. To pick the colors yourself, replace the default fill scale with `scale_fill_manual()`:

```
# Custom discrete colors manually
ggplot(data = october_flights) +
  geom_bar(mapping = aes(x = origin, fill = origin)) +
```



```
  scale_fill_manual(values = c("EWR" = "#3182bd",
                               "JFK" = "#e6550d",
                               "LGA" = "#31a354"))
```

Naming each value (`"EWR" = ...`) guarantees each airport keeps its color, even if the order of the data changes. Match the scale to the aesthetic: `scale_fill_manual()` for fills, `scale_color_manual()` for points and lines.

The six-character codes are **hex codes**: two characters each for the red, green, and blue in the color. Choosing colors that work for everyone is harder than it looks. About one man in twelve has some form of color blindness. Websites such as https://colorbrewer2.org suggest sets of colors that work for color-blind readers and in grayscale, and `scale_fill_viridis_d()` uses the viridis palette, which is designed for both.

10.5.2 Adding labels with labs()

A column name such as `dep_delay` means something to you, but not to your reader. `labs()` sets every label in one call:

```
ggplot(data = october_flights) +
  geom_point(mapping = aes(x = dep_delay, y = arr_delay, color = origin),
             alpha = 0.5) +
  labs(
    title = "NYC Flight Delay Comparison",
    subtitle = "October flights departing from JFK, LGA, and EWR",
    x = "Departure delay (minutes)",
    y = "Arrival delay (minutes)",
    color = "Origin airport",
    caption = "Source: nycflights13 package"
  )
```

The argument names match the aesthetics: `x`, `y`, `color`, `fill`. Here `color = "Origin airport"` renames the legend, because color is the mapped aesthetic. A good habit for every plot you hand in: a title, axis labels with units, and the data source in the caption.

10.5.3 Themes

A **theme** controls everything that isn't data: the background, the grid lines, the fonts. Add one as another layer. `theme_minimal()` removes the gray background, and `theme_bw()` and `theme_classic()` are worth trying too:

```
ggplot(data = october_flights) +
  geom_point(mapping = aes(x = dep_delay, y = arr_delay, color = origin)) +
  theme_minimal()
```

10.5.4 Saving a plot with ggsave()

You could export a plot from RStudio's Plots pane, but then it comes out at whatever size the pane happens to be. `ggsave()` makes the size and resolution part of your code, so the figure is the same every time you knit:



```
# Create your plot
ggplot(data = october_flights) +
  geom_point(mapping = aes(x = dep_delay, y = arr_delay, color = origin)) +
  theme_minimal()

# Save it as a high-resolution PNG file
ggsave(filename = "nyc_flights_delays_october.png",
       width = 8,
       height = 6,
       dpi = 300)   # good for print or presentations
```

`ggsave()` saves the last plot you made. `width` and `height` are in inches, and `dpi` is the resolution in dots per inch: `300` is print quality. The file extension sets the type: `.png`, `.pdf`, `.svg`, or `.jpg`. The file goes into your working directory, which for an R Markdown file is the folder the `.Rmd` file is in. To save a plot that isn't the last one, store it first, `p <- ggplot(...)`, and then `ggsave("file.png", plot = p)`.

10.6 Practice: June flights

These steps follow the in-class exercise.

Step 1: set up. Start a new R Markdown file, load the packages, and look at the data:

```
library(nycflights13)
library(tidyverse)

glimpse(flights)

flights |>
  distinct(carrier)
```

Step 2: plot. Graph `air_time` against `distance` for the June flights. Add a trend line, color the points by origin airport, and label the axes with units. Try it first:

```
flights |>
  filter(month == 6)
  # plot air_time against distance
```



Show the answer to step 2

```
flights |>
  filter(month == 6) |>
  ggplot(aes(x = distance, y = air_time)) +
  geom_point(aes(color = origin), alpha = 0.3) +
  geom_smooth() +
  labs(x = "Distance (miles)", y = "Air time (minutes)",
       color = "Origin airport",
       title = "Air time against distance, June 2013")
```

The points fall close to a straight line, because air time grows with distance at roughly cruising speed. Points far off the line are worth asking about. Here `color` is mapped in `geom_point()` only, so there is a single trend line for all three airports.

Step 3: challenge. Which plane (`tailnum`) has the worst on-time record? This is a dplyr question, not a plotting one: you answered a version of it at the end of the last chapter. Once you have a table of planes with their share of late flights, try plotting it: a histogram of `share_late` shows whether a few planes are unusually bad, or whether lateness is spread across the whole fleet.

10.7 Check your understanding

1. In ggplot2, which function connects columns in your data to visual properties such as x, y, and color? ____[1]
2. `geom_point(color = "blue")` is an example of: [2]

- (A) Mapping an aesthetic
- (B) Setting an aesthetic
- (C) Faceting
- (D) A scale

3. Writing `aes(color = "blue")` makes all the points blue. TRUE / FALSE[3]
4. Which geom counts the rows in each category for you? [4]

- (A) geom_col()
- (B) geom_bar()
- (C) geom_point()
- (D) geom_boxplot()

5. For a bar chart, which aesthetic colors the inside of each bar? ___[5]

[1] aes, aes()
[2] Setting an aesthetic
[3] FALSE
[4] geom_bar()
[5] fill



6. Which bar position stretches every bar to the same height, so you can compare shares? [6]

- (A) “stack”
- (B) “dodge”
- (C) “fill”
- (D) “identity”

7. In a boxplot, the thick line across the box shows the: [7]

- (A) Mean
- (B) Median
- (C) Maximum
- (D) 25th percentile

8. `facet_wrap(~ carrier)` draws one panel for each carrier, with shared axes. TRUE / FALSE[8]

9. In a chain of ggplot2 layers, the + goes at the start of each new line. TRUE / FALSE[9]

Show hint for question 3

Inside `aes()`, `"blue"` is treated as a data value, not a color. ggplot2 makes a legend entry called *blue* and uses its default color, a salmon pink. To make the points blue, set the color outside `aes()`: `geom_point(color = "blue")`.

10.8 Summary

- **ggplot2** builds graphs from the **grammar of graphics**: data, **aesthetics** that map columns to visual properties, and **geoms** that draw the marks, joined in **layers** with +.
- The template is `ggplot(data) + geom_function(aes(mappings))`. Mappings in `ggplot()` apply to every layer.
- Choose the geom from the kinds of variables: `geom_point()` and `geom_smooth()` for two continuous variables, `geom_bar()` for counts of a category, `geom_boxplot()` for a continuous variable across categories.
- **Setting** an aesthetic (outside `aes()`) gives every mark the same value. **Mapping** one (inside `aes()`) ties it to a column and makes a legend.
- For bars, `fill` is the inside and `color` the outline. `position = "fill"` compares shares; `position = "dodge"` compares counts.
- **Facets** (`facet_wrap()`, `facet_grid()`) make small multiples, a better choice than many colors.
- Finish every plot with readable colors (`scale_fill_manual()`), labels (`labs()`), and a theme, and save it with `ggsave()`.

[6] “fill”
[7] Median
[8] TRUE
[9] FALSE



10.9 Exercises

Use the `flights` data from nycflights13 for these exercises.

1. **Histogram.** Make a histogram of `distance` with `geom_histogram()`. Change `binwidth` to `100`, then `500`. What do you notice about the distances flown from New York?
2. **Sorted bars.** Make a bar chart of the number of flights from each origin airport. Then make one for the 10 most common destinations (`dest`). (Hint: `count()`, `slice_max()`, and `geom_col()`.)
3. **Delays by month.** Use `group_by()` and `summarize()` to find the mean departure delay for each month, then plot it with `geom_line()` and `geom_point()`. Which months were worst?
4. **Set or map?** Make a scatterplot of `distance` against `air_time` for the January flights. First set every point to dark green. Then map color to `origin`. Then map `size` to `dep_delay`. Which of these makes a useful plot?
5. **Facets.** Make boxplots of `dep_delay` for each `origin`, then facet them by `month`. Use `coord_cartesian(ylim = c(-20, 100))` to zoom in. What does zooming in show you that the full plot doesn't?
6. **Finish and save.** Take your favorite plot from these exercises, give it a title, axis labels with units, and a caption, apply `theme_minimal()`, and save it as a PNG file 7 inches wide and 5 inches tall.
7. **Fix the code.** Each chunk below has one mistake. Find and fix it.

```
ggplot(flights) + geom_point(aes(x = dep_delay, y = arr_delay), color = "origin"
ggplot(flights, aes(x = carrier))
  + geom_bar()
flights |> ggplot(aes(x = carrier)) |> geom_bar()
```

10.10 Further reading

- *R for Data Science* (2nd ed.) (Wickham et al. 2023), free at https://r4ds.hadley.nz. The "Data visualization" chapter introduces ggplot2 with a different data set, and the "Layers" chapter goes further into geoms, facets, and positions.
- The ggplot2 website, https://ggplot2.tidyverse.org, has the reference page for every function in this chapter.
- Posit's *Data visualization with ggplot2* cheatsheet, https://posit.co/resources/cheatsheets/, lists the geoms by the kinds of variables they plot.
- *ggplot2: Elegant Graphics for Data Analysis* (3rd ed.), by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen, free at https://ggplot2-book.org, explains the grammar of graphics in depth.



Part III

Unit 3: Maps and Geographic Data



This unit moves from tables and graphs to maps. It starts by making thematic maps in R with the tmap package, using campus data on UNC's trees, sidewalks, and buildings. It then steps back from the code to ask how the real world becomes geographic data in the first place: the vector and raster data models, and the file formats that store them. Next comes location itself: how a datum, a coordinate system, and a map projection turn places on a round Earth into coordinates on a flat map. The unit ends by getting geographic data from the places that publish it: the US Census Bureau, through the APIs that let your code download its data directly, and the ArcGIS servers that publish many government GIS layers.

Chapter 10: Thematic Maps with tmap

A thematic map shows how one theme varies from place to place. This chapter reads spatial data into R as sf objects, which are tidy data frames with a geometry column, and maps points, lines, and polygons with tmap's `tm_shape()`, `tm_dots()`, `tm_lines()`, and `tm_polygons()`. It covers coloring features by an attribute, choosing classes, combining layers, adding a title, scale bar, and north arrow, and switching between static and interactive maps.

Chapter 11: What Is Geographic Data?

Every map is a model of the world. This chapter follows data abstraction from real-world entities to features a computer can store, and compares the two main data models: vector data (points, lines, and polygons with attributes) and raster data (grids of cells). It explains map scale, cell size and resolution, and the common file formats (shapefile, GeoJSON, KML, and GeoTIFF), and why metadata matter.

Chapter 12: Location and Coordinate Systems

To store a location, a computer needs a model of the Earth, a grid on it, and a way to flatten it. This chapter covers geodesy and the ellipsoid, the datums used in North America (NAD27, NAD83, and WGS84), latitude and longitude, and converting between decimal degrees and degrees, minutes, and seconds. It explains why every map projection distorts shape, area, distance, or direction, how to choose one for a task, and how to check and change a layer's coordinate reference system in R with `st_crs()` and `st_transform()`.

Chapter 13: Census Data and APIs

US Census data come from the decennial census, a full count every ten years, and the American Community Survey, a continuous sample with a margin of error on every estimate. This chapter explains how Census geography nests from states down to tracts and blocks, how APIs work, and how the tidycensus package downloads tidy, map-ready Census data. It also shows how to carry margins of error through calculations and compare data across years.



Chapter 14: ArcGIS Data with arcpullr

Governments publish many of their GIS layers on ArcGIS REST services. This chapter shows how to find a layer's URL and read its REST page, then download it into R as an sf object with arcpullr's `get_spatial_layer()`. It covers SQL `where` clauses that download only the features you need, the server's record limit, and the other ways arcpullr can query a layer.

11 Thematic Maps with tmap

> **Learning objectives**
>
> By the end of this chapter you should be able to:
>
> 1. Load point, line, and polygon layers into R with `read_sf()`, and explain what an sf object is.
> 2. Build a map in layers with `tm_shape()` and `tm_dots()`, `tm_lines()`, or `tm_polygons()`.
> 3. Color features by an attribute, and control the classes with `n` and `style`.
> 4. Combine several layers in one map, and add a title, a scale bar, and a north arrow.
> 5. Switch between static and interactive maps with `tmap_mode()`.

The last chapter turned tables into graphs with ggplot2. This chapter turns them into *maps*. A **thematic map** shows how one theme, such as tree height, the age of buildings, or the population of cities, varies from place to place. It is the most common kind of map in spatial data science, and in R the quickest way to make one is the **tmap** package.

The examples use three layers of data about the UNC Chapel Hill campus: its trees, its sidewalks, and its buildings. Between them they cover the three kinds of vector data, **points**, **lines**, and **polygons**, so by the end of the chapter you will have mapped all three.

> **Run this chapter in RStudio**
>
> The code in this chapter needs the tmap and sf packages and data files from your computer, so it doesn't run in the browser like the code in earlier chapters. Follow along in an R Markdown file in RStudio. The maps you see here were made with exactly the code printed above them.

11.1 Spatial data are tidy data

11.1.1 Features and attributes

Spatial data follow the same tidy rules as any other data frame. In a spatial layer, each row is one **feature**: one tree, one stretch of sidewalk, one building. Each column is one **attribute** of those features: a tree's species or height, a building's year built or number of floors.

What makes the data spatial is one extra column. The **sf** package ("simple features") stores each feature's shape in a **geometry column**, as a point, a line, or a polygon. Everything else about an sf object is an ordinary data frame. That means the tidyverse verbs you already know, such as `filter()`, `mutate()`, and `count()`, work on spatial data too, and they keep the geometry attached to each row.



11.1.2 tmap: the thematic mapper

tmap was written by Martin Tennekes. Like ggplot2, it builds a map in layers, joined with +:

- `tm_shape()` names the data to draw.
- A layer function draws it: `tm_dots()` for points, `tm_lines()` for lines, and `tm_polygons()` for polygons.
- More functions add the parts of a finished map: `tm_title()`, `tm_scalebar()`, `tm_compass()`, and `tm_layout()`.

tmap reads sf objects directly, and the same code makes a static map for a report or an interactive web map you can zoom into. This book uses **tmap version 4**, which changed many argument names. If you find older code online that uses `col =` for polygon colors or `tm_layout(title = ...)`, it was written for version 3. It will often still run, but tmap prints a message suggesting the new form.

Load three packages at the top of every script:

```
library(tidyverse)
library(sf)
library(tmap)
```

11.1.3 Our data: three campus layers

The three layers come from UNC's open data portal, https://gisdata-uncadmin.opendata.arcgis.com (Table 11.1). Download each one as a GeoJSON file and save it in the same folder as your R Markdown file.

Table 11.1: The three UNC campus layers.

Layer	Geometry	One feature is	Attributes used here
Tree Inventory	Points	One campus tree	`COMMONNAME`, `HEIGHT`
Sidewalks	Lines	One sidewalk segment	`SURFACE_`
Building Footprints	Polygons	One building outline	`YEARBUILT`, `FLOORS`

`read_sf()` reads each file into an sf object:

```
buildings <- read_sf("Building_Footprints.geojson")
sidewalks <- read_sf("Sidewalks.geojson")
trees <- read_sf("Tree_Inventory.geojson")
```

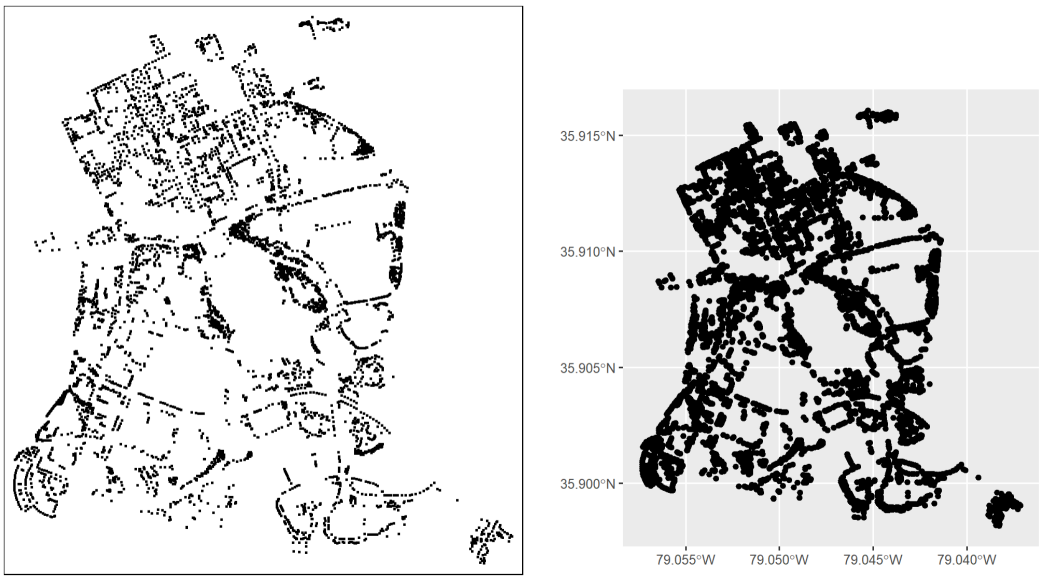
Run `glimpse(trees)` to see the attributes, and check that the last column is `geometry`. Run `st_crs(trees)` to see the layer's **coordinate system**. GeoJSON files always use longitude and latitude, so all three layers share a coordinate system and will line up when you draw them together.

11.1.4 tmap and ggplot2 on the same trees

ggplot2 can draw maps too, with `geom_sf()`. Here is the same tree layer drawn by each package (Figure 11.1):

```
trees |>
  tm_shape() + tm_dots()

trees |>
  ggplot() + geom_sf()
```



(a) tmap: `tm_shape() + tm_dots()` (b) ggplot2: `ggplot() + geom_sf()`

Figure 11.1: The same trees drawn by tmap and by ggplot2.

Both start from the same sf object. tmap gives a clean map frame by default, and has built-in scale bars, north arrows, and an interactive mode. ggplot2 adds axes and a gray panel, as it would for any plot. ggplot2 is the more general tool, and you can make fine maps with it, but for everyday thematic maps tmap takes less code. The rest of this chapter uses tmap.

11.2 Points: mapping trees with tm_dots()

11.2.1 Color trees by height

Each tree in the inventory is one point. To color the trees by height, pass the name of a column to `fill`, in quotes:

```
trees |>
  tm_shape() +
  tm_dots(fill = "HEIGHT") +
  tm_layout(bg.color = "gray")
```

Two details matter. First, the column name goes in quotes, `fill = "HEIGHT"`. (ggplot2's `aes()` takes column names without quotes, but tmap wants them as text.) Second, because `HEIGHT` is a number, tmap uses a **sequential palette**, a ramp from light to dark, so that taller trees look darker. It also breaks the heights into a few **classes**, `0`–`19`, `20`–`39` and so on, and builds the legend.

`tm_layout()` controls the map as a whole rather than one layer. Here `bg.color = "gray"` makes the background gray, so the palest dots still show up.

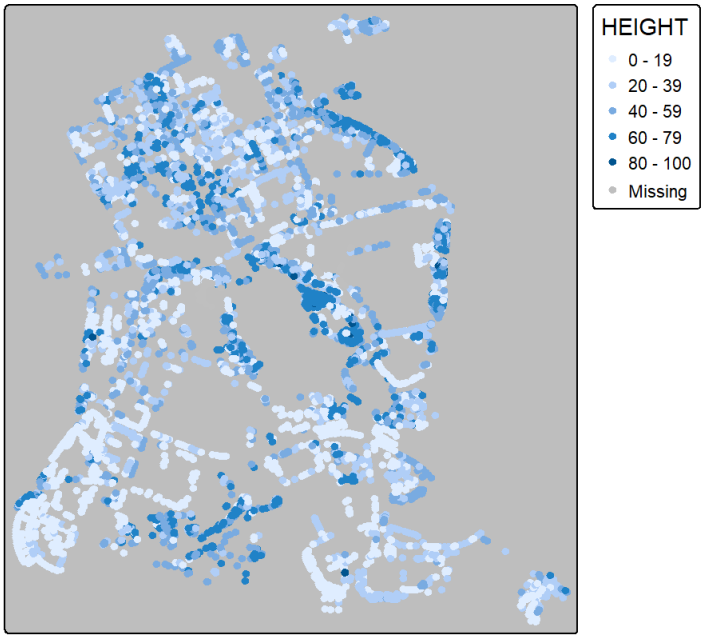


Figure 11.2: Campus trees colored by height. tmap breaks the heights into classes and colors them from light (short) to dark (tall).

Think about it

Look at Figure 11.2. Where on campus are the tallest trees? Can you think of a reason why?

11.2.2 Color trees by species

Now try a text column, the species name:

```
trees |>
  tm_shape() +
  tm_dots(fill = "COMMONNAME")
```

For a text column, tmap uses a **qualitative palette**: distinct colors with no order, one per category. That works for a handful of categories. With 158 species, the colors repeat and the legend takes up as much room as the map, so you can't tell which tree is which (Figure 11.3). There are two fixes: map only the most common species, or lump the rare ones together.

11.2.3 Fix 1: map only the top five species

First, find the most common species with `count()`:

```
trees |>
  st_drop_geometry() |>
  count(COMMONNAME, sort = TRUE)
```

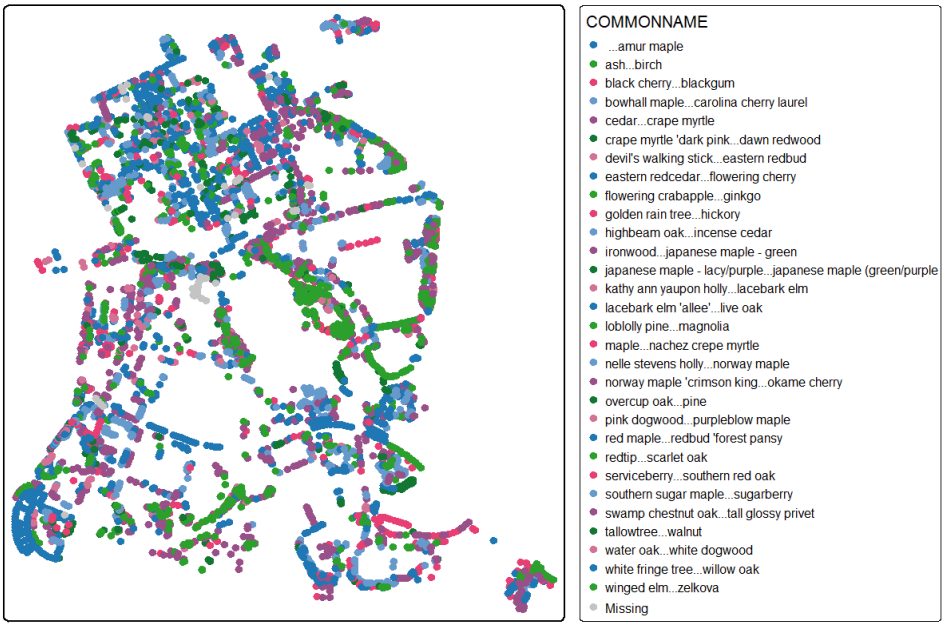


Figure 11.3: Campus trees colored by species. With 158 species, colors repeat and the legend crowds the map.

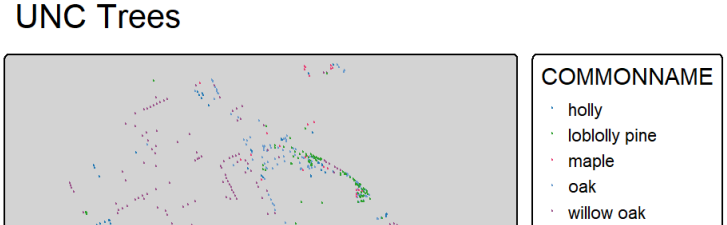
Table 11.2: The ten most common values of COMMONNAME.

COMMONNAME	n
oak	778
holly	570
loblolly pine	513
NA	498
maple	259
willow oak	234
pine	229
crape myrtle	223
hickory	217
dogwood	183

`st_drop_geometry()` turns the sf object back into an ordinary data frame. That isn't required, but it makes counting much faster, because otherwise sf tries to merge the points of every species into one shape. Oak, holly, and loblolly pine lead, and 498 trees have no species name (Table 11.2).

Then `filter()` to the top five, and map them:

```
trees |>
  filter(COMMONNAME %in% c("oak", "holly", "loblolly pine", "maple", "willow oak")) |
  tm_shape() +
  tm_dots(fill = "COMMONNAME", size = 0.05) +
  tm_title("UNC Trees") +
  tm_layout(bg.color = "lightgray")
```



11.2.4 Fix 2: group rare species as “other”

Instead of dropping the rare species, give them a new label. `mutate()` makes a new column, `name`, that keeps the nine most common species and calls everything else “other”:

```
top_nine <- c("oak", "willow oak", "pine", "crape myrtle", "hickory",
              "dogwood", "holly", "loblolly pine", "maple")

trees |>
  mutate(name = if_else(COMMONNAME %in% top_nine, COMMONNAME, "other")) |>
  tm_shape() +
  tm_dots(fill = "name", size = 0.05) +
  tm_title("UNC Trees") +
  tm_layout(bg.color = "lightgray")
```

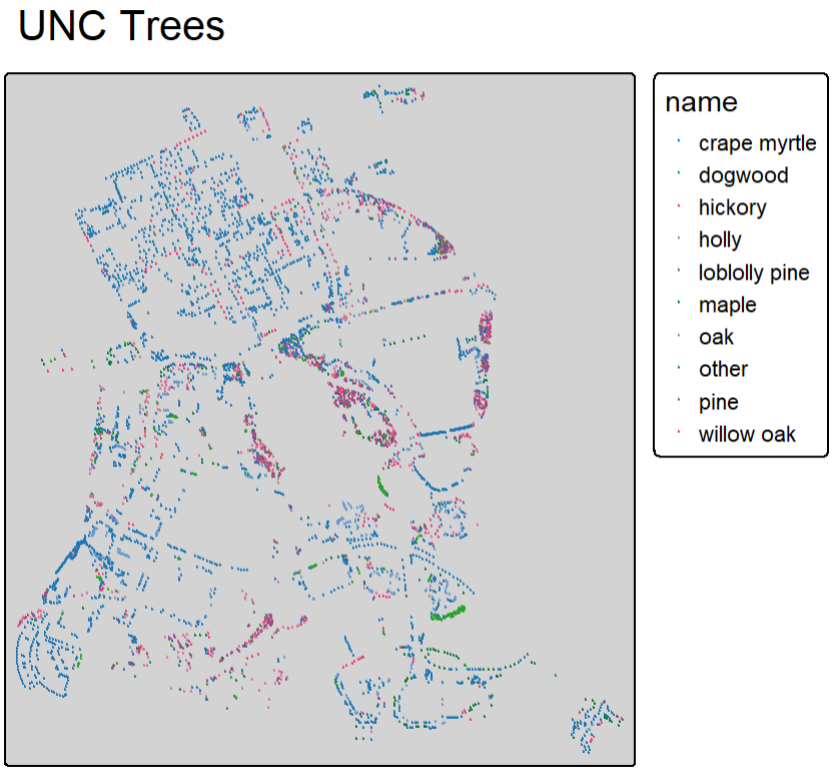


Figure 11.5: Nine species plus “other”. Every tree stays on the map, and the legend has ten entries.

`if_else(condition, yes, no)` checks each tree: if its species is in `top_nine`, it keeps the name; if not, it becomes “other”. The 498 trees with no name go to “other” too, because `NA %in% top_nine` is FALSE. Every tree stays on the map, and ten categories keep the legend readable (Figure 11.5).

11.3 Lines: mapping sidewalks with tm_lines()

11.3.1 Draw the sidewalk network

The sidewalk layer is a network of line segments. `tm_lines()` is the line version of `tm_dots()`:

```
sidewalks |>
  tm_shape() +
  tm_lines()
```



Figure 11.6: UNC sidewalk segments, one line each, with the default style.

With no arguments, every segment is the same thin black line (Figure 11.6). The gaps on the map are places with no sidewalk *data*, not places with no pavement: the layer only covers what has been surveyed.

11.3.2 Color sidewalks by surface

```
sidewalks |>
  tm_shape() +
  tm_lines(col = "SURFACE_")
```

A line has no inside to fill, so lines take `col =` rather than `fill =`. SURFACE_ is a text column, so tmap picks a qualitative palette. (The underscore at the end of the column name comes from the source data; you must type it exactly.) Brick and concrete make up most of central campus (Figure 11.7).

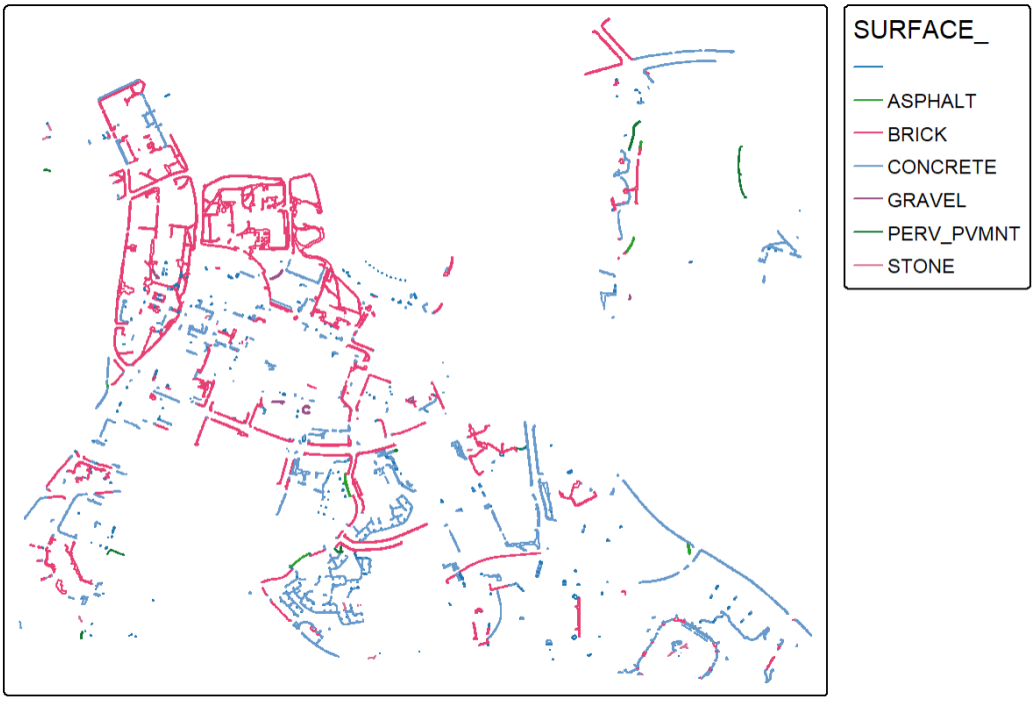


Figure 11.7: UNC sidewalks colored by surface material. Brick (pink) and concrete (blue) make up most of central campus.

11.4 Polygons: mapping buildings with tm_polygons()

11.4.1 Draw the building footprints

Building footprints are areas, so `tm_polygons()` draws each one with a fill and an outline:

```
buildings |>
  tm_shape() +
  tm_polygons()
```

With no arguments, every building gets the same gray fill (Figure 11.8). That already shows the shape of campus, before any attribute is mapped.

11.4.2 Shade buildings by year built

Map `fill` to a numeric column, and you have a **choropleth map**: areas shaded by a value.

```
buildings |>
  tm_shape() +
  tm_polygons(fill = "YEARBUILT")
```

By default, tmap chooses "pretty" breaks, round numbers, here 50-year classes from 1750 to 2050 (Figure 11.9). Buildings with no year recorded are drawn gray and labeled *Missing* in the legend.



Figure 11.8: UNC building footprints, with the default gray fill.

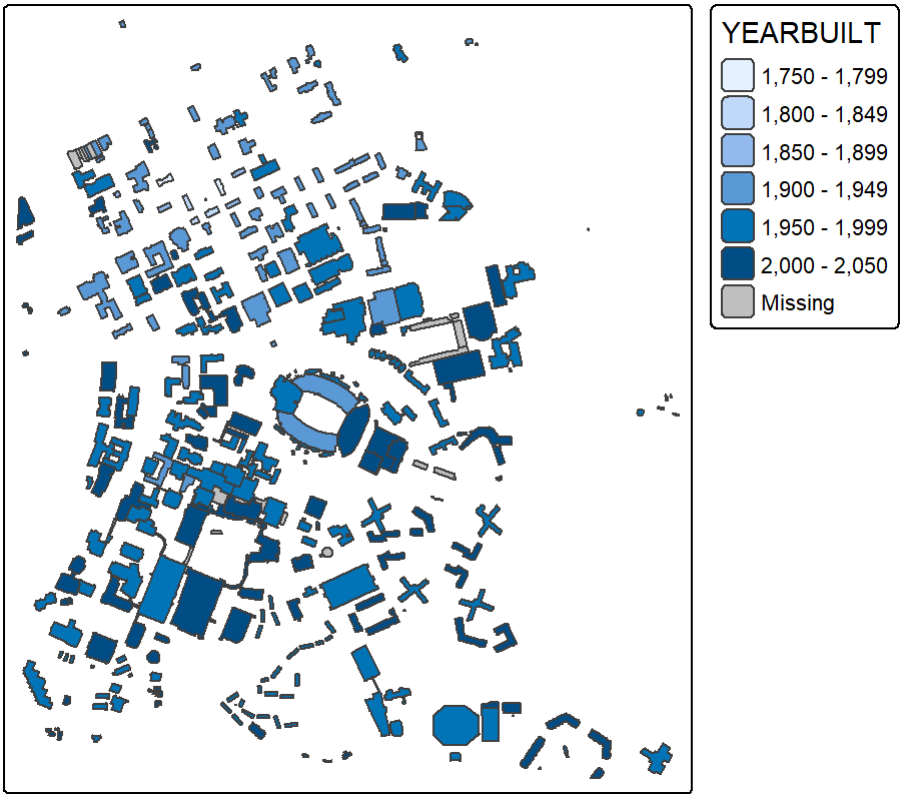


Figure 11.9: UNC buildings by year built. By default, tmap breaks the years into 50-year classes.

Think about it

Where are the oldest buildings on campus? The newest? What does the map suggest about how the campus grew?

11.4.3 Control the classes with n and style

How many classes the map uses, and where the breaks between them fall, change what the reader sees. This is called **classification**, and two settings control it:

- **n** sets the number of classes.
- **`style`** sets the method for choosing the breaks. `"equal"` makes **equal intervals**: every class covers the same range of values.

In tmap version 4, both go in a **scale** function, `tm_scale_intervals()`, given to the `fill.scale` argument:

```
buildings |>
  tm_shape() +
  tm_polygons(fill = "YEARBUILT",
              fill.scale = tm_scale_intervals(n = 4, style = "equal"))
```

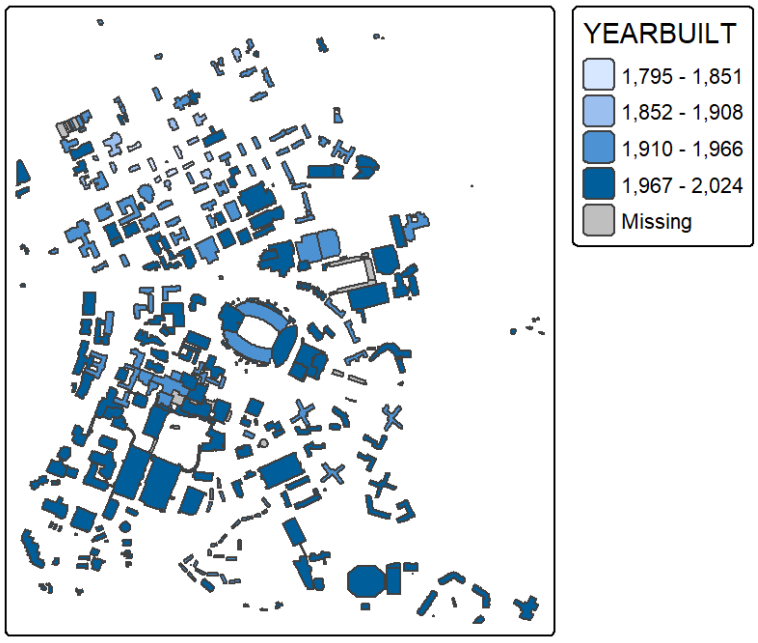


Figure 11.10: Year built in four equal-interval classes. Each class covers about 57 years, from the oldest building (1795) to the newest (2024).

Equal intervals split the range from the oldest building to the newest into four equal parts (Figure 11.10). You can also write `tm_polygons(fill = "YEARBUILT", n = 4, style = "equal")`, the shorter form from version 3. It still works, but tmap prints a message suggesting `fill.scale`. Other styles include `"quantile"` (the same number of buildings in each class) and `"jenks"` (**Jenks natural breaks**, which puts the breaks in the biggest gaps between values). Later chapters return to classification, because the choice can change the story a map tells.

11.5 Putting a map together

11.5.1 Combine points, lines, and polygons

To draw several layers in one map, save each layer as its own object, then add them together with +:

```
# 1. Define independent layers
map_buildings <- buildings |>
  tm_shape() +
  tm_polygons(fill = "FLOORS")

map_sidewalks <- sidewalks |>
  tm_shape() +
  tm_lines()

map_trees <- trees |>
  tm_shape() +
  tm_dots(fill = "HEIGHT", size = 0.02)

# 2. Add them together, placing buildings first to set the map extent
map_buildings + map_sidewalks + map_trees +
  tm_title("UNC Chapel Hill Campus Map") +
  tm_layout(bg.color = "#eeeeee")
```

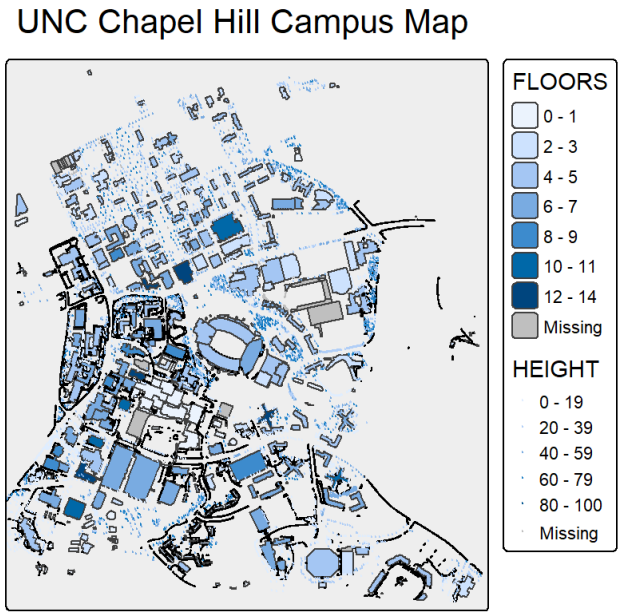


Figure 11.11: Buildings shaded by number of floors, sidewalks, and trees colored by height, in one map.

Each layer draws on top of the ones before it, so put the polygons first, then the lines, then the points; otherwise the buildings would cover the trees. The first layer also sets the map's **extent**, the area the map shows. `tm_title()` and `tm_layout()` style the whole map, so they go at the end, after all the layers.

11.5.2 Add a scale bar and north arrow

A reader expects a finished map to have a scale bar and a north arrow:

```
map_buildings + map_sidewalks + map_trees +
  tm_title("UNC Chapel Hill Campus Map") +
  tm_layout(bg.color = "#eeeeee") +
  tm_scalebar(position = c("left", "bottom")) +
  tm_compass(type = "8star", position = c(0.7, 1))
```

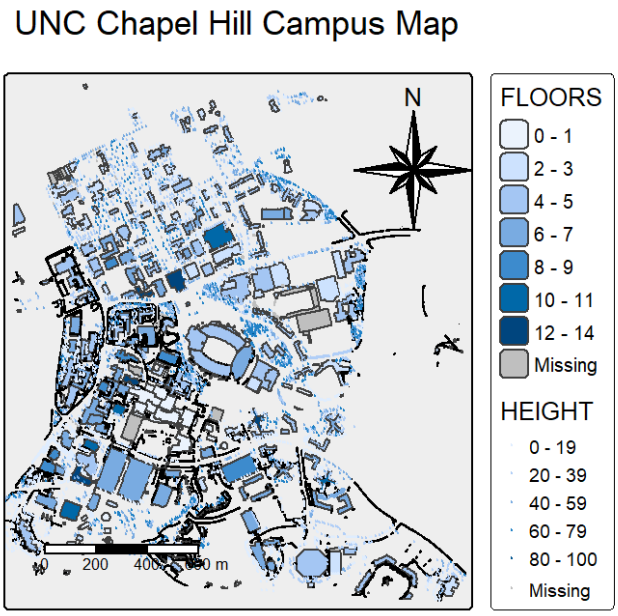


Figure 11.12: The campus map with a scale bar at the bottom left and an eight-pointed north arrow at the top.

`tm_scalebar()` adds a **scale bar**, which shows how distances on the map relate to distances on the ground. `tm_compass()` adds a **north arrow**; `type = "8star"` is one of several styles. `position` places each one, either by name, such as `c("left", "bottom")`, or by numbers from 0 to 1 that give the share of the way across and up the map frame. Put them where they don't cover any data.

11.5.3 Static or interactive: tmap_mode()

tmap has two modes, and the same code works in both:

```
tmap_mode("view")   # interactive web map
tmap_mode("plot")   # static map (the default)
```

After `tmap_mode("view")`, every map becomes an **interactive map** on top of a **basemap** of streets and place names (Figure 11.13). You can pan, zoom, and click on any feature to see its attributes. It is the best way to explore your data. The basemap tiles load from the web, so view mode needs an internet connection. `tmap_mode("plot")` switches back to static maps, which are what you want for reports, slides, and anything printed.

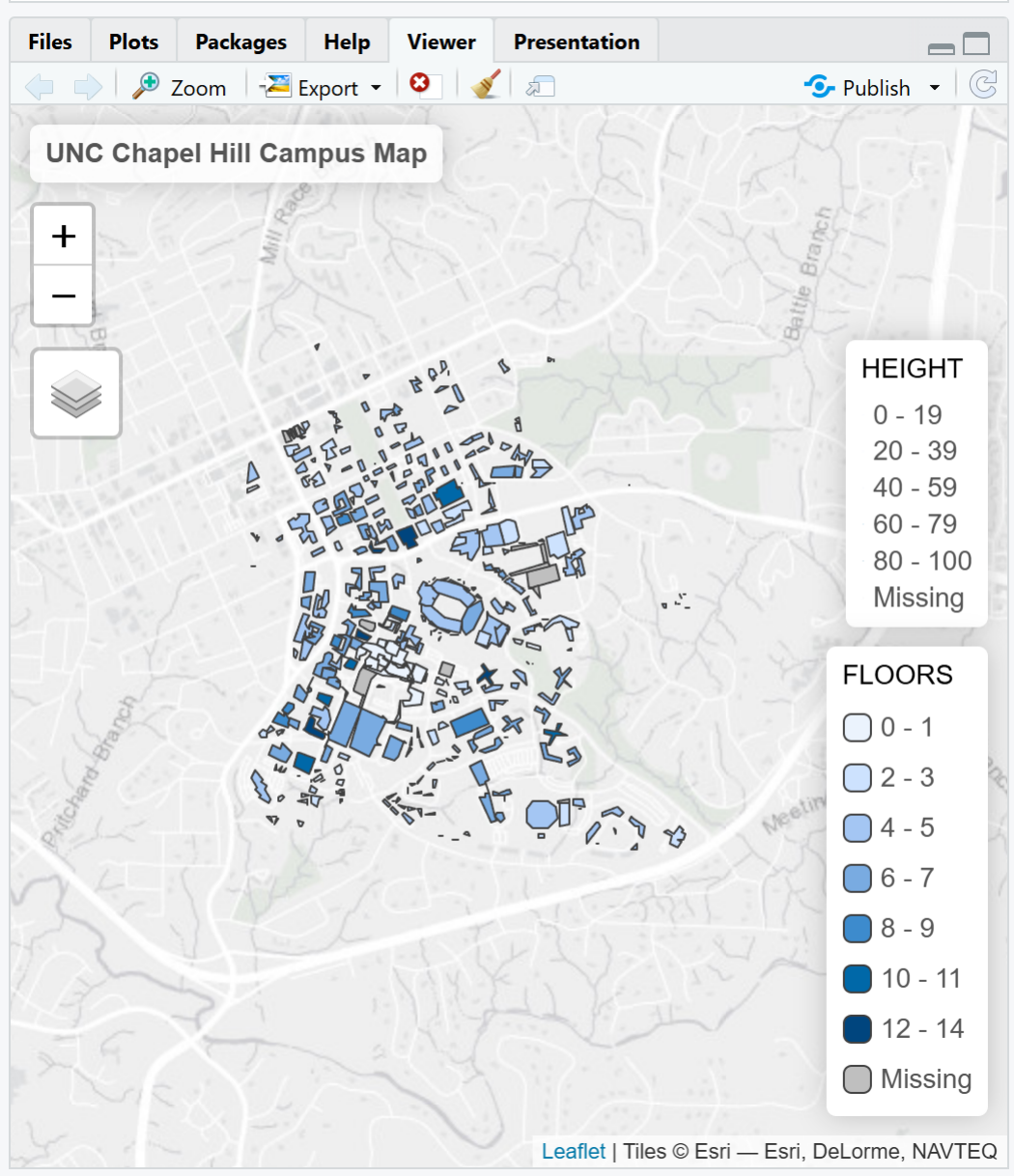


Figure 11.13: The campus map in view mode, in RStudio's Viewer pane. You can pan, zoom, and click a feature to see its attributes.



11.6 Practice: map the world

These steps follow the in-class exercise. tmap comes with three world layers, so you don't need to download anything.

Step 1: set up. Start a new R Markdown file, and load the packages and data:

```
library(tidyverse)
library(sf)
library(tmap)

data(World)
data(metro)
data(rivers)
```

Step 2: look at the data. Use `glimpse()` on `World`, `rivers`, and `metro`. Which is a point layer, which is a line layer, and which is a polygon layer? Which column of `metro` holds the projected population in 2030?

Step 3: make the map. Make one tmap that shows the countries, rivers, and metro areas. As a stretch, size the metro areas by their projected 2030 population.

Show one solution

```
# 1. Create individual maps
map_rivers <- rivers |>
  tm_shape() +
  tm_lines(col = "lightblue", lwd = 0.7)

map_metro <- metro |>
  tm_shape() +
  tm_bubbles(size = "pop2030", fill = "orange")

map_world <- World |>
  tm_shape() +
  tm_polygons(fill = "name", fill.legend = tm_legend_hide())

# 2. Plot all the layers together
map_world + map_rivers + map_metro +
  tm_title("2030 Projected Metro Population") +
  tm_layout(bg.color = "lightblue")
```

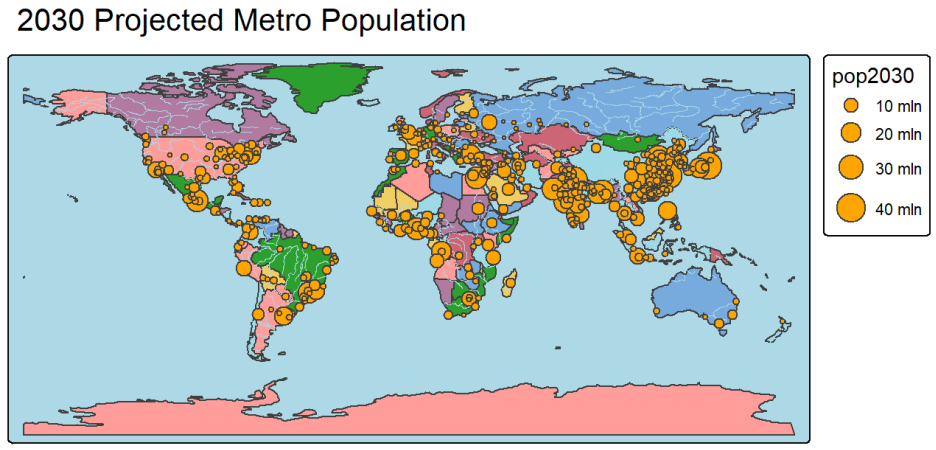


Figure 11.14: Metro areas sized by projected 2030 population, over the world's countries and rivers.

The layers go in order: countries first, then rivers, then the metro bubbles on top. `tm_bubbles()` draws circles sized by a column, here `pop2030`. Notice the difference between *setting* and *mapping*, just as in ggplot2: `fill = "orange"` is a color, so every bubble is orange, while `size = "pop2030"` is a column, so the size comes from the data. Filling the countries by `name` gives each one its own color, and `fill.legend = tm_legend_hide()` hides the legend, which would otherwise list 177 countries.



11.7 Check your understanding

1. In an sf object, each row is one: [1]

- (A) Attribute
- (B) Feature
- (C) Coordinate
- (D) Layer

2. Which tmap function names the data to draw, at the start of each layer? _____[2]
3. Which layer function draws polygons? [3]

- (A) tm_dots()
- (B) tm_lines()
- (C) tm_polygons()
- (D) tm_shape()

4. To color sidewalk lines by a column, you use `col =` rather than `fill =`, because a line has no inside to fill. TRUE / FALSE[4]
5. For a numeric column such as HEIGHT, tmap's default colors are a: [5]

- (A) Qualitative palette, one color per value
- (B) Sequential palette, from light to dark
- (C) Single color
- (D) Random set of colors

6. In `tm_scale_intervals(n = 4, style = "equal")`, what does n set? [6]

- (A) The size of the dots
- (B) The number of classes
- (C) The number of layers
- (D) The number of the north arrow style

7. When you add layers together, the first layer sets the extent of the map. TRUE / FALSE[7]
8. Which function adds a north arrow? [8]

[1] Feature
[2] tm_shape, tm_shape()
[3] tm_polygons()
[4] TRUE
[5] Sequential palette, from light to dark
[6] The number of classes
[7] TRUE
[8] tm_compass()

- (A) tm_scalebar()
- (B) tm_compass()
- (C) tm_title()
- (D) tm_layout()

9. Which mode makes an interactive map that you can pan and zoom? [9]

- (A) tmap_mode("plot")
- (B) tmap_mode("view")
- (C) tmap_mode("web")
- (D) tmap_mode("static")

Show hint for question 4

`fill` colors the inside of a dot or a polygon. A line is only an outline, so tmap colors it with `col`, the same name ggplot2 uses for the color of lines and outlines.

11.8 Summary

- An **sf** object is a tidy data frame of spatial **features**, one per row, with their **attributes** in the columns and their shapes in a **geometry column**. Read a file into one with `read_sf()`.
- **tmap** builds a **thematic map** in layers: `tm_shape()` names the data, and `tm_dots()`, `tm_lines()`, or `tm_polygons()` draws it. Layers are joined with +.
- Color features by an attribute with `fill = "column"` (dots and polygons) or `col = "column"` (lines). Numbers get a **sequential palette**; text gets a **qualitative palette**.
- Too many categories make a map unreadable: `filter()` to the most common, or lump the rest together with `mutate()` and `if_else()`.
- **Classification** decides how many classes a map uses and where the breaks fall: `fill.scale = tm_scale_intervals(n = 4, style = "equal")`.
- Combine layers with +, with polygons first and points last. Finish with `tm_title()`, `tm_scalebar()`, `tm_compass()`, and `tm_layout()`.
- `tmap_mode("view")` makes interactive web maps for exploring; `tmap_mode("plot")` makes static maps for reports.

11.9 Exercises

Use the three campus layers for exercises 1 to 4, and tmap's `World` data for 5 and 6.

1. **Floors.** Make a choropleth map of the buildings, shaded by FLOORS. Then try `n = 3` and `n = 7`. Which do you find easiest to read?
2. **Classification styles.** Map YEARBUILT with `style = "equal"`, `"quantile"`, and `"jenks"`, each with five classes. How do the breaks differ? Which map makes campus look oldest?

[9] tmap_mode("view")



3. **One species.** Map only the willow oaks, as dots on top of the building footprints. Give the map a title, a scale bar, and a north arrow.
4. **Sidewalk lengths.** How many sidewalk segments are made of each surface? Use `st_drop_geometry()` and `count()`. Then try `st_length()` inside `mutate()` to find the total length of each surface. Is the answer different?
5. **A world choropleth.** `World` has a column called `life_exp`, life expectancy. Make a choropleth of it with five classes. Which regions stand out?
6. **Explore.** Switch to `tmap_mode("view")` and draw your map from exercise 5. Click on a few countries. Then switch back with `tmap_mode("plot")`.

11.10 Further reading

- The tmap website, https://r-tmap.github.io/tmap/, has a getting-started guide and many examples, all in version 4 syntax.
- *Geocomputation with R* (2nd ed.), by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, free at https://r.geocompx.org. Its "Making maps with R" chapter covers tmap in depth, including faceted and animated maps.
- The sf website, https://r-spatial.github.io/sf/, explains simple features and has a cheat-sheet for its functions.

12 What Is Geographic Data?

> **Learning objectives**
>
> By the end of this chapter you should be able to:
>
> 1. Explain how data abstraction turns real-world entities into features a computer can store.
> 2. Tell continuous data from discrete objects, and spatial data from attributes.
> 3. Describe the vector and raster data models, and choose between them for a task.
> 4. Use the parts of vector data (points, nodes, vertices, line segments, polygons) and of raster data (cells, cell size, origin) correctly.
> 5. Recognize common geographic file formats, and explain why metadata matters.

In the last chapter you mapped campus trees, sidewalks, and buildings. But a map, or a layer of data in R, is never the world itself. A tree on the map is a dot; a building is a flat outline. Each is a **model** of reality, and every model makes choices about what to keep and what to leave out.

This chapter steps back from the code to ask how geographic information gets from the real world into a computer. It is organized around four questions:

1. How does a map represent the world?
2. What types of geographic data are there?
3. What are the advantages and disadvantages of each type?
4. How are the data stored in files that R can read?

The answers will help you choose the right data for a question, and understand what the data can and can't tell you.

12.1 From the real world to the computer

12.1.1 Entities and features

Everything starts with the real world. Walk across campus and you pass trees, light poles, buildings, sidewalks, and lawns. Each of these real things is an **entity**. When we put them on a map or in a database, they become **features**: simplified representations of those entities.

Not every entity makes it onto the map. A typical campus map keeps the buildings and sidewalks, and leaves out the light poles and most of the trees. Choosing what to keep is the first step of abstraction.

12.1.2 Data abstraction

Data abstraction is the process of making the real world readable by a computer. It happens in steps, each further from what people see and closer to what a machine stores (Table 12.1).

Table 12.1: The levels of data abstraction, from the human end to the machine end.

Level	What it is	Example: a wetland
Real world	Too detailed to store as it is	The marsh itself, with its water, plants, and edges
Data model	We choose which objects and properties to keep	Wetland areas, each with an ID, an area, and a type
Data structure	Tables and lists of coordinates organize the model	A list of x, y coordinates for each outline, and a table of attributes
Machine code	Everything is stored as 0s and 1s	`10011101 00110110 ...`

The top of the table is meaningful to people, and the bottom to machines. Geographic information science works in the middle, with **data models** and **data structures**, where both can read it.

12.1.3 Continuous data and discrete objects

Geographic information comes in two broad types, and the type helps decide how to model it:

- **Continuous data** exist everywhere on the map. Every location has a value, even if we only measured some of them, and the value changes from place to place. Examples: elevation, temperature, precipitation.
- **Discrete objects** exist in one place and not in another. They are either present or absent, with clear edges. Examples: roads, houses, states.

Think about it

Is population density continuous or discrete? You could think of it as a value that exists everywhere and changes smoothly across a city. Or you could count the people in each census tract, which gives one number per area with sharp edges between them. Which is more realistic? Which is easier to work with?

12.1.4 Geographic data models

A **geographic data model** is "a formal (meaning mathematical) means of representing and manipulating spatially-referenced information." People understand the world through ideas like *river* or *neighborhood*; computers understand numbers and rules. A data model lives in the overlap: precise enough for a computer, and meaningful enough for us. Every data model loses something, so it is always worth asking what was left out.

Geographic information reaches the computer in three main forms:

- **Vector** data store discrete features as coordinates: points for fire hydrants, lines for streets, polygons for land parcels.

- **Raster** data store a grid of cells, each holding a value, like the pixels of a satellite image or an elevation model.
- **Tabular** data hold the attributes of features: for one house, its address, year built, square footage, and number of bedrooms. Attribute tables are usually joined to vector features.

The next two sections take vector and raster data in turn.

12.2 Vector data

12.2.1 One object, three representations

The same real-world objects can be stored as different kinds of vector features. Take the buildings in an air photo (Figure 12.1):

- As **points**, they record *where* each building is.
- As **lines**, they record each building's outline.
- As **areas** (polygons), they record the outline *and* the space inside it.

What is the difference between the lines and the areas, when both trace the same outline? Enclosure. A line is just a boundary; a polygon knows it has an inside, so you can ask how big it is, color it in, or find out whether a point falls within it.

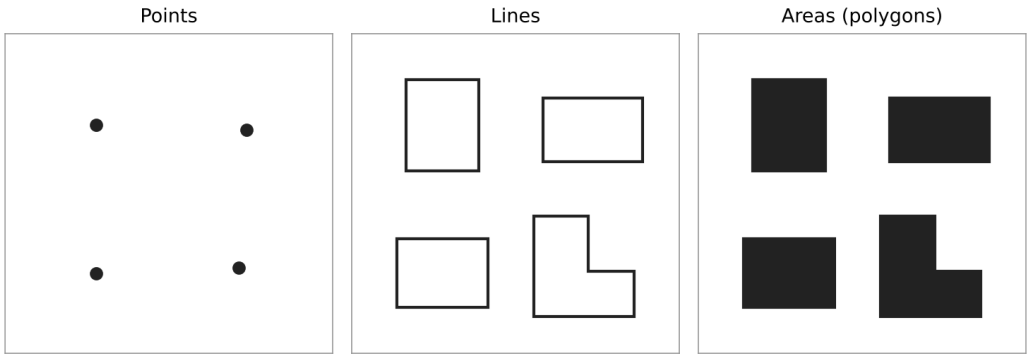


Figure 12.1: The same four buildings stored as points, as lines, and as areas.

The right choice depends on the map's purpose and its scale.

12.2.2 Generalization and map scale

When a map shows a place at a smaller scale, detail has to go. This is called **generalization**. Imagine three maps of the same city:

- A **large-scale map**, zoomed in on a neighborhood, draws the streets and buildings as areas.
- An intermediate map, of the region around the city, draws the city as an area and the roads as lines.
- A **small-scale map**, of the whole country, draws the city as a single point.



The names are easy to mix up. Map scale is a fraction: a large-scale map has a large fraction, such as 1:10,000, so it shows a small area in detail. A small-scale map has a small fraction, such as 1:10,000,000, so it shows a large area with little detail. The same feature can need a different geometry at a different scale, so a data model must be chosen with the map's purpose in mind.

12.2.3 What makes something an entity?

To store an entity, we need to know five things about it:

1. A **location**: an address or a set of coordinates.
2. An **extent**: its size, boundary, or dimensions.
3. Non-spatial **attributes**: its name, number, qualities, or color.
4. **Time**: when it existed, or when the data were collected.
5. A **geometry**: its shape.

Think of a lake. Its location and extent come from an air photo, its geometry is a polygon traced around the shore, and its name, depth, and water quality are attributes.

12.2.4 Spatial data and attributes

Spatial data have a location: you could put a pin on a map for each campus building, each rain gauge in North Carolina, or each interstate highway. **Non-spatial data** have no location of their own: the parts of a bicycle, student ID numbers, or class grades.

The two meet in attributes. When we describe a spatial entity with non-spatial information, such as a building's name or the year it was built, that information is called an **attribute**. Attribute data:

- are always linked to a spatial feature;
- are usually stored in tables, with one row per feature and one column per attribute;
- have the same columns for every feature in a layer.

Follow a fire hydrant into a database. The hydrant becomes a point in a layer of hydrants, and everything we know about it goes into a row of the **attribute table**: its ID, its number of outlets, its color, its rating (Table 12.2). The ID is what ties each row back to its point. Because every hydrant has the same columns, we can query the layer, for example, "show all hydrants with a low rating".

Table 12.2: An attribute table for a layer of fire hydrants. Each row is one hydrant (one point on the map), and each column is one attribute.

ID	Outlets	Color	Rating
1	3	red	high
2	3	red	high
3	2	green	low
4	3	red	mid

This is the same structure as the sf objects in the last chapter: each row is a feature, each column an attribute, and the geometry column holds the points.

12.2.5 The parts of vector data

These terms come up whenever you edit or analyze vector data (Figure 12.2):

- A **point** is a single location that stands alone.
- A **node** is the start or end point of a line.
- A **vertex** is a point in between that gives the line its shape.
- A **line segment** is the straight piece between two vertices.
- A **polygon** is a closed ring of line segments.

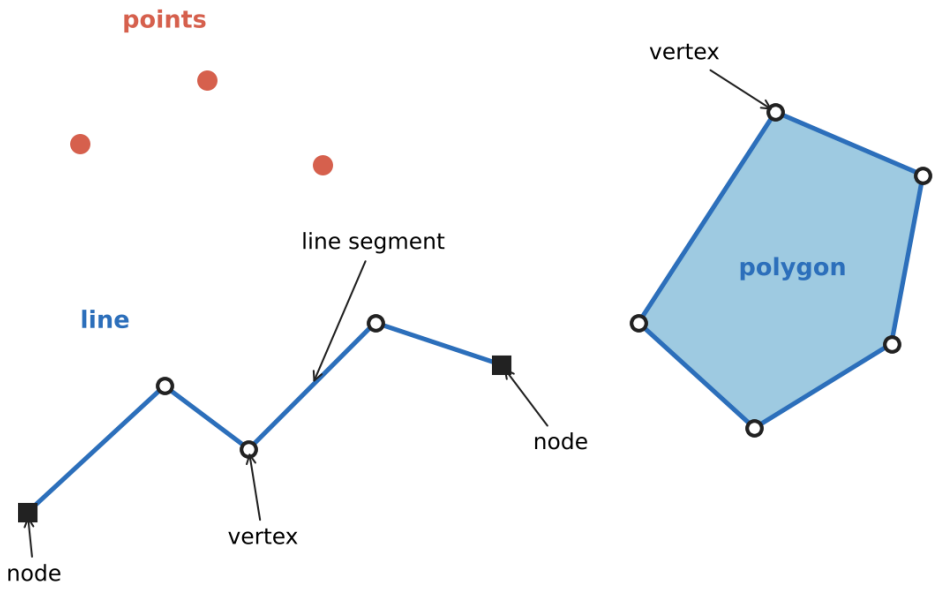


Figure 12.2: The parts of vector data: points, a line that starts and ends at nodes and bends at vertices, and a polygon.

Nodes matter for networks. Roads connect at nodes, which is how routing software knows two streets meet. Vertices matter for accuracy: more vertices make a smoother, more accurate line, but a bigger file.

12.2.6 Thematic layers

Geographic data are organized into **thematic layers**. All the roads go in one layer, all the soils in another, and elevation in a third. Each layer holds one theme, and similar features go together.

Because every layer uses the same **coordinate system**, the layers stack exactly on top of each other, and a map is just a set of layers drawn together. That is exactly what you did in the last chapter, when you drew buildings, then sidewalks, then trees. Stacking layers is also the basis of **overlay analysis**: for example, finding the wetlands on a particular type of soil within a kilometer of a road.

12.3 Raster data

12.3.1 Vector and raster compared

Rasters divide space into a regular grid of cells, and each cell stores one value. Figure 12.3 shows the same points, line, and area stored both ways. On the left, as vector data, the points are exact coordinates, the line is a chain of vertices, and the area is a polygon. On the right, as a raster, each feature is approximated by grid cells: the line becomes a staircase of cells, and the area a block of them.

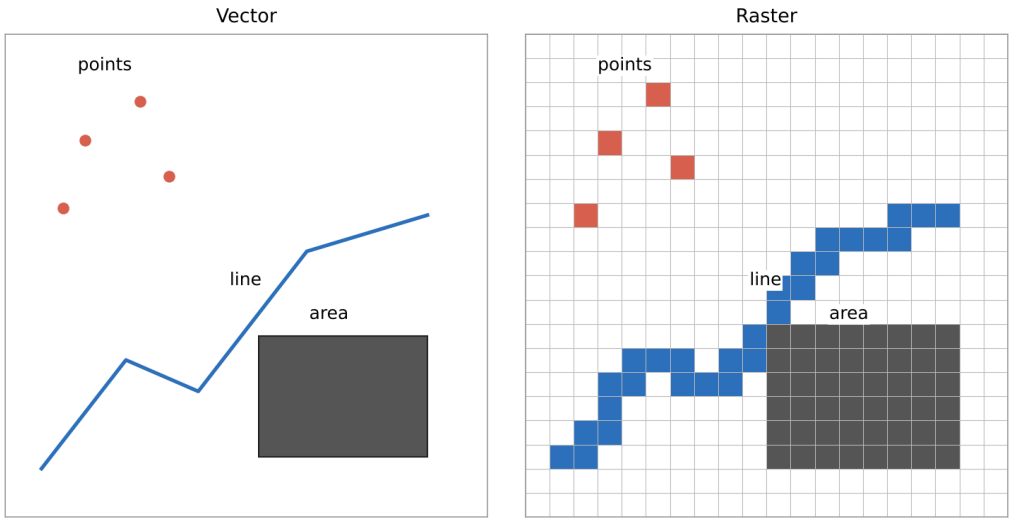


Figure 12.3: The same points, line, and area stored as vector data (left) and as a raster (right).

How accurate the raster version is depends entirely on the size of the cells. Table 12.3 sums up the two models.

Table 12.3: The vector and raster data models compared.

	Vector	Raster
Geometry	Points, lines, polygons	A regular grid of cells
Best for	Discrete objects	Continuous data
Location	Stored as coordinates	Set by the cell size and position in the grid
Values	Each feature has attributes	Each cell holds a value
Examples	Roads, parcels, wells	Elevation, satellite imagery, rainfall
Strengths	Precise and compact for features with clear edges	Simple to analyze; handles surfaces that vary everywhere
Weaknesses	Awkward for values that change continuously	Files grow quickly as cells get smaller; edges become staircases

12.3.2 A raster is a grid of cell values

Every **cell** in a raster holds exactly one value (Figure 12.4). The value can be a number, like an elevation in meters, or a code for a category, like 1 for forest and 2 for water. Places with no measurement are flagged as **no data**, here as *N*. Every raster format has a way to mark those cells, and in R they become NA.

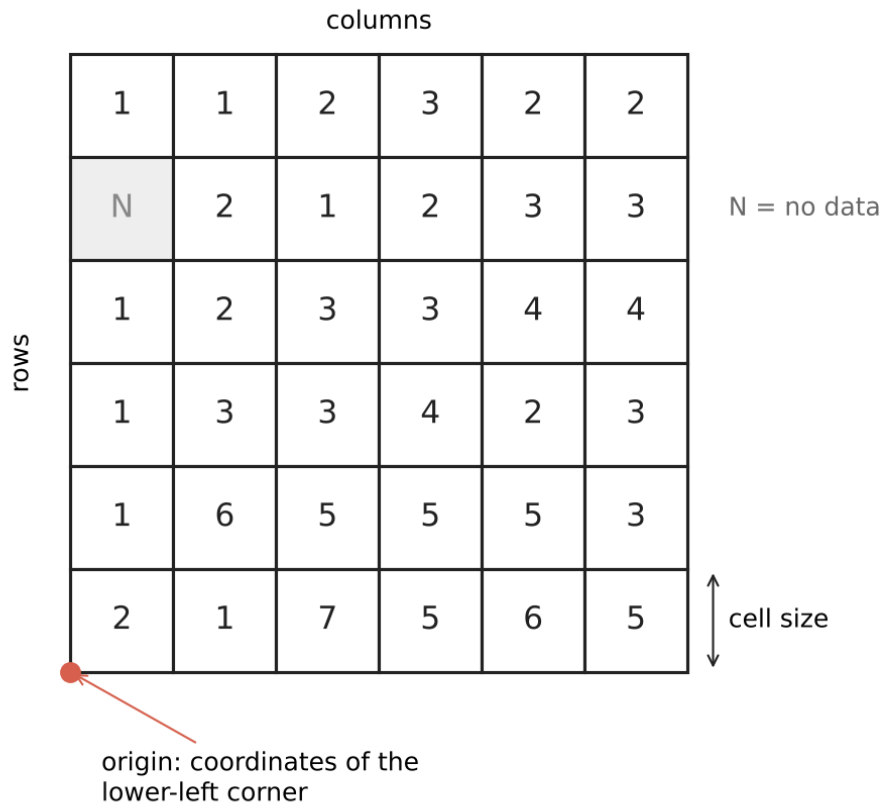


Figure 12.4: A raster is a grid of cell values. Its location is fixed by the coordinates of the lower-left corner and the cell size.



A raster doesn't store coordinates for every cell. It stores the coordinates of one corner, the **origin**, usually the lower-left corner, and the **cell size**: the length of one side of a cell in real-world units, such as meters. From those, the computer works out where any cell is from its row and column:

$$N_{\text{cell}} = N_{\text{lower-left}} + \text{row} \times \text{cell size}$$

$$E_{\text{cell}} = E_{\text{lower-left}} + \text{column} \times \text{cell size}$$

Here N is the northing (the y coordinate) and E the easting (the x coordinate), with rows counted up from 0 at the bottom and columns counted from 0 at the left. Try it: a raster of 30 m cells has its lower-left corner at easting 600,000 m and northing 3,950,000 m. Where is the corner of the cell in row 4, column 10?

```
easting_ll  <- 600000
northing_ll <- 3950000
cell_size   <- 30

row <- 4
col <- 10

northing_ll + row * cell_size
easting_ll + col * cell_size
```

Because a raster is just a grid of numbers, R can hold a small one in a **matrix**. This box makes the grid in Figure 12.4, with NA for the no-data cell, and draws it with base R's `image()`:

```
cells <- matrix(c(1, 1, 2, 3, 2, 2,
                  NA, 2, 1, 2, 3, 3,
                  1, 2, 3, 3, 4, 4,
                  1, 3, 3, 4, 2, 3,
                  1, 6, 5, 5, 5, 3,
                  2, 1, 7, 5, 6, 5),
                nrow = 6, byrow = TRUE)
cells
mean(cells, na.rm = TRUE)

# image() draws row 1 at the bottom, so flip the rows first
image(t(cells[6:1, ]), col = hcl.colors(7, "Greens", rev = TRUE),
      axes = FALSE, asp = 1)
```

Real rasters are far bigger, with millions of cells, and in R you would use the **terra** package for them. But the idea is the same.

12.3.3 Cell size and resolution

The cell size is the raster's **resolution**. Figure 12.5 shows the same 200 m square at three cell sizes. Going from 100 m to 50 m to 25 m cells takes it from 4 to 16 to 64 cells.

Smaller cells capture more detail, but the number of cells grows with the *square* of the resolution: halve the cell size and you have four times as many cells, and roughly four times as much data.

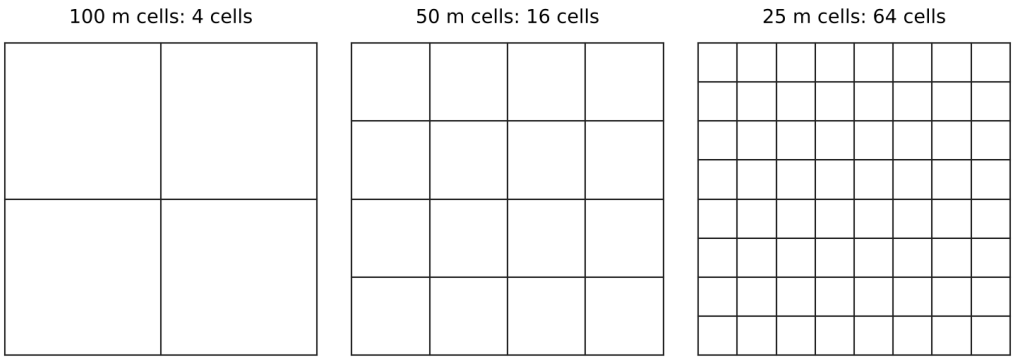


Figure 12.5: The same 200 m square at three cell sizes. Halving the cell size makes four times as many cells.

Think about it

A TV screen is a raster, and each pixel is a cell. If you were buying a new TV, which of the three grids in Figure 12.5 would give the sharpest picture? A 4K TV has about four times as many pixels as a 1080p TV of the same size. What does that say about the size of its pixels, and about how much data it needs?

12.3.4 Data structures

The data model is the idea; the **data structure** is how that idea is laid out in memory or on disk. A polygon, for example, can be stored as a set of **arcs**, each a list of coordinate pairs. A pointer at the end of one arc tells the computer where the next one begins. Neighboring polygons can share an arc, so their common boundary is stored once rather than twice. Data structures that record how features connect and share boundaries like this are called **topological** data structures.

12.4 Geographic file formats

Data models and structures are stored in files, and you will meet several formats when you load spatial data into R (Table 12.4).

Table 12.4: Common geographic file formats.

Format	Extension	Data model	Notes
Shapefile	`.shp`, plus several others	Vector	Esri's format, from the early 1990s; still very common
GeoJSON	`.geojson`	Vector	Open, plain-text standard; the default for web maps
KML	`.kml`, `.kmz`	Vector	The format behind Google Earth
GeoTIFF	`.tif`	Raster	An image with its location stored inside

12.4.1 Shapefiles

A **shapefile** is the most common way vector data are shared, and its name is misleading: it isn't one file, but a group of files with the same name and different extensions (Table 12.5).

Table 12.5: The files that make up one shapefile, called `roads`.

File	What it holds
`roads.shp`	The geometry of each feature
`roads.shx`	A positional index, for faster access
`roads.dbf`	A database table of the attributes
`roads.prj`	The projection (coordinate system)
`roads.sbn`, `roads.sbx`	A spatial index of the features
`roads.shp.xml`	The metadata for the layer

Only `.shp`, `.shx`, and `.dbf` are required, but without the `.prj` file the software can't place the data correctly on the Earth. Keep all the files together, in the same folder: a `.shp` on its own is not usable. When you email or download a shapefile, zip the whole set.

Shapefiles have limits worth knowing. Attribute column names can be at most 10 characters long, which is why you will see names like `SURFACE_` or `POP2020EST`, and each file can be at most 2 GB. Newer formats, such as the GeoPackage (`.gpkg`), avoid these limits.

12.4.2 GeoJSON

GeoJSON is an open standard for simple geographic features, written in **JSON**, the plain-text format that web APIs use. You can open it in any text editor, read it, and edit it. It stores points, lines, polygons, or collections of features, and it has become the default format for web maps. The campus layers in the last chapter were GeoJSON files.

Here is a small GeoJSON file with one feature, a point at the Old Well:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [-79.0512, 35.9121]
      },
      "properties": {
        "name": "Old Well"
      }
    }
  ]
}
```

A `FeatureCollection` holds a list of `Features`, and each feature has a `geometry` (its type and coordinates) and `properties` (its attributes). Notice that the coordinates are written **longitude first**, then latitude. GeoJSON always uses longitude and latitude on the **WGS84** datum, the same system GPS uses. (Coordinate systems get a chapter of their own



later in the course.) The free website https://geojson.io lets you draw features on a map and see their GeoJSON appear beside it.

12.4.3 KML

KML, Keyhole Markup Language, is the format behind Google Earth. It was created by Keyhole, the company whose Earth viewer Google bought in 2004, and it became an Open Geospatial Consortium standard in 2008. KML is written in XML, as nested tags, and stores points, paths, polygons, images, and even 3D models. Here is the same Old Well as a KML placemark:

```
<Placemark>
  <name>Old Well</name>
  <Point>
    <coordinates>-79.0512,35.9121,0</coordinates>
  </Point>
</Placemark>
```

Like GeoJSON, KML uses WGS84 longitude and latitude, plus an altitude. A `.kmz` file is a zipped KML file, usually bundled with its icons or images. KML is great for sharing and display, but less so for analysis, so it is usually converted to another format before you work with it in R.

12.4.4 GeoTIFF

GeoTIFF is the standard format for raster data: satellite images, air photos, and elevation models. It is an ordinary TIFF image with extra information inside the file saying where on Earth each pixel is, and which coordinate system it uses. It is a public-domain, open standard.

GeoTIFFs get big fast, often hundreds of megabytes. A newer variety, the **Cloud Optimized GeoTIFF (COG)**, arranges the cells in tiles with built-in zoomed-out copies, so that a web map or R can read only the area and the level of detail it needs, instead of downloading the whole file.

12.4.5 Metadata

Metadata are data about the data. The `.shp.xml` file in a shapefile is one example. Good metadata answer these questions:

- Who created the data?
- When were they created?
- How were they created, and to what specifications?
- Where can the data be found?
- What coordinate system do they use, and how accurate are they?

In the US, federal agencies follow the **FGDC** metadata standard, from the Federal Geographic Data Committee; internationally, the standard is **ISO 19115**. Without metadata, you can't judge whether data are fit for your purpose. A practical rule: if you can't find out when and how a dataset was made, be very cautious about using it to make a decision.

12.4.6 Spatial data in R

How does R handle all of this? With data frames. In the last chapter, `read_sf()` read GeoJSON files into sf objects, which are data frames with an extra column that holds each feature's geometry. The same function reads shapefiles (point it at the `.shp` file, with the other files beside it), KML, and GeoPackages. Rasters, such as GeoTIFFs, are read with the terra package. Later chapters put these to work.

12.5 Check your understanding

1. A real-world object, such as a tree on campus, is called an: [1]

- (A) Entity
- (B) Attribute
- (C) Vertex
- (D) Origin

2. Elevation is an example of: [2]

- (A) A discrete object
- (B) Continuous data
- (C) Non-spatial data
- (D) Metadata

3. Which data model stores features as points, lines, and polygons? [3]

- (A) Vector
- (B) Raster
- (C) Tabular
- (D) Metadata

4. A point in the middle of a line, where the line changes direction, is called a: [4]

- (A) Node
- (B) Vertex
- (C) Cell
- (D) Origin

[1] Entity
[2] Continuous data
[3] Vector
[4] Vertex

5. A large-scale map, such as 1:10,000, shows a small area in a lot of detail. TRUE / FALSE[5]
6. A raster has 50 m cells. You change it to 25 m cells over the same area. How many times as many cells does it have now? _[6]
7. Which file in a shapefile holds the attribute table? [7]

- (A) .shp
- (B) .shx
- (C) .dbf
- (D) .prj

8. GeoJSON writes coordinates with latitude first, then longitude. TRUE / FALSE[8]
9. Which format is the standard for raster data such as satellite images? [9]

- (A) KML
- (B) GeoJSON
- (C) Shapefile
- (D) GeoTIFF

Show hint for question 6

Halving the cell size halves the width of each cell *and* its height, so each old cell becomes 2 × 2 = 4 new cells.

12.6 Summary

- A map or a GIS layer is a **model** of the world. **Data abstraction** turns real-world **entities** into **features**, through a **data model** and a **data structure**, down to machine code.
- **Continuous data**, such as elevation, exist everywhere; **discrete objects**, such as roads, are either present or absent.
- **Vector** data store discrete features as **points**, **lines**, and **polygons**, built from **nodes**, **vertices**, and **line segments**. The same entity can be a polygon at a large scale and a point at a small one (**generalization**).
- **Attributes** are non-spatial information about features, stored in a table with one row per feature. Layers of one theme each stack into a map because they share a **coordinate system**.
- **Raster** data store a grid of **cells**, each with one value. Its location comes from the **origin** and the **cell size**; smaller cells mean more detail, and more data.
- Vector suits discrete objects; raster suits continuous surfaces. Common formats are the **shapefile**, **GeoJSON**, and **KML** for vector data, and **GeoTIFF** for rasters.
- **Metadata** tell you who made the data, when, how, and how accurate they are. Without them you can't judge whether data fit your purpose.

[5] TRUE
[6] 4
[7] .dbf
[8] FALSE
[9] GeoTIFF

12.7 Exercises

1. **Entities on campus.** List ten entities you pass on your way to class. For each, say whether you would store it as a point, a line, a polygon, or a raster, for a campus map at 1:5,000. Would any change for a map of all of Orange County?
2. **Continuous or discrete?** Classify each as continuous data or a discrete object, and explain your choice: air temperature, county boundaries, soil type, noise from an airport, bus stops, tree canopy cover.
3. **Vector or raster?** For each task, choose vector or raster data and explain why: (a) finding the shortest driving route between two addresses; (b) mapping the slope of the land across a county; (c) counting the trees within 10 m of a sidewalk; (d) mapping land cover from a satellite image.
4. **Cell arithmetic.** A raster covers an area 3 km wide and 2 km tall with 10 m cells. How many cells does it have? How many would it have with 30 m cells? Use R to check your answers.
5. **Make a GeoJSON file.** Go to https://geojson.io and draw a point at your favorite place on campus, a line along your route to class, and a polygon around a building. Look at the GeoJSON text. Which coordinates come first? Save the file and read it into R with `sf::read_sf()`.
6. **Read the metadata.** Find a dataset on NC OneMap, https://www.nconemap.gov, and look at its metadata. Who made it, when, and in what coordinate system? Would you trust it to make a decision?

12.8 Further reading

- *GIS Fundamentals: A First Text on Geographic Information Systems*, by Paul Bolstad, covers data models, data structures, and file formats in more depth, with many diagrams.
- *Geocomputation with R* (2nd ed.), by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, free at https://r.geocompx.org. Its “Geographic data in R” chapter explains how sf and terra store vector and raster data.
- The GeoJSON standard, RFC 7946, at https://datatracker.ietf.org/doc/html/rfc7946, is short and readable.

13 Location and Coordinate Systems

Learning objectives

By the end of this chapter you should be able to:

1. Explain why every location in a computer needs a datum, a coordinate system, and a projection.
2. Describe the ellipsoid and the datums used in North America (NAD27, NAD83, and WGS84).
3. Convert between decimal degrees and degrees, minutes, and seconds.
4. Use latitude, longitude, parallels, meridians, and the graticule correctly.
5. Name the four kinds of map distortion and the projection families that preserve each.
6. Choose a projection for a task, and check and change a layer's coordinate reference system in R.

The last chapter showed that a map is a model of the world. This chapter takes on what I call the first central problem of spatial data in a computer: **how do we represent the world in a computer?**

Computers store numbers, not places. So every location has to become a pair of coordinates. Three building blocks make that possible:

1. A **datum**: a model of the Earth's size and shape, tied to the ground.
2. A **coordinate system**: a grid laid on that model.
3. A **map projection**: a way to flatten the grid onto a flat map or screen.

We take them in that order.

13.1 Where are we?

How would you tell someone where you are right now? You might give:

- an **address**, such as 1600 Pennsylvania Ave.;
- a pair of **coordinates**, such as 35° 54′ 47″ N, 79° 3′ 21″ W;
- a **landmark**, such as "turn left at the cairn";
- a **relative location**, such as the "you are here" dot on a mall directory.

All of these are valid ways to say where something is. But only coordinates are unambiguous, and easy for a computer to store, compare, and measure between. A landmark only helps people who know it, and "you are here" only works on that one map. Location always depends on what you measure from, and at what scale.



13.2 Geodesy: the shape of the Earth

13.2.1 Measuring the Earth

Geodesy is the science of measuring the Earth's size, shape, orientation, and gravity. Every coordinate you will ever use rests on a geodetic model of the Earth.

People have measured the Earth for a long time. Around 240 BCE, the Greek scholar Eratosthenes noticed that at noon on the summer solstice, the sun shone straight down a well in Syene, in southern Egypt. At the same moment in Alexandria, to the north, a vertical stick cast a shadow at an angle of about 7.2°. That is one-fiftieth of a circle, so he multiplied the distance between the cities by 50 to estimate the Earth's circumference. His answer was close to the modern value of about 40,000 km. Just how close depends on the length of the *stadion*, the unit he used, which historians still argue about.

13.2.2 Sphere or ellipsoid?

Is the Earth a sphere? Close, but not quite. Because it spins, the Earth bulges at the Equator and is flattened at the poles. Its shape is an **ellipsoid** (also called a **spheroid**): a sphere squashed a little along its axis.

- The **semi-major axis**, *a*, runs from the center to the Equator: about 6,378 km.
- The **semi-minor axis**, *b*, runs from the center to a pole: about 6,357 km.

The difference is only 21 km, but it matters for precise mapping. We describe it with the **flattening**:

$$f = \frac{a - b}{a} \approx \frac{1}{298}$$

A sphere is fine for a small-scale map of the world. For anything precise, we use an ellipsoid. Different ellipsoids were fitted at different times and for different places. **Clarke 1866** was fitted for North America; **GRS80** and **WGS84** fit the whole globe, and are almost identical to each other.

The real Earth is lumpier still. The surface that gravity defines, roughly where mean sea level would be if the oceans flowed under the continents, is called the **geoid**. The ellipsoid is the simplest smooth shape that comes close to it.

You can work out the flattening yourself from the WGS84 axes:

```
a <- 6378137        # semi-major axis, meters (WGS84)
b <- 6356752.314    # semi-minor axis, meters

f <- (a - b) / a
f
1 / f               # the "1 in 298" you'll see quoted
(a - b) / 1000      # the difference, in km
```

13.2.3 Datums

An ellipsoid is a shape floating in space. A **datum** pins it to the real Earth. A geodetic datum says which ellipsoid to use and exactly where it sits relative to the Earth, so that a latitude and longitude point to one place on the ground.

On the ground, a datum is made real by thousands of surveyed **control points**. The brass disks you can find set into bridges, sidewalks, and mountain tops are **benchmarks** placed by the National Geodetic Survey. A witness post nearby often warns people not to disturb them.

Three datums show up again and again in North American data:

Table 13.1: Common datums in North America.

Datum	Ellipsoid	Notes
NAD27 (North American Datum of 1927)	Clarke 1866	Surveyed from one origin point, at Meades Ranch, Kansas
NAD83 (North American Datum of 1983)	GRS80	Centered on the Earth's mass; readjusted several times; now maintained through **CORS**, a network of Continuously Operating Reference Stations
WGS84 (World Geodetic System 1984)	WGS84	The global datum that GPS uses

In North America, NAD83 and WGS84 differ by only a meter or two. But moving from NAD27 to NAD83 can shift a point by tens of meters or more. Change the datum and the same coordinates point to a different place on the ground, so always check which datum your data use.

13.3 Building a coordinate system

With a model of the Earth in hand, we can lay a grid on it. Building a coordinate system takes two anchors, and both come from the Earth's rotation:

1. **The poles.** The North and South Poles are where the Earth's spin axis meets the surface.
2. **The Equator.** The circle halfway between the poles. It is 0° latitude, and it divides the Northern Hemisphere from the Southern.

Then we draw the grid (Figure 13.1):

- **Parallels** are lines of **latitude**. They run east–west, parallel to the Equator, and measure from 0° at the Equator to 90° north or south at the poles.
- **Meridians** are lines of **longitude**. They run from pole to pole, meet at the poles, and measure from 0° to 180° east or west.

Unlike the Equator, the starting line for longitude has no natural place. It had to be agreed on. In 1884, an international conference chose the meridian through the Royal Observatory in Greenwich, England, as the **Prime Meridian**, 0° longitude. (So the Prime Meridian is 0° W, or 0° E. And the Equator is 0° N, or 0° S. Zero is zero.)

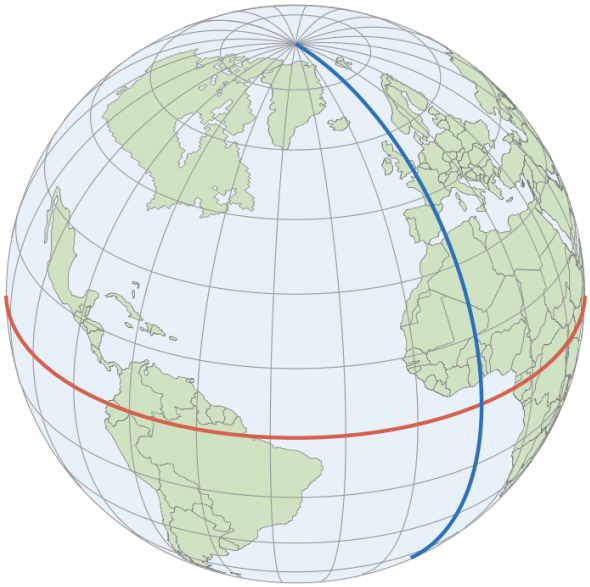


Figure 13.1: The graticule on a globe. The Equator is in red and the Prime Meridian in blue. Parallels and meridians are drawn every 15°.

A full circle of longitude around the Earth is 360°. Latitude only covers 180°, from pole to pole.

13.3.1 The graticule

The **graticule** is the network of parallels and meridians drawn on a map. It is a reference grid, not data. Which lines should you show? Match the interval to the scale: every 30° on a world map, every 1° or less on a map of a state. Keep the lines thin and light so they don't compete with the data. On many thematic maps you can leave them off.

13.3.2 A degree isn't a fixed distance

Meridians meet at the poles, so the parallels get shorter as you move away from the Equator. One degree of latitude is always about 111 km. But one degree of longitude is about 111 km at the Equator, and shrinks to zero at the poles. It is roughly 111.32 km times the cosine of the latitude:

```
lat <- c(0, 30, 36, 60, 89)   # Chapel Hill is near 36 degrees north
km_per_degree_lon <- 111.32 * cos(lat * pi / 180)
round(km_per_degree_lon, 1)
```

At Chapel Hill's latitude, a degree of longitude is about 90 km. This is why you can't treat degrees as if they were meters. We'll come back to it in Section 13.7.

13.4 Writing coordinates

There are two common ways to write the same angle:



- **Decimal degrees (DD)** are a single signed number, such as –104.54°. Negative means south latitude or west longitude. This is the format GIS software and GPS units store.
- **Degrees, minutes, and seconds (DMS)** split the angle into whole degrees, minutes (′), and seconds (″), such as 104° 32′ 24″ W. A letter, N, S, E, or W, replaces the sign.

There are 60 minutes in a degree and 60 seconds in a minute, so 3,600 seconds in a degree.

DD to DMS. Take the whole number of degrees. Multiply the decimal part by 60 to get minutes. Multiply the decimal part of that by 60 to get seconds. For –104.54°: 0.54 × 60 = 32.4′, and 0.4 × 60 = 24″. So –104.54° is 104° 32′ 24″ W.

DMS to DD. Add the parts back up: degrees + minutes / 60 + seconds / 3600, then add the sign.

```
# DMS to decimal degrees
dms_to_dd <- function(deg, min, sec) {
  deg + min / 60 + sec / 3600
}
-dms_to_dd(104, 32, 24)   # west, so negative

# Decimal degrees to DMS
dd  <- abs(-104.54)
deg <- floor(dd)
min <- floor((dd - deg) * 60)
sec <- ((dd - deg) * 60 - min) * 60
c(deg, min, round(sec, 2))
```

GIS software almost always wants decimal degrees, so this conversion comes up when you type in coordinates from older maps or documents.

13.5 Map projections

A grid on a globe is still not a flat map. Before we flatten it, it helps to see how the pieces stack up. Read Table 13.2 from the bottom up: each layer rests on the one below it.

Table 13.2: Building a projection, from the Earth up.

Layer	What it adds
Map projection	Flattens the curved grid onto a plane, in meters or feet
Geographic coordinate system	Latitude and longitude, in degrees
Datum	Pins the ellipsoid to the real Earth
Ellipsoid	A smooth mathematical model of the Earth's shape
The Earth	Lumpy and irregular

If you change a lower layer, everything above it shifts. That is why two datasets must share a datum and a projection before they will line up.

13.5.1 Projecting the globe

A **map projection** is a method for moving the graticule from the globe to a flat surface. Picture a clear globe with a light bulb inside it. The bulb throws the shadow of the grid onto a sheet of

paper. Wrap the paper into a cylinder around the globe, roll it into a cone, or hold it flat against one point, and you get the three classic projection surfaces: **cylindrical**, **conic**, and **planar** (or azimuthal).

13.5.2 Distortion

You can't flatten an orange peel without stretching or tearing it, and you can't flatten the globe without distortion. Every projection distorts at least one of four properties:

1. **Shape**
2. **Area**
3. **Distance**
4. **Direction**

The Mercator projection, for example, keeps local shapes and angles true, which made it ideal for navigation. But it badly inflates areas near the poles. On a Mercator map, Greenland looks about as big as Africa. In fact Africa is about 14 times larger. Greenland is about the same size as the Democratic Republic of the Congo, roughly 2.2 million km² against 2.3 million km². Compare the two red countries in Figure 13.2.

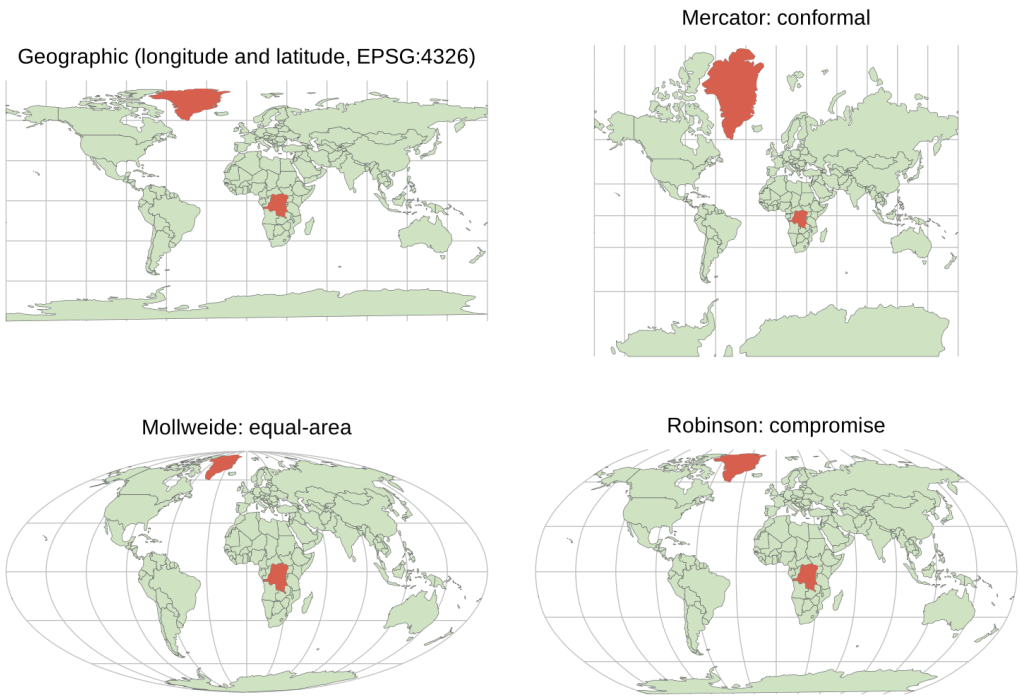


Figure 13.2: The world in four projections. Greenland and the Democratic Republic of the Congo, in red, have almost the same area.

The first map in Figure 13.2 is not really a projection at all. It plots longitude as x and latitude as y, as if degrees were a flat grid. Many programs draw unprojected data this way, which is why a map of raw latitude and longitude looks stretched east–west away from the Equator.

13.5.3 Why distance distorts

Imagine a flat map surface that cuts through the ellipsoid along two lines, like a knife slicing an orange. A surface that cuts the globe this way is called **secant**. Along the two lines where

the surfaces meet, called **standard lines**, the scale is exactly true. Between them, distances on the map shrink. Outside them, distances stretch. The farther you go from a standard line, the bigger the distortion. This is why projections designed for small areas, with standard lines close by, distort so little.

13.5.4 Projection families

Every projection is designed to keep some properties at the cost of others (Table 13.3).

Table 13.3: The four projection families and what each preserves.

Property	Preserves	Examples
Conformal	Local shape and angles	Mercator, Lambert Conformal Conic
Equal-area	Area	Albers Equal-Area Conic, Mollweide
Equidistant	Distance from one point, or along chosen lines	Azimuthal Equidistant, Equidistant Conic
Azimuthal	Direction from a center point	Lambert Azimuthal Equal-Area, Gnomonic

The key trade-off: **no projection can be both conformal and equal-area.** Some **compromise** projections, such as Robinson, preserve nothing exactly but keep every distortion moderate, which is why they are popular for world maps.

> **Think about it**
>
> If you had to pick one projection for every map you will ever make, which would you choose, and why? What would go wrong with your choice for a map of the whole world? For a map of Chapel Hill?

13.5.5 The State Plane Coordinate System

For local work in the United States, surveyors and governments use the **State Plane Coordinate System**: a set of local projections, with one or more zones per state. Each zone is small enough to keep scale error under 1 part in 10,000. Zones that run long north–south use **Transverse Mercator**; zones that run long east–west use **Lambert Conformal Conic**. Coordinates are in feet or meters, not degrees.

North Carolina is wide from east to west, so it has a single Lambert Conformal Conic zone. That is the projection we use in this book: NAD83 / North Carolina, EPSG code **32119**, in meters.

13.6 Choosing a projection

There is no single best projection. Two rules of thumb help you choose:

Fit your area. The smaller the area, the less distortion.

- Working in Chapel Hill or Raleigh? The North Carolina State Plane zone (EPSG:32119).



- Working across the Southeast? A UTM zone, such as UTM Zone 17N, or an Albers projection for the region.
- Working across the lower 48 states? Albers Equal-Area Conic (EPSG:5070) is the standard.

Preserve what you measure.

- Comparing areas or densities? Use an equal-area projection. Over a small area, such as a city, a conformal projection is close enough.
- Measuring distances? Use an equidistant projection, or an azimuthal equidistant one centered on your starting point.
- Measuring directions from a point? Use an azimuthal projection centered on that point.
- Keeping shapes? Use a conformal projection.

And if you work with elevation data, make sure the horizontal units match the vertical ones. Mixing feet and meters wrecks slope and hillshade calculations.

13.7 Coordinate reference systems in R

In R, the datum, the coordinate system, and the projection travel together as a **coordinate reference system** (CRS). Every sf object carries one. Most are identified by an **EPSG code**, a number from a public registry: 4326 is WGS84 latitude and longitude, and 32119 is NC State Plane in meters.

Check a layer's CRS with `st_crs()`, and change it with `st_transform()`:

```
library(sf)

nc <- st_read(system.file("shape/nc.shp", package = "sf"))

st_crs(nc)$epsg          # 4267: NAD27, in degrees
st_crs(nc)$units         # NULL: a geographic CRS has no linear units

nc_sp <- nc |> st_transform(32119)
st_crs(nc_sp)$units      # "m"
```

`st_transform()` recalculates every coordinate. It is different from `st_set_crs()`, which only changes the label and leaves the numbers alone. Use `st_set_crs()` only when a layer has no CRS and you know what it should be.

13.7.1 Degrees are not meters

Why does it matter? Chapel Hill and Raleigh are about 0.44 degrees apart. That number is not a distance you can use: a degree of longitude here is about 90 km, but a degree of latitude is about 111 km, so there's no single factor to convert it. Measured properly, the two cities are about 40.5 km apart:

```
towns <- st_sfc(st_point(c(-79.0558, 35.9132)),   # Chapel Hill
                st_point(c(-78.6382, 35.7796)),   # Raleigh
                crs = 4326)

st_distance(towns)                        # about 40,465 m, on the ellipsoid
```

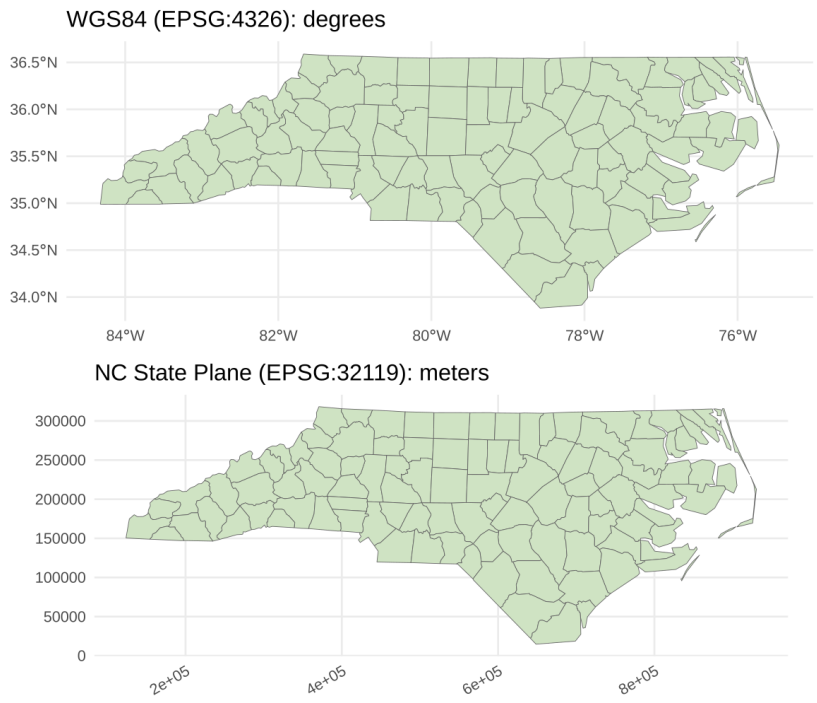


Figure 13.3: North Carolina in WGS84 latitude and longitude (top) and in NC State Plane (bottom). The shape changes, and so do the units on the axes.

```
st_distance(st_transform(towns, 32119))  # about 40,530 m, in State Plane
```

For latitude and longitude, sf measures distance on the curved Earth, so `st_distance()` still returns meters. But many other operations, and many other programs, treat coordinates as if they were flat. Before you measure, buffer, or join by distance, check the CRS with `st_crs()` and project your data with `st_transform()`. And before you combine two layers, make sure they share the same CRS.

13.8 Check your understanding

1. The science of measuring the Earth's size, shape, orientation, and gravity is called: [1]

- (A) Cartography
- (B) Geodesy
- (C) Geology
- (D) Topography

2. The Earth's equatorial radius is larger than its polar radius. TRUE / FALSE[2]
3. Which datum does GPS use? [3]

- (A) NAD27

[1] Geodesy
[2] TRUE
[3] WGS84

- (B) Clarke 1866
- (C) WGS84
- (D) State Plane

4. Lines of latitude are called: [4]

- (A) Parallels
- (B) Meridians
- (C) Graticules
- (D) Datums

5. Convert –79.25° (a longitude) to degrees, minutes, and seconds. How many minutes? _ [5]

6. Which property does the Mercator projection preserve? [6]

- (A) Area
- (B) Shape (it is conformal)
- (C) Distance everywhere
- (D) Nothing at all

7. A projection can be both conformal and equal-area. TRUE / FALSE[7]

8. Which function changes the coordinates of an sf layer to a new CRS? [8]

- (A) st_crs()
- (B) st_set_crs()
- (C) st_transform()
- (D) st_distance()

Show hint for question 5

Take the decimal part, 0.25, and multiply by 60. The answer is 79° 15′ 0″ W.

13.9 Summary

- To store a location, a computer needs a **datum** (a model of the Earth tied to the ground), a **coordinate system** (a grid on that model), and a **map projection** (a way to flatten it).

[4] Parallels
[5] 15
[6] Shape (it is conformal)
[7] FALSE
[8] st_transform()



- **Geodesy** measures the Earth. Its shape is an **ellipsoid**, flattened at the poles by about 1/298. Common North American datums are **NAD27**, **NAD83**, and **WGS84**.
- **Latitude** runs from 0° at the **Equator** to 90° at the poles along **parallels**; **longitude** runs 0° to 180° east or west of the **Prime Meridian** along **meridians**. The grid of both is the **graticule**.
- Coordinates can be written in **decimal degrees** or **degrees, minutes, and seconds**. A degree of longitude shrinks toward the poles, so degrees are not a distance.
- Every projection distorts **shape**, **area**, **distance**, or **direction**. **Conformal**, **equal-area**, **equidistant**, and **azimuthal** projections each protect one property, and none is both conformal and equal-area.
- Choose a projection that fits your area and preserves what you measure. For North Carolina, use **State Plane** (EPSG:32119).
- In R, check a layer's **CRS** with `st_crs()` and change it with `st_transform()`.

13.10 Exercises

1. **Your own coordinates.** Find the latitude and longitude of the building you're in, using a phone or an online map. Write them in decimal degrees and in degrees, minutes, and seconds.
2. **Eratosthenes.** The distance from Syene to Alexandria was about 5,000 stadia, and the shadow angle was 7.2°. What circumference did Eratosthenes get, in stadia? If a stadion was 157.5 m, how far off was he from 40,075 km?
3. **The true size.** Use https://www.thetruesize.com to drag Greenland, Alaska, and North Carolina toward the Equator. What happens to their size on the Mercator map, and why?
4. **Pick a projection.** Choose a projection, and say why, for (a) a map comparing forest area in every country; (b) a map showing flight distances from RDU airport; (c) a map of parcels in Orange County.
5. **Check the CRS.** Load the `nc` data from the sf package (see Section 13.7). What datum is it in? Transform it to EPSG:32119 and to EPSG:5070, and plot all three. What changes?
6. **Degrees and distance.** Pick two towns in North Carolina, one north of the other, and two at the same latitude, about the same number of degrees apart. Use `st_distance()` to compare the distances. Explain the difference.

13.11 Further reading

- *GIS Fundamentals: A First Text on Geographic Information Systems*, by Paul Bolstad, has a full chapter on geodesy, datums, map projections, and State Plane coordinates.
- *Geocomputation with R* (2nd ed.), by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, free at https://r.geocompx.org. See the chapter "Reprojecting geographic data".
- *Map Projections: A Working Manual*, by John P. Snyder (U.S. Geological Survey, 1987), is the classic reference, free from the USGS.
- The EPSG registry at https://epsg.io lets you search for any coordinate reference system by name or code.

14 Census Data and APIs

Learning objectives

By the end of this chapter you should be able to:

1. Explain where US Census data come from, and how the decennial census differs from the American Community Survey (ACS).
2. Describe the Census geographic hierarchy and read a GEOID.
3. Choose an appropriate geography, survey, and variable for a research question.
4. Explain what an API is and build a Census API request by hand.
5. Download Census data with boundaries attached using **tidycensus**, and map it.
6. Work responsibly with margins of error, percentages, and comparisons across years.

Most of the spatial data science you will do about people in the United States starts with Census data. Population, income, housing, commuting, education: if a question involves who lives where, the Census Bureau has probably measured it, for small areas, across the whole country, for free.

This chapter has two halves. The first covers *what* the data are: where they come from, how Census geography is organized, and which variables you will use most. The second covers *how* to get them: what an API is, how to make a request by hand, and how the **tidycensus** package turns that request into one line of R that returns a map-ready data frame.

14.1 A brief history of the US Census

Article I of the Constitution requires a count of the population every ten years so that seats in the House of Representatives can be apportioned among the states. The first census, in 1790, was carried out by U.S. marshals on horseback and counted 3.9 million people. The count has run every decade since.

The Census Bureau has repeatedly pushed new technology into being because the job of counting a growing country kept outgrowing the old tools (Table 14.1). Herman Hollerith's punch-card tabulating machines, built for the 1890 census, led to the company that became IBM. A century later, the 1990 census introduced TIGER, one of the first national digital map databases. The boundary files you will download later in this chapter descend directly from it.

Table 14.1: Milestones in the history of the US Census.

Year	Milestone	What changed
1790	First census	U.S. marshals count 3.9 million people, as the Constitution requires



Year	Milestone	What changed
1890	Punch cards	Herman Hollerith's machines speed up tabulation
1902	Permanent bureau	Congress makes the Census Office permanent
1940	Sampling begins	Some questions go to a 5% sample of households
1990	TIGER	Digital street and boundary files for the whole country
2005	ACS launches	A yearly survey replaces the decennial "long form"
2020	Online response	First census answered online; differential privacy added

For our purposes the most important change is the one from 2005. Before then, detailed social and economic questions (income, education, commuting) were asked on a "long form" sent to a sample of households once a decade. Since 2005 those questions have moved to the **American Community Survey**, which samples households every year. That gives two distinct sources of Census data, and you will choose between them constantly:

- The **decennial census** is a full count of everyone, every ten years. It asks only a few questions (age, sex, race, Hispanic origin, household relationship, housing tenure), but it covers every person.
- The **American Community Survey (ACS)** is a continuous sample survey. It asks many more questions, but every number it publishes is an *estimate*, with a margin of error (Section 14.7.1).

14.1.1 The moving center of population

Each census, the Bureau calculates the **mean center of population**: the point where a flat map of the United States would balance if every resident weighed the same. In 1790 it sat in Kent County, Maryland, on the Eastern Shore. By 2020 it had moved about 884 miles, in a straight line, to Wright County, Missouri (Figure 14.1) (U.S. Census Bureau, n.d.).

```
lower48 <- states(cb = TRUE, resolution = "20m", year = 2020) |>
  filter(!STUSPS %in% c("AK", "HI", "PR")) |>
  st_transform(5070)

# Draw each series as its own path, plus a dashed link between the two 1950 points
paths <- centers |>
  group_by(extent) |>
  summarize(do_union = FALSE) |>
  st_cast("LINESTRING")

link_1950 <- centers |>
  filter(year == 1950) |>
  summarize(do_union = FALSE) |>
  st_cast("LINESTRING")

labeled <- centers |>
  filter(year %in% c(1790, 1850, 1900, 1950, 2020), extent == "Lower 48" | year == 20
```

```
# Zoom to the eastern two-thirds of the country, where the path lies
zoom <- st_bbox(st_buffer(centers, 600000))

ggplot() +
  geom_sf(data = lower48, fill = "#f6efe0", color = "#b9ad97", linewidth = 0.3) +
  geom_sf(data = link_1950, color = "#3b5b7a", linetype = "dashed", linewidth = 0.5) +
  geom_sf(data = paths, color = "#3b5b7a", linewidth = 0.7) +
  geom_sf(data = centers, aes(fill = year), shape = 21, size = 2.6, color = "white",
  geom_sf_label(
    data = labeled, aes(label = year), nudge_y = 75000,
    size = 3.2, label.size = 0, fill = alpha("white", 0.8)
  ) +
  scale_fill_gradient(low = "#9fb7cc", high = "#1d3550", name = "Census year") +
  coord_sf(xlim = zoom[c("xmin", "xmax")], ylim = zoom[c("ymin", "ymax")], expand = F
  theme_void() +
  theme(legend.position = "bottom", legend.key.width = unit(2, "cm"))
```

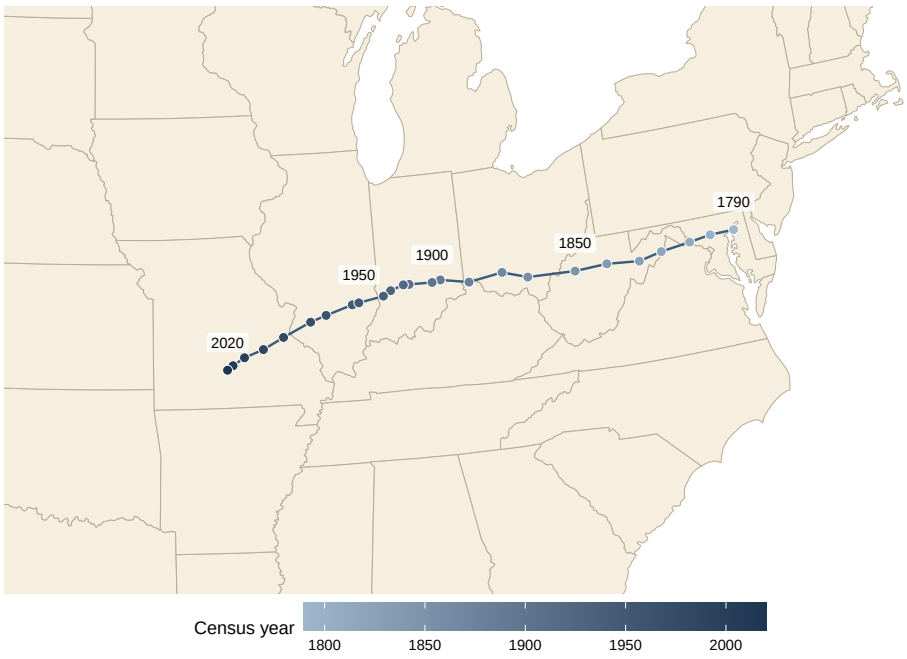


Figure 14.1: Mean center of US population at each census, 1790–2020. Centers through 1950 are for the lower 48 states; from 1950 on they include Alaska and Hawaii, which pulls the 1950 point slightly west and south (dashed link). Source: U.S. Census Bureau, Centers of Population, 1790–2020.

The path tells the story of American settlement. Through the 1800s the center moved almost straight west as settlement pushed across the continent, fastest before 1900. After 1950 it bent southwest as the Sun Belt grew, and it has been in Missouri since 1980.

Think about it

Which direction do you expect the center of population to move by 2030? What trends in migration, birth rates, or immigration would push it further west, further south, or back east?

14.2 Census geography

Census data are always published for *areas*, never for individual people or addresses. Understanding how those areas are defined, and how they fit together, is the first step in using the data well.

Standard Hierarchy of Census Geographic Entities

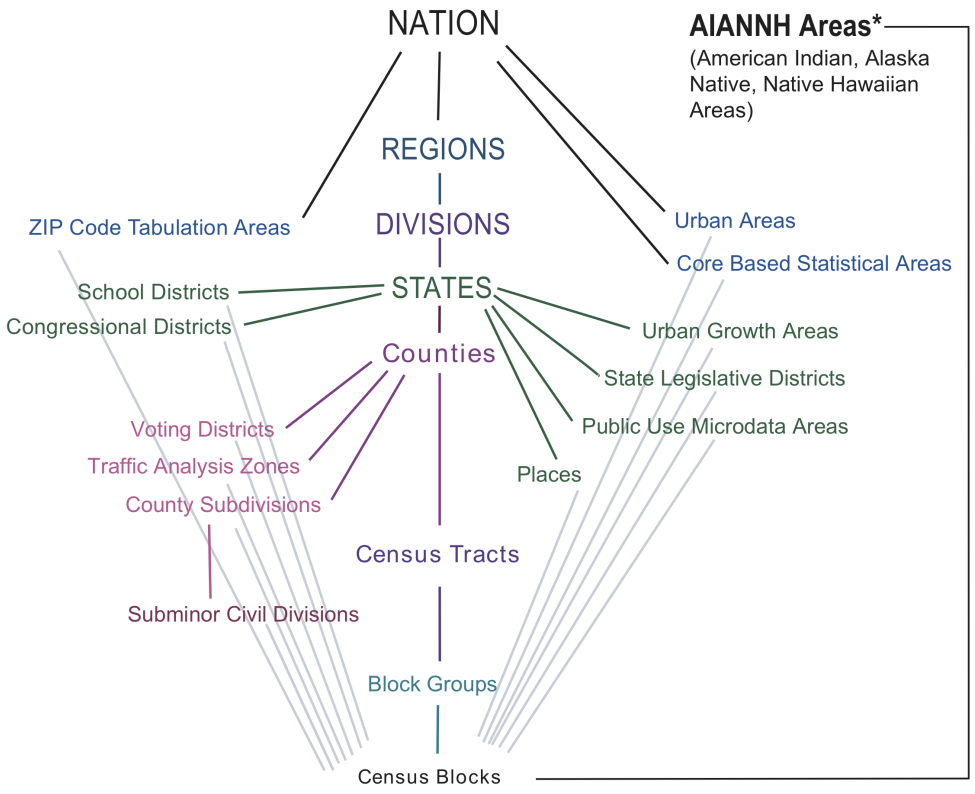


Figure 14.2: The standard hierarchy of Census geographic entities. The central spine, from the nation down to blocks, nests completely. The entities branching off it do not. Source: U.S. Census Bureau.

Figure 14.2 shows the standard hierarchy. The central spine runs from the nation to regions, divisions, states, counties, census tracts, block groups, and finally blocks. Every unit on the spine **nests** perfectly: each block lies inside exactly one block group, which lies inside exactly one tract, one county, and one state. That nesting is what makes the spine so useful. You can add blocks up to block groups, block groups to tracts, and so on, and never double-count anyone.

The geographies that branch off the spine, such as ZIP Code Tabulation Areas, places (cities and towns), and congressional districts, are built from blocks too, but they do not nest neatly inside counties or tracts. A ZIP code can cross county lines, and a city can include parts of several tracts. When you have a choice, work on the spine.

The three smallest units on the spine are the ones you will use most for local analysis. Figure 14.3 shows all three for Orange County, North Carolina.

```
orange_tracts <- tracts("NC", "Orange", year = 2020)
orange_bgs    <- block_groups("NC", "Orange", year = 2020)
orange_blocks <- blocks("NC", "Orange", year = 2020)

geo_levels <- bind_rows(
  orange_tracts |> transmute(level = "Tracts"),
  orange_bgs    |> transmute(level = "Block groups"),
  orange_blocks |> transmute(level = "Blocks")
) |>
  mutate(level = fct_inorder(level))

ggplot(geo_levels) +
  geom_sf(fill = "#f6efe0", color = "#3b5b7a", linewidth = 0.1) +
  facet_wrap(~level) +
  theme_void()
```

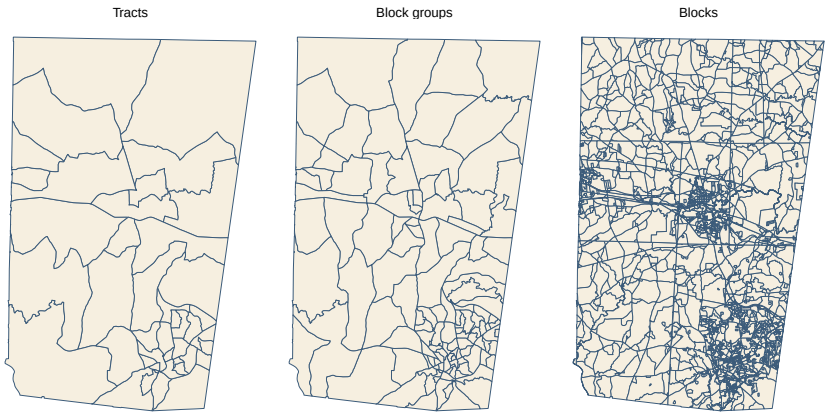


Figure 14.3: Census tracts, block groups, and blocks in Orange County, North Carolina (2020 boundaries). Each level subdivides the one above it.

Census blocks are the smallest unit. A block is often a single city block bounded by streets, but blocks are bounded by any visible feature (railroads, streams, shorelines) and in rural areas can be large. Only the decennial census is published for blocks, and in 2020 block-level data carry the most differential-privacy noise (Section 14.7.3), so use them with care.

Block groups combine several blocks and usually hold 600 to 3,000 people. They are the smallest geography for which ACS 5-year estimates are published, which makes them the finest level available for most social and economic variables. That detail comes at a cost: block group estimates have large margins of error.

Census tracts combine block groups and typically hold 1,200 to 8,000 people, about 4,000 on average. They are drawn to be fairly similar in population and stable over time, which makes them the workhorse unit for neighborhood analysis. Their boundaries are still revised each decade, though, so comparisons across censuses take care (Section 14.7.4).

14.2.1 FIPS codes and GEOIDs

Every Census area has a unique identifier called a **GEOID**, built by stringing together **FIPS codes** (Federal Information Processing Standards codes) for each level of the hierarchy. FIPS codes are standard numeric IDs for states and counties: North Carolina is 37 and Orange County is 135.

Table 14.2: Reading a GEOID, using a tract in Orange County, NC. The tract number is illustrative. A block GEOID appends a four-digit block number to the *tract* GEOID, and the first digit of that block number is the block group.

Part	Example	Digits	GEOID so far
State	37 (North Carolina)	2	37
County	135 (Orange)	3	37135
Tract	011600	6	37135011600 (11 digits)
Block group	2	1	371350116002 (12 digits)
Block	2004	4	371350116002004 (15 digits)

Read Table 14.2 from the top down. Because the parts always have the same number of digits, you can recover the state, county, and tract of any block just by slicing its GEOID, for example with `str_sub(GEOID, 1, 5)` for the county.

> **Keep GEOIDs as text**
>
> GEOIDs look like numbers, but treat them as text. Alabama's state FIPS code is `01`. Read it as a number and it becomes `1`, every Alabama GEOID loses a digit, and your joins quietly fail. When you read a CSV, set the column type explicitly, for example `read_csv(file, col_types = cols(GEOID = col_character()))`.

The GEOID is also the key you will join on. Census tables and Census boundary files both carry a `GEOID` column, so attaching data to shapes is a single `left_join()` on that field. (Chapter 15 covers joins in detail.)

14.2.2 What data are available at each level

Not every table is published for every geography. As areas get smaller, the Bureau publishes fewer tables and adds more noise to protect privacy. Table 14.3 summarizes the 2020 decennial products.

Table 14.3: Variables available in the 2020 Census redistricting (PL 94-171) and DHC products, by geographic level.

Geographic level	Redistricting (PL 94-171)	Demographic and Housing Characteristics (DHC)	Notes
Nation, state, county	All basic counts and race/voting-age splits	All (2,400+ variables)	Full data availability
Place, MCD, metro area	All basic counts	Most DHC tables	Standard demographic and housing tables
Census tract	All basic counts	Extensive, including single-year age	Good balance of detail and privacy noise
Block group	All basic counts	Subset of DHC tables	Detailed race and age cross-tabs omitted or generalized

Geographic level	Redistricting (PL 94-171)	Demographic and Housing Characteristics (DHC)	Notes
Block	Basic population, race, voting-age, and housing counts	Core counts only (P1, H1, basic tables)	Most affected by differential privacy; detailed cross-tabs absent

14.2.3 ACS 1-year and 5-year estimates

The ACS publishes estimates for different periods (Table 14.4). **1-year** estimates are the most current but are only published for areas of 65,000 people or more: large counties, cities, and metro areas. **5-year** estimates pool 60 months of responses. They are the most reliable and the only ACS product published for tracts and block groups. In tidycensus you choose between them with the `survey` argument: `"acs1"` or `"acs5"` (the default).

Table 14.4: ACS period estimates. *The 3-year product was discontinued after the 2011–2013 release. Source: U.S. Census Bureau, American Community Survey.

	1-year	1-year supplemental	3-year*	5-year
Collection period	12 months (e.g., 2023)	12 months (e.g., 2023)	36 months (e.g., 2011–2013)	60 months (e.g., 2019–2023)
Areas covered	Populations of 65,000+	Populations of 20,000+	Populations of 20,000+	All areas
Sample size	Smallest	Smallest	Larger than 1-year	Largest
Reliability	Lowest	Lower than 5-year	Middle	Highest
Currency	Most current	Most current	Middle	Least current
Released	Annually since 2005	Annually since 2014	2007–2013	Annually since 2009

A 5-year estimate labeled 2022 describes the whole period 2018–2022, not the year 2022. Keep that in mind when you write about the results.

14.3 Commonly used variables

Census variables have codes rather than names. An ACS code such as `B19013_001` has two parts: the **table** (`B19013`, median household income) and the **line** within it (`001`). Line `_001` is usually the table total. In raw API output, ACS codes end in `E` for the estimate or `M` for the margin of error (`B19013_001E`, `B19013_001M`). tidycensus adds these suffixes for you, so the tables below omit them. Decennial codes look different (`P1_001N`).

The tables below list the variables that come up most often in student projects. To find others, see Section 14.6.1.

Table 14.5: Population and race/ethnicity variables.

Variable	Code	Source	Smallest geography
Total population	B01003_001	ACS 5-year	Block group
Total population (full count)	P1_001N	2020 Census (PL)	Block
Race: White, Black, Asian alone	B02001_002, _003, _005	ACS 5-year	Block group
Hispanic or Latino (any race)	B03002_012	ACS 5-year	Block group
Non-Hispanic White alone	B03002_003	ACS 5-year	Block group

Race and Hispanic origin are separate questions in the Census: a person can be Hispanic and of any race. If you add up the race groups in B02001, Hispanic residents are spread across them. When you want mutually exclusive groups that sum to the total (Hispanic, non-Hispanic White, non-Hispanic Black, and so on), use table B03002.

Table 14.6: Age and household variables.

Variable	Code	Source	Smallest geography
Median age	B01002_001	ACS 5-year	Block group
Population by sex and age group	B01001_002 to _049	ACS 5-year	Block group
Total households	B11001_001	ACS 5-year	Block group
Average household size	B25010_001	ACS 5-year	Block group
Housing units (full count)	H1_001N	2020 Census (PL)	Block

Table B01001 splits the population by sex and by age group, so many common measures require adding several lines together. The population under 5, for example, is males under 5 (B01001_003) plus females under 5 (B01001_027). You will do exactly this in Section 14.6.

Table 14.7: Income and poverty variables. For poverty at the block group level, sum C17002_002 and _003 (income under 100% of the poverty line). B17001 and B19083 are published for tracts and larger.

Variable	Code	Source	Smallest geography
Median household income	B19013_001	ACS 5-year	Block group
Per capita income	B19301_001	ACS 5-year	Block group
Income below poverty level	C17002_002 + _003	ACS 5-year	Block group
Individuals below poverty level	B17001_002	ACS 5-year	Tract
Gini index of income inequality	B19083_001	ACS 5-year	Tract



Medians don't add up

You can add counts across areas, but not medians. The median income of two tracts combined is not the sum, or even the average, of their two medians. If you need a median for a larger area, download it for that area directly.

Table 14.8: Education, work, and housing variables. "No vehicle" combines owner (_003) and renter (_010) households.

Variable	Code	Source	Smallest geography
Bachelor's degree, age 25+	B15003_022	ACS 5-year	Block group
Unemployed, civilian labor force	B23025_005	ACS 5-year	Block group
Renter-occupied housing units	B25003_003	ACS 5-year	Block group
Median home value	B25077_001	ACS 5-year	Block group
Median gross rent	B25064_001	ACS 5-year	Block group
No vehicle available	B25044_003 + _010	ACS 5-year	Block group

Most of these are counts. To compare places of different sizes, divide by the right denominator, which is usually the _001 total of the same table: bachelor's degree holders over the population aged 25 and older (B15003_001), or unemployed people over the civilian labor force (B23025_003). Section 14.7.2 shows how to do this in one step.

Think about it

Write down one question about your own community that Census data could help answer. Which variable would you need, at what geography, and from which survey?

14.4 What is an API?

You could get any of the variables above by clicking through data.census.gov and downloading a CSV. For a one-off table that works. For anything you want to repeat, share, or scale up, it is far better to ask for the data from code, through an API.

An **API** (Application Programming Interface) is a set of rules for how one program asks another for data or services. Your R script is one program; the Census Bureau's server is the other. Your script sends a **request**, and the server sends back **data** (Figure 14.4).

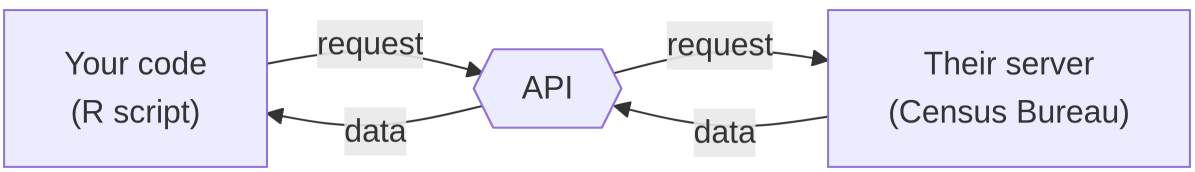


Figure 14.4: An API sits between your code and someone else's server. Your code sends a request in an agreed format, and the server returns data.



A restaurant menu is a useful analogy. The menu lists what you can order and how to ask for it. You never see the kitchen, and you get back exactly what you ordered, as long as you ordered something on the menu.

14.4.1 Web APIs: URLs in, JSON out

Most data APIs on the web follow a style called **REST**. Three ideas cover almost everything you need:

- **Every dataset or service has its own web address**, called an **endpoint**. The 2022 ACS 5-year data live at `https://api.census.gov/data/2022/acs/acs5`.
- **Requests are ordinary HTTP requests**, the same thing your browser sends when it loads a page. A `GET` request reads data; a `POST` request sends data. Data APIs mostly use `GET`.
- **Responses usually come back as JSON** (JavaScript Object Notation), structured text that R can turn into lists or data frames.

14.4.2 Anatomy of a request

A REST request is just a URL with two parts: the endpoint, which says which service to call, and the **parameters**, which say what you want from it. A `?` separates the two, each parameter is written `name=value`, and an `&` separates parameters.

Here is a request to Esri's geocoding service that looks up one address:

```
https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddressCandidat
```

Everything before the `?` is the endpoint. `singleLine` is the address to look up (`%20` is how a space is written inside a URL), and `f=pjson` asks for pretty-printed JSON. Paste that URL into a browser and you will see the JSON response.

Here is a Census API request with the same structure. It asks for median household income for every county in North Carolina:

```
https://api.census.gov/data/2022/acs/acs5?get=NAME,B19013_001E&for=county:*&in=state:37
```

`get` lists the variables, `for` picks the geography (`county:*` means every county), and `in` limits the request to state 37, North Carolina.

14.4.3 Why use an API?

APIs take a little more setup than a download button. They pay that back in four ways:

- **Reproducible.** Your script is an exact record of what you downloaded, so you, a classmate, or a reviewer can rerun it and get the same data.
- **Targeted.** Parameters return only the variables and places you need.
- **Current.** You pull fresh data each time you run the script, instead of keeping stale files around.
- **Scalable.** One loop can request every state, or every year.

These benefits might not seem like much the first time you download a table. They matter the second time you need the same data, or when someone asks exactly how you got it.



14.5 Calling an API from R

There are two ways to call an API from R: build the request yourself, or use a package that wraps the API for you. It is worth seeing both, because a wrapper package won't always exist.

14.5.1 Step 0: get and store a key

The Census API asks you to identify yourself with a free key. Request one at https://api.census.gov/data/key_signup.html; it arrives by email within a few minutes. Then store it once:

```
tidycensus::census_api_key("YOUR KEY HERE", install = TRUE)
```

`install = TRUE` writes the key to your `.Renviron` file as CENSUS_API_KEY, so R loads it at the start of every session. Restart R after you run this. From then on, your code reads the key with `Sys.getenv("CENSUS_API_KEY")`.

> **Keep keys out of your scripts**
>
> Never paste an API key directly into a script or Quarto document you might share or push to GitHub. Anyone who has your key can use it as you. Store it in `.Renviron`, which stays on your computer.

14.5.2 By hand, with httr2

Every manual API call follows the same steps, whatever the service (Table 14.9).

Table 14.9: The steps of a manual API call.

Step	Task	R functions	What happens
0	Get a key	`Sys.setenv()` or `.Renviron`	Sign up once; keep the key out of your script
1	Build the request	`request()`, `req_url_query()`	Endpoint URL plus parameters
2	Send it	`req_perform()`	R sends an HTTP GET and waits for the reply
3	Parse the response	`resp_body_json()`	Turn JSON text into a list or matrix
4	Analyze	dplyr, sf, tmap	Clean, join to geometry, and map

Here are steps 1 to 3 for the North Carolina income request, using **httr2**:

```
library(httr2)

resp <- request("https://api.census.gov/data/2022/acs/acs5") |>
  req_url_query(
    get   = "NAME,B19013_001E",
    `for` = "county:*",
```



```
    `in`  = "state:37",
    key   = Sys.getenv("CENSUS_API_KEY")
  ) |>
  req_perform()

nc_raw <- resp_body_json(resp, simplifyVector = TRUE)
head(nc_raw)
```

```
     [,1]                                [,2]          [,3]    [,4]
[1,] "NAME"                              "B19013_001E" "state" "county"
[2,] "Alamance County, North Carolina"   "60866"       "37"    "001"
[3,] "Alexander County, North Carolina"  "62764"       "37"    "003"
[4,] "Alleghany County, North Carolina"  "42115"       "37"    "005"
[5,] "Anson County, North Carolina"      "42000"       "37"    "007"
[6,] "Ashe County, North Carolina"       "49176"       "37"    "009"
```

`for` and `in` are reserved words in R, so they need backticks when used as argument names. Look at what comes back: a matrix of text, with the column names in the first row and every value, including income, stored as a string. Step 4 starts with cleaning it:

```
colnames(nc_raw) <- nc_raw[1, ]

nc_manual <- as_tibble(nc_raw[-1, ]) |>
  transmute(
    GEOID  = paste0(state, county),
    NAME,
    income = as.numeric(B19013_001E)
  )

nc_manual
```

```
# A tibble: 100 x 3
   GEOID NAME                              income
   <chr> <chr>                              <dbl>
 1 37001 Alamance County, North Carolina    60866
 2 37003 Alexander County, North Carolina   62764
 3 37005 Alleghany County, North Carolina   42115
 4 37007 Anson County, North Carolina       42000
 5 37009 Ashe County, North Carolina        49176
 6 37011 Avery County, North Carolina       53513
 7 37013 Beaufort County, North Carolina    56081
 8 37015 Bertie County, North Carolina      41652
 9 37017 Bladen County, North Carolina      40476
10 37019 Brunswick County, North Carolina   71193
# i 90 more rows
```

14.5.3 With a package: tidycensus

tidycensus (Walker and Herman, n.d.) does steps 1 through 3, and most of step 4, for you. This single call builds the same URL behind the scenes, sends it, and returns a tidy data frame:



```r
nc_income <- get_acs(
  geography = "county",
  variables = "B19013_001",
  state     = "NC",
  year      = 2022
)

nc_income
```

```
# A tibble: 100 x 5
   GEOID NAME                                variable   estimate   moe
   <chr> <chr>                               <chr>         <dbl> <dbl>
 1 37001 Alamance County, North Carolina     B19013_001    60866  2024
 2 37003 Alexander County, North Carolina    B19013_001    62764  4812
 3 37005 Alleghany County, North Carolina    B19013_001    42115  6307
 4 37007 Anson County, North Carolina        B19013_001    42000  4659
 5 37009 Ashe County, North Carolina         B19013_001    49176  3543
 6 37011 Avery County, North Carolina        B19013_001    53513  4190
 7 37013 Beaufort County, North Carolina     B19013_001    56081  2803
 8 37015 Bertie County, North Carolina       B19013_001    41652  4732
 9 37017 Bladen County, North Carolina       B19013_001    40476  4526
10 37019 Brunswick County, North Carolina    B19013_001    71193  2217
# i 90 more rows
```

The result has one row per county with `GEOID`, `NAME`, `variable`, `estimate`, and `moe` (margin of error) columns. The rest of this chapter uses tidycensus. Similar wrapper packages exist for other services: **tigris** for Census boundary files and **arcpullr** for Esri REST services, for example. When there is no package for the API you need, the httr2 pattern above will always work.

14.6 Worked example: mapping young children in North Carolina

This section puts the pieces together. The goal is to map the share of each North Carolina census tract's population that is under 5 years old.

14.6.1 Find the variables

`load_variables()` downloads the full list of variables for a dataset and year:

```r
vars_acs5 <- load_variables(2022, "acs5", cache = TRUE)

vars_acs5 |>
  filter(str_detect(name, "^B01001_")) |>
  head(5)
```

```
# A tibble: 5 x 5
  name       label                                        concept   geography universe
```

```
  <chr>      <chr>                                    <chr>    <chr>     <chr>
1 B01001_001 Estimate!!Total:                         Sex by ~ block gr~ Total p~
2 B01001_002 Estimate!!Total:!!Male:                  Sex by ~ block gr~ Total p~
3 B01001_003 Estimate!!Total:!!Male:!!Under 5 years   Sex by ~ block gr~ Total p~
4 B01001_004 Estimate!!Total:!!Male:!!5 to 9 years    Sex by ~ block gr~ Total p~
5 B01001_005 Estimate!!Total:!!Male:!!10 to 14 years Sex by ~ block gr~ Total p~
```

The result has a `name` column (the code), a `label` describing the line, a `concept` naming the table, and the smallest `geography` it is published for. In RStudio, `View(vars_acs5)` opens it in a searchable viewer. You can also filter it in code, for example `filter(vars_acs5, str_detect(concept, "SEX BY AGE"))`. `cache = TRUE` saves the list so it loads instantly next time.

Searching the labels shows that table `B01001`, *Sex by Age*, has what we need: the total (`_001`), males under 5 (`_003`), and females under 5 (`_027`).

> **Check the concept, not just the label**
>
> Many ACS tables come in lettered versions for single race groups. `B01001A` is *Sex by Age (White Alone)* and `B01001B` is *Sex by Age (Black or African American Alone)*. Their lines have the same labels as `B01001`, so it is easy to grab the wrong one. Always check the `concept` column.

14.6.2 Download data with geometry

`get_acs()` with `geometry = TRUE` downloads the tract boundaries along with the data and joins them on `GEOID`, so the result is an `sf` object, ready to map:

```
nc_tracts <- get_acs(
  geography = "tract",
  variables = c(total    = "B01001_001",
                male_u5  = "B01001_003",
                female_u5 = "B01001_027"),
  state     = "NC",
  year      = 2022,
  output    = "wide",
  geometry  = TRUE
)

nc_tracts
```

```
Simple feature collection with 2672 features and 8 fields (with 12 geometries empty)
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -84.32187 ymin: 33.84232 xmax: -75.46062 ymax: 36.58812
Geodetic CRS:  NAD83
First 10 features:
         GEOID                                                  NAME totalE
1  37037020600     Census Tract 206; Chatham County; North Carolina   4810
2  37105030102      Census Tract 301.02; Lee County; North Carolina   5810
3  37165010300    Census Tract 103; Scotland County; North Carolina   4442
4  37133000800        Census Tract 8; Onslow County; North Carolina   1166
```

```
5  37133002500       Census Tract 25; Onslow County; North Carolina   3501
6  37013930200   Census Tract 9302; Beaufort County; North Carolina   5519
7  37001020100    Census Tract 201; Alamance County; North Carolina   4477
8  37065020700   Census Tract 207; Edgecombe County; North Carolina   1672
9  37129010600 Census Tract 106; New Hanover County; North Carolina   4054
10 37129011000 Census Tract 110; New Hanover County; North Carolina   2190
    totalM male_u5E male_u5M female_u5E female_u5M
1      606       37       37        135        106
2      693      220      170        208        165
3      417      173       76        200         74
4      244      101       76         41         41
5      524       99       65        109         97
6      645      127      111        138         98
7      502      223      102        162        177
8      495        2        4         24         25
9      545       62       54         76         74
10     460       28       40         59         67
                          geometry
1  MULTIPOLYGON (((-79.40216 3...
2  MULTIPOLYGON (((-79.23063 3...
3  MULTIPOLYGON (((-79.48963 3...
4  MULTIPOLYGON (((-77.35379 3...
5  MULTIPOLYGON (((-77.30802 3...
6  MULTIPOLYGON (((-77.19577 3...
7  MULTIPOLYGON (((-79.45802 3...
8  MULTIPOLYGON (((-77.733 36....
9  MULTIPOLYGON (((-77.92597 3...
10 MULTIPOLYGON (((-77.94434 3...
```

Naming the variables (`total = "B01001_001"`) gives the output columns readable names. `output = "wide"` puts each variable in its own pair of columns: `totalE` and `totalM` for the estimate and margin of error, and so on.

14.6.3 Calculate and map

Now add the two under-5 counts, divide by the total, and map the result. `moe_sum()` and `moe_prop()` carry the margins of error through the calculation (Section 14.7.1):

```
nc_under5 <- nc_tracts |>
  mutate(
    under5E   = male_u5E + female_u5E,
    under5M   = map2_dbl(male_u5M, female_u5M, \(a, b) moe_sum(c(a, b))),
    pct_u5    = under5E / totalE * 100,
    pct_u5_moe = moe_prop(under5E, totalE, under5M, totalM) * 100
  )

tm_shape(nc_under5) +
  tm_polygons(
    fill       = "pct_u5",
    fill.scale = tm_scale_intervals(style = "jenks"),
    fill.legend = tm_legend(title = "% under 5"),
    lwd        = 0.2
```

```
)
```

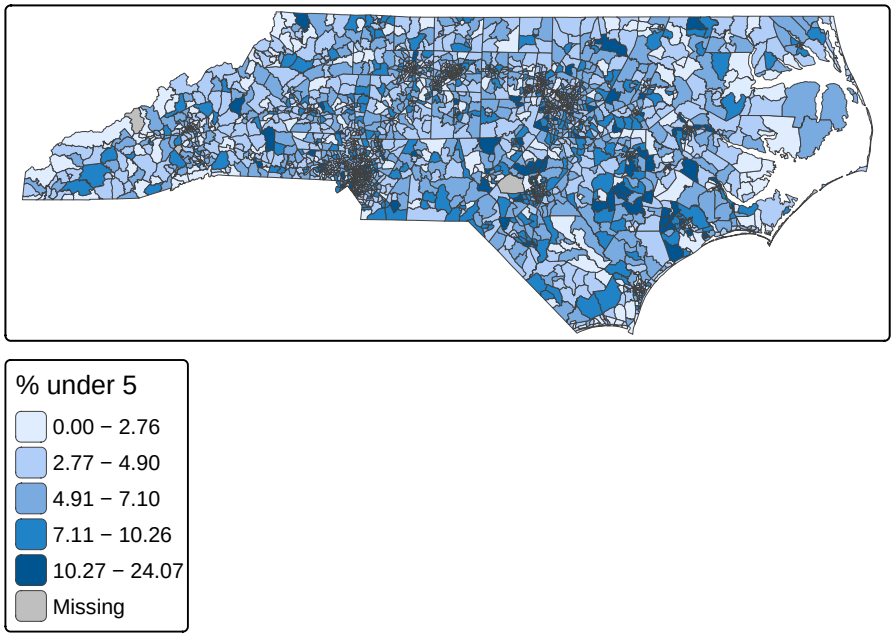


Figure 14.5: Share of the population under 5 years old, North Carolina census tracts, 2018–2022 ACS 5-year estimates. Classes use Jenks natural breaks.

A handful of tracts have no value. These are tracts where no one lives, such as national forests, military training areas, or open water, so the percentage would mean dividing by zero. Before you interpret the map, remember that each tract's value is an estimate. Small tracts can have margins of error nearly as large as the estimate itself, so a single dark tract may reflect sampling noise as much as a real pattern.

> **Try it**
>
> Change the code above to map the share of each county's population that is 65 or older. You will need several lines of `B01001` for each sex. Use `load_variables()` to find them.

14.7 Going further with tidycensus

The worked example covers the most common case. This section covers what you need to know once you are past your first download.

14.7.1 Margins of error

Every ACS value is an estimate from a sample, and tidycensus returns a margin of error (`moe`) alongside each `estimate`. ACS margins of error use a 90% confidence level: there is a 90% chance that the true value lies within `estimate ± moe`.

The smaller the area, the smaller the sample behind it, and the larger the margin of error relative to the estimate. A useful way to judge reliability is the **coefficient of variation (CV)**:



the standard error as a percentage of the estimate. Divide the MOE by 1.645 to get the standard error, then divide by the estimate:

$$CV = \frac{MOE/1.645}{\text{estimate}} \times 100$$

```
nc_income |>
  mutate(cv = (moe / 1.645) / estimate * 100) |>
  arrange(desc(cv)) |>
  head()
```

```
# A tibble: 6 x 6
  GEOID NAME                                variable   estimate   moe    cv
  <chr> <chr>                               <chr>         <dbl> <dbl> <dbl>
1 37095 Hyde County, North Carolina         B19013_001    43724 13650 19.0
2 37177 Tyrrell County, North Carolina      B19013_001    55341 11686 12.8
3 37043 Clay County, North Carolina         B19013_001    57065 12015 12.8
4 37073 Gates County, North Carolina        B19013_001    55750 10503 11.5
5 37187 Washington County, North Carolina   B19013_001    38927  7294 11.4
6 37005 Alleghany County, North Carolina    B19013_001    42115  6307  9.10
```

There is no single official cutoff, but a common rule of thumb treats estimates with a CV above about 30% as unreliable: report them with caution, or combine areas or years to get a larger sample (U.S. Census Bureau 2020).

When you combine estimates you must also combine their margins of error, and you can't just add them. tidycensus provides functions for the standard formulas: `moe_sum()` for sums, `moe_prop()` for proportions, `moe_ratio()` for ratios, and `moe_product()` for products.

14.7.2 Calculating percentages

Raw counts mostly tell you where people live. A map of the number of renters looks a lot like a map of population. To compare places, convert counts to shares.

The `summary_var` argument adds a second variable, usually the table total, to every row as `summary_est` and `summary_moe`. That puts the numerator and denominator side by side, so the share is a single division:

```
nc_white <- get_acs(
  geography   = "county",
  variables   = "B02001_002",
  summary_var = "B02001_001",
  state       = "NC",
  year        = 2022
) |>
  mutate(
    pct     = estimate / summary_est * 100,
    pct_moe = moe_prop(estimate, summary_est, moe, summary_moe) * 100
  )

nc_white |> select(NAME, estimate, summary_est, pct, pct_moe)
```

```
# A tibble: 100 x 5
   NAME                                estimate summary_est   pct pct_moe
   <chr>                                  <dbl>       <dbl> <dbl>   <dbl>
 1 Alamance County, North Carolina       112619      171779  65.6      NA
 2 Alexander County, North Carolina       32093       36505  87.9      NA
 3 Alleghany County, North Carolina       10270       10989  93.5      NA
 4 Anson County, North Carolina           10218       22200  46.0      NA
 5 Ashe County, North Carolina            25585       26759  95.6      NA
 6 Avery County, North Carolina           15773       17679  89.2      NA
 7 Beaufort County, North Carolina        31510       44711  70.5      NA
 8 Bertie County, North Carolina           6188       17818  34.7      NA
 9 Bladen County, North Carolina          16874       29807  56.6      NA
10 Brunswick County, North Carolina      116031      139721  83.0      NA
# i 90 more rows
```

14.7.3 Decennial Census data

`get_decennial()` works like `get_acs()` but pulls from the full-count decennial census. Because these are counts rather than sample estimates, there is no `moe` column. For 2020, choose the summary file with `sumfile`: `"pl"` for the redistricting counts (available down to blocks) or `"dhc"` for more detailed age, sex, household, and housing tables.

```
nc_pop <- get_decennial(
  geography = "tract",
  variables = "P1_001N",
  state     = "NC",
  year      = 2020,
  sumfile   = "dhc"
)

nc_pop
```

```
# A tibble: 2,672 x 4
   GEOID       NAME                                                  variable value
   <chr>       <chr>                                                 <chr>    <dbl>
 1 37071033303 Census Tract 333.03; Gaston County; North Carolina P1_001N   3602
 2 37071033202 Census Tract 332.02; Gaston County; North Carolina P1_001N   2351
 3 37071033203 Census Tract 332.03; Gaston County; North Carolina P1_001N   2521
 4 37071033204 Census Tract 332.04; Gaston County; North Carolina P1_001N   3666
 5 37071033306 Census Tract 333.06; Gaston County; North Carolina P1_001N   1554
 6 37071033308 Census Tract 333.08; Gaston County; North Carolina P1_001N   1311
 7 37071033309 Census Tract 333.09; Gaston County; North Carolina P1_001N   1933
 8 37071033310 Census Tract 333.10; Gaston County; North Carolina P1_001N   1940
 9 37071033311 Census Tract 333.11; Gaston County; North Carolina P1_001N    789
10 37071033312 Census Tract 333.12; Gaston County; North Carolina P1_001N   3800
# i 2,662 more rows
```

"Full count" does not mean "exact." To protect respondents' privacy, the 2020 census applied **differential privacy**: carefully calibrated statistical noise added to published tables. The noise is negligible for states and counties but can be substantial for blocks, especially for small population groups. Treat individual block counts with caution, and aggregate blocks where you can.



14.7.4 Comparing across years

Change over time is one of the most interesting things to measure with Census data, and one of the easiest to get wrong. Four traps to watch for:

1. **Boundaries change.** Tract and block group boundaries are redrawn every ten years. A GEOID from 2010 and the same GEOID in 2020 may not describe the same place. ACS releases from 2020 on use 2020 boundaries.
2. **5-year periods overlap.** The 2017–2021 and 2018–2022 estimates share four years of responses, so differences between them are muted. Compare non-overlapping periods, such as 2013–2017 and 2018–2022.
3. **Dollars are nominal.** Income, rent, and home values are reported in dollars of their own period. Adjust for inflation before comparing.
4. **Codes change.** Tables and lines are occasionally renumbered or redefined. Check `load_variables()` for every year you use.

`get_acs()` returns one year per call. To pull several, loop over the years with **purrr** and stack the results:

```
years <- c(2012, 2017, 2022)  # non-overlapping 5-year periods

nc_income_years <- map(years, \(y) {
  get_acs(
    geography = "county",
    variables = "B19013_001",
    state     = "NC",
    year      = y
  ) |>
    mutate(year = y)
}) |>
  list_rbind()

nc_income_years |> filter(NAME == "Orange County, North Carolina")
```

```
# A tibble: 3 x 6
  GEOID NAME                              variable   estimate   moe  year
  <chr> <chr>                             <chr>         <dbl> <dbl> <dbl>
1 37135 Orange County, North Carolina B19013_001    55241  2576  2012
2 37135 Orange County, North Carolina B19013_001    65522  2492  2017
3 37135 Orange County, North Carolina B19013_001    85785  2626  2022
```

The 2012, 2017, and 2022 releases cover 2008–2012, 2013–2017, and 2018–2022, so they do not overlap. The income values are still in nominal dollars.

14.7.5 More tidycensus functions

tidycensus can do much more than `get_acs()` and `get_decennial()`. Table 14.10 lists some of the most useful extras.

Table 14.10: Other tidycensus functions and arguments.

Function or argument	What it returns	Example use
`get_estimates()`	Annual population estimates and components of change	County population growth since 2020
`get_pums()`	Individual and household ACS microdata	Custom tabulations not in published tables
`get_flows()`	County-to-county migration flows	Where people moving into a county came from
`table = "B19001"`	Every variable in one ACS table	Full household income distribution in one call
`survey = "acs1"`	1-year ACS estimates	Recent data for large counties and metros
`tigris::erase_water()`	Geometry with water areas removed	Cleaner maps along coasts and lakes

14.8 Summary

- US Census data come from two main sources: the **decennial census**, a full count every ten years with few questions, and the **American Community Survey**, a continuous sample with many questions and a margin of error on every estimate.
- Census geography nests from the nation down to **tracts**, **block groups**, and **blocks**. Every area has a **GEOID** built from FIPS codes. Keep GEOIDs as text and join on them.
- Smaller areas mean more detail but fewer published tables, more privacy noise, and larger margins of error. ACS 5-year estimates are the only ACS product for tracts and block groups.
- An **API** lets your code request data from a server. A REST request is an endpoint URL plus parameters, and the response is usually JSON.
- **tidycensus** wraps the Census API: `get_acs()` and `get_decennial()` return tidy data frames, and `geometry = TRUE` makes them map-ready.
- Convert counts to shares with `summary_var`, carry margins of error through calculations with the `moe_*()` functions, and compare across years only with non-overlapping periods, consistent boundaries, and inflation-adjusted dollars.

14.9 Exercises

1. **Reading GEOIDs.** The GEOID `370630020271003` identifies a Census block. Which state, county, tract, and block group does it belong to? How many digits does each part have?
2. **Choosing a source.** For each question, name the survey (decennial, ACS 1-year, or ACS 5-year) and the smallest geography you could use. (a) How many housing units are in each block in Durham? (b) What was median household income in Wake County last year? (c) What share of residents in each Chapel Hill neighborhood rent their home?
3. **Build a URL.** Write by hand the Census API URL that requests total population (`B01003_001E`) for every county in Virginia (state FIPS `51`) from the 2022 ACS 5-year data. Paste it into a browser to check it. Then reproduce the same request with httr2.
4. **Reliability.** Download median household income for every block group in Orange County, NC. What share of block groups have a CV above 30%? Repeat for tracts and compare.



5. **A rate map.** Map the share of households with no vehicle available for tracts in your home county, or in Mecklenburg County, NC. Use `B25044` and include the margin of error of the share.
6. **Change over time.** Compare median gross rent (`B25064_001`) for North Carolina counties in the 2013–2017 and 2018–2022 ACS. Which counties saw the largest increase? Explain why you should adjust for inflation, and what else could make your comparison misleading.
7. **Your own question.** Return to the question you wrote down in Section 14.3. Download the data you need, make a map, and write one paragraph interpreting it, including what the margins of error tell you about your confidence.

14.10 Further reading

- Kyle Walker, the author of tidycensus, has written a free online book, *Analyzing US Census Data: Methods, Maps, and Models in R* (Walker 2023). It covers everything in this chapter in more depth and is the best next step for any project that uses Census data.
- The tidycensus documentation, at https://walker-data.com/tidycensus, has vignettes on spatial data, margins of error, and microdata.
- The Census Bureau's ACS handbooks explain how the survey works and how to handle margins of error (U.S. Census Bureau 2020).
- To browse tables without code, use data.census.gov, then bring the variable codes you find back to `load_variables()` and tidycensus.

15 ArcGIS Data with arcpullr

Learning objectives

By the end of this chapter you should be able to:

1. Explain how agencies publish GIS layers on ArcGIS portals such as NC OneMap.
2. Tell a feature service from other kinds of items, and find a layer's REST endpoint URL.
3. Read the useful details on a layer's REST Services Directory page.
4. Download a layer into R as an sf object with `get_spatial_layer()` from the **arcpullr** package.
5. Filter a layer on the server with a `where` clause, and explain why that is better than filtering after you download.

Chapter 13 showed how an **API** lets your code ask another program's server for data, using the Census Bureau's API and the tidycensus package. This chapter uses the same idea for a different, and very common, source of GIS data: layers published with Esri's ArcGIS software. By the end of it you'll load a layer from North Carolina's statewide GIS portal, NC OneMap, straight into R and map it.

15.1 Recap: APIs

An API is a set of rules that lets your code request data from another program's server. The pattern is always the same: **request, then response**. Your code sends an **endpoint** URL, often with **parameters** that say what you want, and the server sends data back, usually as JSON.

Why bother, when you could click a Download button? Two reasons from Chapter 13 still apply:

- **Reproducibility.** Your script is the record of exactly where the data came from and how you got them. Anyone can rerun it.
- **Efficiency.** You download only the data you need, not a whole statewide file.

tidycensus is a wrapper package for the Census API. Today's wrapper package is **arcpullr**, for ArcGIS servers.

15.2 ArcGIS portals and feature services

Thousands of government agencies, universities, and companies publish their GIS data online through **ArcGIS Online** or ArcGIS Enterprise, Esri's systems for sharing maps and data on the web. Many run a public **portal**: a website where you can search, preview, and download their layers. North Carolina's is **NC OneMap**, https://www.nconemap.gov/, which gathers layers

from state agencies, counties, and cities: roads, parcels, schools, hospitals, flood zones, and much more.

Behind the website, each published layer lives on an ArcGIS server. A layer published as a **feature service** (or as one layer of a map service) has its own **REST endpoint**: a URL that returns the layer's features and attributes when your code asks for them. arcpullr is an R package that sends a request to that endpoint and returns the layer as an **sf** object, ready for the dplyr, sf, and tmap tools you already know.

When you search a portal, look at the type of each result. "Feature Service" or "Feature Layer" marks a layer arcpullr can read (Figure 15.1). Other items, such as documents, web maps, and apps, are not.

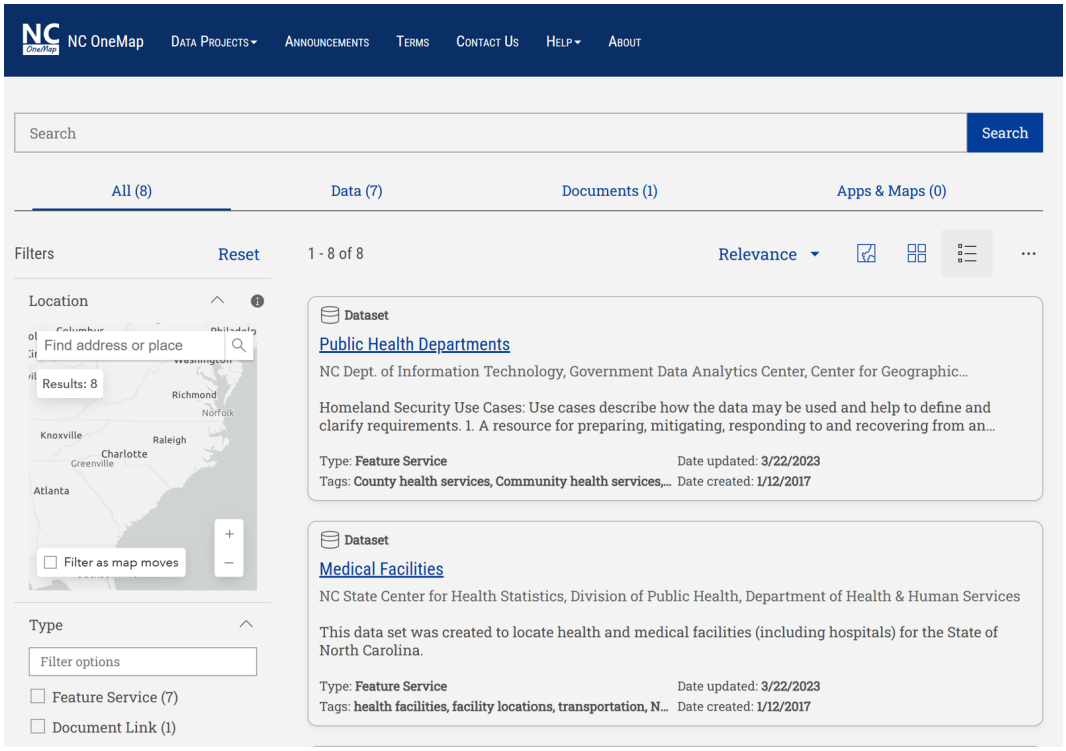


Figure 15.1: Searching NC OneMap for health data. Each result's Type line says what kind of item it is; "Feature Service" marks a layer arcpullr can read.

15.3 Finding a layer's REST URL

arcpullr needs one thing from you: the layer's REST URL. Here is how to find it on NC OneMap, using the Medical Facilities layer. The same steps work on most ArcGIS Online and ArcGIS Enterprise portals.

1. **Search** NC OneMap for a dataset, such as "medical facilities".
2. **Open its item page.** Check that the details panel on the left says it is a feature layer.
3. **Click "I want to use this"** at the bottom of the details panel (Figure 15.2).
4. **Choose "View Data Source"** from the list that opens (Figure 15.3). It opens the layer's REST service page in a new tab.
5. **Copy that page's URL** from the address bar. That URL is what arcpullr needs.

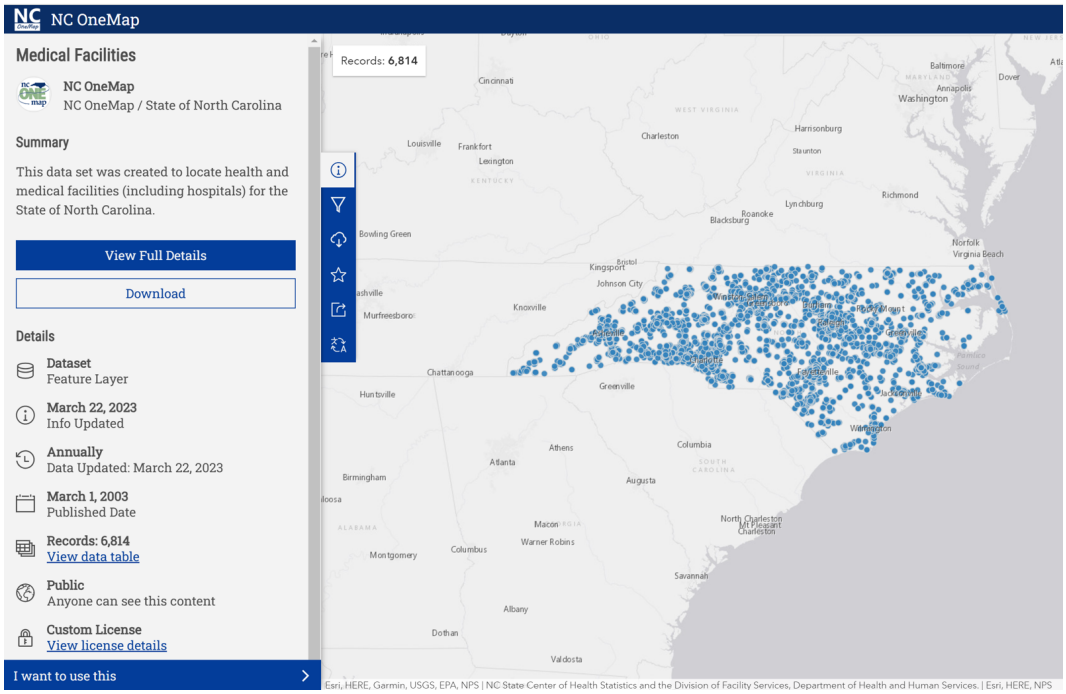


Figure 15.2: The Medical Facilities item page on NC OneMap. The details panel on the left describes the layer; "I want to use this" is at its bottom.

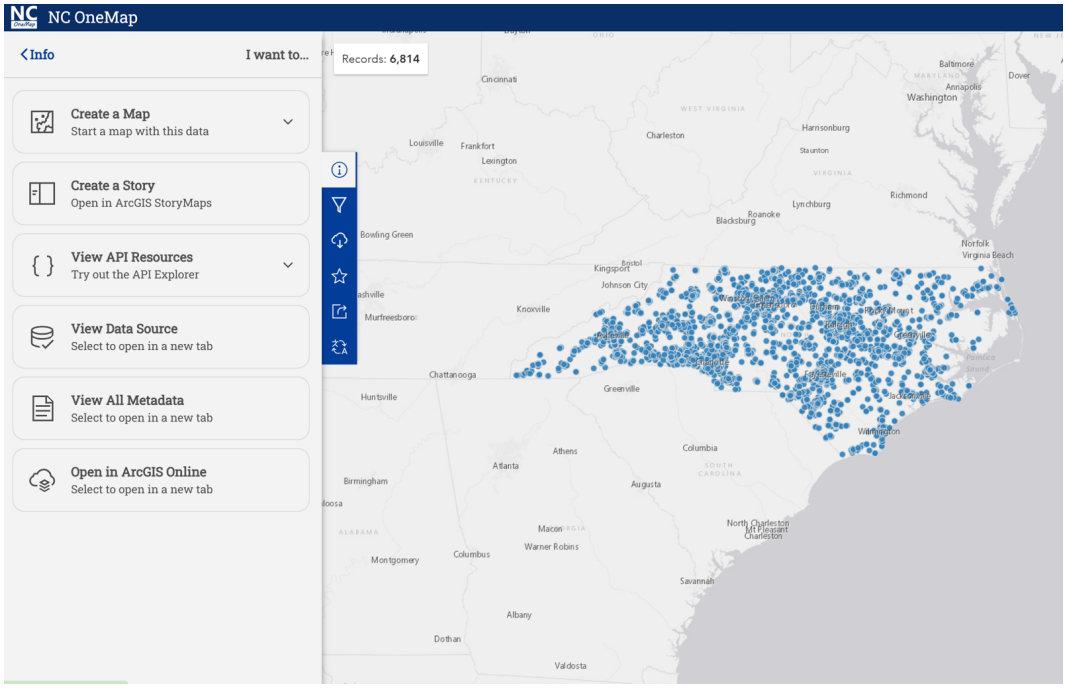


Figure 15.3: The "I want to..." menu. "View Data Source" opens the layer's REST service page.

15.3.1 Reading the REST Services Directory page

The page that opens is the layer's page in the **ArcGIS REST Services Directory** (Figure 15.4). It is plain, but it holds everything you need to know before you write any code:

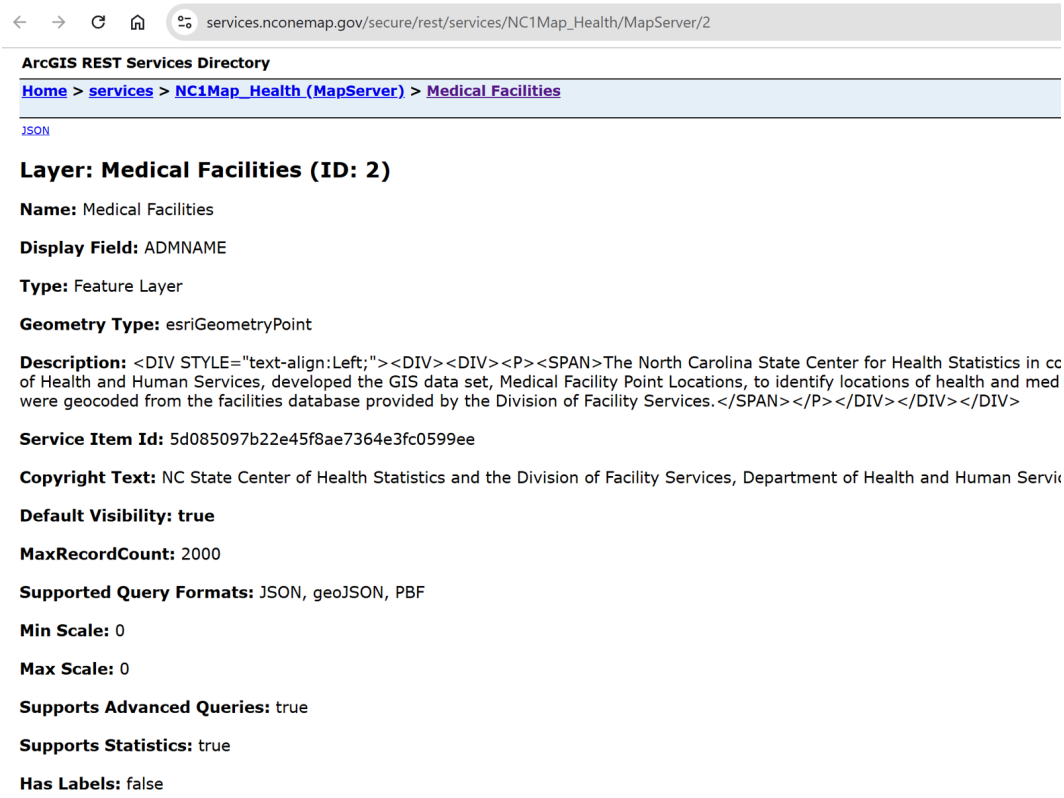


Figure 15.4: The ArcGIS REST Services Directory page for the Medical Facilities layer. Its URL is the layer's REST endpoint.

- **The URL.** `https://services.nconemap.gov/secure/rest/services/NC1Map_Health/MapServer/2` The number at the end, 2, is the **layer ID**: this service holds several layers, and Medical Facilities is number 2. If you copy a URL without the number, you have the whole service, not one layer.
- **Geometry Type.** `esriGeometryPoint`, so each feature is a point. Others are polylines and polygons.
- **MaxRecordCount.** 2,000: the most features the server returns for one request.
- **Fields.** Further down the page, a list of the attribute columns and their types. You need these names to filter the layer.

The item page reports 6,814 records, more than the 2,000 the server returns at once. You don't need to worry about that: arcpullr asks for the features in batches and puts them back together. Still, check that the number of rows you get matches what the portal reports.

15.4 Anatomy of an arcpullr request

Install arcpullr once, in the Console, with `install.packages("arcpullr")`. Then:

```
library(arcpullr)
library(sf)
```



```
library(tmap)

url <- "https://services.nconemap.gov/secure/rest/services/NC1Map_Health/MapServer/2"
medical.centers <- get_spatial_layer(url, where = "fcounty = 'Orange'")
```

Line by line:

- `url` stores the REST endpoint you copied from the service page.
- `get_spatial_layer()` sends the request and returns the layer as an sf object.
- `where =` is a filter, written in **SQL**, that runs on the server. Here `fcounty = 'Orange'` returns only the facilities in Orange County. Without a `where` argument, you get every feature in the layer.

15.4.1 Writing a where clause

The `where` argument is a small piece of SQL, the language of databases, that the server applies before it sends anything back. It uses the field names from the REST page, not the ones you might rename later in R. A few rules cover most needs:

Table 15.1: Common `where` clauses. The field names are examples; use the fields listed on your layer's REST page.

You want	`where =`
Text equal to a value	`"fcounty = 'Orange'"`
One of several values	`"fcounty IN ('Orange', 'Durham', 'Wake')"`
A number above a value	`"beds > 100"`
Two conditions	`"fcounty = 'Orange' AND beds > 100"`
Text that starts with something	`"fcounty LIKE 'Ch%'"`

Note the quotes: the whole clause is an R string in double quotes, and text values inside it go in single quotes. SQL uses =, not ==, and AND and OR, not & and |.

Why filter on the server instead of downloading everything and using `filter()` in R? For a statewide layer of a few thousand points, either works. But many layers are far bigger: every parcel or building in the state. Filtering on the server means less data crosses the internet, the request is faster, and you never hit the server's limits. It also makes your script say exactly which part of the data you used.

Think about it

The REST page says the Medical Facilities layer was "geocoded from the facilities database". What could go wrong with locations made that way (Chapter 13)? How might you check a few of them?

15.5 Two requests, one map

Here are two requests to the same layer: one for every medical facility, and one for Orange County only. tmap draws all of them in black, then the Orange County facilities in red on top (Figure 15.5):

```
url <- "https://services.nconemap.gov/secure/rest/services/NC1Map_Health/MapServer/2"
all.medical.centers    <- get_spatial_layer(url)
orange.medical.centers <- get_spatial_layer(url, where = "fcounty = 'Orange'")

tm_shape(all.medical.centers) +
  tm_dots() +
  tm_shape(orange.medical.centers) +
  tm_dots(fill = "red")
```

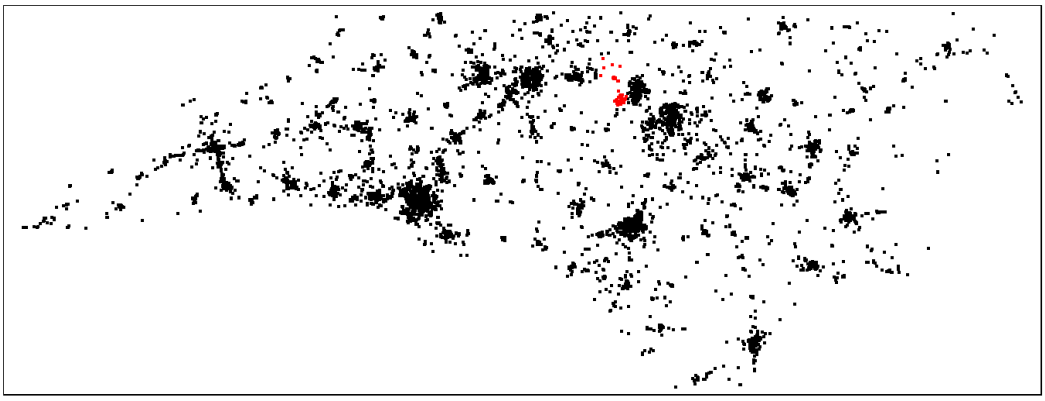


Figure 15.5: Medical facilities from NC OneMap: every facility in black, and those in Orange County, from a second, filtered request, in red.

The dots alone trace the shape of the state, with dense clusters around Charlotte, the Triad (Greensboro, Winston-Salem, and High Point), and the Triangle (Raleigh, Durham, and Chapel Hill), where most people live.

15.5.1 Before you analyze

The result is an ordinary sf object, so everything from earlier chapters applies:

- **Check it.** `nrow()`, `names()`, and `glimpse()` show how many features came back and what the columns are called.
- **Check the coordinate system.** Run `st_crs()` on the result. Before measuring distances or areas, project it to North Carolina State Plane with `st_transform(32119)` (Chapter 12).
- **Note the date.** Portal layers are **dynamic**: agencies update them, sometimes daily. Rerunning your script next month may give different data. Record the date you downloaded it, and save a copy with `st_write()` if your results must be exactly reproducible.

15.5.2 More arcpullr functions

`get_spatial_layer()` is the one you'll use most. arcpullr has others for other jobs:

- `get_table_layer()` downloads a table that has no geometry.
- `get_layer_by_poly()` downloads only the features that fall inside a polygon you give it, such as one county's boundary. Like `where`, it filters on the server, but by location instead of by attribute.
- `get_image_layer()` and `get_map_layer()` download raster images and map services.

15.6 Try it

1. Go to https://www.nconemap.gov/ and browse the data by theme, or search for a topic you care about.
2. Find a dataset that is a feature service.
3. Track down its REST URL: “I want to use this”, then “View Data Source”.
4. Paste the URL into `get_spatial_layer()` and map the result with tmap.
5. Bonus: add a `where` filter for your own county. Which field holds the county name in your layer?

15.7 Check your understanding

1. arcpullr returns a layer as: [1]

- (A) A CSV file
- (B) An sf object
- (C) A web map
- (D) A raster

2. On NC OneMap, which menu item opens a layer’s REST service page? [2]

- (A) Create a Map
- (B) View API Resources
- (C) View Data Source
- (D) Open in ArcGIS Online

3. In `.../NC1Map_Health/MapServer/2`, what is the 2? [3]

- (A) The version of the service
- (B) The layer ID
- (C) The number of fields
- (D) The page of results

4. What is the name of the arcpullr function that downloads a feature layer? _____[4]

5. A `where` clause is applied by the server, before any data are sent to R. TRUE / FALSE[5]

6. Which `where` clause returns only facilities in Durham County? [6]

---

[1] An sf object
[2] View Data Source
[3] The layer ID
[4] get_spatial_layer, get_spatial_layer()
[5] TRUE
[6] “fcounty = ‘Durham’ ”



- (A) “fcounty == Durham”
- (B) “fcounty = ‘Durham’ ”
- (C) fcounty = Durham
- (D) “fcounty & ‘Durham’ ”

7. MaxRecordCount tells you: [7]

- (A) How many layers the service has
- (B) The most features the server returns for one request
- (C) How many fields the layer has
- (D) How many people have downloaded the layer

8. Data downloaded from a portal today will always be identical to data downloaded next year. TRUE / FALSE[8]

15.8 Summary

- Agencies publish GIS layers on **ArcGIS Online** and ArcGIS Enterprise **portals**. **NC OneMap** is North Carolina’s.
- Each published layer has a **REST endpoint** URL. On NC OneMap, find it with “I want to use this” and then “View Data Source”.
- The **REST Services Directory** page shows the **layer ID** (the number at the end of the URL), the geometry type, the **MaxRecordCount**, and the field names.
- **arcpullr**’s `get_spatial_layer(url)` downloads a layer as an sf object.
- A `where` clause, written in **SQL**, filters the layer on the server: `where = "fcounty = 'Orange'"`. Filtering on the server is faster and sends less data.
- Portal layers change over time. Check what you got, project it before measuring, and record the download date.

15.9 Exercises

1. **Your county.** Download the NC OneMap Medical Facilities layer for your home county (or Orange County) with a `where` clause. How many facilities are there? Compare your count with the full layer filtered in R with `filter()`.
2. **Two counties.** Use `IN` to download the facilities for Orange and Durham counties in one request. Map them, colored by county.
3. **A new layer.** Find a different feature service on NC OneMap, such as schools, parks, or fire stations. Record its REST URL, geometry type, MaxRecordCount, and three of its field names, then download and map it.
4. **Polygons.** Find a polygon layer on NC OneMap. Download it, project it with `st_transform(32119)`, and calculate the area of each feature with `st_area()`.

[7] The most features the server returns for one request
[8] FALSE

5. **Join it up.** Count the medical facilities in each county, then join the counts to a county boundary layer (Chapter 15) and make a choropleth map of facilities per 10,000 people, using county populations from tidycensus (Chapter 13).

15.10 Further reading

- The arcpullr page on CRAN, https://cran.r-project.org/package=arcpullr, links to the package manual and vignettes, which list every function with examples.
- NC OneMap, https://www.nconemap.gov/, North Carolina's portal for state and local GIS data.
- Esri's *ArcGIS REST APIs* documentation, https://developers.arcgis.com/rest/, explains what a feature service is and every parameter a query can use, including `where`.



Part IV

Unit 4: Spatial Analysis



This unit puts data together and starts to analyze it. It joins tables by a shared column and by location, then explores data the way an analyst first meets a new dataset: describing each variable's center and spread, plotting its distribution, and mapping it. It ends with statistics that describe where features are, and puts your own R functions to work calculating them.

Chapter 15: Joins: Attribute and Spatial

A join combines tables by matching rows. This chapter covers dplyr's inner, left, right, and full joins, naming the key with `join_by()`, and chaining joins across several tables. It explains cardinality, and why you should know it before you join. It then joins by location with sf's `st_join()`, compares simple and summarized, inside and distance spatial joins, and shows why distances need a projected coordinate system.

Chapter 16: Exploratory Spatial Data Analysis

Before modeling data, get to know it. This chapter places exploratory data analysis in the data science workflow and adds the spatial question: how do values vary across space? It covers checking a data table, measures of central tendency (mean, median, mode) and dispersion (range, IQR, standard deviation), histograms, boxplots, and density plots, distribution shapes and outliers, and mapping rates rather than counts.

Chapter 17: Spatial Descriptive Statistics

Spatial descriptive statistics describe where features are. This chapter covers the mean center, the weighted mean center (including the U.S. center of population), the central feature, and the standard distance, each matched to the familiar statistic it extends. It then reviews writing functions in R, from a one-line `addSeven()` to a `meanCenter()` function for any layer of polygons.



16 Joins: Attribute and Spatial

Learning objectives

By the end of this chapter you should be able to:

1. Explain what a join does, and combine two tables with dplyr's `inner_join()`, `left_join()`, `right_join()`, and `full_join()`.
2. Name the key of a join with `join_by()`, and chain joins to combine three or more tables.
3. Describe the cardinality of a relationship, and predict how many rows a join will return.
4. Explain how a spatial join uses location instead of a shared column, and run one with `st_join()`.
5. Choose between simple and summarized, inside and distance spatial joins.
6. Avoid wrong distances in a spatial join by checking and projecting the coordinate reference system.

Data rarely arrive in one table. Student names live in one table and the courses they take in another. Census counts live in a spreadsheet and county boundaries in a shapefile. EMS calls are points and neighborhoods are polygons. To answer most questions, you have to put two tables together.

A **join** combines two or more tables by matching rows. This chapter covers two kinds:

- An **attribute join** matches rows that share a value in a common column, such as a student ID or a county FIPS code.
- A **spatial join** matches features by their location, such as which neighborhood each EMS call falls inside.

16.1 What is a join?

Recall that a data table has one row per observation and one column per variable. A join lines up the rows of two tables wherever a shared column has the same value. That shared column is the **key**.

Here are two small tables. `students` has one row per student. `takes` has one row for each course a student took. Both have an `id` column. Run the box to make them:

```
library(dplyr)

students <- tibble(
  id   = c("00128", "12345", "11122", "22211"),
  name = c("Zhang", "Shankar", "Parr", "McKeon"),
  dept = c("Comp. Sci.", "History", "Geography", "Geography")
)
```



```
takes <- tibble(
  id        = c("00128", "00128", "12345", "33456", "45678"),
  course_id = c("CS-347", "CS-101", "HIS-222", "GEOG-215", "GEOG-215"),
  sec_id    = c(1, 2, 1, 1, 2)
)

students
takes
```

Look closely before you join. Zhang took two courses. Parr and McKeon took none. And two rows of `takes` belong to IDs, 33456 and 45678, that aren't in `students` at all. Each type of join treats these unmatched rows differently.

16.2 Types of joins

It helps to think of each table as a collection of rows, drawn as a circle. Where the circles overlap are the rows whose key appears in both tables. A table with no duplicate rows is a **set**; one that may have duplicates is a **bag**, and R data frames behave like bags.

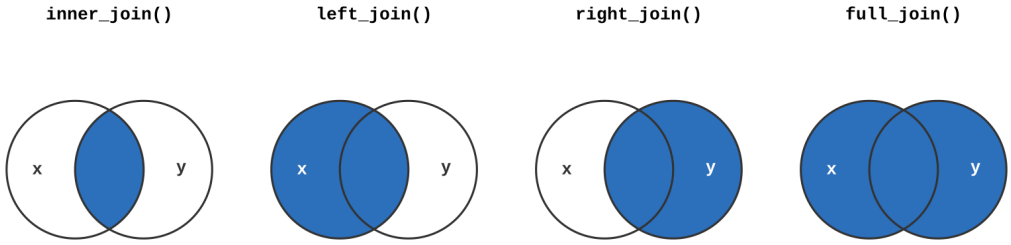


Figure 16.1: The four main joins. The blue part of each diagram is the rows the join keeps.

16.2.1 Inner join

An **inner join** keeps only the rows whose key appears in both tables:

```
students |> inner_join(takes, by = join_by(id))
```

Parr and McKeon are dropped because they took no courses, and the two GEOG-215 rows are dropped because no student matches them. Zhang appears twice, because Zhang took two courses.

The `by = join_by(id)` part names the key. You could leave it out and dplyr would guess, but naming it is the habit to build.

16.2.2 Left join

A **left join** keeps every row of the left table, the first one, and adds the matching columns from the right table:

```
students |> left_join(takes, by = join_by(id))
```



Parr and McKeon now appear, with NA in the course columns. This is the join you will use most, because it never drops a row from the table you care about. But check the row count afterwards: it can grow when a key has several matches. `students` has 4 rows, and the result has 5.

16.2.3 Right join

A **right join** is the mirror image. It keeps every row of the right table:

```
students |> right_join(takes, by = join_by(id))
```

Now the GEOG-215 rows appear, with NA for name and department. Any right join can be written as a left join by swapping the order of the tables, and most people find the left join easier to read.

16.2.4 Full join

A **full join** keeps every row from both tables. It matches rows where it can and fills in NA where it can't, so nothing is dropped:

```
students |> full_join(takes, by = join_by(id))
```

16.2.5 Natural joins and cross joins

Two more joins are worth knowing about, mostly so you can avoid them by accident.

A **natural join** is an inner join on every column the two tables share by name. In dplyr, if you leave out `by`, `inner_join()` does exactly this and prints a message, such as `Joining with `by = join_by(id)``, to tell you which columns it used. It's convenient but fragile. If both tables also had a column called `name` or `year`, it would silently join on that too. In your own code, spell out the key.

A **cross join** pairs every row of one table with every row of the other: the cartesian product. In dplyr it is `cross_join()`. Our 4 students and 5 course rows give 20 rows. It grows fast: two tables of 1,000 rows give a million.

You may also meet a **self join**, which joins a table to itself, such as a table of employees joined to their managers in the same table.

> **Think about it**
>
> Suppose the keys were stored differently in the two tables: `"00128"` as text in one and `128` as a number in the other. Would they match? How would you find out, and how would you fix it?

16.2.6 Joining more than two tables

Joins chain. Pipe the result of one join into the next to combine three or more tables. Here we add an `instructor` table, keyed by `course_id`:



```
instructor <- tibble(
  course_id  = c("CS-347", "CS-101", "HIS-222", "GEOG-215"),
  instructor = c("Okafor", "Lindqvist", "Moreno", "Parr")
)

roster <- students |>
  inner_join(takes, by = join_by(id)) |>
  inner_join(instructor, by = join_by(course_id))

roster
```

Notice that the key changes from step to step: `id` for the first join and `course_id` for the second. Each inner join can only drop rows, so if records go missing, check the row count after each step to see where.

16.3 Cardinality

Before you run a join, ask how many rows on each side can match. **Cardinality** is the relationship between the rows of two tables, and it tells you how big the result will be.

Table 16.1: The four kinds of cardinality.

Cardinality	Meaning	Examples
One-to-one	Each row matches at most one row in the other table	Governor and state; person and passport
One-to-many	One row matches many rows in the other table	Book and pages; state and cities
Many-to-one	The same relationship, seen from the other side	Cities and state; students and advisor
Many-to-many	Rows on both sides can match many rows	Students and classes

`students` to `takes` is one-to-many: one student can take many courses. That is why Zhang's row was copied in the result. A many-to-many join multiplies rows on both sides, and can make a result far bigger than either table. dplyr warns you when it finds one.

So here is the **rule of joining**: know the cardinality before you join. It tells you how many rows to expect, and if the result has a different number, something about your keys isn't what you thought. Cardinality will matter again when we design databases later in the course.

16.4 Why join by location?

So far the key has been a shared column. But often two tables share no ID at all, only location. Consider three questions:

- **How many EMS calls fall in each neighborhood?** The calls are points and the neighborhoods are polygons.
- **How many students attend school in each district?** The schools are points with an enrollment, and the districts are polygons.



- **What is a coffee chain's revenue in each sales region?** The stores are points with revenue, and the regions are polygons.

All three have the same shape. Join each point to the polygon it falls inside, then summarize by polygon. The key is location.

16.5 Spatial joins

16.5.1 From attribute joins to spatial joins

In GIS terms, an attribute join has a **destination table** that receives new columns, a **source table** that supplies them, and a common field that links the two. The output is the destination table with the source's columns added.

A **spatial join** has the same structure, but the link is a **spatial relationship** instead of a common field. Two relationships cover most needs:

- **Inside**: one feature falls inside, or intersects, another. Each school gets the attributes of the county it is in.
- **Distance**: one feature is closest to another. Each city gets the attributes of the nearest airport.

16.5.2 Spatial joins in R

In R, the sf package does spatial joins with `st_join()`:

```
st_join(x, y, join = st_intersects)
```

`x` is the destination and `y` the source. The `join` argument sets the relationship:

- `st_intersects` (the default) or `st_within` for one feature inside another;
- `st_nearest_feature` for the closest feature.

By default `st_join()` keeps every row of `x`, like a left join. Set `left = FALSE` to get an inner join, which drops the features of `x` that matched nothing.

Here is a real example, using two datasets from the spData package: the 742 bicycle hire stations in London, as points, and London's 33 boroughs, as polygons. Which borough is each station in?

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

# Project both layers to the British National Grid, in meters
stations <- cycle_hire |> st_transform(27700)
boroughs <- lnd |> st_transform(27700)

stations_joined <- st_join(stations, boroughs["NAME"])
nrow(stations_joined)    # 742: one row per station
```

Each station now carries a `NAME` column: the borough it falls inside. One station sits just outside every borough polygon, so its `NAME` is `NA`; `left = FALSE` would drop it.



16.5.3 Four kinds of spatial join

Two decisions define a spatial join: the relationship (inside, or distance), and whether you attach one match per feature or **summarize** all of them (Table 16.2).

Table 16.2: Four kinds of spatial join. The arrow points from the source layer to the destination layer that receives its attributes.

	Simple: attach one matching record	**Summarized**: aggregate all matches
Inside	Which county is each school in? (schools → counties)	How many schools are in each county? (counties → schools)
Distance	Which attraction is closest to each hotel, and how far is it? (hotels → attractions)	How many attractions are closest to each hotel? (hotels → attractions)

A **simple join** attaches the attributes of one matching feature. Each septic system gets the geology formation underneath it; each city gets its nearest airport and the distance to it. In both cases, every destination feature has exactly one match, and one source feature can match many destination features, so the cardinality is many-to-one. There is no ambiguity about which fields to attach.

A **summarized join** handles one-to-many and many-to-many relationships. When one county contains many schools, a simple join can't pick just one. Instead, a summarized join counts or aggregates the matching features, such as the number of schools in each county, or their total enrollment, and attaches that.

The rule of joining applies to spatial joins too. Simple joins are one-to-one or many-to-one. Summarized joins handle one-to-many and many-to-many by aggregating.

16.5.4 Summarized joins in R

In R, a summarized join is `st_join()` followed by `group_by()` and `summarize()`, or `count()`. To count the stations in each borough:

```
station_counts <- stations_joined |>
  st_drop_geometry() |>
  count(NAME, name = "n_stations")

boroughs <- boroughs |>
  left_join(station_counts, by = join_by(NAME))
```

The last step is an attribute join. Dropping the geometry first turns the joined stations into a plain table, and `left_join()` keeps all 33 boroughs, with NA for the 22 that have no hire stations. Westminster has the most, with 171 (Figure 16.2).

For a distance join, use `st_nearest_feature`, then measure how far away the match is with `st_distance()`:

```
cities_airports <- st_join(cities, airports, join = st_nearest_feature)

nearest <- st_nearest_feature(cities, airports)
```

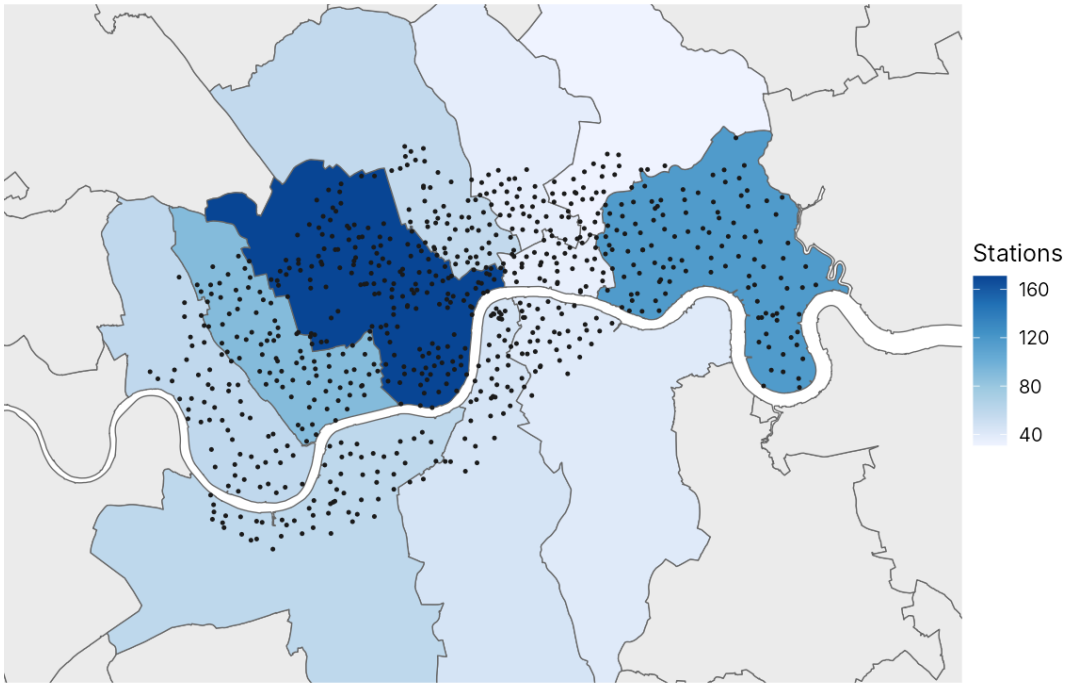


Figure 16.2: London's bicycle hire stations (black points) and the number in each borough, from a summarized spatial join. Gray boroughs have none.

```
cities_airports$dist <- st_distance(cities, airports[nearest, ],
                                    by_element = TRUE)
```

16.6 Spatial joins and the coordinate system

What units is a distance in? Software reports distances in the units of the data's coordinate system. If the data are in a geographic coordinate system, those units are decimal degrees, and a distance of 0.47 means 0.47 degrees. As Section 13.7 showed, degrees can't simply be converted to miles or kilometers, because a degree of longitude shrinks toward the poles.

Worse, the *match itself* can change. Degrees stretch unevenly, so the feature that is nearest in degrees isn't always the one that is nearest on the ground. A distance join run on latitude and longitude can attach a different feature than the same join run in a projected coordinate system.

So, before any spatial join:

1. Check each layer's coordinate system with `st_crs()`.
2. Make sure both layers use the same one. `st_join()` stops with an error if they don't.
3. For distance joins, project both layers with `st_transform()` to a projection that preserves distance in your study area, such as State Plane for North Carolina (EPSG:32119).

16.7 Check your understanding

1. Which join keeps only the rows whose key appears in both tables? [1]

[1] inner_join()

- (A) inner_join()
- (B) left_join()
- (C) full_join()
- (D) cross_join()

2. You left-join `students` (4 rows) to `takes`. One student took two courses and the others took none or one. Can the result have more than 4 rows? [2]

- (A) Yes, keys with several matches are repeated
- (B) No, a left join never changes the row count
- (C) No, it always has fewer rows

3. Any right join can be rewritten as a left join by swapping the order of the tables. TRUE / FALSE[3]
4. A cross join of a 10-row table and a 30-row table has how many rows? ___[4]
5. What is the cardinality between students and their academic advisor, if each student has one advisor? [5]

- (A) One-to-one
- (B) One-to-many
- (C) Many-to-one
- (D) Many-to-many

6. In `st_join(x, y)`, which layer receives the new attributes? [6]

- (A) x, the destination
- (B) y, the source
- (C) Neither; it makes a new layer with only y's columns

7. Which `join` argument attaches the closest feature? [7]

- (A) st_intersects
- (B) st_within
- (C) st_nearest_feature
- (D) st_distance

[2]Yes, keys with several matches are repeated
[3]TRUE
[4]300
[5]Many-to-one
[6]x, the destination
[7]st_nearest_feature

8. Counting the schools in each county is an example of a: [8]

- (A) Simple inside join
- (B) Summarized inside join
- (C) Simple distance join
- (D) Natural join

9. A distance join on data in latitude and longitude gives the same matches as the same join on projected data. TRUE / FALSE[9]

16.8 Summary

- A **join** combines tables by matching rows. An **attribute join** matches a shared **key** column; a **spatial join** matches by location.
- dplyr's **inner join** keeps matches only; a **left join** keeps every row of the first table; a **right join** keeps every row of the second; a **full join** keeps everything. Name the key with `join_by()`, and chain joins with the pipe.
- A **natural join** matches on every shared column name, which is fragile. A **cross join** pairs every row with every row.
- **Cardinality** (one-to-one, one-to-many, many-to-one, many-to-many) tells you how many rows a join will return. Know it before you join.
- `st_join(x, y, join = ...)` attaches the attributes of `y` to `x` using a spatial relationship: `st_intersects` or `st_within` for inside, `st_nearest_feature` for distance. It keeps every row of `x` unless `left = FALSE`.
- **Simple** spatial joins attach one match; **summarized** joins aggregate many, with `group_by()` and `summarize()` or `count()`.
- Distances come out in the data's units. Check the CRS with `st_crs()` and project with `st_transform()` before joining by distance.

16.9 Exercises

1. **Predict, then run.** Before running each join in Section 16.2, write down how many rows you expect and which names will have NA. Then check.
2. **Find the unmatched rows.** dplyr has two filtering joins: `semi_join()` keeps the rows of `x` that have a match in `y`, and `anti_join()` keeps the rows that don't. Use `anti_join()` to list the students who took no courses, and the course rows with no matching student.
3. **Cardinality.** Give the cardinality of each relationship, and explain: (a) capital city and state; (b) students and phone numbers; (c) counties and ZIP codes; (d) census tracts and counties.
4. **Base R.** Base R's `merge()` also joins tables. Find the arguments to `merge()` that give an inner, a left, and a full join of `students` and `takes`.
5. **London bikes.** Reproduce the station counts in Figure 16.2. Then make a map of stations per square kilometer of each borough. Does the ranking change?

[8] Summarized inside join
[9] FALSE



6. **NC schools.** Find a point layer of North Carolina public schools on NC OneMap, https://www.nconemap.gov. Join it to NC counties and count the schools in each county. Which county has the most schools? The most per 10,000 people?

16.10 Further reading

- *R for Data Science* (2nd ed.), by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund, free at https://r4ds.hadley.nz. The "Joins" chapter covers keys, every dplyr join, and how to check them.
- *Geocomputation with R* (2nd ed.), by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, free at https://r.geocompx.org. The "Spatial data operations" chapter covers spatial joins, distance joins, and spatial aggregation.
- *Database System Concepts*, by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, explains joins and cardinality from the database side.



17 Exploratory Spatial Data Analysis

Learning objectives

By the end of this chapter you should be able to:

1. Explain where exploratory data analysis (EDA) fits in the data science workflow, and what makes it spatial (ESDA).
2. Check the basics of a new data table: its size, its missing values, and the type of each field.
3. Calculate and interpret measures of central tendency (mean, median, mode) and dispersion (range, IQR, standard deviation).
4. Choose a histogram, boxplot, or density plot to show a distribution, and read its shape.
5. Spot outliers and decide what to do about them.
6. Map a variable as a rate or density, and describe what spatial descriptive statistics add.

Someone hands you a data table: a hundred rows, a dozen columns. Now what?

The answer is *not* to jump straight to a model. First, get to know the data: how big it is, what each column holds, what is missing, and how each variable is spread out. That process of getting to know your data is **exploratory data analysis**, or **EDA**. When the data are spatial, we add one more question, about where values occur, and it becomes **exploratory spatial data analysis**, or **ESDA**.

Most examples in this chapter use the `nc` data that come with the sf package: North Carolina's 100 counties, with the number of births in each from 1974 to 1978 (`BIR74`) and from 1979 to 1984 (`BIR79`). The live code boxes use the `flights` data from Chapter 9.

17.1 From EDA to ESDA

17.1.1 The "understand" loop

Recall the data science workflow from *R for Data Science*. We import data and tidy it. Then we spend most of our time in a loop to **understand** it: transform, visualize, model, and back again. Only then do we communicate the results. EDA lives in that loop. It is iterative: each plot or summary raises a new question, which sends you back to transform the data again.

EDA is not one technique. It is everything we do to learn what is in a dataset before we try to explain or predict anything with it:

- cleaning and wrangling the data;
- finding duplicates and missing data (NA);
- calculating and transforming variables;
- making graphs, plots, charts, and maps;
- calculating descriptive statistics.



The payoff is better questions. Good data, known well, lead to good questions, and good questions guide the analysis that follows.

17.1.2 Two questions guide the investigation

EDA can feel open-ended, so use two questions to keep it focused:

1. **What type of variation occurs *within* my variables?** Answer it with central tendency, dispersion, and plots of one variable, such as histograms and boxplots. Most of this chapter answers this question.
2. **What type of variation occurs *between* my variables?** Answer it with plots of two or more variables, such as scatterplots and grouped boxplots, and with correlation.

17.1.3 ESDA adds a geographic question

Spatial data let us ask one more: **what type of *geographic* variation occurs within my variables?** Three tools answer it:

- **Maps**: where are the high and low values?
- **Spatial autocorrelation**: are similar values located near each other? This is Tobler's first law in action.
- **Clustering**: are there hot spots or cold spots?

This chapter covers maps and descriptive statistics. Chapter 17 covers spatial descriptive statistics, and Chapter 22 covers spatial autocorrelation.

17.2 Step one: get to know the data table

Before you look at any one variable, check the basics of the whole table:

- The number of **observations** (rows, or features).
- The number of **complete** observations. How many have NA?
- The number of **fields** (variables, or columns).
- The **data type** of each field. Does it match what the field means?

Useful R functions are `nrow()`, `ncol()`, `glimpse()`, and `summary()`. Try them on `flights`:

```
library(dplyr)
library(nycflights13)

nrow(flights)
ncol(flights)
glimpse(flights)

# How many flights have no air_time?
sum(is.na(flights$air_time))
```

Missing values change what you can calculate: `mean()` returns NA unless you set `na.rm = TRUE`. Types matter too. A FIPS code stored as a number loses its leading zero, and a count stored as text can't be averaged.



For `nc`, the same checks show 100 rows and 15 columns, including the geometry, and no missing values in `BIR74`:

```
library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf")) |>
  st_transform(32119)

nrow(nc)              # 100
sum(is.na(nc$BIR74))  # 0
```

These answers decide which approaches make sense: numeric fields get means and histograms, and categorical fields get counts and modes.

17.3 Describing a variable's center

Central tendency is one value that describes the central position of a set of values.

Table 17.1: Three measures of central tendency.

Measure	What it tells you	Use it for
Mean	The average: the sum of the values divided by the count	Numeric data
Median	The middle value once the values are sorted	Numeric data, especially skewed data
Mode	The most common value	Categorical (nominal) data, or any data

We'll work through each on the same ten values:

$$\{3, 8, 12, 6, 5, 10, 2, 3, 6, 3\}$$

Sorted, they are {2, 3, 3, 3, 5, 6, 6, 8, 10, 12}.

17.3.1 Mean

The **mean** is what most people call the average. Add the values and divide by how many there are:

$$\bar{x} = \frac{\sum x_i}{n} = \frac{58}{10} = 5.8$$

Here $\bar{x}$ ("x-bar") is the mean, $\sum$ means "add up", x_i is each value, and n is the count. Because every value enters the sum, one extreme value can pull the mean a long way.

17.3.2 Median

The **median** is the middle of the sorted values, so the first step is always to sort. With an odd number of values, the median is the single middle one. With an even number, it is the mean of the two middle values. We have 10 values, so we average the 5th and 6th: (5 + 6) / 2 = 5.5.

Half the values are below the median and half above, so extreme values barely move it.

17.3.3 Mode

The **mode** is the most frequent value. In the sorted list, 3 appears three times, more than any other value, so the mode is 3. A variable can have no mode, one, or several.

The mode is the only measure of center that works for categorical data. You can find the most common land-cover class, but you can't average land-cover classes. Base R has no function for the statistical mode: `mode()` returns how R stores an object. Use `table()` or `count()` and take the most frequent value instead.

```
a <- c(3, 8, 12, 6, 5, 10, 2, 3, 6, 3)

mean(a)
median(a)

table(a)                    # counts of each value
names(which.max(table(a)))  # the mode
```

17.4 Describing a variable's spread

The mean describes the center. **Dispersion** describes the spread, or variability, around it. Here are two sets of values with exactly the same mean:

- **A**: {3, 8, 12, 6, 5, 10, 2, 3, 6, 3}. Mean 5.8, from 2 to 12.
- **B**: {−56, 8, −1, 64, −34, 84, −5, 38, 6, −46}. Mean 5.8, from −56 to 84.

If you only reported the mean, these would look identical. They are not. That is why you should always pair a measure of center with a measure of spread.

17.4.1 Range and interquartile range

The **range** is the largest value minus the smallest. It is quick to calculate and easy to explain, but it uses only the two most extreme values, so one outlier changes it.

Quartiles split the sorted values into four groups of equal size. The first quartile, **Q1**, is the 25th percentile; the third, **Q3**, is the 75th; and the median is the second. The **interquartile range** (IQR) is Q3 − Q1: the spread of the middle half of the data. It ignores the extremes, so outliers barely affect it.

Take these 12 values: {3, 1, 2, 8, 12, 6, 5, 10, 2, 3, 6, 3}. Sorted and split in half, they are

$$\{1, 2, 2, 3, 3, 3 \mid 5, 6, 6, 8, 10, 12\}$$

- Range = 12 − 1 = 11.



- Q1 is the median of the lower half, between 2 and 3: 2.5.
- Q3 is the median of the upper half, between 6 and 8: 7.
- IQR = 7 − 2.5 = 4.5.

Software uses slightly different rules for percentiles. R's default rule interpolates between values, so it gives Q1 = 2.75 and Q3 = 6.5, and `IQR()` returns 3.75. With a small sample the difference is visible; with hundreds of values the methods agree closely. When you work by hand, show your method.

17.4.2 Standard deviation

The **standard deviation** is the most common measure of spread. It is roughly the typical distance between a value and the mean. To calculate it:

1. Subtract the mean from each value. This is the deviation.
2. Square each deviation, so negative ones don't cancel positive ones.
3. Add them up and divide by $n - 1$. This is the **variance**.
4. Take the square root, to return to the original units.

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}}$$

For set A the standard deviation is 3.3; for set B it is 45.7. Same mean, about 14 times the spread. Like the mean, the standard deviation uses every value, so it is sensitive to outliers. Check the numbers in R:

```
a <- c(3, 8, 12, 6, 5, 10, 2, 3, 6, 3)
b <- c(-56, 8, -1, 64, -34, 84, -5, 38, 6, -46)

c(mean(a), mean(b))
c(sd(a), sd(b))

x <- c(3, 1, 2, 8, 12, 6, 5, 10, 2, 3, 6, 3)
range(x)
diff(range(x))
quantile(x)
IQR(x)
```

17.4.3 All together

`summarize()` puts the whole toolkit in one call. Here it is for births in North Carolina's counties:

```
nc |>
  st_drop_geometry() |>
  summarize(min    = min(BIR74),
            q1     = quantile(BIR74, 0.25),
            median = median(BIR74),
            mean   = mean(BIR74),
            q3     = quantile(BIR74, 0.75),
            max    = max(BIR74),
```

```
            sd      = sd(BIR74),
            IQR     = IQR(BIR74))
```

Table 17.2: Summary statistics for births per county, 1974–78.

min	Q1	median	mean	Q3	max	sd	IQR
248	1,077	2,180.5	3,299.6	3,936	21,588	3,848.2	2,859

The key comparison is the mean against the median. The mean is about 1.5 times the median, and the standard deviation is larger than the mean. Both are signs of **right skew**: most counties have few births, and a few large urban counties, led by Mecklenburg with 21,588, pull the mean up. For skewed variables like this one, the median and the IQR describe a typical county better than the mean and the standard deviation.

17.5 Visualizing distributions

Summary statistics compress a variable into a few numbers. Plots show the whole **distribution**.

17.5.1 Histograms and boxplots

A **histogram** divides the range of a variable into bins, and draws a bar for the number of values in each. A **boxplot** draws the box from Q1 to Q3 with a line at the median. The whiskers reach to the furthest values within 1.5 times the IQR of the box, and points beyond them are drawn one by one as potential outliers.

```
library(ggplot2)

ggplot(nc, aes(x = BIR74)) + geom_histogram()
ggplot(nc, aes(y = BIR74)) + geom_boxplot()
```

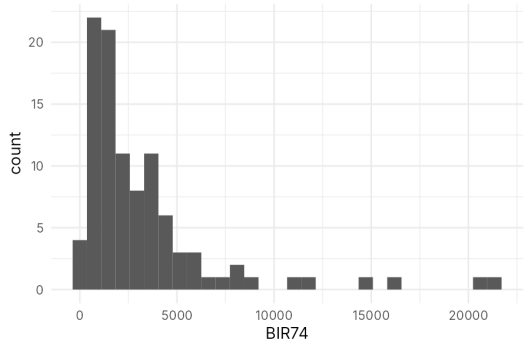


Figure 17.1: Histogram

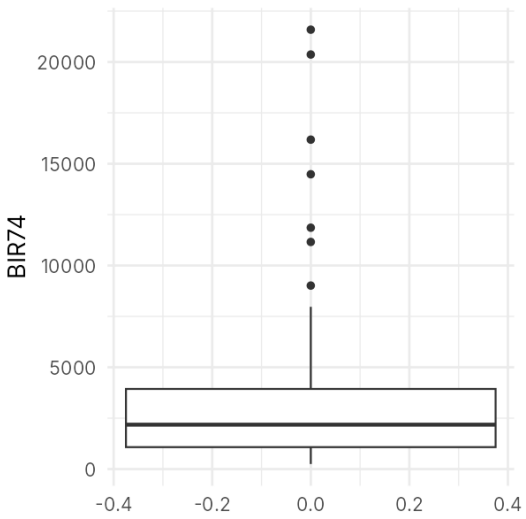


Figure 17.2: Boxplot

Both tell the same story as Table 17.2: births per county are strongly right-skewed.

17.5.2 Choosing a bin width

A histogram depends on one choice you make: the **bin width**. In ggplot, set `binwidth` or `bins` inside `geom_histogram()`. The default of `30` bins prints a message telling you to pick a better value.

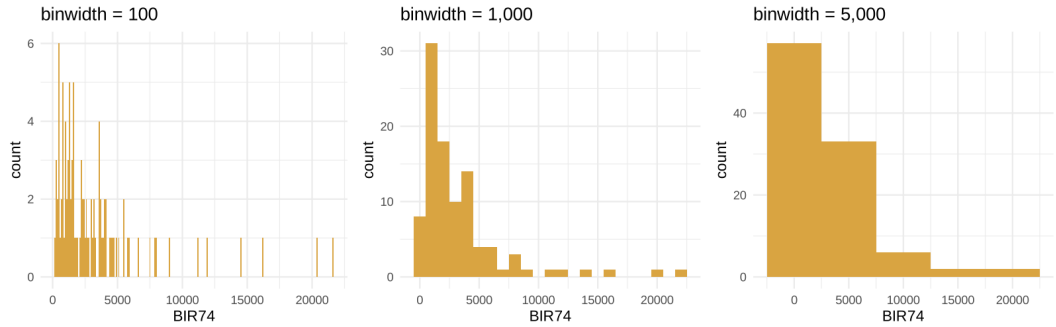


Figure 17.3: The same births data with three bin widths.

Too narrow, and every small fluctuation becomes its own spike; the shape is lost in noise. Too wide, and the bins swallow the detail. Try a few widths, and choose one that shows the structure without the noise.

17.5.3 Shape decides whether the mean is typical

Always look at the shape of a distribution before you trust its mean (Figure 17.4).

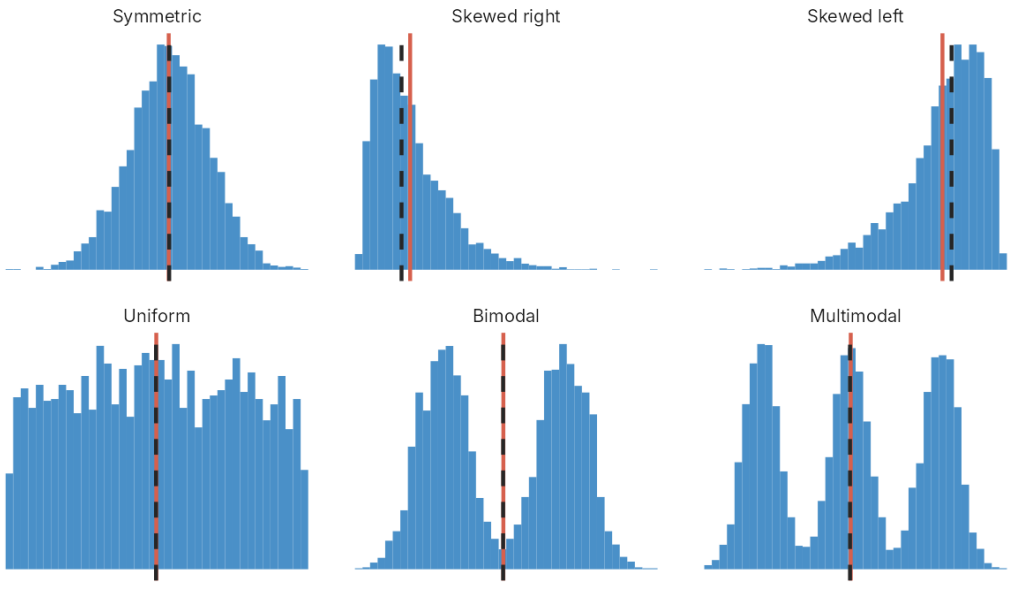


Figure 17.4: Six distribution shapes. The red line is the mean and the dashed line the median.

- **Symmetric**: the mean and the median are about the same.
- **Skewed**: the long tail pulls the mean toward it. Births per county are skewed right.
- **Uniform**: there's no peak at all.

- **Bimodal** or **multimodal**: the mean can land in a dip where few values are. Two or more peaks often mean that different groups are mixed together, such as urban and rural counties.

So ask: does the mean describe a typical value here? Or should I report the median, or split the data into groups?

17.5.4 Outliers

An **outlier** is a value far from the rest. Plots make outliers easy to spot. In a histogram, it is a lone bar far to the right, separated from the others by empty bins. In a boxplot, it is a point beyond the whisker. Seven counties are beyond the upper whisker in Figure 17.2, and Mecklenburg is the highest.

Ask what explains an outlier before deciding what to do. A large urban county is a real value and stays in. A typo or a wrong unit is an error to fix. Either way, outliers are why the mean and standard deviation can be so much larger than the median and IQR. Whatever you decide, say so in your write-up.

17.5.5 Density plots

A **density plot** is a smoothed histogram. Instead of counting values in bins, it estimates a smooth curve. The total area under the curve is 1, so the y-axis is a density, not a count. With a variable measured in thousands, the densities are tiny numbers. Peaks show where values concentrate.

```
ggplot(nc, aes(x = BIR74)) +
  geom_density(fill = "lightblue", alpha = 0.6)
```

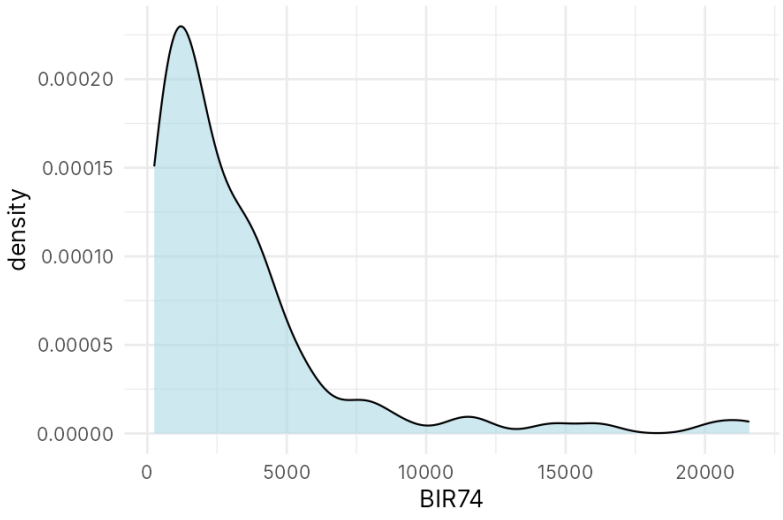


Figure 17.5: Density plot of births per county, 1974–78.

Density plots shine when you compare groups, because several curves can share one set of axes. To compare two columns, first stack them into one column with `pivot_longer()`. Then map the column names to `fill`:

```
library(tidyr)

nc |>
  st_drop_geometry() |>
  select(BIR74, BIR79) |>
  pivot_longer(everything()) |>
  ggplot(aes(x = value, fill = name)) +
  geom_density(alpha = 0.5)
```

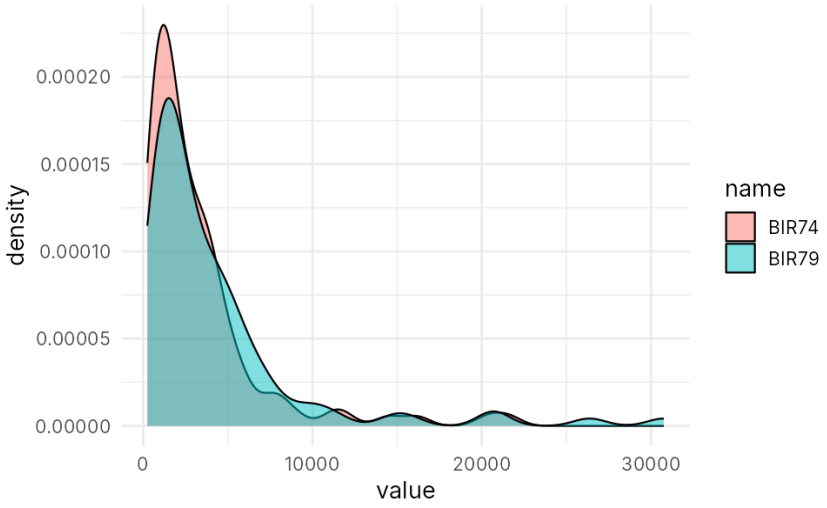


Figure 17.6: Births in 1974–78 and 1979–84, on one set of axes.

The `alpha` argument makes the curves see-through, so the overlap stays visible.

17.5.6 Your turn

Recreate a density plot of `air_time` for two airlines, American (AA) and United (UA). The code filters the flights, drops missing air times, and maps `carrier` to `fill`. Fill in the missing geom:

```
library(ggplot2)
library(dplyr)
library(tidyr)
library(nycflights13)

flights |>
  filter(carrier %in% c("AA", "UA")) |>
  drop_na(air_time) |>
  ggplot(aes(x = air_time, fill = carrier)) +
  geom_histogram()   # replace this with a semi-transparent density
```

Show answer

Replace `geom_histogram()` with `geom_density(alpha = 0.6)`. Why does `air_time` have several peaks? Each peak is a group of common routes with similar distances: a multimodal distribution.

17.6 Maps add the spatial dimension

Everything so far ignores location. A map adds it back, and this is where EDA becomes ESDA.

Map a rate, a density, or a percentage, not a raw count. Big counties have more of almost everything, so a map of raw counts mostly shows which counties are big. Here we divide births by each county's area, from `st_area()`:

```
library(tmap)

nc |>
  mutate(area_km2 = as.numeric(st_area(geometry)) / 1e6,
         births_per_km2 = BIR74 / area_km2) |>
  tm_shape() +
  tm_polygons(fill = "births_per_km2",
              fill.scale = tm_scale_continuous(values = "brewer.oranges"))
```

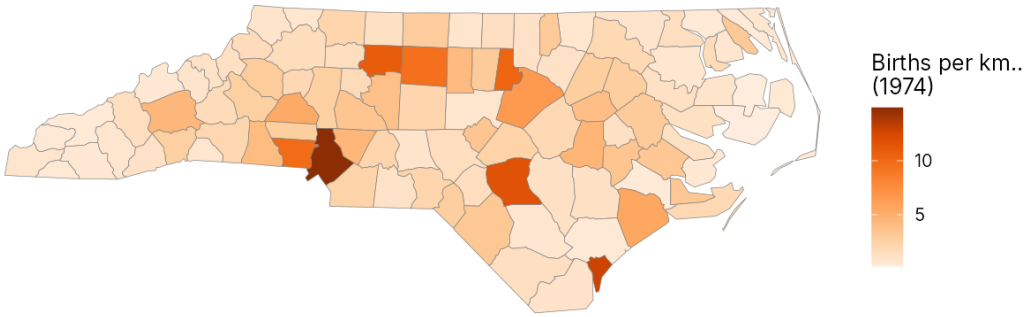


Figure 17.7: Births per square kilometer in each county, 1974–78.

Most of the state is pale, with low density, and a handful of urban counties stand out: Mecklenburg, New Hanover, Cumberland, Forsyth, and Durham, then Gaston and Guilford. Now ask the ESDA question. Are the high values scattered, or do they cluster together?

17.7 Spatial statistics describe location

The statistics in this chapter describe *values*. **Spatial descriptive statistics** describe *locations*:

- The **mean center** is the average x coordinate and the average y coordinate of all the features.
- The **weighted mean center** does the same, but weights each feature by an attribute, such as births or population.
- The **standard distance** is how far features typically sit from the mean center: the spatial version of the standard deviation.

And the question of whether nearby values are similar is measured with **spatial autocorrelation** statistics, such as Moran's I (Chapter 22). Chapter 17 covers the spatial descriptive statistics.

17.8 Check your understanding

1. What is the mean of {2, 4, 4, 10}? _[1]
2. What is the median of {2, 4, 4, 10}? _[2]
3. Which measure of center is the only one that works for categorical data, such as land-cover class? [3]

- (A) Mean
- (B) Median
- (C) Mode
- (D) Standard deviation

4. Which measure of spread is least affected by one extreme value? [4]

- (A) Range
- (B) Standard deviation
- (C) Interquartile range
- (D) Variance

5. A variable's mean is much larger than its median. Its distribution is probably: [5]

- (A) Skewed right
- (B) Skewed left
- (C) Symmetric
- (D) Uniform

6. In a boxplot, the box spans: [6]

- (A) The minimum to the maximum
- (B) Q1 to Q3
- (C) The mean plus and minus one standard deviation
- (D) The 5th to the 95th percentile

7. The area under a density curve is equal to 1. TRUE / FALSE[7]

[1] 5
[2] 4
[3] Mode
[4] Interquartile range
[5] Skewed right
[6] Q1 to Q3
[7] TRUE

8. A choropleth map should usually show raw counts rather than rates or densities. TRUE / FALSE[8]

17.9 Summary

- **Exploratory data analysis** (EDA) is how we learn what's in a dataset before explaining or predicting anything. **ESDA** adds the question of how values vary across space.
- Start with the table: rows, complete rows, columns, and data types.
- Describe each variable's center with the **mean**, **median**, or **mode**, and its spread with the **range**, **interquartile range**, or **standard deviation**. For skewed data, prefer the median and IQR.
- **Histograms**, **boxplots**, and **density plots** show a whole distribution. Choose the **bin width** with care, and read the shape: symmetric, **skewed**, uniform, or multimodal.
- Plots make **outliers** easy to spot. Find out what explains them before deciding what to do.
- **Maps** add location. Map rates or densities, not raw counts, and ask whether high values cluster.
- **Spatial descriptive statistics**, such as the mean center and standard distance, describe where features are.

17.10 Exercises

1. **By hand, then in R.** For {7, 1, 4, 9, 4, 12, 3}, calculate the mean, median, mode, range, and standard deviation by hand. Check each with R.
2. **Same mean, different spread.** Make two sets of 8 numbers with a mean of 10: one with a standard deviation under 2, and one with a standard deviation over 20.
3. **Explore `BIR79`.** Repeat Table 17.2 for `BIR79`. Did births become more or less skewed? Which counties are outliers?
4. **Bin widths.** Make histograms of `distance` in `flights` with three bin widths. Which shows the structure best? Describe the shape.
5. **Rates.** Make a map of `SID74 / BIR74`, the rate of sudden infant deaths per birth. Compare it with a map of raw `SID74`. Which counties change most?
6. **Two variables.** Make a scatterplot of `BIR74` against `BIR79`. What does it tell you about variation *between* the two variables?

17.11 Further reading

- *R for Data Science* (2nd ed.), by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund, free at https://r4ds.hadley.nz. The "Exploratory data analysis" chapter is the source of the two guiding questions.
- *Spatial Analysis Methods and Practice*, by George Grekousis (Cambridge University Press, 2020), covers exploratory spatial data analysis and spatial descriptive statistics.
- *Geographic Data Science with Python*, by Sergio Rey, Dani Arribas-Bel, and Levi John Wolf, free at https://geographicdata.science/book. Its chapters on choropleth mapping and spatial autocorrelation preview what comes next.

[8] FALSE

18 Spatial Descriptive Statistics

Learning objectives

By the end of this chapter you should be able to:

1. Describe where a set of features is with spatial descriptive statistics.
2. Match each spatial statistic to the familiar statistic it extends: the mean, the median, or the standard deviation.
3. Calculate a mean center, a weighted mean center, a central feature, and a standard distance, by hand and in R.
4. Write your own function in R, with inputs and a return value.
5. Read a function that calculates a weighted mean center for a layer of polygons.

Chapter 16 described the *values* of a variable: where its center is and how spread out it is. This chapter does the same for *locations*. Where is the middle of a set of features? Which feature is most central? And how spread out are they?

The answers are **spatial descriptive statistics**. In the second half of the chapter, you will also write your own functions in R, and read one that calculates a weighted mean center.

18.1 Many statistics have a spatial counterpart

Most descriptive statistics you already know have a spatial version (Table 18.1).

Table 18.1: Descriptive statistics and their spatial counterparts.

Statistic	Spatial counterpart	Question it answers
Mean	Mean center; weighted mean center	Where is the average location?
Median	Central feature	Which real feature is most central?
Standard deviation	Standard distance	How spread out are the locations?

A quick refresher on the last row. The standard deviation measures how spread out values are around their mean. In a normal, bell-shaped distribution, about 68% of values fall within one standard deviation of the mean, and about 95% within two. Keep that picture in mind: the standard distance draws the same idea on a map, as a circle.

Throughout the chapter we'll use six points, with x and y in kilometers and a weight, such as population in thousands:

```
x <- c(2, 4, 5, 7, 8, 12)
y <- c(3, 7, 2, 5, 9, 4)
w <- c(10, 50, 5, 20, 5, 10)
```

```
plot(x, y, pch = 16, cex = sqrt(w) / 2, asp = 1,
     xlim = c(0, 13), ylim = c(0, 10))
text(x, y, labels = 1:6, pos = 4)
```

18.2 Mean center

The **mean center** is the simplest spatial statistic. Average all the x coordinates, average all the y coordinates, and that pair is the mean center:

$$\bar{X} = \frac{\sum x_i}{n}, \qquad \bar{Y} = \frac{\sum y_i}{n}$$

```
mean_x <- mean(x)
mean_y <- mean(y)
c(mean_x, mean_y)
```

For our six points the mean center is at (6.33, 5). It doesn't have to be one of the points, and here it isn't.

Like the ordinary mean, the mean center is sensitive to outliers. In Figure 18.1, adding one distant point drags the center toward it.

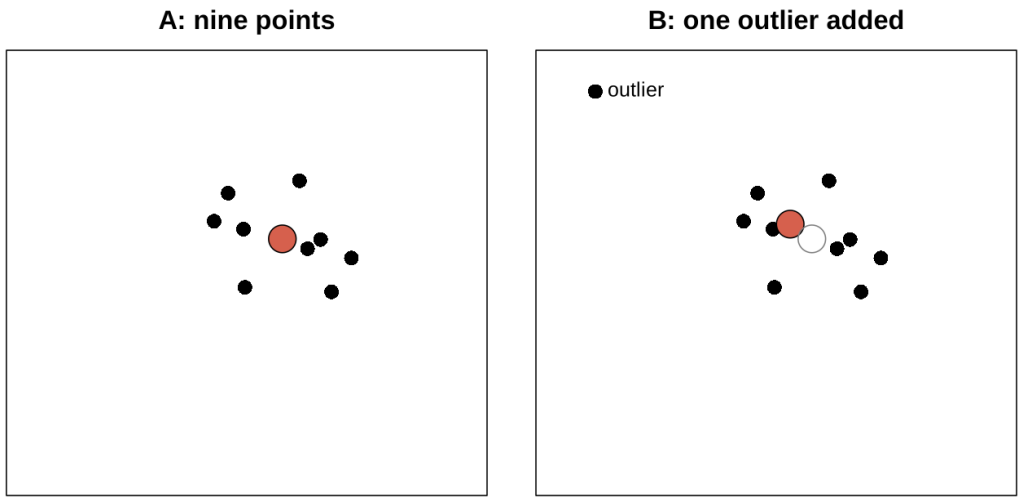


Figure 18.1: The mean center (red) of nine points (A), and of the same points with one outlier added (B). The open circle in B marks where the center was before.

18.2.1 The mean center of North Carolina

Figure 18.2 shows the mean center of North Carolina's counties, calculated from the centroid of each of the 100 counties. It falls in the Piedmont, in Chatham County. Every county counts the same here, so tiny Chowan County pulls as hard as Wake or Mecklenburg.

18.3 Weighted mean center

Often some locations matter more than others. A county with a million people should count for more than one with 4,000. The **weighted mean center** multiplies each coordinate by its weight, *w*, adds them up, and divides by the sum of the weights:

$$\bar{X}_w = \frac{\sum w_i x_i}{\sum w_i}, \qquad \bar{Y}_w = \frac{\sum w_i y_i}{\sum w_i}$$

Base R's `weighted.mean()` does exactly this:

```
wmc_x <- weighted.mean(x, w)
wmc_y <- weighted.mean(y, w)
c(wmc_x, wmc_y)

# The same thing, written out
sum(w * x) / sum(w)
```

Point 2 carries half the total weight, so it pulls the center from (6.33, 5) to (5.45, 5.75), toward itself.

The best-known weighted mean center is the U.S. **center of population**. After every census, the Census Bureau calculates the mean center of the country's population, weighting each location by the number of people who live there. As Figure 14.1 shows, it has moved from Maryland in 1790 to Missouri in 2020: west with settlement, and since about 1950 south as well, with the growth of the Sun Belt.

Now the same for North Carolina, with each county weighted by its births from 1974 to 1978 (Figure 18.2).

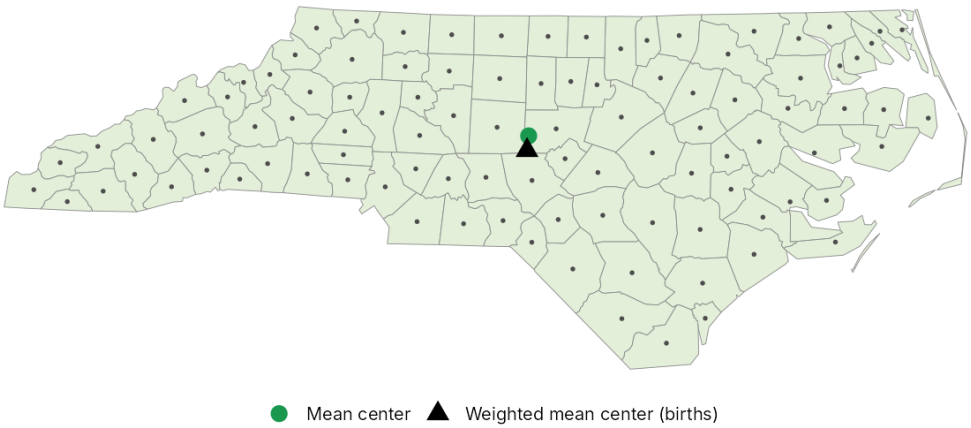


Figure 18.2: The mean center of North Carolina's 100 county centroids (green), and the mean center weighted by births, 1974–78 (black).

The weighted center moves about 12 km, almost due south. Several of the counties with the most births, such as Mecklenburg, Cumberland, Onslow, and New Hanover, lie south of the state's middle, while many counties along the Virginia line had few. East to west, Mecklenburg and the Triangle roughly balance each other. With recent population data, the Triangle's fast growth pulls the weighted center east instead. The weights you choose change the answer, so say which you used.



Think about it

What would the center of North Carolina look like if you weighted each county by its land area instead? By its number of hog farms? What question would each center answer?

18.4 Central feature

The **central feature** is the existing feature that is closest to all the others. To find it, measure the distance from each feature to every other feature, add them up, and pick the feature with the smallest total.

It is the spatial cousin of the median. Instead of averaging, it picks a real feature from the data. That matters when the answer must be a real place: say, choosing which of your existing offices should host a regional meeting.

```
pts <- cbind(x, y)
d <- as.matrix(dist(pts))   # distance from every point to every other
round(d, 1)

total <- rowSums(d)
round(total, 1)
which.min(total)            # the central feature
```

Point 4, at (7, 5), has the smallest total distance, 21.8 km, so it is the central feature. The mean center, (6.33, 5), is close by, but it isn't a point in the data.

For North Carolina, the central feature of the county centroids is Chatham County (Figure 18.3).

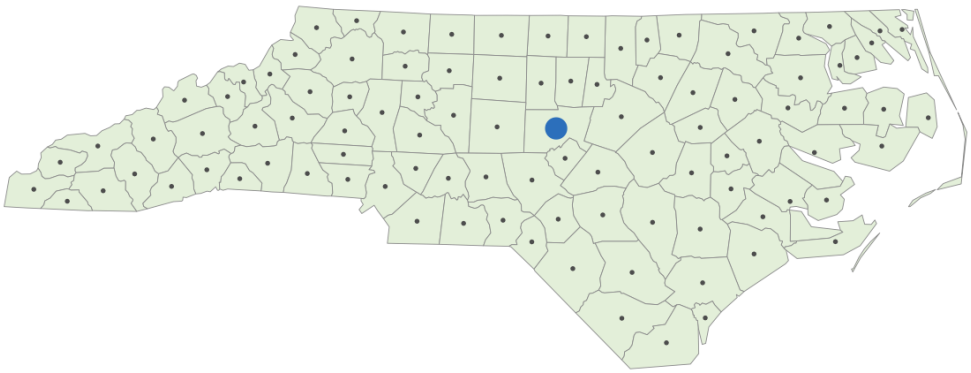


Figure 18.3: The central feature (large blue dot) of North Carolina's county centroids: Chatham County.

18.5 Standard distance

The **standard distance** is the spatial version of the standard deviation: the typical distance between features and their mean center.

$$SD = \sqrt{\frac{\sum (x_i - \bar{X})^2}{n} + \frac{\sum (y_i - \bar{Y})^2}{n}}$$

Take each point's distance from the mean center in x and in y, square them, average, add, and take the square root.

```
sd_dist <- sqrt(sum((x - mean(x))^2) / length(x) +
                sum((y - mean(y))^2) / length(y))
sd_dist

plot(x, y, pch = 16, asp = 1, xlim = c(0, 13), ylim = c(0, 10))
points(mean(x), mean(y), pch = 4, cex = 2, col = "red")
symbols(mean(x), mean(y), circles = sd_dist, inches = FALSE,
        add = TRUE, fg = "red")
```

We draw it as a circle centered on the mean center, with the standard distance as its radius. A large circle means the features are spread out; a small one means they cluster tightly.

Comparing circles is how we compare dispersion: between two weightings, two time periods, or two kinds of features. In Figure 18.4, the solid circle uses every county equally, and has a radius of about 203 km. The dashed circle is weighted by births. It is smaller, about 158 km, because most births happened in a few urban Piedmont counties.

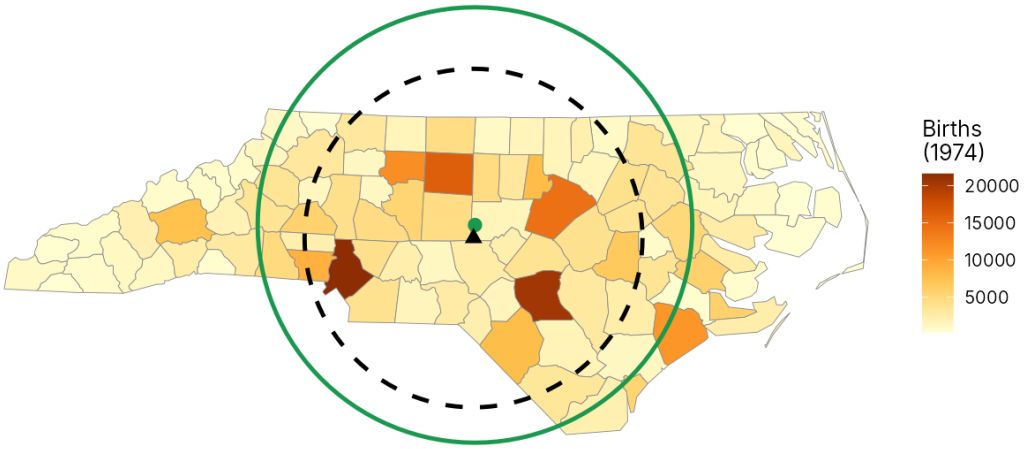


Figure 18.4: Standard distance circles for North Carolina's counties: unweighted (green, solid) and weighted by births (black, dashed). Counties are shaded by births, 1974–78.

18.6 Functions in R

So far we have calculated these statistics one line at a time. A **function** wraps code up so you can run it on any data with one line.

18.6.1 What is a function?

You have used functions all along: `mean()`, `st_read()`, `ggplot()`, and Chapter 5 showed how to write your own. Here is a quick review before we put them to work on spatial statistics. A function is a named, reusable block of code. Every function has:



- a **name**;
- **inputs**, called **arguments**;
- an **output**, called the **return value**.

You call a function by its name, with its inputs in parentheses.

18.6.2 Writing your own

Here is a function named `addSeven`. Its one input is `numbers`, and it returns those numbers plus 7:

```
addSeven <- function(numbers) {
  numbers + 7
}

addSeven(c(1, 2, 3))
addSeven(c(4, 5, 6))
```

The name goes on the left of the assignment arrow. The word `function` is followed by the inputs in parentheses, and the code goes inside the braces. The value of the last line is what the function returns. `addSeven(c(1, 2, 3))` returns 8, 9, 10, because R adds 7 to every element of the vector.

Your turn. Write `timesTwo()`, which doubles its input. Then write `addAny()`, with a second input, `amount`, so that it adds any number you choose:

```
timesTwo <- function(numbers) {
  # your code here
}

timesTwo(c(1, 2, 3))   # should return 2 4 6
```

Show answer

```
timesTwo <- function(numbers) {
  numbers * 2
}

addAny <- function(numbers, amount) {
  numbers + amount
}

addAny(c(1, 2, 3), 10)   # 11 12 13
```

A function can also bundle up a whole calculation. This one returns the standard distance of any set of points:

```
standardDistance <- function(x, y) {
  sqrt(sum((x - mean(x))^2) / length(x) +
       sum((y - mean(y))^2) / length(y))
}
```

```
standardDistance(x, y)
standardDistance(c(0, 1, 0, 1), c(0, 0, 1, 1))
```

18.6.3 A function for the weighted mean center

Here is a function, `meanCenter()`, that finds the weighted mean center of any layer of polygons. Its inputs are a spatial layer and a vector of weights. Its output is a point:

```
library(sf)

meanCenter <- function(layer, weights) {
  # convert the polygons into points
  points <- st_centroid(layer)
  # calculate the weighted mean x and y coordinates
  mc_x <- weighted.mean(st_coordinates(points)[, 1], weights)
  mc_y <- weighted.mean(st_coordinates(points)[, 2], weights)
  # create a point
  mean_center <- st_sfc(st_point(c(mc_x, mc_y)))
  # give it the same coordinate reference system as the layer
  mean_center <- st_set_crs(mean_center, st_crs(layer))
  # return the point
  mean_center
}
```

Walk through it from top to bottom. `st_centroid()` turns each polygon into a point. `st_coordinates()` returns a matrix of those points' coordinates, with `x` in the first column and `y` in the second, and `weighted.mean()` averages each one using the weights. `st_point()` and `st_sfc()` turn the two numbers into a point geometry, and `st_set_crs()` gives it the same coordinate reference system as the input layer. The last line, `mean_center`, is what the function returns.

Notice that `weights` works just like `amount` in `addAny()`: it's a second input that changes the result. Pass a column of 1s, or any constant, and you get the unweighted mean center. Use it like this:

```
nc <- st_read(system.file("shape/nc.shp", package = "sf")) |>
  st_transform(32119)

mc_births <- meanCenter(nc, nc$BIR74)
mc_plain  <- meanCenter(nc, rep(1, nrow(nc)))

st_distance(mc_plain, mc_births)   # about 12,200 m
```

Why project to State Plane first? The mean of latitudes and longitudes isn't a true center, for the same reasons degrees aren't a distance (Section 13.7). Calculate spatial statistics in a projected coordinate system.



18.7 Check your understanding

1. The spatial counterpart of the median is the: [1]

- (A) Mean center
- (B) Central feature
- (C) Standard distance
- (D) Weighted mean center

2. Three points are at (0, 0), (4, 0), and (2, 6). What is the x coordinate of their mean center? _[2]
3. And the y coordinate? _[3]
4. The mean center is always one of the features in the data. TRUE / FALSE[4]
5. Adding one distant outlier moves the mean center toward it. TRUE / FALSE[5]
6. The U.S. center of population is an example of a: [6]

- (A) Central feature
- (B) Weighted mean center
- (C) Standard distance
- (D) Spatial join

7. A larger standard distance circle means the features are: [7]

- (A) More spread out
- (B) More tightly clustered
- (C) Closer to the Equator
- (D) More heavily weighted

8. In `addAny <- function(numbers, amount) { numbers + amount }`, what does `addAny(c(1, 2), 5)` return? [8]

- (A) 6 7
- (B) 5 5
- (C) 8

[1] Central feature
[2] 2
[3] 2
[4] FALSE
[5] TRUE
[6] Weighted mean center
[7] More spread out
[8] 6 7

- (D) 1 2 5

18.8 Summary

- **Spatial descriptive statistics** describe the center and spread of a set of locations.
- The **mean center** is the average x and the average y. Like the mean, it is pulled by outliers.
- The **weighted mean center** weights each feature by an attribute, such as population. The U.S. **center of population** is the best-known example.
- The **central feature** is the real feature with the smallest total distance to all the others: the spatial cousin of the median.
- The **standard distance** is the typical distance from the mean center: the spatial standard deviation, drawn as a circle.
- A **function** packages code you'll reuse. It has a name, **arguments** (inputs), and a **return value** (the last line). `meanCenter()` combines sf functions to find the weighted mean center of any layer.
- Calculate spatial statistics on projected data.

18.9 Exercises

1. **By hand.** Five stores are at (1, 1), (3, 2), (2, 5), (6, 4), and (8, 8), in km. Find the mean center, the central feature, and the standard distance by hand, then check them in R.
2. **Weights.** Give the five stores weights of 100, 20, 20, 20, and 500 (their sales). Find the weighted mean center. Which way did it move, and why?
3. **Your own function.** Write a function, `centralFeature(x, y)`, that returns the number of the central feature. Test it on the six points in this chapter.
4. **Weighted standard distance.** Extend `standardDistance()` to take a third input, `w`, and return the weighted standard distance. (Replace each mean with `weighted.mean()`, and each division by *n* with a division by the sum of the weights.)
5. **Two periods.** Use `meanCenter()` to find the weighted mean center of births in North Carolina for `BIR74` and for `BIR79`. Did it move? How far, and in which direction?
6. **Population.** Once you have read Chapter 13, download the population of each North Carolina county from the American Community Survey with tidycensus, and find the population-weighted mean center. How does it compare with the center for 1970s births?

18.10 Further reading

- *Spatial Analysis Methods and Practice*, by George Grekousis (Cambridge University Press, 2020). Chapter 2 covers the mean center, weighted mean center, central feature, standard distance, and the standard deviational ellipse.
- The U.S. Census Bureau's Centers of Population page, https://www.census.gov/geographies/reference-files/time-series/geo/centers-population.html, has the national, state, and county centers for every census since 1790.
- *R for Data Science* (2nd ed.), by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund, free at https://r4ds.hadley.nz. The "Functions" chapter explains when and how to write your own.

Part V

Unit 5: Statistics and Science



This unit turns data into evidence. It tests whether groups of places differ, models how one variable changes with others, and puts the pieces together into a reproducible spatial data science project, from the research question to the conclusion. It ends by testing Tobler's first law of geography, that near things are more related than distant things: first deciding what "near" means, with distances and neighbors, and then measuring whether values on a map are clustered, random, or dispersed.

Chapter 18: Comparing Groups: p-values and ANOVA

This chapter places statistical tests in the cycle of data analysis. It explains what a p-value does and doesn't tell you, and why the replication crisis changed how scientists report them. It then compares group means with analysis of variance, from a small worked example to median male age across North Carolina's regions, with follow-up tests and alternatives for when ANOVA's assumptions fail.

Chapter 19: Linear Regression

Regression asks how one variable changes as another changes. This chapter covers correlation, fitting a line with `lm()`, slopes, intercepts, residuals, and predictions, and how to read R's model summary, including R^2 and adjusted R^2. It models median house value in North Carolina's counties, then maps the residuals to see where the model fails.

Chapter 20: Writing a Spatial Data Science Project

This chapter steps back to ask how data become knowledge. It covers the data, information, knowledge, wisdom pyramid, big data, choosing sources, and why reproducibility and replicability matter. It then walks through the parts of a project, from the research question and background to methods, results, and conclusions, and ends with what makes a project spatial data science.

Chapter 21: Distance and Neighborhoods

This chapter covers what distance means and the main ways to measure it: Euclidean, great circle, Manhattan, and network distance. It then uses distance and shared borders to decide which features are neighbors, with distance bands, *k* nearest neighbors, and rook and queen contiguity, and stores those relationships in a neighborhood weight matrix, built in R with spdep.



Chapter 22: Spatial Clustering and Spatial Autocorrelation

This chapter tests objectively whether things on a map are clustered. It measures point patterns with the Average Nearest Neighbor ratio, global spatial autocorrelation with Moran's I and a permutation test, and local hot spots and cold spots with Getis-Ord Gi*, using births in North Carolina's counties.

19 Comparing Groups: p-values and ANOVA

Learning objectives

By the end of this chapter you should be able to:

1. Place statistical testing in the cycle of data analysis.
2. Explain what a statistic, a null hypothesis, and a p-value are, and read a test result in three steps.
3. Describe how p-values are misused, and what to report alongside them.
4. Explain how ANOVA compares variation between groups with variation within groups, using the F statistic.
5. Run and interpret an ANOVA in R with `aov()`, and follow it with Tukey's HSD.
6. Explain why observations of neighboring places make ANOVA's p-values less trustworthy.

Western North Carolina's mountain counties are known as a retirement destination. The Piedmont has the big cities, universities, and their young workers. So do men in the west really tend to be older than men in the Piedmont? Or could the difference we see in the numbers be due to chance?

That is the question behind this chapter: **do groups of places really differ, or could the difference be chance?** We'll build up the tools in order. First, what data analysis is, and what a statistic and a p-value are. Then variance, the measure of spread that the method is built on. Then **analysis of variance (ANOVA)**, applied to median male age in North Carolina's 100 counties. We finish with what changes when the observations are places on a map.

19.1 What is data analysis?

Data analysis turns raw observations into evidence. We summarize what the data show, and we test whether the patterns are real or could be due to chance. It is a cycle, not a single test:

Table 19.1: The cycle of data analysis.

Step	What you do
1. Ask	Pose a question the data can answer.
2. Collect	Gather, clean, and organize the observations.
3. Describe	Summarize center, spread, and shape.
4. Test	Check whether a pattern could be due to chance.
5. Communicate	Report findings, maps, and their uncertainty.

Chapter 16 covered step 3, describing data. This chapter is about step 4, testing, and about the measure of spread, variance, that connects the two.

19.2 Statistics and p-values

A **statistic** is a number calculated from a sample that summarizes the data. There are two broad kinds:

- **Descriptive statistics** summarize: the mean, the median, the standard deviation.
- **Test statistics** weigh evidence: *t*, *F*, chi-square.

Because a statistic comes from a sample, it estimates a value for the whole population, and it would change if you drew a different sample. A test asks whether the pattern in *this* sample is bigger than that sample-to-sample wobble would produce by chance.

Every test starts from a **null hypothesis**, written H□: the boring claim that there is no difference, no relationship, nothing going on. The **alternative hypothesis**, H□, is that there is something. The test then works out how surprising your data would be *if the null hypothesis were true*.

19.2.1 Reading a test result

Read the output of any statistical test in three steps:

1. **The test statistic.** How large the pattern is compared with random variation, for example *F*.
2. **The p-value.** The probability of a result at least this extreme *if the null hypothesis were true*, that is, if the groups truly did not differ.
3. **The decision.** By convention, if $p < 0.05$, reject the null hypothesis: the result is called **statistically significant**. If $p \geq 0.05$, you have no evidence of a difference. That is not the same as evidence of *no* difference.

Suppose an ANOVA gives $p = 0.03$. That means: if the groups truly didn't differ, a result this extreme would happen about 3% of the time. It does *not* mean there is a 3% chance the null hypothesis is true, or that the difference between the groups is large.

> ⚠ **Significant is not the same as important**
>
> A tiny difference can be statistically significant if you have lots of data, and a large one can miss $p < 0.05$ if you have little. A p-value says nothing about the size of a difference. Always report the size of the difference too.

19.2.2 p-values and the replication crisis

p-values matter because they are the shared test of "could this be chance?", and for decades journals have used $p < 0.05$ as the bar for publication. That bar created bad incentives:

- **p-hacking:** re-running an analysis in different ways until something crosses 0.05.
- **The file drawer:** results that don't reach 0.05 go unpublished, so the published record overstates how often effects are found.
- **The bright line:** treating 0.049 as "real" and 0.051 as "nothing", as if they were different worlds.

Table 19.2: A century of the p-value. Sources: Ioannidis (2005), *PLOS Medicine*; Open Science Collaboration (2015), *Science*; Wasserstein and Lazar (2016), *The American Statistician*; Amrhein, Greenland, and McShane (2019), *Nature*.

Year	Event
1925	R. A. Fisher suggests $p < 0.05$ as a convenient guide, not a fixed rule.
1933	Jerzy Neyman and Egon Pearson add fixed error rates. Textbooks later blend the two ideas.
2005	John Ioannidis argues that most published research findings are false.
2015	The Open Science Collaboration repeats 100 psychology studies: 97% of the originals had reported $p < 0.05$, but only 36% of the repeats did.
2016	The American Statistical Association warns that p-values are widely misread.
2019	More than 800 scientists sign a call to retire "statistical significance".

The failure of so many published findings to hold up when other researchers repeated them is called the **replication crisis**. The lesson for your own work: report the p-value *with* the size of the effect and a confidence interval, never on its own. And treat $p = 0.051$ and $p = 0.049$ as what they are: almost exactly the same amount of evidence.

19.3 Variance: measuring spread

ANOVA is built on **variance**, which you met in Chapter 16. Variance is the average squared distance of each value from the mean:

$$s^2 = \frac{\sum (x_i - \bar{x})^2}{n - 1}$$

For a sample we divide by $n - 1$ rather than n. Small variance means values cluster near the mean; large variance means they spread widely. The **standard deviation**, s, is the square root of the variance, back in the data's own units.

```
a <- c(3, 4, 5, 5, 6, 7)

sum((a - mean(a))^2) / (length(a) - 1)   # variance by hand
var(a)                                   # the same, with var()
sd(a)                                    # standard deviation = sqrt(variance)
```

19.3.1 Two kinds of variation

Now suppose we have three groups, A, B, and C, of six values each (Figure 19.1). There are two kinds of variation:

- **Within-group variation:** the spread of the values around their *own* group's mean.

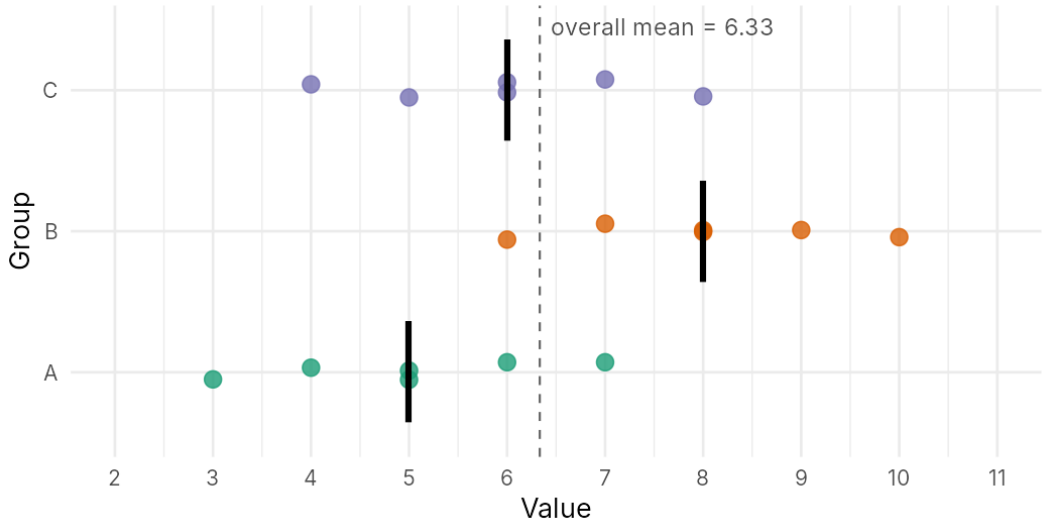


Figure 19.1: Three groups of six values. Each colored dot is one value, the black bar is that group's mean, and the dashed line is the overall mean of all 18 values.

- **Between-group variation:** how far the group means sit from the overall mean.

ANOVA compares the two:

$$F = \frac{\text{between-group variance}}{\text{within-group variance}}$$

If the group means sit far apart *relative to* the noise inside each group, F is large, and the groups differ more than chance alone would produce. If the groups are really just random draws from one population, F hovers around 1.

19.4 ANOVA: are the group means different?

Analysis of variance, or **ANOVA**, tests whether the means of three or more groups differ. (With only two groups, you would use a *t*-test. ANOVA extends the idea to more groups without the extra false positives you'd get from running many *t*-tests.)

The two hypotheses are:

- H₀: μ₁ = μ₂ = μ₃. All the group means are equal.
- H₁: At least one group mean differs.

ANOVA builds F from **sums of squares** (SS), the summed squared distances from the means, each divided by its **degrees of freedom** (df) to give a **mean square** (MS). With k groups and N observations in all:

Table 19.3: How ANOVA builds the F statistic.

Source	df	Mean square (MS)
Between groups	$k - 1$	SS between ÷ $(k - 1)$
Within groups (residuals)	$N - k$	SS within ÷ $(N - k)$
F statistic		MS between ÷ MS within



For the three groups in Figure 19.1, $k = 3$ and $N = 18$. The overall mean is 6.33. Work it through:

- **SS between** = $6 \times (5 - 6.33)^2 + 6 \times (8 - 6.33)^2 + 6 \times (6 - 6.33)^2 = 28$. With $3 - 1 = 2$ df, MS between = 14.
- **SS within**: each group's values are spread around its own mean with a variance of 2, so its squared distances add up to $5 \times 2 = 10$. Three groups give 30. With $18 - 3 = 15$ df, MS within = 2.
- ***F*** = $14 \div 2 = \mathbf{7}$.

`aov()` fits the model, and `summary()` prints the ANOVA table. The formula `value ~ group` reads "value, by group":

```
d <- data.frame(
  group = rep(c("A", "B", "C"), each = 6),
  value = c(3, 4, 5, 5, 6, 7,  6, 7, 8, 8, 9, 10,  4, 5, 6, 6, 7, 8)
)

fit <- aov(value ~ group, data = d)
summary(fit)
```

$F = 7$, with 2 and 15 degrees of freedom, and $p = 0.0071$. If all three groups came from one population, an F this large would turn up less than 1% of the time, so we reject H□: at least one group mean differs.

Where does the p-value come from? If H□ is true, F follows a known shape, the ***F* distribution**, which depends on the two degrees of freedom. The p-value is the share of that distribution beyond your F. In R, `pf()` gives it directly:

```
pf(7, df1 = 2, df2 = 15, lower.tail = FALSE)
```

19.5 Example: median male age by North Carolina region

Back to the question we started with: **is the median age of males different between western, central, and eastern North Carolina?**

- **Data:** American Community Survey 5-year estimates of median age for males (table B01002) for all 100 North Carolina counties, downloaded with tidycensus (Chapter 13).
- **Groups:** the west (the mountains), the center (the Piedmont), and the east (the coastal plain). Each county belongs to one region (Figure 19.2).
- **Unit of analysis:** the county, not the person or the census tract.

Before testing anything, look at the data. Each dot in Figure 19.3 is one county, and each box holds the middle half of the counties in its region:

```
nc_df <- st_drop_geometry(nc)

ggplot(nc_df, aes(region, estimate, fill = region)) +
  geom_boxplot(alpha = 0.6, outlier.shape = NA) +
  geom_jitter(width = 0.15, size = 1.8, alpha = 0.8) +
  scale_fill_manual(values = c("#1b9e77", "#d95f02", "#7570b3")) +
```

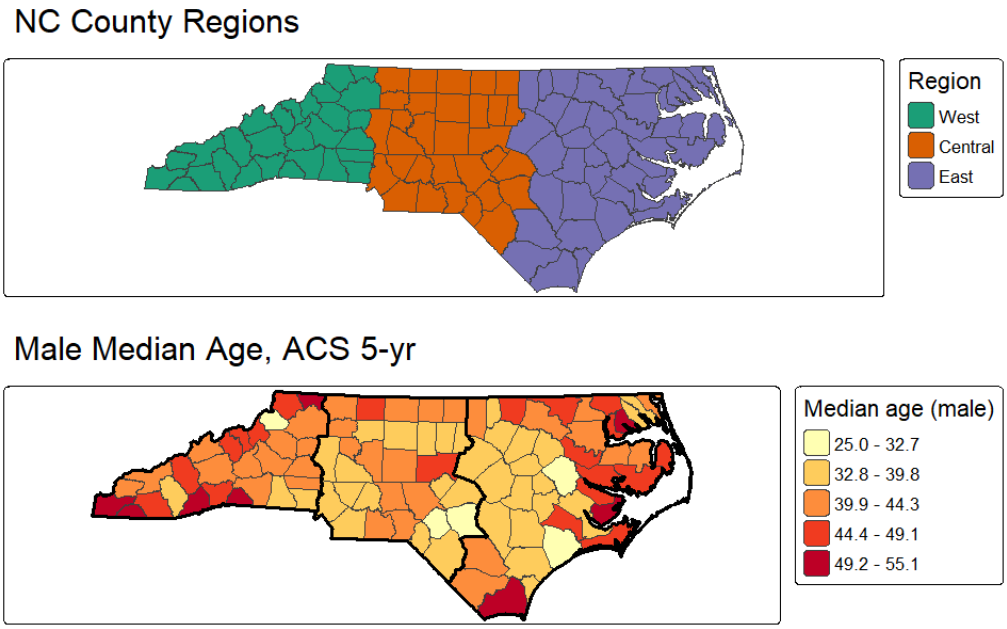


Figure 19.2: North Carolina's counties grouped into three regions (top), and the median age of males in each county from the ACS 5-year estimates (bottom).

```
    labs(title = "Male Median Age by Region, North Carolina Counties",
         x = NULL, y = "Median age (male, years)",
         caption = "Source: ACS 5-year estimates, table B01002") +
    theme_minimal() +
    theme(legend.position = "none")
```

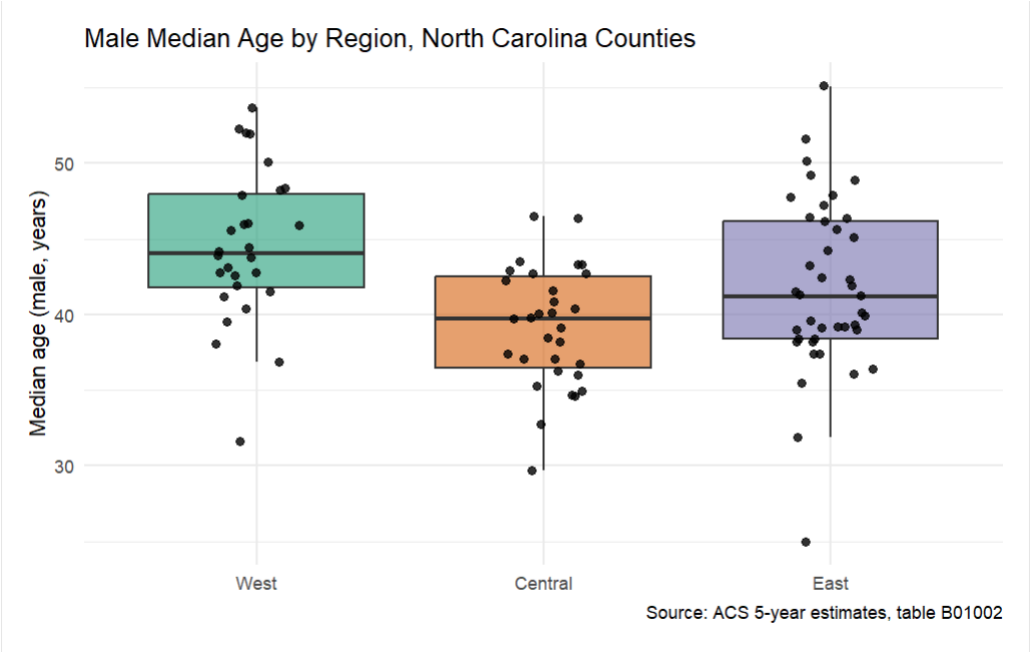


Figure 19.3: Median male age by region. Each dot is one county.

The table puts numbers on the picture:

```
nc_df |>
  group_by(region) |>
  summarize(n = n(),
            mean   = mean(estimate, na.rm = TRUE),
            sd     = sd(estimate, na.rm = TRUE),
            median = median(estimate, na.rm = TRUE))
```

Table 19.4: Median male age, in years, by region.

Region	Counties	Mean	SD	Median
West	28	44.5	5.0	44.1
Central	31	39.2	3.9	39.7
East	41	41.8	5.6	41.2

The West is oldest, with a mean of 44.5 years, and the Piedmont youngest, at 39.2. The East sits between, at 41.8, with the widest spread. But the boxes overlap a lot. Is the gap between the regional means large compared with the spread inside each region? That is exactly what ANOVA tests.

19.5.1 The ANOVA

One line fits the model: `estimate` is the variable we compare, and `region` defines the groups.

```
fit <- aov(estimate ~ region, data = nc_df)
summary(fit)
```

```
             Df Sum Sq Mean Sq F value   Pr(>F)
region        2  420.8  210.41   8.439 0.000418 ***
Residuals    97 2418.5   24.93
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Read it in three steps:

1. **Test statistic.** $F = 210.41 \div 24.93 = 8.44$. The variation between regions is about eight times the variation within them. With $k = 3$ regions and $N = 100$ counties, the degrees of freedom are 2 and 97.
2. **p-value.** 0.000418, far below 0.05. If the three regions really had the same mean, an F this large would almost never happen (Figure 19.4).
3. **Decision.** Reject the null hypothesis. At least two of the regions differ in median male age.

Notice what the result does *not* say. It doesn't say all three regions differ, only that some do. And it doesn't say *why*: region doesn't cause age. Retirement migration, college towns, and military bases are all possible explanations that the test can't tell apart.

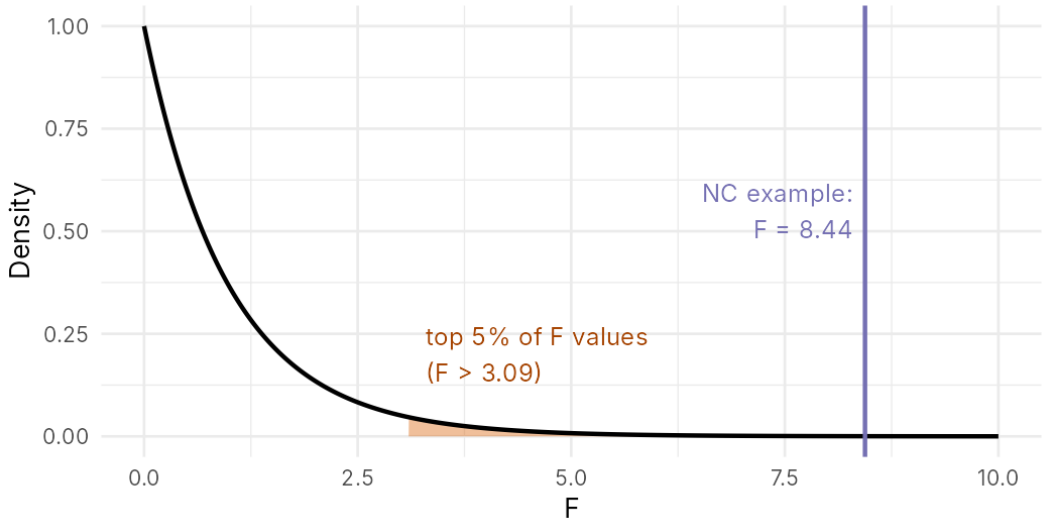


Figure 19.4: The F distribution with 2 and 97 degrees of freedom: the values of F you'd expect if the regions didn't differ. The shaded tail is the top 5%; the NC example's F = 8.44 lies far beyond it.

Think about it

Before you saw the data, which region would you have guessed has the oldest men? What features of a county (a university, a military base, a retirement community) would pull its median age down or up?

19.6 After ANOVA: which groups differ?

ANOVA tells us that at least two groups differ, not which ones. To find out, run a **post-hoc test**. **Tukey's honestly significant difference (HSD)** test compares every pair of groups and adjusts the p-values, so that making three comparisons doesn't inflate the chance of a false positive:

```
d <- data.frame(
  group = rep(c("A", "B", "C"), each = 6),
  value = c(3, 4, 5, 5, 6, 7,  6, 7, 8, 8, 9, 10,  4, 5, 6, 6, 7, 8)
)
fit <- aov(value ~ group, data = d)

TukeyHSD(fit)
```

For the three groups, only B and A differ significantly: B's mean is 3 higher, with an adjusted p-value of 0.006. C and A (p = 0.46) and C and B (p = 0.066) do not. In the North Carolina example, `TukeyHSD(fit)` on the regional model shows which pairs of regions differ; try it in Exercise 2.

19.6.1 Check the assumptions

ANOVA's p-value is only trustworthy if three assumptions hold:



1. **Independence:** each observation tells you nothing about the others.
2. **Normality:** the values in each group, or more exactly the model's residuals, are roughly normal.
3. **Equal variances:** the groups have similar spreads.

The **Shapiro-Wilk test** checks normality. Its null hypothesis is that the residuals *are* normal, so a large p-value is good news. For the North Carolina model, `shapiro.test(residuals(fit))` gives $W = 0.98$ and $p = 0.18$: no evidence that the residuals aren't normal.

If the assumptions fail, there are standard fallbacks:

- **Welch's ANOVA**, `oneway.test()`, does not assume equal variances.
- The **Kruskal-Wallis test**, `kruskal.test()`, compares the groups by rank, so it does not need normal data.

```
d <- data.frame(
  group = rep(c("A", "B", "C"), each = 6),
  value = c(3, 4, 5, 5, 6, 7,  6, 7, 8, 8, 9, 10,  4, 5, 6, 6, 7, 8)
)

shapiro.test(residuals(aov(value ~ group, data = d)))
oneway.test(value ~ group, data = d)
kruskal.test(value ~ group, data = d)
```

All three tests agree here: the groups differ ($p < 0.05$).

19.7 What makes it spatial data science?

The first assumption, independence, is where spatial data cause trouble. The observations in our example are places, and places are connected.

- **Nearby places are alike.** By the **first law of geography**, near things are more related than distant things. Neighboring counties share age profiles, so the 100 counties are not 100 independent pieces of evidence. Treating them as independent tends to make ANOVA's p-value look smaller than it should.
- **Boundaries shape the answer.** Drawing the three regions differently, or analyzing census tracts instead of counties, could change the result. That is the **modifiable areal unit problem**.
- **Spatial tools account for neighbors.** Moran's I (Chapter 22) tests whether a model's residuals cluster on the map, and spatial regression and permutation tests build neighbors into the model to give more honest p-values.

The bottom line: before you trust a p-value from map data, map the residuals and check them for spatial autocorrelation. Our ANOVA says median male age differs across North Carolina's regions ($F = 8.44$, $p < 0.001$), with the West oldest. Because the counties are neighbors on a map, treat that p-value with some caution.

19.8 Check your understanding

1. An ANOVA gives $p = 0.03$. This means: [1]

[1] If the groups truly didn't differ, a result this extreme would happen about 3% of the time

- (A) If the groups truly didn't differ, a result this extreme would happen about 3% of the time
- (B) There is a 3% chance the null hypothesis is true
- (C) The difference between the groups is large
- (D) 97% of the data support the hypothesis

2. The null hypothesis of an ANOVA is that: [2]

- (A) All the group means are equal
- (B) All the group means are different
- (C) At least one group mean differs
- (D) The groups have different variances

3. A statistically significant result is always a large or important one. TRUE / FALSE[3]
4. The F statistic is between-group variance divided by: [4]

- (A) Within-group variance
- (B) The overall mean
- (C) The number of groups
- (D) The p-value

5. An ANOVA compares 4 groups with 40 observations in all. How many degrees of freedom does the between-groups row have? _[5]
6. ANOVA gives $F = 8.44$ and $p = 0.0004$ for three regions. What can you conclude? [6]

- (A) All three regions differ from each other
- (B) The West is older than the East
- (C) Region causes differences in age
- (D) At least two regions differ in median male age

7. Which test compares every pair of groups after an ANOVA? [7]

- (A) Shapiro-Wilk
- (B) Tukey's HSD

[2] All the group means are equal
[3] FALSE
[4] Within-group variance
[5] 3
[6] At least two regions differ in median male age
[7] Tukey's HSD

- (C) Kruskal-Wallis
- (D) Moran's I

8. Neighboring counties being alike breaks ANOVA's assumption of: [8]

- (A) Normality
- (B) Equal variances
- (C) Independence
- (D) Large samples

19.9 Summary

- Data analysis is a cycle: ask, collect, describe, **test**, and communicate.
- A **statistic** summarizes a sample. A test starts from a **null hypothesis** of no difference, and the **p-value** is the chance of a result at least this extreme if the null hypothesis were true. By convention, $p < 0.05$ is **statistically significant**.
- p-values are easy to misuse. Report them with the size of the effect and a confidence interval, and don't treat 0.05 as a bright line.
- **Variance** is the average squared distance from the mean. **ANOVA** compares between-group with within-group variance: F = MS between ÷ MS within. A large F and small p mean at least one group mean differs.
- In R: `fit <- aov(value ~ group, data = d)`, then `summary(fit)`. Follow it with `TukeyHSD(fit)` to find which groups differ. Use `oneway.test()` or `kruskal.test()` if the assumptions fail.
- In North Carolina, median male age differs by region ($F = 8.44$, $p = 0.0004$), with the West oldest. Because neighboring counties are alike, check spatial data for spatial autocorrelation before trusting a p-value.

19.10 Exercises

1. **By hand.** Three groups have the values (2, 4, 6), (5, 7, 9), and (8, 10, 12). Calculate the group means, SS between, SS within, and F by hand, then check your answer with `aov()`.
2. **Which regions?** Run `TukeyHSD()` on the North Carolina regional model. Which pairs of regions differ significantly? Does that match what you see in Figure 19.3?
3. **Female age.** Download the median age for females (variable `B01002_003`) for North Carolina's counties with tidycensus, assign each county to the same regions, and run an ANOVA. Do the regions differ for women too?
4. **Draw the lines differently.** Move the boundary between the central and eastern regions one tier of counties east or west, and rerun the ANOVA. How much does F change? What does that tell you about the modifiable areal unit problem?
5. **Births.** Using the `nc` data from the sf package (Chapter 16), make a variable for births per square kilometer, group the counties into three regions by the longitude of their centroids, and test whether births per square kilometer differ by region. Check the assumptions, and use `kruskal.test()` if they fail.

[8] Independence

19.11 Further reading

- Wasserstein, R. L., and Lazar, N. A. (2016). “The ASA Statement on p-Values: Context, Process, and Purpose.” *The American Statistician* 70(2): 129–133. Six short principles for using p-values well.
- Amrhein, V., Greenland, S., and McShane, B. (2019). “Scientists Rise Up Against Statistical Significance.” *Nature* 567: 305–307.
- Open Science Collaboration (2015). “Estimating the Reproducibility of Psychological Science.” *Science* 349(6251): aac4716.
- *OpenIntro Statistics* (4th ed.), by David Diez, Mine Çetinkaya-Rundel, and Christopher Barr, free at https://www.openintro.org/book/os/. Chapter 7 covers ANOVA.

20 Linear Regression

Learning objectives

By the end of this chapter you should be able to:

1. Describe how data become information and then knowledge, and why spatial data need their own methods.
2. Tell a dependent variable from an independent variable, and correlation from regression.
3. Write a regression formula and fit a linear regression in R with `lm()`.
4. Read a model summary: coefficients, p-values, R^2, and adjusted R^2.
5. Build a multivariate model and compare it with a simpler one.
6. Map a model's residuals and explain what clusters of similar residuals mean.

Chapter 18 asked whether groups of places differ. This chapter asks a different question: **how does one variable change as another changes?** Do counties where more people work from home have higher house values? By how much? And how well does that explain the differences between counties?

The tool is **regression**. We'll start with where regression fits in turning data into knowledge, build it up on a small example you can run, and then follow a North Carolina project on median house values from research question to residual map.

20.1 From data to knowledge

Data science is using data to gain information or knowledge about the world. Data become knowledge in steps, each adding context:

Table 20.1: Each step adds context, ending in a tested explanation.

Step	What it is	North Carolina example
Data	Raw values without context	Median house value and the share of people working from home in each county
Information	Data organized and summarized	A map showing which counties have the highest house values
Knowledge	Information explained and tested	A regression showing that counties with more remote workers have higher house values

Regression is one of the main ways we take that last step: from a pattern on a map to a tested explanation.

20.1.1 Spatial data are special

Spatial data need their own methods, for four reasons:

- **Location is a core attribute, and location is messy:** projections, geocoding errors, and boundaries that change.
- **Nearby things are more alike.** This **spatial autocorrelation** breaks the **independence** assumption behind standard statistics, including the regression in this chapter.
- **Spatial data are visual,** so they call for maps and other visualizations.
- **Spatial data often change over time,** which adds a temporal side to the analysis.

Spatial data science draws on many fields to deal with these: environmental and sociodemographic data with location and time attached; remote sensing, geostatistics, spatial databases, and geographic information science; cloud platforms and APIs; and programming in R, Python, and SQL.

20.2 Regression explains how one variable changes with others

Regression analysis fits a model that estimates one variable from one or more others.

- The **dependent variable**, *y*, is the outcome you want to explain: here, median house value.
- The **independent variables**, *x*, are the variables you think explain it: the percent working from home, median income, median age.

A regression estimates three things about each independent variable:

- **Direction:** does *y* rise or fall as *x* rises?
- **Strength:** how much of the variation in *y* does *x* explain?
- **Size:** how much does *y* change for each one-unit change in *x*?

20.2.1 Correlation and regression

Correlation shows whether two variables move together. The **correlation coefficient**, *r*, runs from –1 (a perfect negative straight-line relationship) through 0 (none) to 1 (a perfect positive one) (Figure 20.1).

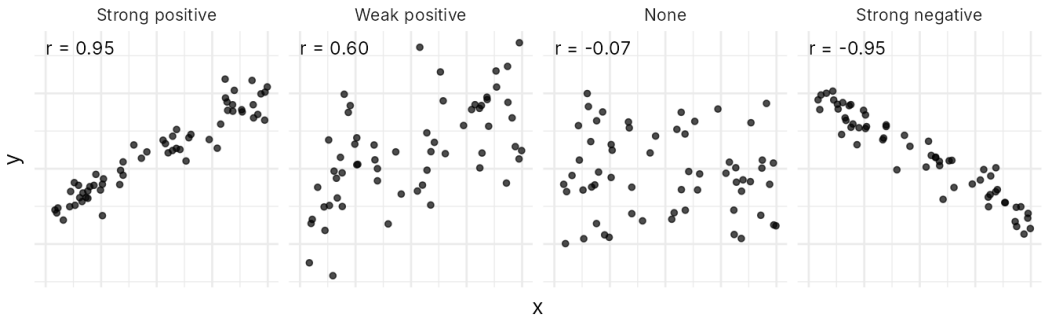


Figure 20.1: Four sets of points and their correlation coefficients.



Regression goes further. It estimates *how much y* changes for each unit of *x*, and it draws the **line of best fit**: the straight line that comes closest to the points, in the sense that the squared vertical distances from the points to the line add up to as little as possible. That's why it is called **ordinary least squares**.

The line has the familiar form

$$y = b_0 + b_1 x$$

where b_0 is the **intercept** (the value of *y* when *x* is 0) and b_1 is the **slope**: how much *y* changes when *x* goes up by one. The intercept and slope are the model's **coefficients**.

20.3 Writing a regression formula

In R, a regression formula puts the dependent variable on the left of the tilde, ~, and the independent variables on the right, joined by +:

```
dependent ~ independent1 + independent2 + ... + independentN
```

For example:

```
ice_cream_sales ~ temperature
weight ~ height + sex + cheeseburgers_per_day + steps_per_day + phone_number
```

The second formula makes a point: every variable needs a reason to be there. Height, sex, diet, and exercise all plausibly affect weight. A phone number has no plausible link at all, even if, by chance, it happened to correlate with weight in your data. Choose variables from what you know about the process, not by trying everything.

`lm()`, for **linear model**, fits the regression:

```
lm(median.house.value ~ works.from.home.percent, data = ncCounties)
```

20.3.1 A small example to try

An ice cream stand records its high temperature and the number of cones sold on 12 days:

```
temp  <- c(62, 65, 68, 70, 73, 75, 78, 80, 83, 85, 88, 91)
cones <- c(180, 205, 190, 230, 260, 245, 290, 300, 310, 345, 330, 380)

cor(temp, cones)          # correlation: do they move together?

model <- lm(cones ~ temp) # regression: by how much?
model
```

The correlation is 0.98: temperature and sales move together very closely. The regression says

$$\text{cones} = -245.7 + 6.77 \times \text{temperature}$$

so each extra degree goes with about 6.8 more cones sold. The intercept, −245.7, is the predicted sales at 0 °F. It's needed to place the line, but it means nothing on its own: no one buys ice cream at 0 °F, and the stand never opened on a day that cold.

Use the model to predict with `predict()`:

```
temp  <- c(62, 65, 68, 70, 73, 75, 78, 80, 83, 85, 88, 91)
cones <- c(180, 205, 190, 230, 260, 245, 290, 300, 310, 345, 330, 380)
model <- lm(cones ~ temp)

predict(model, data.frame(temp = c(77, 100)))
```

At 77 °F the model predicts about 275 cones. At 100 °F it predicts 431, but be careful: that is **extrapolation**, a prediction outside the range of the data (62 to 91 °F). The straight line may not hold there.

20.4 Anatomy of a regression

Figure 20.2 shows the parts of a fitted regression:

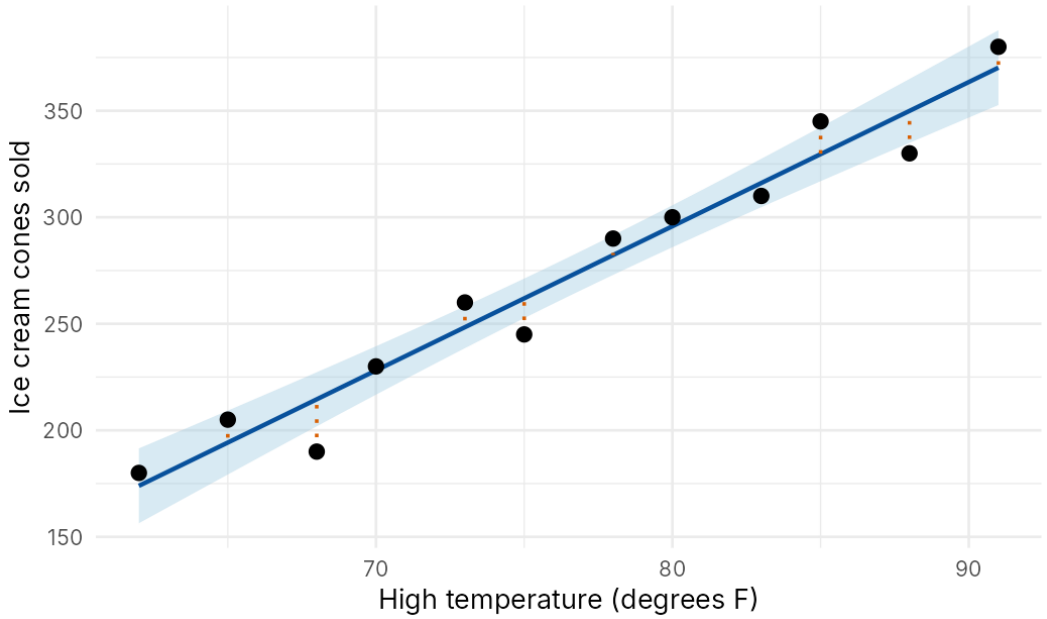


Figure 20.2: The ice cream regression. The blue line is the line of best fit and the shaded band its 95% confidence band. Dotted orange lines are the residuals: each day's distance from the line.

- **The line of best fit** is the best estimate of *y* for any value of *x*.
- **R^2** (R-squared) is the share of the variance in *y* that the model explains, from 0 to 1. Here $R^2 = 0.96$: temperature explains 96% of the day-to-day variation in sales. As a rough guide, below about 0.3 the two variables are barely connected.
- **The 95% confidence band** shows how uncertain the line itself is. It's narrowest near the middle of the data and widens toward the ends.
- **Residuals** are the vertical distances from each point to the line: observed minus predicted. Points far from the line are **outliers** worth understanding, not deleting. On the 68 °F day, the stand sold about 25 fewer cones than predicted. Was it raining?



```
temp  <- c(62, 65, 68, 70, 73, 75, 78, 80, 83, 85, 88, 91)
cones <- c(180, 205, 190, 230, 260, 245, 290, 300, 310, 345, 330, 380)
model <- lm(cones ~ temp)

round(residuals(model), 1)
```

20.5 Reading the model summary

`summary()` reports the statistical results:

```
temp  <- c(62, 65, 68, 70, 73, 75, 78, 80, 83, 85, 88, 91)
cones <- c(180, 205, 190, 230, 260, 245, 290, 300, 310, 345, 330, 380)
model <- lm(cones ~ temp)

summary(model)
```

Work down the output:

- **Call:** the model you fitted.
- **Residuals:** a five-number summary of the residuals. Roughly symmetric around 0 is good.
- **Coefficients:** one row per coefficient.
  - **Estimate:** the intercept, and the slope for each independent variable. Here 6.77 cones per degree.
  - **Std. Error:** how uncertain each estimate is.
  - **t value:** the estimate divided by its standard error.
  - **Pr(>|t|):** the p-value for the null hypothesis that the true coefficient is 0, that is, that x has no relationship with y. Stars flag small p-values: `***` means $p < 0.001$. For temperature, $p = 4.5 \times 10^{-8}$ (R writes `4.51e-08`), so the relationship is very unlikely to be chance.
- **Residual standard error:** the typical size of a residual, about 14 cones.
- **Multiple R-squared** and **Adjusted R-squared:** the share of variation explained. **Adjusted R^2** lowers R^2 a little for each independent variable in the model, so that adding useless variables can't make a model look better. Use it to compare models with different numbers of variables.
- **F-statistic:** a test of whether the model as a whole explains anything, with its p-value.

20.6 Example: house values in North Carolina

Now a real project, written up as an R Markdown report. Its research question: **what demographic variables may influence median house value in North Carolina counties in 2020?** In particular, is the share of people working from home related to house values, and by how much?

The data are Census Bureau estimates for North Carolina's 100 counties for 2020, loaded as an sf object called `ncCounties`. (Names ending in `E`, such as `median.house.valueE`, are estimates from the American Community Survey, as tidycensus names them.) Before modeling, map each independent variable (Figure 20.3):

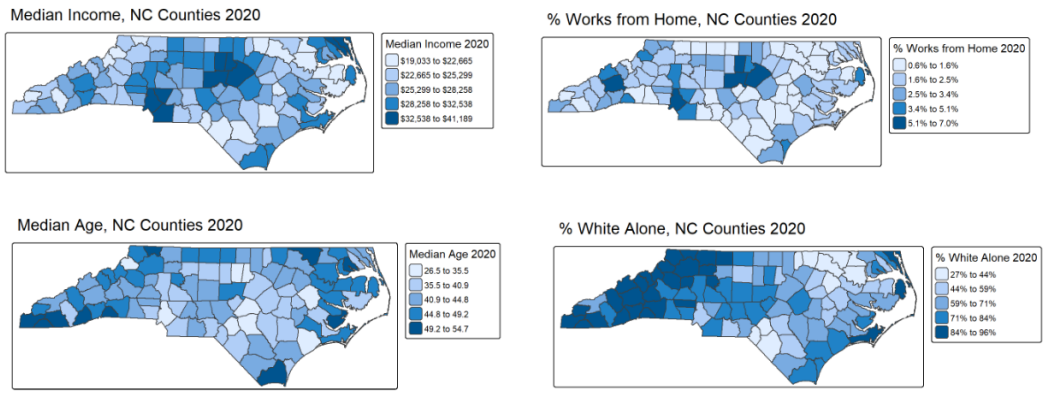


Figure 20.3: The independent variables: median income, the percent working from home, median age, and the percent who are white alone, for North Carolina's counties in 2020.

Higher incomes and more remote work stand out around the Triangle and Charlotte. Older populations are in the mountains and along parts of the coast.

Think about it

Why might counties where more people work from home have higher house values? Think of at least two explanations. Does working from home push prices up, or do the jobs that allow remote work simply pay more?

20.6.1 The first model

```
first.model <- lm(median.house.valueE ~ works.from.home.percent, data = ncCounties)
summary(first.model)
```

```
Call:
lm(formula = median.house.valueE ~ works.from.home.percent, data = ncCounties)

Residuals:
   Min     1Q Median     3Q    Max
-60097 -19670  -5395  13763 102485

Coefficients:
                        Estimate Std. Error t value Pr(>|t|)
(Intercept)                73974       5839   12.67   <2e-
16 ***
works.from.home.percent    36564       2185   16.73   <2e-
16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 29220 on 98 degrees of freedom
Multiple R-squared:  0.7408,    Adjusted R-squared:  0.7381
F-statistic:   280 on 1 and 98 DF,  p-value: < 2.2e-16
```

Read the three key numbers:



- **The coefficient.** Each additional percentage point of people working from home goes with a median house value about **$36,564** higher.
- **The p-value.** Below 2×10^{-16}, effectively zero. Three stars: $p < 0.001$.
- **Adjusted R^2.** 0.7381: the share working from home alone accounts for about 74% of the variation in median house value between counties.

20.6.2 Adding variables: a multivariate model

A model with more than one independent variable is a **multivariate** (or multiple) regression. The second model adds median age, median income, and the percent white:

```
second.model <- lm(median.house.valueE ~ works.from.home.percent +
                     median.ageE + median.incomeE + white.percent,
                   data = ncCounties)
summary(second.model)
```

```
Coefficients:
                          Estimate Std. Error t value Pr(>|t|)
(Intercept)             -7.949e+04  2.596e+04  -3.062  0.00286 **
works.from.home.percent  2.434e+04  2.324e+03  10.476  < 2e-
16 ***
median.ageE              1.839e+02  4.794e+02   0.384  0.70218
median.incomeE           4.313e+00  7.233e-01   5.964 4.19e-
08 ***
white.percent            8.154e+02  1.432e+02   5.695 1.37e-
07 ***
---
Residual standard error: 21620 on 95 degrees of freedom
Multiple R-squared:  0.8624,    Adjusted R-squared:  0.8567
F-statistic: 148.9 on 4 and 95 DF,  p-value: < 2.2e-16
```

R writes large and small numbers in scientific notation: `2.434e+04` means 2.434×10^{4}, or 24,340. What changed?

- **Adjusted R^2 rose from 0.74 to 0.86.** Together the four variables explain about 86% of the variation.
- **Median age is not significant** ($p = 0.70$). Once the other variables are in the model, age adds little.
- **The other three are significant.** Holding the others constant, each extra dollar of median income goes with $4.31 more house value (so $4,313 per $1,000 of income), and each extra percentage point white goes with about $815.
- **The work-from-home coefficient shrank,** from $36,564 to about $24,340. Part of what the first model credited to remote work really went with income: counties where more people work from home also tend to have higher incomes. A coefficient in a multivariate model is the change in y for one unit of x *with the other variables held constant*.



> ⚠ **Correlation is not causation**
>
> Even a strong, significant coefficient doesn't show that *x* causes *y*. Here it isn't clear which way the relationship runs: working from home could drive up house prices, or people whose jobs allow remote work may earn more and so buy more expensive houses. A good write-up says so.

20.7 Mapping residuals

A regression isn't finished until you check its residuals, and with spatial data that means **mapping** them. A residual is a county's observed house value minus the model's prediction. Residuals near 0 mean the model fits that county well; large positive or negative residuals mean a poor fit.

```
ncCounties$resids <- second.model$residuals

tm_shape(ncCounties) +
  tm_polygons(fill = "resids",
              fill.scale = tm_scale_intervals(midpoint = 0),
              fill.legend = tm_legend("Residuals 2020",
                                      position = tm_pos_out("right"))) +
  tm_title("Model Residuals, NC Counties 2020")
```

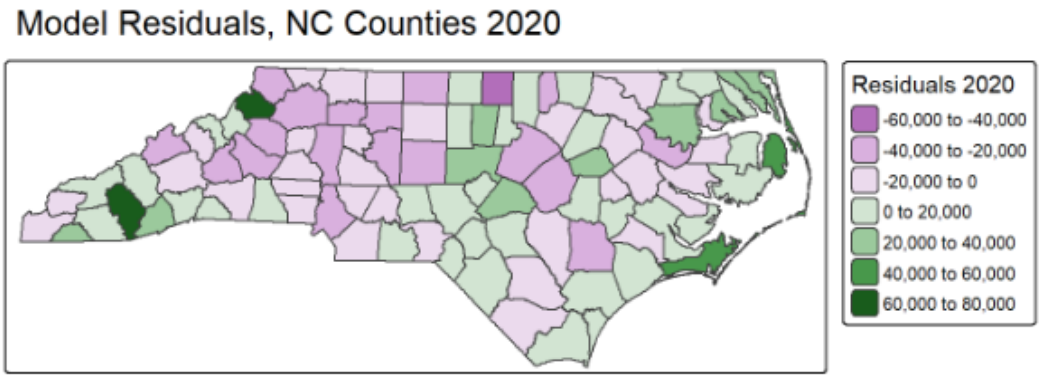


Figure 20.4: Residuals from the second model, by county. Purple counties have lower house values than the model predicts; green counties have higher values.

Look for **clusters**. If counties with large positive residuals sit next to each other, the model is missing something spatial: here, perhaps second homes and vacation markets in the mountains and on the beach, which raise house values beyond what local incomes predict. Clustered residuals also mean the observations aren't independent, so the model's standard errors and p-values are too small.

The fix is a cycle:

1. **Fit the model.**
2. **Map the residuals.**
3. **Test them with Moran's I** (Chapter 22), which measures whether the residuals cluster more than chance would produce.

4. **Revise the model:** add the missing variables, or move to a spatial regression model, such as a **spatial lag model** or a **spatial error model**, that builds neighbors into the model.

Repeat steps 2 to 4 until the residuals no longer cluster.

20.8 Check your understanding

1. In `lm(house_value ~ income, data = d)`, the dependent variable is: [1]

- (A) house_value
- (B) income
- (C) d
- (D) lm

2. Correlation tells you how much *y* changes for each unit of *x*. TRUE / FALSE[2]
3. A slope of 36,564 for `works.from.home.percent` means each extra percentage point working from home goes with a median house value that is: [3]

- (A) $36,564 higher
- (B) $36,564 lower
- (C) 36,564% higher
- (D) 36,564 times higher

4. An adjusted R^2 of 0.74 means the model explains about what percent of the variation in *y*? __[4]
5. Which number should you use to compare models with different numbers of independent variables? [5]

- (A) Multiple R-squared
- (B) Adjusted R-squared
- (C) The intercept
- (D) The residual standard error

6. In the second model, median age has $p = 0.70$. This means: [6]

- (A) Median age adds little once the other variables are in the model

[1] house_value
[2] FALSE
[3] $36,564 higher
[4] 74, 74%
[5] Adjusted R-squared
[6] Median age adds little once the other variables are in the model

- (B) Median age explains 70% of house value
- (C) Median age is the most important variable
- (D) The model is wrong

7. A residual is: [7]

- (A) The observed value minus the predicted value
- (B) The predicted value minus the mean
- (C) The slope of the line
- (D) The p-value of a coefficient

8. Clusters of similar residuals on a map suggest the model is missing something spatial. TRUE / FALSE[8]

20.9 Summary

- Data become **information** when organized and **knowledge** when explained and tested. Regression is a main tool for the last step.
- Spatial data need their own methods: location is messy, nearby things are alike (**spatial autocorrelation**), and the data are visual and change over time.
- **Regression** models a **dependent variable** (*y*) from one or more **independent variables** (*x*). **Correlation** says whether two variables move together; regression says by how much, and draws the **line of best fit**.
- In R: `lm(y ~ x1 + x2, data = d)`, then `summary()`. Read each **coefficient**, its p-value, and the **adjusted R^2**. Every variable needs a reason to be in the model.
- In North Carolina in 2020, each percentage point working from home went with about $36,564 higher median house value (adjusted R^2 = 0.74). Adding income and percent white raised adjusted R^2 to 0.86 and shrank that coefficient to about $24,340.
- Regression shows association, not causation.
- **Map the residuals.** Clusters of similar residuals mean the model misses something spatial; test them with Moran's I, then revise the model.

20.10 Exercises

1. **Ice cream by hand.** Using the ice cream model, predict sales at 70 °F and at 85 °F from the equation, then check with `predict()`. Which of the 12 days has the largest positive residual?
2. **Fahrenheit to Celsius.** Convert `temp` to degrees Celsius and refit the model. What happens to the slope, the intercept, and R^2? Why?
3. **Your own variables.** Download median house value (`B25077_001`), median household income (`B19013_001`), and one variable of your choice for North Carolina's counties with tidycensus (Chapter 13). Fit a model with income, then with both variables. Did adjusted R^2 go up?

[7] The observed value minus the predicted value
[8] TRUE



4. **Map the residuals.** Map the residuals from your model in Exercise 3. Where does the model over-predict and under-predict? Do the residuals look clustered?
5. **Births.** With the sf package's `nc` data, regress `BIR79` on `BIR74`. Interpret the slope and R^2. Why is this a less interesting model than it looks?

20.11 Further reading

- *OpenIntro Statistics* (4th ed.), by David Diez, Mine Çetinkaya-Rundel, and Christopher Barr, free at https://www.openintro.org/book/os/. Chapters 8 and 9 cover linear and multiple regression.
- *Geocomputation with R* (2nd ed.), by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, free at https://r.geocompx.org. Its chapter on statistical learning applies models to spatial data.
- *Spatial Regression Models* (2nd ed.), by Michael D. Ward and Kristian Skrede Gleditsch (SAGE, 2018), introduces the spatial lag and spatial error models.

21 Writing a Spatial Data Science Project

Learning objectives

By the end of this chapter you should be able to:

1. Tell data from information, and place both in the DIKW pyramid.
2. Describe the data science workflow and where data come from.
3. Explain reproducibility, replicability, and open science.
4. Explain why spatial data need their own methods, and what to check because of it.
5. Write each section of a spatial data science project, from research question to references.
6. Explain what makes a project spatial data science, and not just data science with a map.

You have now learned to get data, tidy it, map it, describe it, and test it. This chapter is about putting those pieces together into a project that someone else can read, trust, and rerun. We start with the ideas underneath: what data, information, and knowledge are, where data come from, and what makes an analysis science. Then we walk through a real project, on house values in North Carolina, section by section.

21.1 No information without data

The writer and programmer Daniel Keys Moran put the first idea briefly: you can have data without information, but you can't have information without data. Raw data become information only after they are processed, organized, and interpreted.

Table 21.1: Data and information.

	Data	Information
What it is	Raw and unprocessed	Processed, organized data with meaning
What it's for	The basis for analysis	Supporting decisions and giving insight
Value	Little on its own	Directly valuable and actionable
Examples	Numbers, dates, measurements, text	Reports, summaries, insights, conclusions

A list of hourly temperatures is data. A forecast that says "bring an umbrella this afternoon" is information.

21.1.1 The DIKW pyramid

The **DIKW pyramid** takes the idea further, in four levels (Figure 21.1). Each builds on the one below:

- **Data:** facts and figures.
- **Information:** data organized to be useful.
- **Knowledge:** information built up and tested over time and across many situations.
- **Wisdom:** principles, reasoning, and knowledge applied to judge what is true and right.

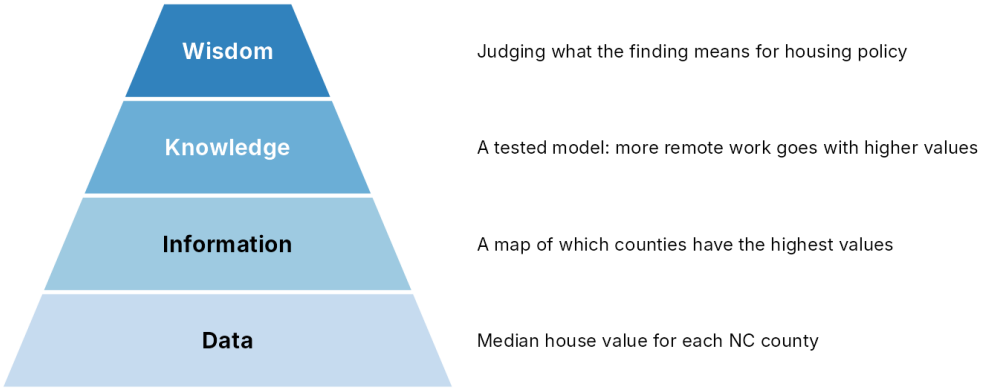


Figure 21.1: The DIKW pyramid, with an example of each level from this chapter's project on North Carolina house values.

Data science is using data to gain information or knowledge about the world. It mostly works on the bottom levels of the pyramid, and helps people climb toward the top.

21.2 The data science process

Chapter 6 introduced the **data science workflow** from *R for Data Science*: **import** the data, **tidy** it, then cycle through **transform**, **visualize**, and **model** until you **understand** it, and finally **communicate** the results. The middle loop repeats, and programming runs through every step. A project write-up is the "communicate" step, but a good one shows all the others too.

21.2.1 Big data

Big data is often described by three Vs:

- **Volume:** the sheer amount, from terabytes of records, transactions, and files up.
- **Velocity:** how fast it arrives, from batches to near-real-time feeds and live streams.
- **Variety:** the mix of structured tables, semi-structured JSON, and unstructured text and images.

Many spatial datasets, such as satellite imagery or the GPS traces from phones, score high on all three.

21.2.2 Where data come from

Table 21.2: Where data come from, and the formats they arrive in.

Sources	Formats
Authoritative: federal, state, county, and local government	Files: GeoJSON, shapefiles, CSV
Web scraping: social media, live feeds, news	API calls: data on demand (Chapters 13 and 14)
The Internet of Things: sensors streaming readings	Images, text, and 3D formats
Real-time analytics feeds	

Government agencies are usually the most **authoritative** sources. Also note whether a dataset is **static**, a fixed snapshot, or **dynamic**, continually updated, because a dynamic source may give different data the next time you run your code.

21.3 Science: reproducibility and replicability

Spatial data science is still science. It builds on a body of knowledge, follows the scientific method, and rests on empirical evidence. Two words describe whether others can trust and build on a result:

- **Reproducible:** the same data and methods give the same results. Anyone can rerun your analysis and get your numbers.
- **Replicable:** a new study, with different data, answering the same question, gets consistent results.

Open science makes both possible: open access (papers anyone can read), open data, open methods, open-source software such as R and RStudio, open peer review, and open educational resources (like this book).

R Markdown, from Chapter 2, is a reproducibility tool. It keeps data loading, code, results, and write-up in one document, so anyone with the file and the data can knit it and get the same report.

21.4 Why spatial data are special

Location links observations together, so spatial data need their own methods.

What makes them special:

- **The first law of geography:** near things are more related than distant things.
- **Spatial autocorrelation:** values cluster, so observations are not independent. Neighboring counties tend to have similar house values.
- **Spatial heterogeneity:** relationships change from place to place. Remote work may matter more for house values near cities than in rural counties.

What it means for your analysis:

- **Clustered residuals make standard tests overconfident.** Map your model's residuals and test them with Moran's I (Chapter 22).



- **Scale and zoning matter.** County and tract results can differ: the **modifiable areal unit problem**.
- **Projection matters.** Distances and areas depend on the coordinate system (Chapter 12).

21.5 Building blocks of a project

Every project has the same building blocks:

Table 21.3: The building blocks of a spatial data science project.

Section	What goes in it
Introduction	Your research question and why it matters
Background	What earlier work found, and how it relates to your question
Data and methods	Your data, cited and described; the code that tidies them; maps and descriptive statistics
Results	Maps, graphs, and statistics that help answer the question
Conclusion	Your answer, as well as you can give it; limitations; what you'd do differently
References	Every source you used

We'll go through each one using an example project, written in R Markdown, titled *What Demographic Variables May Influence Median House Value for North Carolina in 2020?* You met its models in Chapter 19.

21.5.1 Introduction: the research question

The research question drives everything else. A good one is:

- **Specific and focused.** Include the place and time where they matter.
- **Answerable,** even if you end up not able to answer it fully.
- **Worth asking.** Explain why it matters.

It doesn't have to be phrased as a question, but that often helps.

The example's introduction says that the project examines which demographic factors might relate to median home value in North Carolina's counties in 2020: in particular, whether the percentage of people working from home is related to home value, and by how much it raises or lowers it. It names the other variables it will bring in (percent white alone, median age, and median income) and says the project will explore the data, build a model, and discuss how well the model fits, statistically and spatially, using residuals. A reader knows the place, the time, the variables, and the plan by the end of the second paragraph.

> **Think about it**
>
> Which is the better research question, and why? (a) "What affects house prices?" (b) "Is the share of residents who work from home related to median house value in North Carolina's counties in 2020?" How would you make (b) more spatial?



21.5.2 Background: build on earlier research

The background shows you've read what came before. What have others found? How does it shape your question, your data, or your methods?

The example's background notes that house prices and cost of living have clear geographic patterns, but are not a common topic for geographers. It summarizes a study by Campbell and James that built a housing cost index for North Carolina's counties, the ratio of median home value to median household income, and tested whether second homes in the mountains and at the beach push costs up. That earlier work suggests variables to include and a spatial pattern to look for.

21.5.3 Data and methods: show your work

In data and methods, you show your work:

1. **Cite the source** of your data: which agency, which survey, which year, and how you got it.
2. **Include the code** that loads and tidies the data.
3. **Describe the data** with summary statistics and maps.

```
# Load the libraries used in this research
library(tidyverse)
library(sf)
library(tmap)

# Load the 2020 NC counties data
ncCounties <- read_sf("...")
glimpse(ncCounties)
```

`glimpse()` shows 100 rows, one per county, and the columns. Then a summary and a map of the dependent variable, median home value, and a map of each independent variable (Figure 20.3). Mapping each variable before modeling it is how you notice patterns, such as higher incomes and more remote work around the Triangle and Charlotte, and problems, such as a county with a missing value.

21.5.4 Results: model your question

A **model** is a simplified representation of a real-world process. The results section fits models that test your question, and reports them with maps, graphs, and statistics.

The example's first model tests whether the share of people working from home is related to median house value. Each additional percentage point working from home goes with about $36,564 higher median house value, with an adjusted R^2 of 0.74. A second model adds median income, median age, and percent white, and raises the adjusted R^2 to 0.86 (Chapter 19).

21.5.5 Check the model: map the residuals

A spatial results section doesn't end with the model summary. Check the model by mapping its **residuals**, each county's observed value minus the model's prediction:

1. **Fit the model.**



2. **Map the residuals.** Where does the model over- or under-predict?
3. **Test them with Moran's I** (Chapter 22). Do the residuals cluster more than chance would produce?
4. **Revise the model.** If they cluster, add explanatory variables or fit a spatial lag or spatial error model.

Repeat steps 2 to 4 until the residuals no longer cluster.

21.5.6 A second method: comparing regions with ANOVA

Regression asks how one variable changes with another. **ANOVA** (Chapter 18) asks whether a variable's mean differs across three or more groups. In a project on North Carolina, the groups can be regions, such as the mountains, the Piedmont, and the coastal plain, and the variable median male age by county. `aov(age ~ region)` tests whether the regions differ, and `TukeyHSD()` shows which ones. Because the groups are places, map them, so the reader can see the regional pattern.

21.5.7 Conclusion: answer the question

The conclusion should:

- **Answer your question,** or explain honestly why you couldn't.
- **Discuss limitations.** What could make the results misleading?
- **Say what you'd do differently** next time, and what future work should try.

The example's conclusion does all three. It answers the question: three variables (percent working from home, percent white, and median income) are related to median house value, and each extra percentage point working from home adds potentially over $20,000. It is honest about the spatial fit: the residuals vary a lot across the state, with some mountain, coastal, and rural counties not following the model, which suggests missing variables. It names limitations: medians hide the range within each county, and the direction of the relationship is unclear (does remote work raise prices, or do people who can work from home simply earn more?). And it proposes future work: smaller units such as census tracts, more variables such as job type and education, and comparisons over time or with similar states.

21.5.8 References

List every source you used, in a consistent style, and cite them in the text. That includes data sources, such as "U.S. Census Bureau, American Community Survey 5-year estimates", and R packages, which you can cite with `citation("tidycensus")`.

21.6 So, what is spatial data science?

Spatial data science combines three things:

- **spatial data,**
- **maps and visualizations,** and
- **spatial analysis.**



Done well, it is also **open, reproducible, and replicable**, and it is often built on big data.

One caution. Using spatial data doesn't by itself make a project spatial data science. A regression on county data that never maps or tests its residuals is ordinary statistics that happens to use places. You also need spatial analysis: a question with spatial aspects, such as the regional ANOVA, the residual map, or a test for spatial autocorrelation.

21.7 Check your understanding

1. A list of every county's median house value, with no summary or map, is: [1]

- (A) Data
- (B) Information
- (C) Knowledge
- (D) Wisdom

2. Which level sits at the top of the DIKW pyramid? _____ [2]
3. Running the same code on the same data and getting the same results is called: [3]

- (A) Reproducibility
- (B) Replicability
- (C) Velocity
- (D) Spatial heterogeneity

4. The three Vs of big data are volume, velocity, and: [4]

- (A) Variety
- (B) Validity
- (C) Value
- (D) Visualization

5. Which section of a project states the research question? [5]

- (A) Introduction
- (B) Background
- (C) Results
- (D) References

[1] Data
[2] Wisdom
[3] Reproducibility
[4] Variety
[5] Introduction

6. A research question should always be phrased as a question. TRUE / FALSE[6]
7. In which section do you discuss what you would do differently next time? [7]

- (A) Data and methods
- (B) Results
- (C) Conclusion
- (D) Background

8. Any project that uses spatial data is spatial data science. TRUE / FALSE[8]

21.8 Summary

- **Data** are raw values; **information** is data organized and interpreted. The **DIKW pyramid** adds **knowledge** and **wisdom** on top.
- The **data science workflow** is import, tidy, a loop of transform, visualize, and model, then communicate. **Big data** is described by volume, velocity, and variety.
- Data come from authoritative agencies, web scraping, sensors, and feeds, as files or through APIs, and can be static or dynamic.
- Science should be **reproducible** (same data and methods, same results) and **replicable** (new data, consistent results). Open science and R Markdown make that easier.
- Spatial data are special because of the first law of geography, **spatial autocorrelation**, and **spatial heterogeneity**. Map and test residuals, and watch scale, zoning, and projection.
- A project has an **introduction** (the research question), **background**, **data and methods**, **results**, a **conclusion**, and **references**.
- Spatial data science combines spatial data, maps, and spatial analysis. Spatial data alone aren't enough.

21.9 Exercises

1. **Climb the pyramid.** Pick a dataset you've used this semester. Give one example of it at each level of the DIKW pyramid.
2. **Sharpen a question.** Rewrite "Is pollution bad for health?" as a specific, answerable, spatial research question about North Carolina. Name the place, time, units of analysis, and variables.
3. **Find the background.** Search Google Scholar for one published study related to your question from Exercise 2. Write two sentences on what it found and how it shapes your own question or methods.
4. **Data and methods.** For your question, list each dataset you'd need, who publishes it, whether it's static or dynamic, and how you'd get it into R.
5. **Peer review.** Swap project drafts with a classmate. For each building block in Table 21.3, write one thing that works and one thing to improve. Is their project spatial data science, by the definition in this chapter?

[6] FALSE
[7] Conclusion
[8] FALSE



21.10 Further reading

- *R for Data Science* (2nd ed.), by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund, free at https://r4ds.hadley.nz. The introduction describes the data science workflow, and the "Quarto" chapters cover writing reproducible reports.
- National Academies of Sciences, Engineering, and Medicine (2019). *Reproducibility and Replicability in Science*. National Academies Press. Free at https://doi.org/10.17226/25303. The source of the definitions used in this chapter.
- Campbell, H. S., Jr., and James, R. D. (2021). "Place, Space, and Cost of Living in North Carolina: A Research Note." *Southeastern Geographer* 61(1): 81–104. The study the example project builds on.

22 Distance and Neighborhoods

> **Learning objectives**
>
> By the end of this chapter you should be able to:
>
> 1. Tell physical from social distance, and explain distance decay.
> 2. Calculate Euclidean, Manhattan, and great circle distances, and say when to use network distance or travel time instead.
> 3. Explain why spatial statistics need a definition of neighbors.
> 4. Define neighbors by a distance band, by inverse distance, by the k nearest features, and by rook or queen contiguity.
> 5. Read a neighborhood weight matrix, and row-standardize it.
> 6. Build neighbor lists and spatial weights in R with spdep.

How far is Chapel Hill from Charlotte? About 179 km in a straight line, but more like 230 km by road, and two and a half hours by car, longer at rush hour. Which answer is right depends on your question.

This chapter has two parts. The first is about **distance**: what it means, why it matters, and the main ways to measure it. The second uses distance, and shared borders, to decide which features are **neighbors**, and stores those relationships in a **neighborhood weight matrix**. That matrix is the input to nearly every spatial statistic in the rest of this book, starting with spatial autocorrelation in Chapter 22.

22.1 Distance is a fundamental geographic concept

Distance shapes the relationships and interactions among people, things, and places: where you shop, who you know, which clinic you can reach. Geographers work with two ideas of distance:

- **Physical distance** is the separation measured across the Earth's surface, in meters, kilometers, or miles.
- **Social distance** is separation measured by connection, similarity, or interaction: how connected or isolated people and groups are. It shapes social networks, who interacts with whom, and how information or disease moves through a population.

The two often work together. Residential segregation is the classic case: two neighborhoods can sit side by side on a map yet be far apart socially.

22.1.1 Distance decay

Distance decay is one of the central regularities in geography: interaction between places, actual or potential, decreases as the distance between them increases. Distance has "friction". Overcoming it costs time, money, or effort, so people choose closer options.

- Fewer customers visit a store the farther away they live.
- Migration between two cities shrinks as the distance between them grows.
- Clinic use drops as travel time rises.

Distance decay is often modeled as an inverse power of distance, such as $1/d$ or $1/d^2$, or as an exponential function. Keep that idea in mind: we'll use the same form for continuous neighbor weights later in the chapter.

22.1.2 Why measure distance?

We use distance in three ways:

1. **Explain it:** why are some places farther from services than others?
2. **Measure it** as an input to other calculations, such as accessibility scores or neighbor weights.
3. **Map it:** distance surfaces, service areas, and **isochrones**, lines of equal travel time, such as the area within a 15-minute drive of a hospital.

In GIS, most work with distance falls into three operations:

Table 22.1: Distance-based operations in GIS.

Operation	What it does	Example	In R
Spatial query	Selects features by distance	All schools within 1 km of a highway	`sf::st_is_within_distance()`
Buffer	Turns a distance into an area around a feature	A 500 m zone around each transit stop	`sf::st_buffer()`
Identify neighbors	Finds the features within each feature's neighborhood	The census tracts that border each tract	`spdep::poly2nb()`

The question you are asking decides which distance measure fits.

22.2 Measuring distance

22.2.1 Euclidean distance

Euclidean distance is straight-line distance, from the Pythagorean theorem. The differences in x and y between two points are the legs of a right triangle, and the distance is the hypotenuse:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



It is a simple calculation that doesn't need a GIS, but it assumes a flat surface with the same units in x and y. That means **projected coordinates**. Never use it on latitude and longitude: a degree of longitude is about 111 km at the Equator but shrinks to nothing at the poles (Chapter 12).

```
euclid <- function(x1, y1, x2, y2) {
  sqrt((x2 - x1)^2 + (y2 - y1)^2)
}

euclid(1, 2, 4, 6)   # a 3-4-5 triangle
```

Euclidean distance has three limitations:

- **It assumes a flat Earth.** The error grows with distance. Over hundreds of kilometers, use great circle distance.
- **It assumes flat terrain.** It underestimates the real separation in the mountains.
- **It assumes no barriers.** Rivers, fences, buildings, and missing roads all block straight-line travel. A driver can't cross a lake just because it's the shortest path.

The next measures each relax one of these assumptions.

22.2.2 Great circle distance

Great circle distance is the shortest distance between two points on a sphere, measured along its surface, like the route a long-haul flight takes. Use it for points stored in longitude and latitude, and always when points are far apart. With longitudes λ and latitudes φ, in radians:

$$D = R \cdot \arccos\left[\sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos(\lambda_1 - \lambda_2)\right]$$

where *R* is the Earth's radius, about 6,371 km on average. (The **haversine formula** gives the same answer and is more accurate for very short distances.)

```
greatCircle <- function(lon1, lat1, lon2, lat2, R = 6371) {
  rad <- pi / 180   # degrees to radians
  R * acos(sin(lat1 * rad) * sin(lat2 * rad) +
           cos(lat1 * rad) * cos(lat2 * rad) * cos((lon1 - lon2) * rad))
}

greatCircle(-79.0558, 35.9132, -80.8431, 35.2271)   # Chapel Hill to Charlotte, km
greatCircle(-79.0558, 35.9132, -0.1276, 51.5072)    # Chapel Hill to London, km
```

Chapel Hill to Charlotte is about 179 km, and Chapel Hill to London about 6,252 km. You rarely need to write this yourself: `sf::st_distance()` computes great circle distances automatically when your data are in longitude and latitude, and Euclidean distances when they are projected.

22.2.3 Manhattan distance

Manhattan distance, or city-block distance, accounts for barriers by restricting travel to routes that run north-south and east-west, as on Manhattan's grid of streets. Instead of cutting diagonally, it adds the horizontal and vertical differences:

$$d = |x_2 - x_1| + |y_2 - y_1|$$

```
manhattan <- function(x1, y1, x2, y2) {
  abs(x2 - x1) + abs(y2 - y1)
}

manhattan(1, 2, 4, 6)   # 3 blocks east + 4 blocks north
```

Manhattan distance is never shorter than the Euclidean distance between the same points. For a purely diagonal trip it is about 1.41 times longer (√2). It works poorly where streets don't follow a grid.

22.2.4 Network distance and travel time

Network distance restricts travel to the lines, or arcs, of a network, such as streets, rail lines, or rivers. The route is a chain of arcs, and the distance is the sum of their lengths. Real networks add rules: one-way streets, turn restrictions, and speed limits. Esri's Network Analyst and routing services calculate it, and so do R packages such as **sfnetworks** and **dodgr**.

Travel time goes one step further and measures separation in minutes instead of meters, which is closer to how people experience distance. It needs a network with speeds, or a travel-cost surface for off-road travel, and road speeds, turns, and traffic all matter. Calculating it for many origin-destination pairs can take a lot of computing. Most U.S. studies assume everyone travels by car, which overstates access for people who rely on transit or walking; better studies add other modes.

22.2.5 Straight lines or networks?

People usually move along networks, so true separation shouldn't be measured as a straight line. But Euclidean and network distances are often highly correlated, and where they are, Euclidean distance is a reasonable and much cheaper stand-in. The correlation is weaker in rural areas, near water, and where networks are sparse. Before you choose, compute both for a sample of your data and check how closely they agree.

> **Think about it**
>
> You want to measure each North Carolina county's access to hospitals. Which distance measure would you use for a county in the Triangle? For a county on the Outer Banks, where the nearest hospital may be across a sound? Why?

22.3 Neighborhoods and neighbors

22.3.1 Comparing separation between features

There are two basic ways to say how far apart features are, and both become ways to define neighbors:

- **Absolute distance:** the actual distance separating them. Example: everything within 2 km.



- **Relative distance:** separation compared with other features. Examples: the nearest feature, the nearest *k* features, or features that touch (**topology**).

22.3.2 Why neighbors matter

The **first law of geography** says that near things are more related than distant things. Values at nearby locations tend to be similar, and similarity decreases with distance. This dependence is **spatial autocorrelation**. To measure it, we first have to say which locations are "near": we need neighbors.

A **neighborhood**, in spatial analysis, is the zone of influence around a feature: an idea of nearness (things close to one another) or of connectedness (things that affect one another). **Neighbors** are the features inside a feature's neighborhood.

Three details matter:

- **Every observation gets its own set of neighbors.**
- **Neighbors need not be symmetric.** With nearest neighbors, A can be B's neighbor without B being A's, if B has another feature even closer.
- **Some observations may have no neighbors at all.** An island county, or an isolated point, needs special handling. In spdep, set `zero.policy = TRUE`.

Nearly every spatial method needs a neighbor definition: global spatial autocorrelation (Moran's I, Geary's C), local indicators (local Moran's I, Getis-Ord Gi*), spatial regression (spatial lag and spatial error models), and spatial smoothing and interpolation. In every case, the neighbor definition is a choice you make, and different definitions give different neighbors and different results.

22.3.3 Binary or continuous neighbors

There are two ways to record a neighbor relationship:

- **Binary** (yes or no): a feature either is a neighbor or isn't, stored as 1 or 0. Binary weights are simpler to interpret and are the default in most software.
- **Continuous:** an amount of "neighborliness", usually based on distance, so closer features get larger weights. Some neighbors are strong and others weak. Continuous weights are distance decay applied to neighbors.

22.4 Three ways to define neighbors

Table 22.2: Three ways to define neighbors.

Approach	Neighbors are	In spdep
Absolute distance	Features within a set distance	`dnearneigh()`
Relative distance: nearest features	The nearest one, or the nearest *k*	`knearneigh()` and `knn2nb()`
Relative distance: topology	Polygons that share a border	`poly2nb()`, with `queen = TRUE` or FALSE

22.4.1 Distance band neighbors

With a **distance band**, features are neighbors if they fall within a chosen threshold distance. For points, measure between the points. For polygons, measure between their centroids, which can mislead: two large rural counties may share a long border yet have centroids far apart.

The threshold is the key decision:

- **Too small,** and many features have no neighbors.
- **Too large,** and almost everything is a neighbor.
- **A common start** is the smallest distance that gives every feature at least one neighbor, which is the largest of the nearest-neighbor distances.

The top row of Figure 22.1 shows a distance band on 19 points that are dense near the center and sparse at the edges. With d = 8, the central point has four neighbors. The same d gives a point near the edge none. This is the **edge effect**: features at the boundary of a study area, or in sparse areas, get few or no neighbors, partly because we can't see past the edge of our data. For these 19 points, every point has at least one neighbor only once d reaches about 20.

22.4.2 Inverse distance weights

Distance band neighbors can also be continuous. Each neighbor inside the band gets a weight from an inverse function of distance:

$$w_{ij} = \frac{1}{d_{ij}^{x}}$$

where w_{ij} is the weight from observation i to observation j, d_{ij} is the distance between them, and x is the **distance-decay parameter**: larger values make weights fall off faster. 1 or 2 is common. Features beyond the threshold get a weight of 0.

```
d <- c(2, 4, 8)   # distances to three neighbors

1 / d^1   # x = 1: twice as far, half the weight
1 / d^2   # x = 2: twice as far, a quarter of the weight
```

Inverse distance weights grow very large as distance approaches 0, so check for duplicate or nearly identical points.

22.4.3 k nearest neighbors

The **nearest-neighbor** approach ignores the actual distance and keeps the k closest features. For polygons, it uses their centroids.

- **Every feature gets exactly *k* neighbors**, so no one is left out.
- **It adapts to density.** Neighbors are close in dense areas and far apart in sparse ones (bottom row of Figure 22.1).
- **Relationships are not always symmetric.** My nearest neighbor may have someone else even closer. spdep's `make.sym.nb()` forces symmetry if a method requires it.

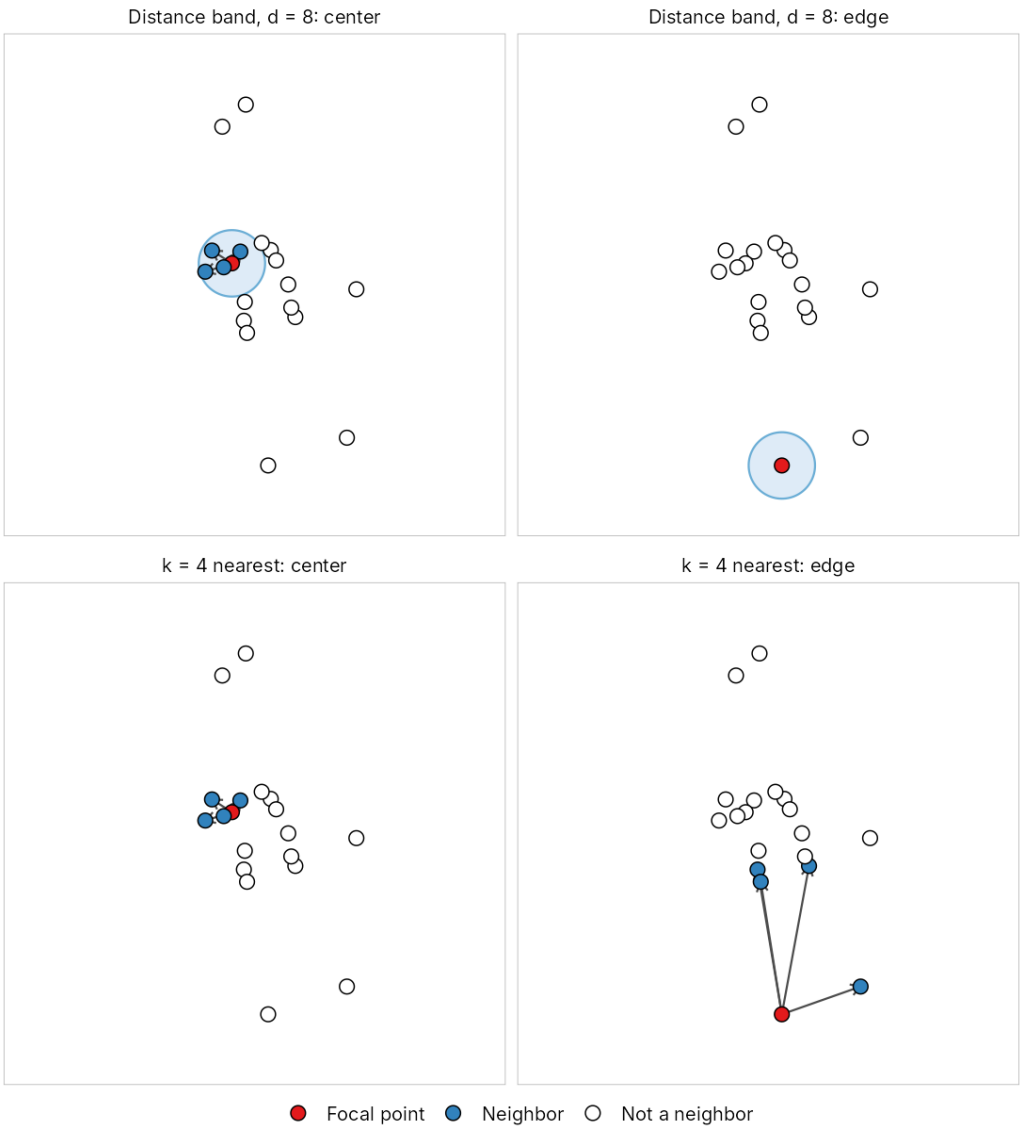


Figure 22.1: Distance bands (top) and k nearest neighbors (bottom) for the same 19 points. Each panel shows one focal point (red) and its neighbors (blue). With a distance band of 8, the central point has four neighbors and the edge point has none. With k = 4, both points have four neighbors, but the edge point's neighborhood stretches much farther.

A distance band gives a constant distance and a variable number of neighbors. *k* nearest neighbors gives a constant number and a variable distance. Choosing *k* is a trade-off: a small *k* keeps local detail, while a larger *k* gives more stable averages but pulls in less related features. Try a few values of *k* and check whether your conclusions change.

22.4.4 Contiguity: rook and queen

Topology-based neighbors ignore distance altogether and ask whether two polygons touch. Points can't share a border, so this only works for polygons. (If you need contiguity for points, first build **Thiessen** or **Voronoi polygons** around them: each polygon holds the area closer to its point than to any other.) There are two common rules, named after how the chess pieces move (Figure 22.2):

- **Rook contiguity:** polygons must share an edge, a boundary segment of some length. On a grid, each cell has four rook neighbors.
- **Queen contiguity:** a shared edge or a single shared corner is enough. On a grid, each cell has eight queen neighbors.

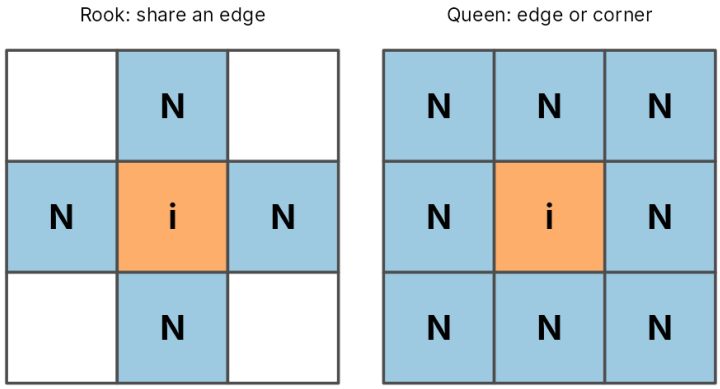


Figure 22.2: Rook and queen neighbors (N) of cell i on a grid.

Every rook neighbor is also a queen neighbor, so the queen set is always the same size or larger. On irregular real boundaries, such as counties, the two sets are often almost the same (Figure 22.3). The neighbors of the neighbors are called **second-order neighbors**.

In the `nc` county boundaries, 14 pairs of counties touch only at a corner, so the queen rule gives 490 neighbor links and the rook rule 462.

22.5 The neighborhood weight matrix

However you define neighbors, the result is stored the same way: a **neighborhood weight matrix**, usually called **W**. It is a table with *n* rows and *n* columns, where *n* is the number of observations. Entry w_{ij}, in row *i* and column *j*, describes the relationship between observation *i* and observation *j*. The diagonal is 0, because no observation is its own neighbor.

Here are four census tracts in a row, A | B | C | D, so each touches only the tracts beside it. With binary rook contiguity:

```
W <- matrix(c(0, 1, 0, 0,
              1, 0, 1, 0,
              0, 1, 0, 1,
```

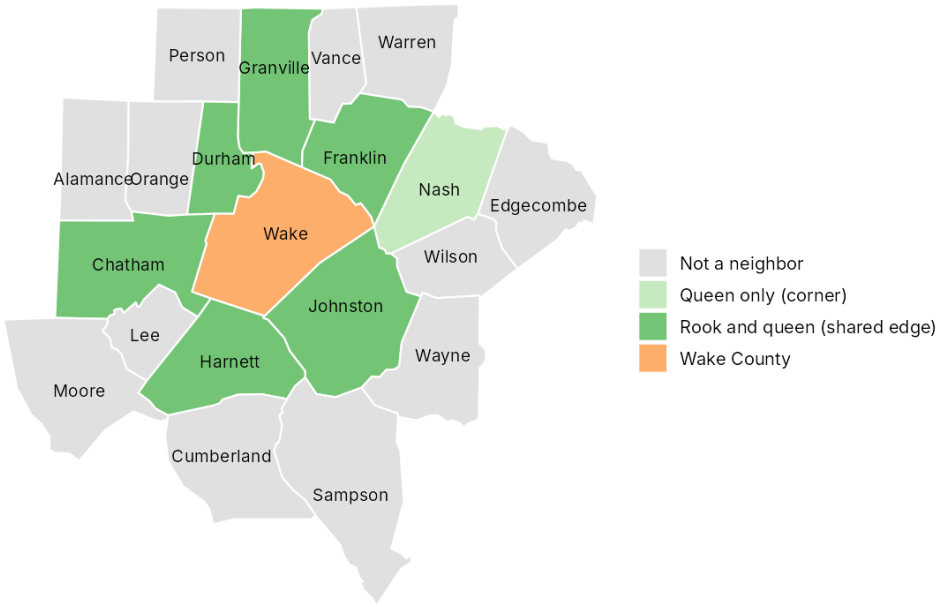


Figure 22.3: Wake County's contiguity neighbors, from the county boundaries in the sf package's nc data. Six counties share an edge with Wake. Nash touches it only at a corner, so it is a queen neighbor but not a rook neighbor.

```
              0, 0, 1, 0),
            nrow = 4, byrow = TRUE,
            dimnames = list(c("A", "B", "C", "D"), c("A", "B", "C", "D")))
W
rowSums(W)   # number of neighbors
```

Read across a row to find a tract's neighbors: B's row has 1s under A and C.

22.5.1 Row standardization

Weights are often **row-standardized**: each weight is divided by its row's total, so every row sums to 1. B's two neighbors then get 0.5 each, and the edge tracts' single neighbors get 1. In spdep this is `style = "W"`.

Why bother? Multiplying a row-standardized matrix by a variable gives each observation the *average* of its neighbors' values. That neighborhood average is called the **spatial lag**:

```
W <- matrix(c(0, 1, 0, 0,
              1, 0, 1, 0,
              0, 1, 0, 1,
              0, 0, 1, 0),
            nrow = 4, byrow = TRUE,
            dimnames = list(c("A", "B", "C", "D"), c("A", "B", "C", "D")))

W_std <- W / rowSums(W)   # row-standardize
W_std

income <- c(A = 10, B = 20, C = 30, D = 40)   # in thousands of dollars
```

```
W_std %*% income                                  # the spatial lag: neighbors' average
```

B's neighbors, A and C, average (10 + 30) ÷ 2 = 20. Moran's I (Chapter 22) compares each value with exactly this kind of neighborhood average.

22.5.2 Building neighbors and weights in R

In spdep, neighbors are stored as a **neighbor list** (class nb), a compact version of W that lists only each observation's neighbors. `nb2listw()` turns any neighbor list into weights. Assuming `library(sf)`, `library(spdep)`, and an sf polygon layer named `tracts` in a projected coordinate system:

```
# Contiguity neighbors (queen is the default)
nb_queen <- poly2nb(tracts, queen = TRUE)
nb_rook  <- poly2nb(tracts, queen = FALSE)

# k nearest neighbors, using tract centroids
coords <- st_coordinates(st_centroid(tracts))
nb_knn <- knn2nb(knearneigh(coords, k = 4))

# Distance band: 0 to 5 km (in the CRS's units, meters)
nb_dist <- dnearneigh(coords, d1 = 0, d2 = 5000)

# Row-standardized spatial weights
lw <- nb2listw(nb_queen, style = "W", zero.policy = TRUE)

# Inverse distance weights (x = 1) on the distance band
idw <- lapply(nbdists(nb_dist, coords), function(d) 1 / d)
lw_idw <- nb2listw(nb_dist, glist = idw, style = "B", zero.policy = TRUE)
```

`dnearneigh()` measures in the units of your coordinate reference system, so project first. `summary(nb_queen)` reports how many neighbors each observation has, and warns about any with none.

22.5.3 Weight matrices can encode social distance

Nothing requires the entries of a weight matrix to be geographic. They just record which observations are related and how strongly. They could be commuting flows between counties, airline routes, shared school districts, similarity in income or age structure, or trade and migration ties between countries. Two counties linked by heavy commuting may influence each other more than two counties that merely touch. Any $n \times n$ matrix of relationships becomes spatial weights with `spdep::mat2listw()`.

22.6 Check your understanding

1. Interaction between places decreasing as the distance between them increases is called:
[1]

[1] Distance decay

- (A) Distance decay
- (B) Spatial heterogeneity
- (C) The edge effect
- (D) Row standardization

2. What is the Manhattan distance from (1, 2) to (4, 6)? _[2]
3. Euclidean distance is fine to calculate directly on latitude and longitude. TRUE / FALSE[3]
4. With *k* nearest neighbors, every feature gets exactly *k* neighbors. TRUE / FALSE[4]
5. Two counties that touch only at a single corner are neighbors under which rule? [5]

- (A) Rook contiguity
- (B) Queen contiguity
- (C) Both rook and queen
- (D) Neither

6. With inverse distance weights and $x = 2$, a neighbor twice as far away gets what share of the weight? [6]

- (A) Half
- (B) A quarter
- (C) Twice as much
- (D) The same

7. In a row-standardized weight matrix, each row sums to: _[7]
8. Which spdep function builds contiguity neighbors from polygons? [8]

- (A) knearneigh()
- (B) dnearneigh()
- (C) poly2nb()
- (D) nb2listw()

[2] 7
[3] FALSE
[4] TRUE
[5] Queen contiguity
[6] A quarter
[7] 1
[8] poly2nb()

22.7 Summary

- **Physical distance** is measured across the Earth's surface; **social distance** by connection and similarity. **Distance decay** says interaction falls as distance grows.
- **Euclidean distance** is the straight line; use it only on projected coordinates. **Great circle distance** follows the curved Earth and suits longitude and latitude. **Manhattan distance** follows a street grid. **Network distance** and **travel time** follow real routes.
- The **first law of geography** is why neighbors matter: to measure spatial autocorrelation, we must first say who is near whom.
- Neighbors can be defined by a **distance band** (absolute distance), by the ***k* nearest neighbors**, or by **contiguity**: **rook** (a shared edge) or **queen** (an edge or a corner). Weights can be **binary** or **continuous**, such as inverse distance.
- Distance bands give a constant distance and a variable count, and leave edge features short. *k* nearest neighbors gives a constant count and a variable distance, and can be asymmetric.
- Neighbor relationships are stored in a **weight matrix**, W. **Row-standardized** weights sum to 1 across each row, so W times a variable gives the **spatial lag**: each observation's neighborhood average.
- In spdep: `poly2nb()`, `knearneigh()` with `knn2nb()`, and `dnearneigh()` build neighbors; `nb2listw()` turns them into weights. Each choice gives a different weight matrix, and can give different results, so justify yours and try more than one.

22.8 Exercises

1. **By hand.** For the points (2, 3) and (8, 11), calculate the Euclidean and the Manhattan distance, then check both with the functions in this chapter.
2. **Great circles.** Find the longitude and latitude of your hometown and calculate its great circle distance from Chapel Hill. Then compare it with the driving distance from an online map. How much longer is the drive, in percent?
3. **Wrong on purpose.** Calculate the "Euclidean distance" from Chapel Hill to Charlotte using their longitudes and latitudes as if they were x and y. What units is the answer in, and why is it meaningless?
4. **Your county.** Using the sf package's `nc` data, build queen and rook neighbors with `poly2nb()`. How many neighbors does your home county (or Orange County) have under each rule? Which county has the most neighbors?
5. **Three definitions.** Project `nc` with `st_transform(32119)`, find the county centroids, and build neighbors with $k = 4$ and with a distance band of 50 km. Compare `summary()` of each. Which counties have no neighbors within 50 km? What is the smallest distance band that gives every county at least one neighbor?
6. **Spatial lag.** Make row-standardized queen weights for `nc`, and calculate the spatial lag of births per square kilometer with `lag.listw()`. Make a scatterplot of each county's value against its spatial lag. What does the pattern suggest?

22.9 Further reading

- *Spatial Data Science: With Applications in R*, by Edzer Pebesma and Roger Bivand (CRC Press, 2023), free at https://r-spatial.org/book/. Chapter 14, "Proximity and Areal Data", covers neighbor lists and spatial weights in spdep.



- *Geographic Information Analysis* (2nd ed.), by David O'Sullivan and David Unwin (Wiley, 2010). Chapter 2 discusses distance, adjacency, interaction, and neighborhoods.
- Tobler, W. R. (1970). "A Computer Movie Simulating Urban Growth in the Detroit Region." *Economic Geography* 46: 234–240. The source of the first law of geography.

23 Spatial Clustering and Spatial Autocorrelation

Learning objectives

By the end of this chapter you should be able to:

1. Explain why the first law of geography means real-world patterns are rarely random.
2. Tell global from local measures of clustering, and spatial pattern from spatial autocorrelation.
3. Calculate and interpret the Average Nearest Neighbor ratio for a set of points.
4. Explain how Moran's I is calculated, and interpret its magnitude, its significance, and its importance.
5. Run Moran's I and a local Getis-Ord Gi* hot-spot analysis in R with spdep, and map the results.
6. Choose the right statistic for a question: Average Nearest Neighbor, Moran's I, or Getis-Ord Gi*.

Look at a map of reported Lyme disease cases in the United States and you'll see cases clustered in the Northeast and the upper Midwest. Look at COVID-19 case rates by ZIP code and you'll see high-rate ZIP codes next to other high-rate ZIP codes. Our eyes are very good at seeing clusters. They are also very good at seeing clusters that aren't there.

This chapter is about testing, objectively, whether things on a map are **clustered, random, or dispersed**. It builds on the neighbors and weight matrices of Chapter 21. We'll cover spatial pattern, measured with the Average Nearest Neighbor ratio; spatial autocorrelation, measured with Moran's I; and local hot-spot analysis, with Getis-Ord Gi*.

23.1 Tobler's first law

Waldo Tobler's **first law of geography** (1970): "everything is related to everything else, but near things are more related than distant things." Values at locations near each other tend to be similar, and the similarity fades with distance. That means phenomena are *not* distributed randomly across space.

Imagine how the world would look if they were. A map of population density with every county's value shuffled at random would look like static: no cities, no empty mountains, no suburbs around them. A random elevation map would have no mountain ranges and no river valleys. Real maps almost never look like that. They have structure, and this chapter's statistics measure it.

23.1.1 Geographic variation

Geographic variation is the change in a phenomenon from place to place. We can list it in a table, but we see it in a map. Three kinds of things vary:

- **Events,** such as disease cases, mapped as points.
- **Locations,** such as hospitals, also points.
- **Values,** such as average income, usually attached to polygons such as counties.

23.1.2 Seeing a pattern is not testing it

Looking at a map is subjective. Two people can look at the same map and disagree about whether it is clustered. And map design changes what we see: the classification breaks and the color ramp (Chapter 10) can make the same data look clustered or not. Spatial statistics give repeatable, testable answers.

23.2 Spatial cluster analysis: global and local

There are two main approaches to finding clusters:

- **Global** methods ask: are the events or values clustered *anywhere* in the study area? They return one result for the whole dataset, but they don't tell us *where* things are happening.
- **Local** methods ask: *which* areas have clusters of high or low values? They calculate a result for every observation, so the results can be mapped.

A global statistic for income inequality in U.S. counties would look at all 3,000-plus counties at once and return one number. A local statistic would give every county its own value, based on its neighbors, and show where high inequality sits next to high inequality.

23.3 Spatial pattern: how objects are arranged

A **spatial pattern** is the arrangement of objects in space. Describing and explaining patterns, how people, objects, and features are distributed across a landscape and why, is one of the central concerns of geography. Based only on where events are, there are three basic patterns (Figure 23.1):

- **Clustered:** events are located near one another.
- **Random:** there is no regular pattern; any location is equally likely.
- **Ordered** or **dispersed:** events are spread out in a regular, repeating way.

23.3.1 The Average Nearest Neighbor ratio

The **Average Nearest Neighbor (ANN) ratio** asks: how are my points arranged, compared with a random arrangement?

$$ANN = \frac{\bar{D}_O}{\bar{D}_E}$$

$\bar{D}_O$ is the observed mean distance from each point to its nearest neighbor. $\bar{D}_E$ is the mean distance you'd expect if the same number of points were placed at random in the same area: $0.5/\sqrt{n/A}$, for *n* points in an area *A*.

- **ANN < 1:** points are closer together than random: **clustered**.
- **ANN > 1:** points are more spread out than random: **dispersed**.
- **ANN ≈ 1: random**.

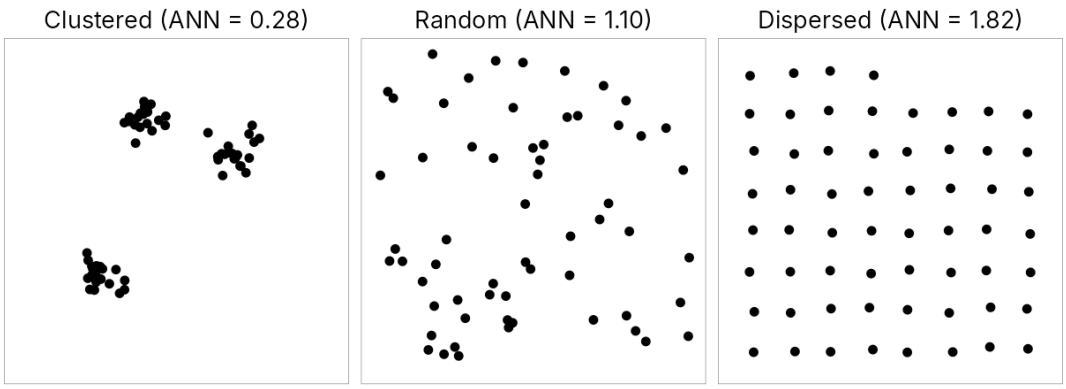


Figure 23.1: Sixty points in three patterns, with their ANN ratios.

```
ann <- function(x, y, area) {
  d <- as.matrix(dist(cbind(x, y)))   # distance from every point to every other
  diag(d) <- Inf                      # a point isn't its own nearest neighbor
  observed <- mean(apply(d, 1, min))  # mean nearest-neighbor distance
  expected <- 0.5 / sqrt(length(x) / area)
  observed / expected
}

set.seed(22)
x <- runif(60, 0, 100)   # 60 random points in a 100 x 100 square
y <- runif(60, 0, 100)
ann(x, y, area = 100 * 100)
```

Sixty random points give an ANN of 1.10: close to 1, as expected. In the lecture example, a set of crime points in a city gave an ANN of 0.57, well below 1, so the crimes are clustered.

23.3.2 Limits of the ANN ratio

- **It ignores values.** ANN uses only locations. Two datasets with the same point locations but very different attribute values get the same ratio.
- **It looks at one dataset by itself.**



- **It is global.** One ratio for the whole pattern says nothing about where the clusters are.
- **It depends on the study area.** The expected distance depends on A, so a larger boundary makes the same points look more clustered. Always report the study area with an ANN ratio.

To account for values, we need spatial autocorrelation.

23.4 Spatial autocorrelation

The same three patterns apply to attribute **values**, not just locations. Values can be clustered (similar values next to each other), random, or dispersed (neighbors tend to be different, like a checkerboard). **Spatial autocorrelation** is the degree of similarity between objects that are located near each other. It describes how values are arranged across a landscape, and it can be measured, for a whole region (a global statistic) or for smaller areas within it (local statistics).

The key distinction:

- **Spatial pattern** cares only about location and arrangement. Use ANN.
- **Spatial autocorrelation** cares about location *and* values. Use Moran's I and Getis-Ord Gi.

Notice the "auto": *self*. Spatial autocorrelation describes how a *single* variable relates to itself at nearby locations. Correlation between two different variables, such as income and education, is a different question, answered with regression (Chapter 19).

23.4.1 Which data suit spatial autocorrelation?

It works for areal (polygon), point, or raster data. In each case we measure how the *values* are arranged. Avoid raw counts, unless the populations generating the counts are the same everywhere: a county with more people will have more cases simply because of its size. Use rates or densities instead, such as cases per 1,000 residents or births per square kilometer.

23.4.2 It depends on who counts as a neighbor

Because spatial autocorrelation is about similarity between things that are *near* each other, we must define neighbors first: by contiguity (rook or queen), by a distance band, or by the k nearest neighbors (Chapter 21). Queen contiguity is the usual default for polygons such as counties. Different neighbor definitions can give different answers, so report the one you used.

23.5 Moran's I: a global test

Moran's I is the most widely used statistic for describing and testing spatial autocorrelation in a region. It is global: it considers the whole study area, and returns a single value, I, that measures the magnitude of spatial autocorrelation, along with a p-value.

23.5.1 How Moran's I is calculated

$$I = \frac{N}{W} \cdot \frac{\sum_i \sum_j w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_i (x_i - \bar{x})^2}$$

- N is the number of observations.
- w_{ij} is the spatial weight between i and j: 1 if they are neighbors and 0 if not, in the simplest case.
- W is the sum of all the weights.
- $x_i - \bar{x}$ is how far observation i is from the mean.

You won't compute it by hand, but know what drives it. For every pair of neighbors, multiply how far each is from the mean. If both are above the mean, or both below, the product is positive. If one is above and the other below, it is negative. So when neighbors tend to be alike, the sum is positive and I rises toward 1; when neighbors tend to differ, it is negative and I falls toward –1. The denominator standardizes by the overall variance.

23.5.2 Moran's I ranges from –1 to 1

- **–1:** perfectly dispersed, like a checkerboard.
- **0:** random.
- **+1:** perfectly clustered.

More precisely, the value expected under complete spatial randomness is $-1/(N-1)$, which is close to 0 for large datasets. Figure 23.2 shows five arrangements of the same 64 cells, half dark and half light. Only the arrangement changes, and arrangement is all Moran's I responds to.

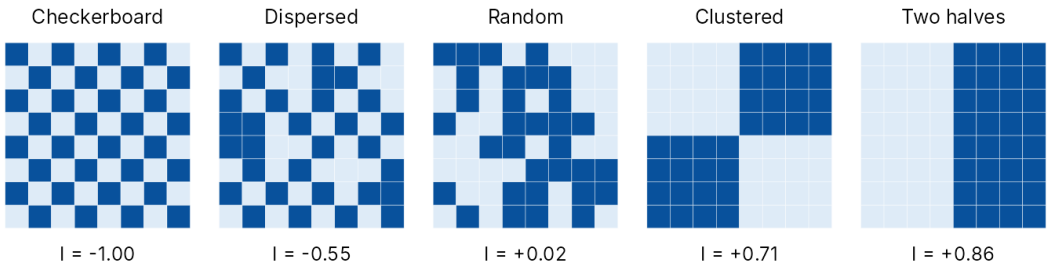


Figure 23.2: Moran's I for five arrangements of the same 64 cells, using rook neighbors.

You can check these values with a few lines of base R. Each cell's rook neighbors are the cells exactly 1 unit away:

```
cells <- expand.grid(row = 1:8, col = 1:8)
W <- (as.matrix(dist(cells)) == 1) * 1   # 1 if rook neighbors, 0 if not

moranI <- function(x, W) {
  z <- x - mean(x)
  n <- length(x)
  (n / sum(W)) * sum(W * outer(z, z)) / sum(z^2)
}

checkerboard <- (cells$row + cells$col) %% 2
halves <- as.integer(cells$col > 4)
```

```
moranI(checkerboard, W)
moranI(halves, W)
```

23.5.3 Is it significant?

How do we know whether an I is bigger than chance? The **Monte Carlo** (or **permutation**) test answers it directly: shuffle the values across the locations many times, calculate I each time, and see where the observed I falls among those random results.

```
cells <- expand.grid(row = 1:8, col = 1:8)
W <- (as.matrix(dist(cells)) == 1) * 1
moranI <- function(x, W) {
  z <- x - mean(x)
  (length(x) / sum(W)) * sum(W * outer(z, z)) / sum(z^2)
}
halves <- as.integer(cells$col > 4)

set.seed(215)
random_I <- replicate(999, moranI(sample(halves), W))   # 999 shuffles
hist(random_I, main = "Moran's I for 999 random shuffles", xlab = "I")
abline(v = moranI(halves, W), col = "red", lwd = 2)

mean(random_I)   # close to -1/63
```

The shuffled values give I near 0 (about −0.013, close to −1/63). None comes anywhere near 0.857, so the two-halves pattern is very unlikely to be chance.

23.6 Interpreting Moran's I

Interpret a Moran's I result in three steps: its magnitude, its significance, and its importance.

23.6.1 Magnitude: the I value

The closer to 1, the more clustered the values; the closer to −1, the more dispersed. There's no official scale for describing strength, but this rule of thumb from the GEOG 215 lectures is a useful guide:

Table 23.1: A rule of thumb for describing the strength of Moran's I.

I value	Interpretation
> 0.5	Highly clustered
0.3 to 0.5	Moderately clustered
0.1 to 0.3	Slightly clustered
0 to 0.1	Barely clustered (≈ random)
0 to −0.1	Barely dispersed (≈ random)
−0.1 to −0.3	Slightly dispersed
−0.3 to −0.5	Moderately dispersed
< −0.5	Highly dispersed

23.6.2 Significance: the p-value

The p-value is the probability of an I this extreme if the values were arranged at random. If $p < 0.05$, the result is statistically significant. If $p > 0.05$, the pattern can't be distinguished from random.

23.6.3 Importance

Beware of a significant but unimportant result. The p-value depends on the number of observations, so:

- With many observations, a tiny deviation from random can be significant: $I = 0.04$, $p < 0.001$ is significant, but essentially random in practice.
- With few observations, a small amount of clustering, say $I = 0.11$, may not reach significance.

Always report and interpret *both* I and p (Chapter 18).

> **Think about it**
>
> Would you expect Moran's I for median household income in North Carolina's census tracts to be positive, near 0, or negative? What about elevation? The number of fast-food restaurants per person? Why?

23.7 Moran's I in R

In R, the **spdep** package does the work. Here is Moran's I for births per square kilometer in North Carolina's counties from 1974 to 1978, using the sf package's `nc` data (Chapter 16) and queen contiguity (Figure 23.3):

```
library(sf)
library(spdep)

nc <- st_read(system.file("shape/nc.shp", package = "sf")) |>
  st_transform(32119)
nc$births_per_km2 <- nc$BIR74 / (as.numeric(st_area(nc)) / 1e6)

nb <- poly2nb(nc)                    # queen contiguity neighbors
lw <- nb2listw(nb, style = "W")      # row-standardized weights

moran.test(nc$births_per_km2, lw)

set.seed(215)
moran.mc(nc$births_per_km2, lw, nsim = 499)   # Monte Carlo version
```

```
    Monte-Carlo simulation of Moran I

data:  nc$births_per_km2
weights: lw
number of simulations + 1: 500
```

```
statistic = 0.12675, observed rank = 487, p-value = 0.026
alternative hypothesis: greater
```

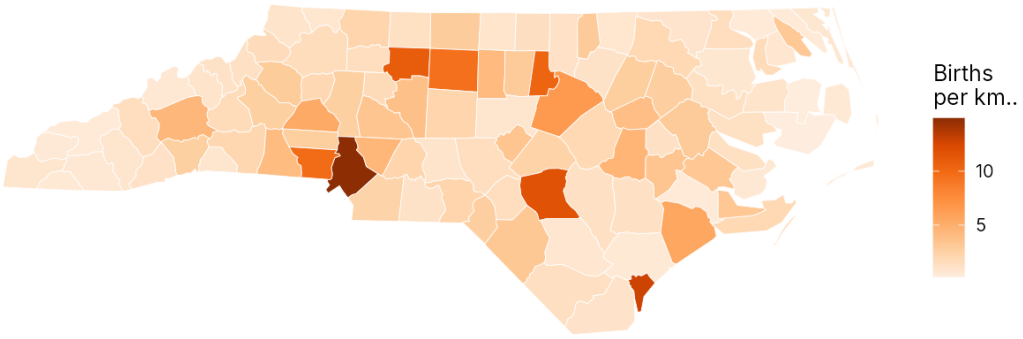


Figure 23.3: Births per square kilometer in North Carolina's counties, 1974–78.

The two numbers to report are the statistic, $I = 0.127$, and the p-value, 0.026. By the rule of thumb, 0.127 is **slightly clustered**, and because $p < 0.05$ the result is statistically significant. (`moran.test()`, which uses a formula instead of shuffling, gives the same I and $p = 0.015$.) Counties with many births per square kilometer tend to be near each other, but the effect is modest.

The lecture used the population of North Carolina's counties from the ACS and got a very similar result: $I = 0.158$, $p = 0.028$, slightly clustered and significant.

23.8 Local hot-spot analysis: Getis-Ord Gi*

A **global statistic** gives one number for one variable, such as Moran's I for county population. A **local statistic** gives a value for *each observation*: Getis-Ord Gi*, or local Moran's I, for each county. Because local results are attached to places, we can map them and see where the clusters are.

The **Getis-Ord G** statistic asks a slightly different question from Moran's I: are *high values near high values*, or *low values near low values*? Like Moran's I, it has a global and a local version. The local version gives every observation a **z-score**:

- A **hot spot** is an area of high values surrounded by high values: a large positive z-score.
- A **cold spot** is an area of low values surrounded by low values: a large negative z-score.
- Values near 0 are not clustered.

There are two versions. **Gi** leaves the observation itself out and looks only at its neighbors. **Gi*** includes the observation along with its neighbors; it is the version most software uses by default. As a rule of thumb, a z-score above 1.96 or below –1.96 is significant at the 0.05 level, and beyond ±2.58 at the 0.01 level.

23.8.1 Local Getis-Ord Gi* in R

The workflow has three steps:

1. **Define neighbors.** For counties, queen contiguity; for points, the *k* nearest neighbors, such as `knn2nb(knearneigh(coords, k = 10))`.
2. **Convert them to binary weights.** For Gi*, `include.self()` adds each observation to its own neighborhood, and `style = "B"` makes every neighbor a 1.
3. **Run `localG()`** to get a z-score for every observation, then map it.

```
nb_self <- include.self(poly2nb(nc))
lw_b <- nb2listw(nb_self, style = "B")

nc$gi <- as.numeric(localG(nc$births_per_km2, lw_b))

nc$gi_class <- cut(nc$gi, c(-Inf, -1.96, 1.96, 2.58, Inf),
                   labels = c("Cold spot (z < -1.96)", "Not significant",
                              "Hot spot (z 1.96 to 2.58)", "Hot spot (z > 2.58)"))

tm_shape(nc) +
  tm_polygons(fill = "gi_class",
              fill.scale = tm_scale_categorical(
                values = c("#2166ac", "grey85", "#f4a582", "#b2182b")),
              fill.legend = tm_legend("Gi* z-score"))
```

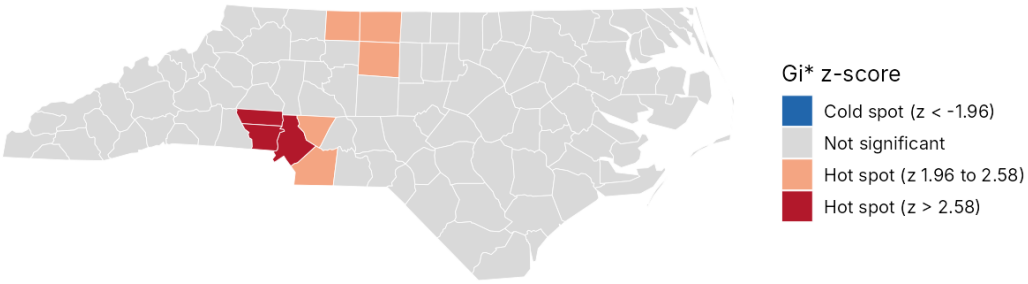


Figure 23.4: Getis-Ord Gi* hot spots of births per square kilometer, 1974–78, with queen contiguity.

Eight counties are significant hot spots (Figure 23.4). The strongest, with z-scores above 2.58, are Gaston (3.9), Lincoln (3.4), and Mecklenburg (3.2): the Charlotte area, where high-density counties are surrounded by other high-density counties. A second group of hot spots is in the Triad: Guilford, Rockingham, and Stokes. No county is a significant cold spot: the lowest z-scores, in the far west and the northeast, reach only about −1.8.

Notice that Wake and Durham, which have high densities themselves, are *not* hot spots. Their neighbors, such as Johnston, Harnett, Chatham, and Person, had far fewer births per square kilometer in the 1970s, so the neighborhood as a whole isn't unusual. Gi* finds clusters of high values, not single high values.

The lecture ran the same analysis on county population from the ACS: counties around Charlotte formed the strongest hot spot (z from 3.0 to 4.0), with a second band of positive values across the north-central Piedmont, and low-population cold spots in the northeast and far west.

23.9 Summary: choosing a spatial statistic

Table 23.2: Choosing a spatial statistic.

Method	Uses	Scope	Answers
Average Nearest Neighbor	Locations only	Global	Are points clustered, random, or dispersed?
Moran's I	Locations and values	Global	Are similar values near each other overall?
Getis-Ord Gi / Gi*	Locations and values	Local	Where are the hot spots and cold spots?

Whichever you use: define your neighbors first, report both the statistic and its p-value, and map local results.

23.10 Check your understanding

1. A spatial statistic that returns one value for the whole study area is: [1]

- (A) Global
- (B) Local
- (C) Dispersed
- (D) Contiguous

2. An Average Nearest Neighbor ratio of 0.57 means the points are: [2]

- (A) Clustered
- (B) Random
- (C) Dispersed
- (D) Impossible to tell

3. ANN takes the attribute values of the points into account. TRUE / FALSE[3]

4. A perfect checkerboard of two values has a Moran's I of: [4]

- (A) -1
- (B) 0
- (C) 0.5
- (D) 1

[1] Global
[2] Clustered
[3] FALSE
[4] -1

5. Moran's I = 0.158 with p = 0.028 describes a pattern that is: [5]

- (A) Slightly clustered and statistically significant
- (B) Highly clustered and not significant
- (C) Dispersed and significant
- (D) Random

6. Why should you usually use rates instead of raw counts for spatial autocorrelation? [6]

- (A) Places with more people have more cases just because of their size
- (B) Counts can't be mapped
- (C) Moran's I only works on percentages
- (D) Rates are always normally distributed

7. A county with a large positive Gi* z-score is a: [7]

- (A) Hot spot
- (B) Cold spot
- (C) Outlier
- (D) Random value

8. A Gi* z-score of 2.3 is significant at the 0.05 level. TRUE / FALSE[8]

23.11 Summary

- The **first law of geography** means real-world values are rarely random across space. Seeing a pattern on a map is subjective; spatial statistics test it.
- **Global** statistics give one answer for the whole study area; **local** statistics give one per observation, which you can map.
- **Spatial pattern** uses locations only. The **Average Nearest Neighbor ratio** compares the observed mean nearest-neighbor distance with the random expectation: below 1 is clustered, above 1 dispersed. It depends on the study area.
- **Spatial autocorrelation** uses locations and values: the similarity of a single variable at nearby locations. Use rates, not counts, and define neighbors first.
- **Moran's I** is global and runs from –1 (dispersed) through 0 (random) to +1 (clustered). Interpret its magnitude, its p-value, and whether it matters. In R: `poly2nb()`, `nb2listw()`, then `moran.test()` or `moran.mc()`.
- **Getis-Ord Gi*** is local. A large positive z-score is a **hot spot**, a large negative one a **cold spot**; $|z| > 1.96$ is significant at 0.05. In R: `include.self()`, `nb2listw(style = "B")`, then `localG()`.

[5] Slightly clustered and statistically significant
[6] Places with more people have more cases just because of their size
[7] Hot spot
[8] TRUE



- Births per square kilometer in North Carolina's counties, 1974–78, are slightly clustered (I = 0.127, p = 0.026), with hot spots around Charlotte and in the Triad.

23.12 Exercises

1. **ANN by hand.** Five points are at (1, 1), (2, 1), (1, 2), (8, 8), and (9, 8), in a 10 × 10 study area. Find each point's nearest-neighbor distance, the observed mean, the expected mean, and the ANN ratio. Then double the study area to 20 × 20. What happens to the ratio, and why?
2. **Shuffle a map.** Shuffle the values of `births_per_km2` across North Carolina's counties with `sample()`, map the result, and calculate Moran's I. Compare both with the real map and the real I.
3. **Neighbors matter.** Recalculate Moran's I for births per square kilometer with rook contiguity, with k = 4 nearest neighbors, and with a 50 km distance band (Chapter 21). How much does I change?
4. **Sudden infant deaths.** Calculate the rate of sudden infant deaths per 1,000 births, `SID74 / BIR74 * 1000`, and test it for spatial autocorrelation. Is it more or less clustered than births per square kilometer? Then map its Gi* hot and cold spots.
5. **Residuals.** Fit a regression of `BIR79` on `BIR74` (Chapter 19), and test its residuals with `lm.morantest()`. Are the residuals spatially autocorrelated? What would you do next?
6. **Local Moran's I.** Run `localmoran()` on births per square kilometer, and compare which counties stand out with the Gi* map. What does local Moran's I find that Gi* doesn't?

23.13 Further reading

- *Spatial Data Science: With Applications in R*, by Edzer Pebesma and Roger Bivand (CRC Press, 2023), free at https://r-spatial.org/book/. Chapter 15, "Measures of Spatial Autocorrelation", covers Moran's I and local indicators in spdep.
- Getis, A., and Ord, J. K. (1992). "The Analysis of Spatial Association by Use of Distance Statistics." *Geographical Analysis* 24(3): 189–206. The paper that introduced the G statistics.
- Goodchild, M. F. (1986). *Spatial Autocorrelation*. Concepts and Techniques in Modern Geography (CATMOG) 47. Geo Books. A short, classic introduction, with grid examples like Figure 23.2.
- Tobler, W. R. (1970). "A Computer Movie Simulating Urban Growth in the Detroit Region." *Economic Geography* 46: 234–240.



Part VI

Unit 6: Next Steps



This unit looks beyond the course. It first turns your maps and analysis into interactive dashboards that anyone can explore, then shows where the skills from this book are used, previews the advanced topics that build on them, and maps out the courses, credentials, and careers that come next.

Chapter 23: Interactive Map Dashboards

This chapter builds an interactive dashboard of income and poverty by state. It makes interactive maps with leaflet (or tmap in view mode), lays out a map, value box, chart, and table with flexdashboard, links them with crosstalk in a single HTML file, and adds Shiny so the dashboard recalculates when readers make choices. It ends with how to share each kind of dashboard.

Chapter 24: What's Next in Spatial Data Science

This chapter shows spatial data science at work in public health, environmental science, planning, social science, and business. It previews three advanced areas: remote sensing and NDVI, space-time modeling, and GeoAI, including why spatial autocorrelation matters for machine learning. It ends with a path through UNC's GIS courses, the GIS minor and Geospatial Data Science major, and careers in GIS and spatial data science.

24 Interactive Map Dashboards

Learning objectives

By the end of this chapter you should be able to:

1. Explain what a dashboard is, how it differs from a report, and what makes a good one.
2. Make an interactive choropleth map with leaflet, with a palette function, legend, labels, and popups, or with tmap in view mode.
3. Lay out a dashboard with flexdashboard: a sidebar, columns, and panels holding a map, a value box, a chart, and a table.
4. Link the views with crosstalk, and work around its limits on polygons.
5. Add Shiny inputs, reactive expressions, and render functions so a dashboard recalculates.
6. Choose between a static and a Shiny dashboard, and share each one.

All semester you have made maps and analyses and written them up as R Markdown reports. This chapter turns that work into an interactive **dashboard**: a single web page where a reader can explore a map, a chart, and a table that respond to each other. We build one example dashboard step by step, first from interactive widgets that run in any browser, and then with Shiny. Everything is still an R Markdown file, so it fits the workflow you already know.

This chapter's code runs in RStudio, not in the book

leaflet, flexdashboard, crosstalk, and Shiny build web pages, so they can't run in the live code cells of this book. The code here is shown as plain blocks, with screenshots of the results. Copy it into RStudio to try it. Install the packages once with:

```
install.packages(c("flexdashboard", "leaflet", "DT", "plotly", "crosstalk", "shi
```

24.1 What is a dashboard?

A **dashboard** shows key information on one screen so people can monitor and explore it. It usually has several **linked views**, such as a map, charts, a table, and a few headline numbers, and it is interactive: readers can zoom, hover, filter, and select. You have certainly used one. During COVID-19, case dashboards from universities and state health departments were checked daily by millions of people, and most were built around a map. Air quality, weather, and traffic dashboards, and city and state open data portals, work the same way. A map is often the anchor, because readers first ask "where?"

24.1.1 Reports and dashboards do different jobs

Table 24.1: Reports and dashboards.

	R Markdown report	Dashboard
Order	Tells a story in a fixed order	Reader explores in any order
Maps and charts	Static	Interactive and linked
Text	Explains every result	Short labels; the visuals carry the message
Best for	An argument or analysis write-up	Monitoring and exploring data

Your assignments have been reports: you guide the reader through an analysis step by step. A dashboard hands control to the reader. That means less text and more careful design, because the visuals have to explain themselves. A good project often has both: a report that explains the analysis (Chapter 20) and a dashboard that lets people explore the data.

24.1.2 What makes a good dashboard

- **Answer a few clear questions,** and leave out everything else. Write down two or three questions before you start, and cut anything that doesn't help answer them.
- **Put the most important view, often the map, largest and first.**
- **Use the map design rules you already know** (Chapter 10): sensible classification, a clear legend with units, and a data source.
- **Link the views,** so a filter or selection updates them together.
- **Design for everyone.** Never use color alone to carry meaning; add labels and a table. A table beside the map gives screen reader users and colorblind readers the same information. Check color contrast, and test with a keyboard and on a phone.

Think about it

Think of a dashboard you have used: for weather, sports, a class, or public health. What two or three questions does it answer? What would you cut, and what is missing?

24.2 The R dashboard toolkit

Table 24.2: Packages for dashboards in R.

Package	What it does
flexdashboard	Lays out an R Markdown file as a dashboard
leaflet	Interactive web maps (tmap makes them too, with `tmap_mode("view")`)
DT	Searchable, sortable data tables
plotly	Interactive charts
crosstalk	Links widgets, so a filter updates them all
shiny	Runs R code in response to what the reader does

leaflet, DT, and plotly are all **htmlwidgets**: R functions that produce JavaScript, which runs in the browser with no R needed.

24.2.1 Two kinds of interactivity

The most important design decision is which of two kinds of dashboard to build:

Table 24.3: Static and Shiny dashboards.

	htmlwidgets + crosstalk	Shiny
How it works	Knit once to a single HTML file; interactivity runs in the browser	R runs on a server while people use it
What readers can do	Show and hide the data that was knitted in	Trigger any R code: recalculate statistics, rebuild maps, query APIs
Sharing	No R needed; share it like any file	Needs a server

A static dashboard can be emailed, uploaded to Canvas, or hosted on GitHub Pages, and it works with no R installed. But it can only show and hide the data that was baked in when you knitted it. Shiny can do anything R can do in response to a click, like recalculating a weighted average or refitting a model, but it needs a running R process. Start static, and move to Shiny only when you need it.

24.2.2 Our example: income and poverty by state

Figure 24.1 shows where we are headed. A sidebar holds filters. The large map shows median income by state. On the right are a value box, a scatter plot of poverty rate against median income, and a table.

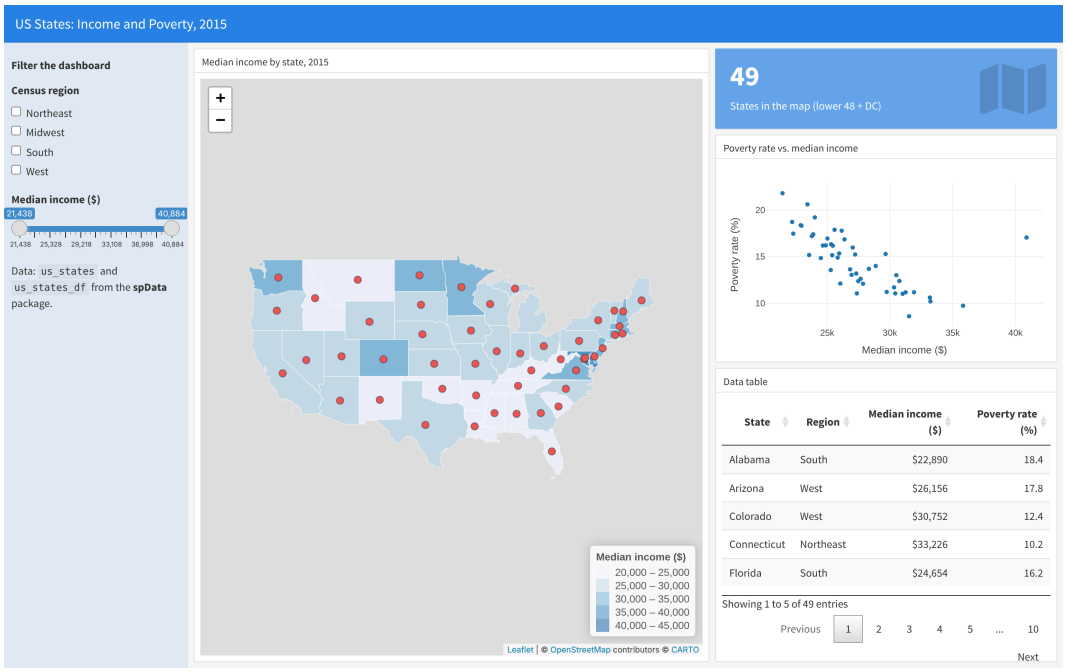


Figure 24.1: The finished static dashboard. The map's gray background is only because the basemap tiles didn't load when this screenshot was taken; on your computer you will see a light basemap.



The data are `us_states` and `us_states_df` from the spData package, the same data you joined in Chapter 15, so everything loads with `library()` and nothing needs downloading. Preparing the data looks just like the joins chapter:

```
library(sf)
library(tidyverse)
library(spData)

states <- us_states |>
  left_join(us_states_df, by = c("NAME" = "state")) |>
  mutate(
    REGION = fct_recode(REGION, Northeast = "Norteast"),  # fix a typo in spData
    poverty_rate_15 = poverty_level_15 / total_pop_15 * 100
  ) |>
  st_transform(4326)  # leaflet expects WGS84 longitude and latitude
```

Three details. spData spells the Northeast region "Norteast", so `fct_recode()` from forcats fixes the label. `poverty_level_15` is a count of people, so dividing by population gives a rate. And leaflet works in longitude and latitude on WGS84, EPSG:4326 (Chapter 12), so we transform the data. Doing the data preparation once, at the top of the file, keeps the rest of the dashboard simple.

24.3 Interactive maps with leaflet

The map is the heart of the dashboard.

24.3.1 Your first leaflet map

```
library(leaflet)

leaflet(states) |>
  addProviderTiles(providers$CartoDB.Positron) |>
  addPolygons(color = "white", weight = 1,
              fillColor = "steelblue", fillOpacity = 0.6)
```

A leaflet map is built in layers, like tmap and ggplot2, but the layers are joined with the pipe instead of +. `leaflet()` creates an empty map from the data. `addProviderTiles()` adds **basemap tiles**; CartoDB Positron is a light gray basemap that works well under choropleth colors. `addPolygons()` draws the states. Run this in RStudio and the map appears in the Viewer pane, where you can already zoom and pan.

24.3.2 Coloring a choropleth

Unlike tmap, leaflet doesn't classify data for you. First you build a **palette function**: a function that takes data values and returns colors.

```
pal <- colorBin("Blues", domain = states$median_income_15, bins = 5)
```

```
leaflet(states) |>
  addProviderTiles(providers$CartoDB.Positron) |>
  addPolygons(fillColor = ~pal(median_income_15),
              fillOpacity = 0.8, color = "white", weight = 1) |>
  addLegend(pal = pal, values = ~median_income_15,
            title = "Median income ($)")
```

`colorBin()` takes a ColorBrewer scheme, here Blues, and the range of the data, and returns a function that turns values into colors. The tilde in `~pal(median_income_15)` tells leaflet to look up the column in the data. Use the same palette in `addLegend()` so the legend matches the map. The other palette functions are `colorNumeric()` for a continuous ramp, `colorQuantile()` for quantile classes, and `colorFactor()` for categories.

24.3.3 Labels, popups, and highlighting

Interactivity is what makes a web map worth the effort:

```
addPolygons(
  fillColor = ~pal(median_income_15), fillOpacity = 0.8,
  color = "white", weight = 1,
  label = ~NAME,                                    # shows on hover
  popup = ~paste0("<b>", NAME, "</b><br>Poverty rate: ",
                  round(poverty_rate_15, 1), "%"),   # shows on click
  highlightOptions = highlightOptions(
    weight = 3, color = "#13294B", bringToFront = TRUE)
)
```

A **label** appears when the reader hovers over a state, and a **popup** when they click. Popups accept simple HTML, so the state name can be bold, with a line break after it. `highlightOptions()` outlines the state under the mouse, so readers know exactly which polygon they are on. Labels also help accessibility, because readers don't have to match a color to the legend.

24.3.4 A shortcut: tmap in view mode

If you prefer tmap, you don't have to learn leaflet first. Chapter 10 showed that `tmap_mode("view")` turns tmap code into an interactive map, and that map *is* a leaflet map:

```
library(tmap)
tmap_mode("view")   # interactive; tmap_mode("plot") for static

tm_shape(states) +
  tm_polygons(
    fill = "median_income_15",
    fill.scale = tm_scale_intervals(n = 5, style = "jenks",
                                    values = "brewer.blues"),
    fill.legend = tm_legend(title = "Median income ($)"),
    id = "NAME",
    popup.vars = c("median_income_15", "poverty_rate_15"))
```



`id` sets the hover label, and `popup.vars` chooses the variables shown on click. tmap is quicker to write and classifies the data for you. leaflet gives you more control and works with crosstalk, which we use below. Both work in a dashboard.

24.4 Laying out a flexdashboard

24.4.1 A flexdashboard is an R Markdown file

A **flexdashboard** is an R Markdown file with a different output format in its YAML header. You knit it the same way you knit your assignments.

```
---
title: "US States: Income and Poverty, 2015"
output:
  flexdashboard::flex_dashboard:
    orientation: columns
    vertical_layout: fill
---
```

`orientation: columns` lays out the page as side-by-side columns; `rows` is the other option. `vertical_layout: fill` stretches the panels to fill the browser window; `scroll` lets the page scroll instead. In RStudio, File > New File > R Markdown > From Template > Flex Dashboard starts one for you.

24.4.2 Columns, panels, and a sidebar

The layout comes from Markdown headings:

```
Inputs {.sidebar}
-----------------------------------------

Column {data-width=600}
-----------------------------------------
### Median income by state, 2015

Column {data-width=400}
-----------------------------------------
### States {data-height=120}
### Poverty rate vs. median income
### Data table
```

- A heading underlined with dashes starts a **column**.
- A `###` heading starts a **panel** inside that column, and its text becomes the panel's title. Under each panel heading goes one R chunk that draws the map, chart, or table (left out above).
- The attributes in curly braces set relative widths and heights. `600` and `400` mean the map column gets 60% of the space after the sidebar.
- The special `{.sidebar}` column holds inputs and filters on the left.

Figure 24.2 shows how those headings become the page. Sketching the layout on paper before you code is a good habit.

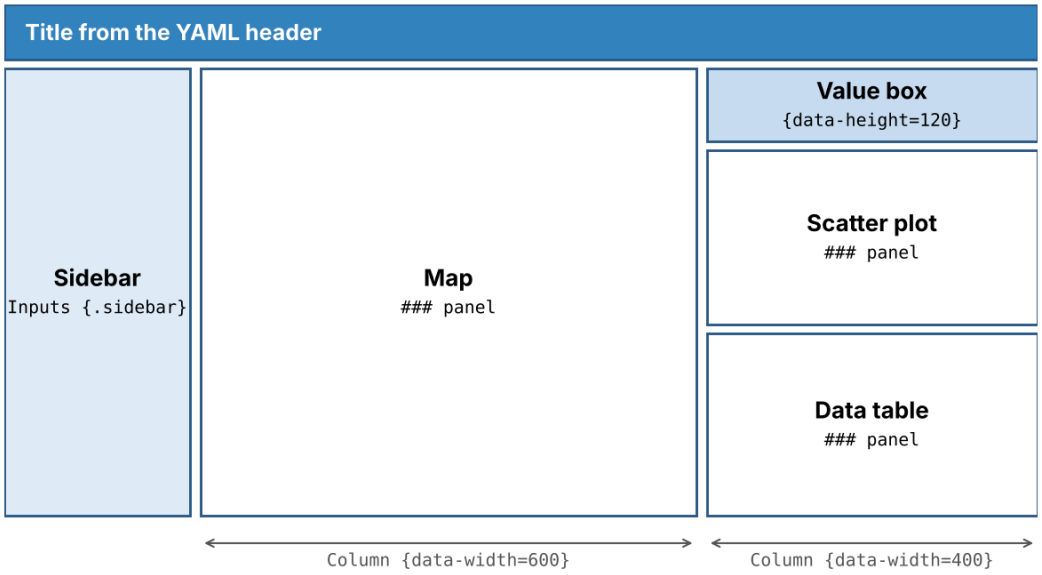


Figure 24.2: How the headings above lay out the dashboard: a sidebar, a wide map column, and a narrower column that stacks a value box, a scatter plot, and a data table.

24.4.3 Value boxes, tables, and charts

Each of the other panels gets one of these:

```
# Headline number
valueBox(nrow(states), caption = "States in the map", icon = "fa-map")

# Searchable, sortable table
datatable(state_table, rownames = FALSE, options = list(pageLength = 5))

# Interactive scatter plot
plot_ly(state_table, x = ~median_income_15, y = ~poverty_rate_15,
        text = ~NAME, type = "scatter", mode = "markers")
```

A **value box** shows a single headline number with an icon from Font Awesome. `datatable()` from DT makes a table readers can search and sort, and `plot_ly()` makes a scatter plot with hover labels. The table and chart use `state_table`, a copy of the data without geometry, made with `st_drop_geometry()` and `select()`. Tables and charts don't need polygons, and dropping the geometry keeps the file small.

24.4.4 Linking views with crosstalk

crosstalk makes the views talk to each other:

```
library(crosstalk)

# Same key and group = linked widgets
shared_table <- SharedData$new(state_table, key = ~NAME, group = "states")

# Filters go in the sidebar
filter_checkbox("region", "Census region", shared_table, ~REGION)
```

```
filter_slider("income", "Median income ($)", shared_table, ~median_income_15)
```

Wrap a data frame in `SharedData$new()`, with a **key** that identifies each row, here the state name, and a **group** name. Pass `shared_table` instead of `state_table` to `datatable()` and `plot_ly()`, and every widget built from a `SharedData` object in the same group is linked: filtering one filters the others, and selecting points in the scatter plot highlights the same states in the table. `filter_checkbox()` and `filter_slider()` make filter controls that work without Shiny; they go in the sidebar.

24.4.5 crosstalk and leaflet: points, not polygons

Here is a limitation you will run into: leaflet links only point layers, markers and circles, with crosstalk. A crosstalk filter won't hide polygons, so a region filter can't hide states on a choropleth.

A simple workaround is one point per state. Draw the choropleth normally as context, then add the points as a shared layer on top:

```
state_points <- states |>
  st_transform(5070) |>        # find points in a projected CRS...
  st_point_on_surface() |>
  st_transform(4326) |>        # ...then go back to longitude and latitude
  select(NAME, REGION, median_income_15, poverty_rate_15)

shared_points <- SharedData$new(state_points, key = ~NAME, group = "states")

# in the map chunk, after addPolygons():
addCircleMarkers(data = shared_points, radius = 5, label = ~NAME)
```

`st_point_on_surface()` is better than `st_centroid()` here, because its point always falls inside the polygon. Run it on projected data, such as EPSG:5070, and transform the points back to 4326 afterward.

Figure 24.3 shows the result. The reader has checked only the South: the points on the map, the scatter plot, and the table all show only the 17 southern states. The choropleth colors and the value box didn't change, because value boxes are fixed when you knit. That is the static dashboard's limit. Even so, this single HTML file is already a useful, shareable product.

> **Think about it**
>
> In Figure 24.3, a reader filtered to the South but the value box still says 49. Is that misleading? How could you change the dashboard's labels so a reader isn't confused, without switching to Shiny?

24.5 Adding Shiny

24.5.1 When do you need Shiny?

Use **Shiny** when a reader's choice should trigger new R calculations, not just hide data:

- Filter the polygons themselves on the map.

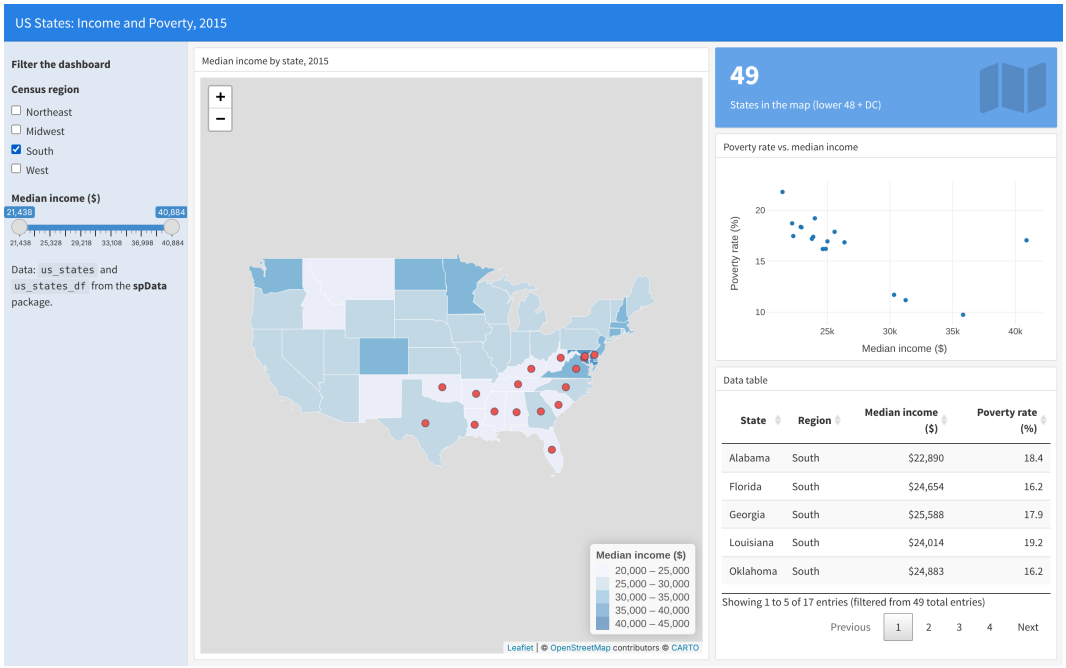


Figure 24.3: The static dashboard filtered to the South. The red points, scatter plot, and table show 17 states; the choropleth and value box are unchanged.

- Update headline numbers, such as an average for the selected states.
- Let readers choose which variable to map.
- Rerun a model or query an API with the reader's choices.

In a flexdashboard, add one line, `runtime: shiny`, to the YAML header. Then you need three pieces: **inputs** that collect the reader's choices, **reactive expressions** that recompute data, and **render functions** that redraw the outputs.

24.5.2 Inputs in the sidebar

```
variables <- c("Median income, 2015 ($)" = "median_income_15",
               "Poverty rate, 2015 (%)"  = "poverty_rate_15",
               "Population, 2015"        = "total_pop_15")

selectInput("variable", "Map variable", choices = variables)

checkboxGroupInput("region", "Census region",
                   choices  = levels(states$REGION),
                   selected = levels(states$REGION))
```

`selectInput()` makes a drop-down menu. The named vector shows readers friendly labels while giving your code the column names. `checkboxGroupInput()` makes the region checkboxes, all checked to start. Every input has an ID, its first argument, and anywhere else in the dashboard `input$variable` or `input$region` holds the reader's current choice.

24.5.3 Reactive data and render functions

```
selected_states <- reactive({
  req(input$region)
  states |> filter(REGION %in% input$region)
})

renderLeaflet({
  df  <- selected_states()
  val <- df[[input$variable]]
  pal <- colorBin("Blues", domain = val, bins = 5)
  leaflet(df) |> addPolygons(fillColor = pal(val))
})
```

A **reactive expression** is code that reruns automatically when the inputs it uses change. `selected_states` filters the data to the checked regions, and `req()` stops it from running if no region is checked. You call it like a function: `selected_states()`.

Render functions, such as `renderLeaflet()`, `renderValueBox()`, and `renderDT()`, draw outputs. Because this `renderLeaflet()` uses both `selected_states()` and `input$variable`, Shiny redraws the map whenever either changes. You don't write any code to watch the inputs; Shiny tracks the dependencies for you. That is the core idea of reactive programming. (The full app, at the end of the chapter, also adds basemap tiles, labels, and a legend.)

Figure 24.4 shows the Shiny version. The reader chose poverty rate and unchecked the Northeast and West. Now the polygons themselves are filtered, the color classes are recalculated for just these states, and the value boxes show 29 states and a population-weighted poverty rate of 15.6% for them.

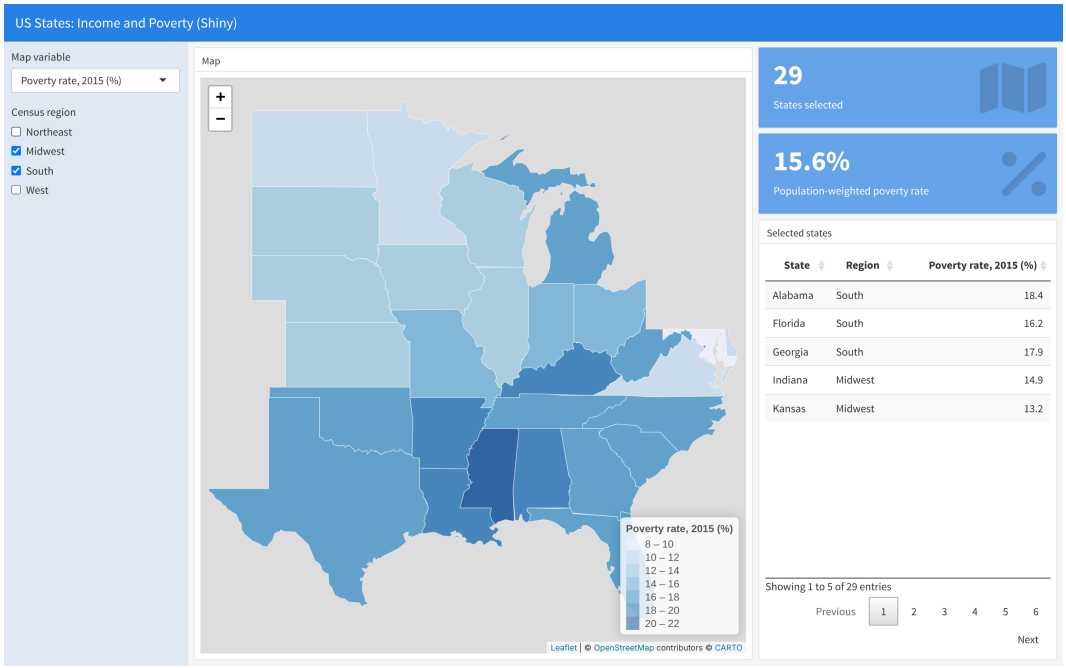


Figure 24.4: The Shiny dashboard mapping poverty rate for the Midwest and South. The polygons, the value boxes, and the table all recalculate.

24.6 Sharing your dashboard

Table 24.4: How to share each kind of dashboard.

Static (crosstalk)	Shiny
Knit to one self-contained HTML file	Needs R running on a server
Email it or upload it to Canvas	shinyapps.io or Posit Connect Cloud (free tiers)
Host it free on GitHub Pages or RPubs	Or a campus or company Shiny server
Open it in any browser	Run it locally with RStudio's Run Document button

24.6.1 Common pitfalls and fixes

- **The map is blank or in the wrong place:** transform to EPSG:4326 first.
- **The file is huge or slow:** simplify the polygons and drop columns you don't need. Use `st_simplify()` on projected data, or `rmapshaper::ms_simplify()`.
- **The legend and map disagree:** use the same palette function for both.
- **Missing values:** set `na.color` in the palette, and say what gray means.
- **A filter doesn't hide polygons:** crosstalk only filters leaflet points.
- **The design isn't accessible:** add labels and a table; never rely on color alone.

24.6.2 Looking ahead: Quarto dashboards

Quarto, the successor to R Markdown (and what built this book), has its own dashboard format: `format: dashboard` in the YAML header. The ideas are the same: rows, columns, cards instead of panels, value boxes, and a sidebar. It works with R and Python, and with leaflet, DT, plotly, and Shiny, so everything in this chapter carries over.

24.7 The complete example files

These are the two complete, tested dashboards from this chapter. Copy one into a new `.Rmd` file in RStudio and knit it (the static version) or click Run Document (the Shiny version).



states-dashboard.Rmd: the static dashboard

````
---
title: "US States: Income and Poverty, 2015"
output:
  flexdashboard::flex_dashboard:
    orientation: columns
    vertical_layout: fill
---

```{r setup, include=FALSE}
library(flexdashboard)
library(sf)
library(tidyverse)
library(spData)
library(leaflet)
library(DT)
library(plotly)
library(crosstalk)

# Join state polygons to the income and poverty table, then compute rates
states <- us_states |>
  left_join(us_states_df, by = c("NAME" = "state")) |>
  mutate(
    REGION = fct_recode(REGION, Northeast = "Norteast"),  # fix a typo in spData
    poverty_rate_15 = poverty_level_15 / total_pop_15 * 100,
    pop_change = (total_pop_15 - total_pop_10) / total_pop_10 * 100
  ) |>
  st_transform(4326)   # leaflet expects WGS84 longitude/latitude

# A plain table (no geometry) for the chart and the data table
state_table <- states |>
  st_drop_geometry() |>
  select(NAME, REGION, median_income_15, poverty_rate_15)

# One point per state: leaflet can filter points with crosstalk, not polygons
# (find points in a projected CRS, then go back to lon/lat)
state_points <- states |>
  st_transform(5070) |>
  st_point_on_surface() |>
  st_transform(4326) |>
  select(NAME, REGION, median_income_15, poverty_rate_15)

# SharedData objects in the same group are linked
shared_points <- SharedData$new(state_points, key = ~NAME, group = "states")
shared_table  <- SharedData$new(state_table,  key = ~NAME, group = "states")
````
````



states-dashboard-shiny.Rmd: the Shiny dashboard

````
---
title: "US States: Income and Poverty (Shiny)"
output: flexdashboard::flex_dashboard
runtime: shiny
---

```{r setup, include=FALSE}
library(flexdashboard)
library(shiny)
library(sf)
library(tidyverse)
library(spData)
library(leaflet)
library(DT)

states <- us_states |>
  left_join(us_states_df, by = c("NAME" = "state")) |>
  mutate(
    REGION = fct_recode(REGION, Northeast = "Norteast"),
    poverty_rate_15 = poverty_level_15 / total_pop_15 * 100
  ) |>
  st_transform(4326)

# Choices for the variable menu: label shown = column name used
variables <- c("Median income, 2015 ($)" = "median_income_15",
               "Poverty rate, 2015 (%)"  = "poverty_rate_15",
               "Population, 2015"        = "total_pop_15")
```

Inputs {.sidebar}
-----------------------------------------------------------------------

```{r inputs}
selectInput("variable", "Map variable", choices = variables)

checkboxGroupInput("region", "Census region",
                   choices = levels(states$REGION),
                   selected = levels(states$REGION))
```

```{r reactive-data}
# Re-runs only when the region checkboxes change
selected_states <- reactive({
  req(input$region)
  states |> filter(REGION %in% input$region)
})
```

Column {data-width=650}
-----------------------------------------------------------------------

### Map

```{r map}
renderLeaflet({
  df  <- selected_states()
  val <- df[[input$variable]]
````
````

24.8 Summary

- A **dashboard** puts a map, charts, a table, and key numbers on one interactive screen, with **linked views**. A report tells a story in order; a dashboard lets the reader explore.
- **leaflet** builds the map, in layers joined with the pipe: **basemap tiles**, polygons colored with a **palette function**, a legend, **labels** on hover and **popups** on click. `tmap_mode("view")` makes the same kind of map.
- **flexdashboard** lays out an R Markdown file as a dashboard: a **sidebar**, **columns**, and **panels**, with **value boxes**, DT tables, and plotly charts.
- **crosstalk** links **htmlwidgets** in a single HTML file that anyone can open, using `SharedData` with a key and a group. It filters only leaflet points, not polygons.
- **Shiny** (`runtime: shiny`) reruns R code when readers make choices, with **inputs**, **reactive expressions**, and **render functions**, but it needs a server.
- Start simple and static; add Shiny only when you need it.

24.9 Exercises

1. **Questions first.** Before building anything, write the two or three questions a dashboard about North Carolina's counties should answer. Which views would answer each one?
2. **Map a change.** Start from `states-dashboard.Rmd`. Add a column for the change in median income from 2010 to 2015, and map it instead of 2015 income. Which palette function fits a variable that can be negative?
3. **Another headline.** Add a second value box that shows the highest-income state.
4. **Another filter.** Add a `filter_select()` to the static dashboard, or a new Shiny input to the Shiny version.
5. **tmap instead.** Replace the leaflet map with `tmap_mode("view")` and the tmap code from Section 24.3. What did you gain, and what did you lose?
6. **Usability test.** Knit your dashboard and give it to a classmate with your questions from Exercise 1. Can they answer them without asking you anything?

24.10 Further reading

- *flexdashboard for R*, the package documentation, at https://pkgs.rstudio.com/flexdashboard.
- *Leaflet for R*, at https://rstudio.github.io/leaflet. Covers basemaps, palettes, popups, and more.
- *Mastering Shiny*, by Hadley Wickham, free at https://mastering-shiny.org. The best introduction to inputs, outputs, and reactivity.
- *Interactive web-based data visualization with R, plotly, and shiny*, by Carson Sievert, free at https://plotly-r.com. Includes a chapter on linking views with crosstalk.
- Quarto dashboards, at https://quarto.org/docs/dashboards.

25 What's Next in Spatial Data Science

Learning objectives

By the end of this chapter you should be able to:

1. Name applications of spatial data science in public health, environmental science, and other fields, and the course methods each one uses.
2. Describe remote sensing, its sensors and four kinds of resolution, and how NDVI turns pixels into information.
3. Explain how space-time modeling extends Tobler's first law and spatial autocorrelation to time.
4. Describe GeoAI, and explain why spatial autocorrelation matters when you train and test machine learning models.
5. Plan your next GIS courses at UNC, and the GIS minor or Geospatial Data Science major.
6. Describe careers that use spatial skills, and how to start building a portfolio.

This chapter looks forward instead of teaching a new method. You can now do a full spatial data science workflow in R:

- **Get data:** CSV files, shapefiles, the Census API, and ArcGIS REST services (Chapters 13 and 14).
- **Wrangle it:** the tidyverse, the pipe, and table and spatial joins (Chapters 6–9 and 15).
- **Map it:** ggplot2 and tmap, with sensible coordinate systems (Chapters 9, 10, and 12).
- **Analyze it:** exploratory data analysis, ANOVA, regression, distance, neighbors, and spatial autocorrelation (Chapters 16, 18, 19, 21, and 22).
- **Communicate it:** reproducible R Markdown reports and interactive dashboards (Chapters 2, 20, and 23).

That is the core workflow of spatial data science. This chapter shows where it is used, previews three advanced topics that build on it, and maps out what to take next at UNC and where spatial skills can take you. As you read, notice which application excites you most. That is a good clue to which course to take next.

25.1 Spatial data science in action

Almost any question that involves "where" is a spatial question:

- Where are disease cases clustering?
- Which neighborhoods are farthest from a clinic or grocery store?
- How fast is forest being cleared, and where?
- Who lives in the flood zone?
- Where should a new store, bus route, or fire station go?

The methods are shared; the questions change by discipline. For each application below, notice which methods from this book show up: joins, distance, neighbors, autocorrelation, regression, or mapping.

25.1.1 Public health: where disease happens

Spatial epidemiology studies how health outcomes vary across places. Its classic origin story is John Snow's map of the 1854 Broad Street cholera outbreak in London, which tied deaths to a single water pump.

Today we don't just look at the map; we test whether a cluster is bigger than chance would produce. Local Moran's I and hot spot statistics such as Getis-Ord Gi* (Chapter 22) find unusual clusters of cases, and **scan statistics**, in software such as SaTScan, search through both space and time for outbreaks. During COVID-19, case maps and dashboards were updated daily, and wastewater testing added a new signal for tracking infection by sewershed. A sewershed is just another polygon you can join data to.

25.1.2 Public health: access to care

Access to care depends on distance, travel time, and who can make the trip. Chapter 21 compared straight-line, network, and travel-time distance, and health access is where that choice matters most: a clinic 10 miles away is not the same for someone with a car and someone who relies on a bus. Analysts draw drive-time and transit-time areas around hospitals, pharmacies, and clinics, and track how rural hospital and maternity unit closures lengthen trips for patients.

Accessibility models weigh supply against demand. The **two-step floating catchment area** (2SFCA) method is a common example: it combines how many providers there are, how many people need them, and how long it takes to get there.

25.1.3 Public health: environmental exposure and justice

Where people live shapes what they breathe, drink, and are exposed to. Researchers link air pollution monitors and satellite estimates to asthma rates, and measure distance to highways, factories, and hazardous waste sites. **Environmental justice** asks who bears that burden.

A well-known North Carolina example comes from UNC epidemiologists, including the late Steve Wing, who showed that industrial hog operations in eastern North Carolina were concentrated near Black, Latino, and low-income communities. The methods are ones you know: spatial joins, buffers, census data, and regression, controlling for spatial autocorrelation. Maps can reveal inequities, and spatial data science has real policy consequences.

25.1.4 Environmental science: watching the planet change

Satellites give a consistent, repeated view of the whole Earth. Hansen and colleagues (2013) used Landsat imagery to map forest loss worldwide at 30-meter resolution, and Global Forest Watch now serves that kind of data to the public. Satellite data also show land cover change, such as farms turning into suburbs or wetlands into development, and **urban heat islands**: paved, treeless neighborhoods that run hotter, which is a health issue too.

Ecologists use **species distribution models**, which are essentially regression with spatial predictors, to predict where plants and animals can live, and GPS collars and tags to record how animals move through landscapes.

25.1.5 Environmental science: hazards and resilience

GIS is central to preparing for and responding to disasters: flood maps, evacuation routes, and shelter locations before a storm, and rapid damage mapping from aerial and satellite imagery after it. North Carolina has plenty of examples:

- Hurricane Helene's 2024 flooding in western North Carolina, when imagery and GIS were used to map flooding, landslides, and damaged roads.
- Sea-level rise and shoreline change along the Outer Banks.
- Statewide lidar elevation data, collected for floodplain mapping.

The key question, who is exposed, is an overlay of hazard zones with census data: exactly the kind of spatial join you did in Chapter 15.

25.1.6 Cities, transportation, and planning

Urban and regional planners use GIS every day, for zoning, land use, and housing affordability. Transportation questions are almost always network questions. Who lives within a 10-minute walk of a bus stop? Where are the bike and pedestrian crash hot spots?

Choosing where to put new schools, parks, or fire stations is a classic GIS problem. **Location–allocation** models take demand locations and candidate sites and choose the sites that minimize travel or serve the most people. Network distance, from Chapter 21, is the core tool.

25.1.7 Social science, history, and politics

- **History.** The Mapping Inequality project digitized the 1930s Home Owners' Loan Corporation "redlining" maps, including several North Carolina cities such as Durham. Researchers join those historic polygons to today's data on health, heat, and wealth.
- **Politics.** Redistricting and gerrymandering are spatial problems by definition, and North Carolina has been at the center of many redistricting court cases. Compactness measures and simulated district maps test whether a plan is fair.
- **Archaeology.** Lidar sees through forest canopy to reveal buried sites.
- **Sociology and economics.** Segregation, mobility, and neighborhood effects are studied with neighbor-based measures like the weight matrices from Chapter 21.

25.1.8 Business, agriculture, and humanitarian work

- **Business.** Site selection, delivery routing, market areas, and real estate, often called **location intelligence**.
- **Agriculture.** Precision farming with GPS-guided equipment, drones, and crop imagery puts water and fertilizer only where they are needed.
- **Humanitarian mapping.** Volunteers trace roads and buildings in OpenStreetMap, and responders use those maps after earthquakes, floods, and conflicts. Anyone can contribute, including you, this week.

- **Insurance and finance.** Insurers price risk from floods, wildfire, and climate change with hazard models.

> **Think about it**
>
> Pick the application in this section that interests you most. Which chapters of this book would you need for a first project on it? What data would you need, and who publishes them?

25.2 Remote sensing: seeing Earth from above

The next three sections preview advanced areas: remote sensing, space-time modeling, and GeoAI. Each is a full course or career on its own, so the goal is vocabulary and a sense of what is possible, not mastery.

Remote sensing is collecting information about Earth without touching it, and it produces most of the raster data in the world (Chapter 11). Sensors come in two kinds:

- **Passive sensors** record reflected sunlight or the heat a surface gives off, like a camera or a multispectral scanner.
- **Active sensors** send their own signal and measure what comes back. **Radar** can see through clouds and at night, and **lidar** measures distance with laser pulses to build 3D models of terrain and trees.

25.2.1 Four kinds of resolution

Every sensor is described by four kinds of resolution:

- **Spatial resolution:** the size of a pixel on the ground.
- **Spectral resolution:** the number and width of wavelength bands it records.
- **Temporal resolution:** how often it revisits the same place.
- **Radiometric resolution:** how finely it records brightness.

Each sensor trades these off. A satellite with tiny pixels usually covers a smaller area or revisits less often. Choosing data means choosing the resolution that fits your question, just as Chapter 21 chose the distance measure that fits the question.

Table 25.1: Remote sensing data you will meet most often. Pixel sizes are approximate.

Sensor	Type	Pixel size	Good for
Landsat (since 1972)	Passive	30 m	Decades of change
Sentinel-2	Passive	10–60 m	Crops, forests, cities
MODIS	Passive	250 m–1 km	Daily fires and snow
Sentinel-1, NISAR	Active radar	About 5–20 m	Floods, ground motion
Lidar	Active laser	Under 1 m	Elevation, tree height
Drones	Mostly passive	Centimeters	Small study sites

Landsat is the longest continuous record of Earth's surface, which makes it ideal for change over decades. Sentinel-2, from the European Space Agency, has finer pixels and frequent

revisits. MODIS is coarse but images the whole planet about every day. Radar, from Sentinel-1 and the NASA–ISRO NISAR mission launched in 2025, works through clouds, so it maps floods during storms. Most of these data are free from NASA, the USGS, and the European Space Agency, a big change from twenty years ago.

25.2.2 From pixels to information

Different surfaces reflect different wavelengths of light, so each has a **spectral signature**. Healthy plants absorb red light, because chlorophyll uses it, and reflect near-infrared (NIR) light strongly. The **normalized difference vegetation index** (**NDVI**) turns that contrast into one number for each pixel:

$$\text{NDVI} = \frac{\text{NIR} - \text{Red}}{\text{NIR} + \text{Red}}$$

NDVI ranges from –1 to 1, and dense green vegetation is high (Figure 25.1).

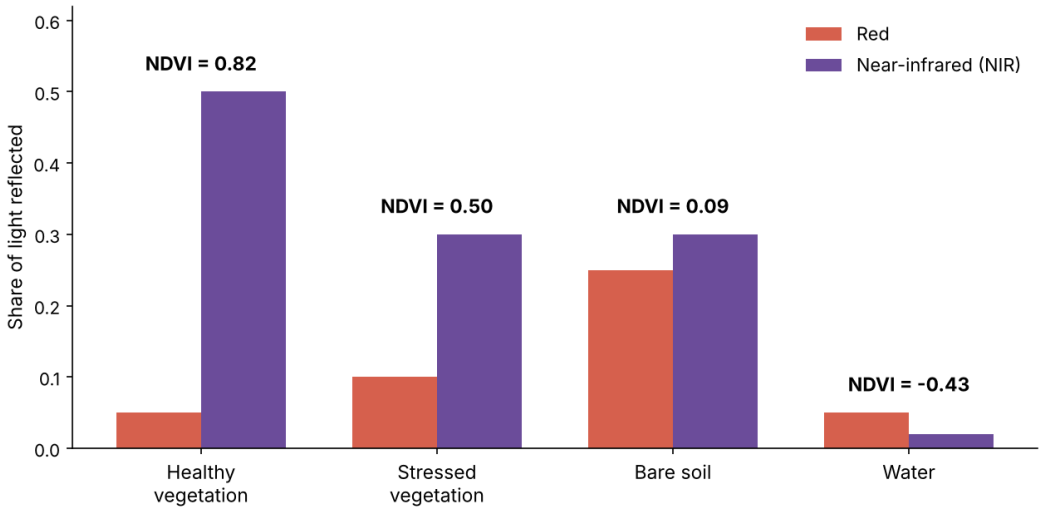


Figure 25.1: Typical shares of red and near-infrared light reflected by four surfaces, and the NDVI each gives. Values are illustrative.

Two more steps turn imagery into maps:

- **Classification** assigns each pixel a land cover class, such as forest, water, or developed land, increasingly with machine learning.
- **Change detection** compares images of the same place from different dates.

In R, the **terra** and **stars** packages handle rasters. NDVI is two lines of raster algebra in terra. Google Earth Engine, taught in GEOG 567, runs the same analyses in the cloud on petabytes of imagery, without downloading anything.

25.3 Space-time modeling

Almost every dataset in this book was a snapshot: one ACS period, or one year. But real processes change over both space and time:

- Disease spreading from county to county.
- Air pollution changing by the hour and by neighborhood.



- Cities growing over decades.

Tobler's first law extends to time: nearby moments are related, just as nearby places are. A useful mental model is the **space-time cube**: a map for every time step, stacked on top of each other, with x, y, and time as its three axes.

Some space-time methods to explore:

- **Emerging hot spot analysis** runs a hot spot statistic, like Gi* from Chapter 22, for every time step and classifies each location's trend: new, intensifying, persistent, or diminishing. The sfdep package has a function for it.
- **Space-time scan statistics** detect disease outbreaks early, and are widely used in disease surveillance.
- **Spatiotemporal kriging** predicts values at places and times with no measurements, for example air pollution exposure.
- **Movement and trajectory analysis** works with GPS tracks of people, vehicles, and animals.

In R, look at the **sfdep**, **stars**, **sftime**, and **gstat** packages. At UNC, ENEC 468, Temporal GIS and Space/Time Geostatistics for the Environment and Public Health, focuses on these methods.

25.4 GeoAI

GeoAI is artificial intelligence applied to spatial data, and it is one of the fastest-growing parts of the field.

- **Deep learning reads imagery at massive scale.** Models trained to recognize objects in images have extracted building footprints for whole countries (Microsoft released its global footprints as open data), and map roads, rooftop solar panels, crops, and flood extent. Work that once took armies of people digitizing by hand is now automatic.
- **Machine learning predicts.** Random forests and gradient boosting, with spatial predictors, estimate air quality, poverty, or crop yield in places with no measurements.

25.4.1 Foundation models and AI assistants

Geospatial foundation models are large models pretrained on huge image archives, which can then be adapted to many tasks. In 2023 NASA and IBM released Prithvi, an open model trained on Landsat and Sentinel-2 imagery, which can be fine-tuned to map floods, burn scars, or crop types with little labeled data. Segmentation models can outline buildings or fields with minimal training.

Large language models can help write R code, explain error messages, and geocode place names in text. They are most useful to people who know enough to check the answer, such as noticing that a model used the wrong coordinate system. The course AI policy asks you to use them that way.

25.4.2 Spatial thinking keeps AI honest

This is where this book pays off directly. Most machine learning assumes observations are independent, and spatial data are not: you measured that with Moran's I in Chapter 22.

- **Random train–test splits mislead.** If you split spatial data at random, near-duplicate neighbors end up in both the training set and the test set. Accuracy looks great, and then the model fails in new places.
- **Spatial cross-validation** holds out whole regions instead, so the test really is a new place. The R packages spatialsample and blockCV do this.
- **Training data can be biased** toward wealthy or well-mapped places, so models do worst where better data are most needed.
- **Location data raise privacy concerns.** A few GPS points can be enough to identify a person.

Understanding spatial data is what separates someone who can run an AI model from someone who can trust its results.

> **Think about it**
>
> A model predicts poverty from satellite images of North Carolina's census tracts. It is trained on a random 80% of tracts and scores 95% accuracy on the other 20%. Why might it do much worse in South Carolina? How would you test it more honestly?

25.4.3 Scaling up: cloud, big data, and open science

Datasets are growing faster than laptops. Planet-wide imagery and billions of GPS points don't fit on one computer, so analysis is moving to the cloud, where the data sit next to the computing:

- **Cloud platforms:** Google Earth Engine, Microsoft Planetary Computer, and AWS open data.
- **Spatial databases: PostGIS** adds spatial queries, such as spatial joins, to SQL.
- **Cloud-ready formats:** Cloud Optimized GeoTIFF and GeoParquet let you read only the piece of a file you need.

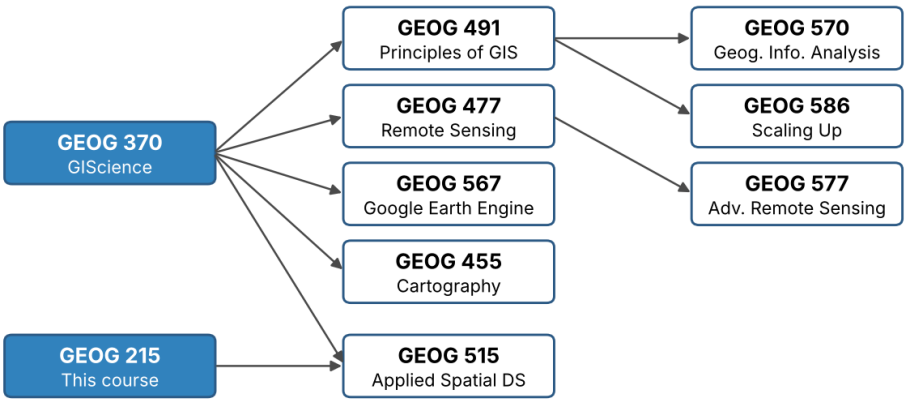
Reproducibility (Chapter 20) matters more as projects grow. The habits you built with R Markdown scale up to Quarto (which built this book), Git, and GitHub, with shared code and data. GEOG 586 Scaling Up Geospatial Analysis and GEOG 592 GIS Programming go deeper.

25.5 Keep learning at UNC

Course titles and requirements in this section come from the 2026–27 UNC undergraduate and graduate catalogs. Always check the catalog and ConnectCarolina for when each course is offered, and meet with a Geography advisor before you plan.

25.5.1 A path through UNC GIS courses

GEOG 215, this course, and GEOG 370, Geographic Information Science, are the two core courses of the GIS minor, so you are already halfway through the core. GEOG 370 is the gateway to most upper-level GIS courses (Figure 25.2). GEOG 515, Applied Spatial Data Science, needs both 215 and 370, and picks up where this book stops: spatial regression, clustering, and interpolation in R.



Filled boxes: the two core courses of the GIS minor. Arrows: prerequisites in the UNC catalog. GEOG 570 also requires GEOG 391, and GEOG 577 can follow GEOG 370 directly.

Figure 25.2: How UNC's GIS courses connect. Filled boxes are the GIS minor's two core courses; arrows are prerequisites in the UNC catalog.

25.5.2 Geography courses by interest

Table 25.2: Geography Department GIS courses by interest. Titles are shortened in a few places.

If you liked...	Take
R and spatial statistics	GEOG 515 Applied Spatial Data Science; GEOG 570 Geographic Information Analysis
Remote sensing	GEOG 477 Introduction to Remote Sensing; GEOG 577 Advanced Remote Sensing; GEOG 567 Google Earth Engine
Health	GEOG 541 GIS in Public Health; GEOG 544 GIS for Impact Evaluation and Health Studies

If you liked...	Take
GIS and programming	GEOG 491 Principles of GIS; GEOG 591 Applied Issues in GIS; GEOG 592 GIS Programming; GEOG 586 Scaling Up Geospatial Analysis
Making maps	GEOG 455 Cartography and Geovisualization; GEOG 456 Geovisualizing Change; GEOG 555 Cartography of the Global South
Field data	GEOG 594 GPS and Applications

25.5.3 GIS courses beyond Geography

GIS is taught across campus. These courses all count as electives in the GIS minor, the Geospatial Data Science major, or the graduate GIS certificate:

- **Environment, Ecology, and Energy:** ENEC 468 Temporal GIS and Space/Time Geostatistics for the Environment and Public Health; ENEC 479 Landscape Analysis; ENEC 340 Environmental Applications of Drones.
- **Earth, Marine, and Environmental Sciences:** EMES 483 GIS and Remote Sensing for Earth and Ocean Sciences.
- **Anthropology:** ANTH 419 Anthropological Application of GIS.
- **Information and Library Science:** INLS 541 Information Visualization; INLS 641 Visual Analytics.
- **City and Regional Planning:** PLAN 672 Urban Data Analytics (graduate).

25.5.4 Turn your interest into a credential

- **GIS minor (15 credit hours).** The core is GEOG 215 and GEOG 370, plus three GIS electives, for example GEOG 455, 477, 491, 541, or 577. It pairs well with any major, and you have already finished GEOG 215.
- **Geospatial Data Science B.S.** A full major. Its core is GEOG 215, 370, 391, 392, 455, 477, 491, and 515, plus math, statistics, or data science courses, GIS electives, and a capstone.
- **Graduate Certificate in Geographic Information Science (18 credit hours),** for graduate students.

25.6 Careers in GIS and spatial data science

Whether or not you major in geography, spatial skills make you more employable in many fields. Job titles vary a lot, so search broadly:

- **GIS analyst or technician:** builds, maintains, and analyzes spatial data. The most common entry point.

- **Geospatial data scientist:** statistics, machine learning, and code at scale.
- **Remote sensing analyst:** turns imagery into maps and measurements.
- **Cartographer or geovisualization designer:** maps, dashboards, and web maps.
- **Geospatial developer:** web mapping apps, spatial databases, and APIs.
- **Domain roles that use GIS:** planner, spatial epidemiologist, environmental consultant, emergency manager.

Many of you will end up in the last group, using GIS as a core skill in another field. The U.S. Bureau of Labor Statistics projects much faster than average growth for data scientists, and spatial skills set you apart within that field.

25.6.1 Where geospatial professionals work

- **State and local government:** nearly every North Carolina city and county has GIS staff, as do NCDOT and the NC Center for Geographic Information and Analysis, which runs NC OneMap.
- **Federal agencies:** USGS, NOAA, NASA, the Census Bureau, CDC, and EPA, which has a large campus in Research Triangle Park.
- **Research and nonprofits:** universities, RTI International, and conservation groups.
- **Private sector:** engineering and environmental consulting, utilities, real estate, tech, and insurance.
- **Software and data companies:** Esri and other mapping, imagery, and analytics firms.

Look at job boards for these employers to see which skills they ask for.

25.6.2 Skills employers look for

Table 25.3: Skills that geospatial job postings ask for.

Technical skills	Professional skills
R or Python for spatial analysis	Clear maps and visual communication
ArcGIS Pro and QGIS	Writing for non-technical audiences
SQL and PostGIS	Asking good spatial questions
Statistics and spatial statistics	Data ethics and privacy
Remote sensing and cloud tools	Working on teams and managing projects
Git and reproducible workflows	

You already have R, spatial statistics, and reproducible reports from this book. Adding ArcGIS Pro or QGIS, through GEOG 370 or 491, makes a strong combination. Don't underestimate the second column: employers consistently say they need people who can explain results clearly, make a good map, and think carefully about ethics.

25.6.3 Start building your portfolio now

You don't need to wait until graduation. Two or three polished projects matter more than a list of courses.

- Put your best maps and R projects on GitHub or a personal website. Your final project, cleaned up with a clear write-up and map, is a strong first piece.
- Contribute to OpenStreetMap or a humanitarian mapping task.

- Look for internships with city, county, and state GIS offices, state agencies, and research centers.
- Join a professional community, such as the NC GIS Conference, URISA, or the American Association of Geographers. Most have student rates.
- Later in your career, consider the **GISP** certification from the GIS Certification Institute and Esri's technical certifications.
- Keep learning: free online books, such as *Geocomputation with R*, pick up right where this book ends.

Think about it

Search a job board for "GIS analyst" in North Carolina. Pick two postings. Which of the skills they ask for do you already have, and which course or project would give you the rest?

25.7 Summary

- Spatial data science answers "where" questions in nearly every field: **spatial epidemiology**, **accessibility** to care, **environmental justice**, environmental change and hazards, planning, history, politics, business, and humanitarian mapping. The methods are the ones in this book.
- **Remote sensing** collects information about Earth from a distance, with **passive** or **active** sensors, each with a **spatial**, **spectral**, **temporal**, and **radiometric** resolution. **NDVI** uses the contrast between red and near-infrared light to measure vegetation; **classification** and **change detection** turn imagery into maps.
- **Space-time modeling** extends Tobler's first law to time. The **space-time cube**, **emerging hot spot analysis**, and **spatiotemporal kriging** are starting points.
- **GeoAI** reads imagery and makes predictions at massive scale, and **foundation models** adapt to new tasks with little labeled data. Because of spatial autocorrelation, honest testing needs **spatial cross-validation**.
- At UNC, take GEOG 370 next, then electives or GEOG 515, and consider the GIS minor or the Geospatial Data Science B.S.
- Spatial skills are in demand in government, research, and the private sector. Start a portfolio now.

25.8 Exercises

1. **Spot the methods.** Choose one application from Section 25.1. List every method from this book it uses, with the chapter where you learned it.
2. **Compute NDVI.** A pixel reflects 0.08 of red light and 0.40 of near-infrared light. Compute its NDVI by hand, then in R. Is it more likely forest, bare soil, or water? Why?
3. **Pick a sensor.** For each question, choose a sensor from Table 25.1 and justify it with one kind of resolution: (a) How much did Wake County's developed land grow from 1990 to 2025? (b) Which roads flooded during last night's storm? (c) How tall are the trees in Duke Forest?
4. **Space-time question.** Write a research question about North Carolina that needs a space-time method. What would one time slice of your space-time cube be?
5. **Plan your path.** Using Figure 25.2 and Table 25.2, write a four-semester plan of GIS courses that fits your interests, and check each course's prerequisites in the UNC catalog.



6. **Portfolio piece.** Choose your best map or analysis from this semester. Write a one-paragraph description of it for a non-technical reader, as you would on a personal website.

25.9 Further reading

- *Geocomputation with R* (2nd ed.), by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, free at https://r.geocompx.org. Picks up where this book ends, with rasters, spatial cross-validation, and more.
- Hansen, M. C., et al. (2013). "High-Resolution Global Maps of 21st-Century Forest Cover Change." *Science* 342(6160): 850–853. The forest change study described in this chapter.
- Mapping Inequality: Redlining in New Deal America, from the University of Richmond's Digital Scholarship Lab, at https://dsl.richmond.edu/panorama/redlining.
- UNC Department of Geography and Environment, programs and courses in the UNC catalog, at https://catalog.unc.edu.

Glossary

Key terms used in this book. In the chapters, a term with a dotted underline shows its definition when you hover over it, tap it, or reach it with the Tab key.

1-year estimates ACS estimates from 12 months of responses. They are the most current but are only published for areas of 65,000 people or more.

5-year estimates ACS estimates that pool 60 months of responses. They are the most reliable ACS product and the only one for tracts and block groups. An estimate labeled 2022 describes 2018 to 2022.

accessibility How easily people can reach a service, such as a clinic, given the distance or travel time, how many providers there are, and how many people need them.

active sensor A sensor that sends out its own signal and measures what comes back, such as radar or lidar.

adjusted R-squared R-squared lowered a little for each independent variable, so adding useless variables can't make a model look better. Use it to compare models.

aesthetic A visual property of a mark, such as its x position, y position, color, fill, size, or shape. In ggplot2, `aes()` connects columns to aesthetics.

alternative hypothesis The claim that there is a difference or relationship, such as "at least one group mean differs".

American Community Survey (ACS) The Census Bureau's continuous survey of a sample of households every year. It asks many more questions than the decennial census, but every value is an estimate with a margin of error.

ANOVA Analysis of variance: a statistical test of whether the averages of three or more groups differ, such as median age in three regions. It compares variation between groups with variation within them.

API Application Programming Interface: a set of rules for how one program asks another for data or services, such as your R script asking the Census Bureau's server for a table.

API key A private code that identifies you to an API. Store it in `.Renviron`, never in a script you share.

apportionment Dividing the seats in the House of Representatives among the states by population, the reason the Constitution requires a census.

ArcGIS Online Esri's systems for publishing and sharing maps and GIS layers on the web. Many agencies' data portals run on them.

arcpullr An R package that downloads layers from ArcGIS REST services as sf objects, with `get_spatial_layer()`.

argument An input to a function, written inside its parentheses.

ascending order Smallest to largest for numbers, A to Z for text. `arrange()` sorts in ascending order unless you wrap the column in `desc()`.

assignment operator `<-`, the arrow that stores the value on its right in the name on its left. Read `x <- 5` as "x gets 5."

attribute A non-spatial fact about a spatial feature, such as a tree's species or a building's year built. Attributes are the columns of a layer's attribute table.

attribute join A join that matches rows using a shared key column.

Average Nearest Neighbor ratio The observed mean distance from each point to its nearest neighbor divided by the mean expected at random. Below 1 is clustered.

azimuthal Describes a projection that preserves direction from a center point. Also called planar.



bar chart A chart that compares counts or amounts across categories with bars. `geom_bar()` counts rows for you; `geom_col()` draws heights you give it.

basemap A background map of streets, places, or imagery, loaded from the web as tiles, that your own data are drawn on top of.

benchmark A surveyed point marked on the ground, often by a brass disk, whose position is precisely known. Datums are made real by thousands of them.

big data Data too large, fast, or varied for ordinary tools, often described by three Vs: volume, velocity, and variety.

bimodal Having two peaks. It often means two different groups are mixed together.

bin width The width of each bar in a histogram. Too narrow shows noise; too wide hides detail.

binary weights Neighbor weights of 1 for a neighbor and `0` for everything else.

block (of code) Lines of code inside curly braces, `{ }`, that R runs as one unit. `if` statements, loops, and functions each control a block.

block group A group of census blocks, usually holding `600` to 3,`000` people. It is the smallest geography with ACS 5-year estimates.

block of code Several lines of code that work together on one task, like the steps of one recipe. In R Markdown, a block usually lives in one code chunk.

Boolean Another name for logical (`TRUE` or `FALSE`) values, after the mathematician George Boole.

boxplot A chart that summarizes a distribution: the box spans the middle half of the values, a line marks the median, and dots beyond the whiskers are outliers.

camelCase A way of naming variables that joins words and capitalizes every word after the first, as in `popChapelHill`.

cardinality How many rows in one table can match a row in another: one-to-one, one-to-many, many-to-one, or many-to-many.

case-sensitive Treating capital and lowercase letters as different. In R, `Months` and `months` are two different names.

cell One square of a raster's grid. Each cell holds exactly one value.

cell size The length of one side of a raster cell in real-world units, such as `30` m. It is the raster's resolution.

census block The smallest Census geography, often a single city block bounded by streets or other visible features. Only decennial census data are published for blocks.

census tract A small area of about 1,`200` to 8,`000` people (4,`000` on average), drawn to be fairly stable over time. The workhorse unit for neighborhood analysis.

central feature The existing feature with the smallest total distance to all the others: the spatial counterpart of the median.

central tendency One value that describes the center of a set of values: the mean, median, or mode.

centroid The geometric center of a polygon. `st_centroid()` in R.

change detection Comparing images of the same place from different dates to find what changed.

character The data type for text, also called strings. Character values always go inside quotation marks, as in `"Chapel Hill"`.

cheatsheet A one-page summary of a package or tool. Posit publishes them for dplyr, ggplot2, RStudio, and more.

choropleth map A map that shades areas, such as counties or buildings, by the value of a variable.

classification Dividing a numeric variable into a few classes for a map, by choosing how many classes there are and where the breaks between them fall.

Cloud Optimized GeoTIFF (COG) A GeoTIFF arranged in tiles with zoomed-out copies, so software can read only the area and detail it needs.

clustering Methods that find groups of similar observations, or groups of places with similar

values that sit close together.

code chunk A block of R code inside an R Markdown file, starting with ```` ```{r} ```` and ending with three backticks.

coefficient A number a regression estimates: the intercept, or the slope for each independent variable.

coefficient of variation (CV) The standard error as a percentage of the estimate. A CV above about 30% is a common sign that an estimate is unreliable.

coercion R converting values from one data type to another. Mixing types in one vector coerces them all to the type that can hold everything, usually character.

cold spot An area of low values surrounded by low values: a large negative Gi* z-score.

colon operator `:`, which makes a sequence of whole numbers: `6:8` is `c(6, 7, 8)`.

comment A note for people that R ignores. In R, a comment starts with #.

comparison operator An operator that compares two values and returns TRUE or FALSE: >, <, >=, <=, == (equal to), and != (not equal to).

compiler A program that translates a whole program into machine code before any of it runs. C and Java are compiled; R is interpreted.

complete spatial randomness Points or values placed entirely at random, the baseline that spatial pattern and autocorrelation statistics compare against.

compromise projection A projection, such as Robinson, that preserves no property exactly but keeps every distortion moderate.

confidence interval A range that is likely to contain the true value of something estimated from data. `geom_smooth()` shades a 95% confidence interval around its trend line.

confidence level How sure you can be that the true value lies within a range. The ACS uses a 90% confidence level for its margins of error.

conformal Describes a projection that preserves local shapes and angles, such as Mercator or Lambert Conformal Conic.

Console The RStudio pane (bottom left) where R runs commands and prints results, warnings, and errors. Nothing typed here is saved.

contiguity Sharing a border. Contiguity neighbors are polygons that touch, by the rook or queen rule.

continuous data Data with a value everywhere on the map that changes from place to place, such as elevation or temperature.

continuous weights Neighbor weights that vary, usually falling with distance, so closer neighbors count more.

coordinate system The rules that tie coordinates to places on the Earth, including how the round Earth is flattened onto a map (the projection). Layers must share a coordinate system before you can combine them.

correlation Whether two variables move together. It doesn't say how much one changes with the other, or which causes which.

correlation coefficient *r*, from –1 to 1: how closely two variables follow a straight line, and in which direction. `cor()` in R.

CORS Continuously Operating Reference Stations: permanent GPS stations across the US that the National Geodetic Survey uses to maintain NAD83.

CRAN The Comprehensive R Archive Network: the official website for downloading R and most R packages.

cross join A join that pairs every row of one table with every row of the other. `cross_join()` in dplyr.

crosstalk An R package that links htmlwidgets in one HTML file, so a filter or selection in one updates the others, with no Shiny.

CSV A plain-text table in which each line is a row and commas separate the values. Read one into R with `read.csv()` or `read_csv()`.

dashboard A web page that shows key information on one screen, usually as several linked, interactive views such as a map, charts, a table, and headline numbers.

data abstraction The process of making the real world readable by a computer: from reality, to a data model, to a data structure, to machine code.

data frame R's table: rows are observations and columns are variables. A tibble is the tidyverse's version of a data frame.

data model A formal way of representing geographic information in a computer, deciding which objects and properties to keep. Vector and raster are the two main ones.

data science workflow The steps of a data analysis: import, tidy, then a loop of transform, visualize, and model, and finally communicate.

data structure How a data model is actually laid out in memory or on disk, such as lists of coordinates and tables.

data type The kind of value R is storing. The four main atomic types are logical, integer, numeric, and character; `class()` tells you which you have.

datum A model that fixes an ellipsoid to the real Earth, so that a latitude and longitude point to one place on the ground. NAD83 and WGS84 are datums.

debugging Finding and fixing the mistakes (bugs) in code: read the error message, find the line, make a guess, and test it.

decennial census The full count of everyone in the US, every ten years. It asks only a few questions but covers every person.

decimal degrees Coordinates written as signed decimal numbers, such as –79.05°. Negative means south latitude or west longitude.

default value The value an argument takes when you don't supply one, set with = in the function definition: `function(x, factor = 1.609)`.

degrees of freedom The number of values free to vary in a calculation. In ANOVA, $k - 1$ between groups and $N - k$ within them.

degrees, minutes, and seconds Coordinates written as whole degrees, minutes (1/60 of a degree), and seconds (1/3,600 of a degree), with a letter N, S, E, or W, such as 79° 3′ 21″ W.

delimiter The character that separates values in a text file of data, such as a comma in a CSV file or a tab.

Demographic and Housing Characteristics (DHC) The 2020 census product with more detailed tables on age, sex, households, and housing than the redistricting data.

denominator The number you divide by to turn a count into a share, usually the total of the same table, such as total population.

density plot A smoothed histogram whose total area is 1. `geom_density()` in ggplot2.

dependent variable The outcome a model explains, *y*. It goes on the left of the ~ in an R formula.

descending order Largest to smallest for numbers, Z to A for text. Ask `arrange()` for it with `desc()`, as in `arrange(desc(dep_delay))`.

destination table In a join, the table that receives new columns: `x` in `st_join(x, y)`.

differential privacy Carefully calibrated random noise added to published 2020 census tables to protect respondents' privacy. It matters most for small areas such as blocks.

DIKW pyramid Data, information, knowledge, wisdom: four levels, each built on the one below, from raw facts to judging what is true and right.

discrete object Something that exists in one place and not another, with clear edges, such as a road, a house, or a state.

dispersion How spread out values are around their center, measured by the range, interquartile range, or standard deviation.

distance band A neighbor definition: every feature within a set threshold distance is a neighbor. `dnearneigh()` in spdep.

distance decay Interaction between places falls as the distance between them grows, because crossing distance costs time, money, or effort.

distance-decay parameter The power *x* in inverse distance weights, 1 / *d*^*x*. Larger values make weights fall off faster.

distribution How a variable's values are spread out: where they are common, where they are rare, and the shape that makes.

dpi Dots per inch: the resolution of a saved image. `300` dpi is print quality.

dplyr The tidyverse package for transforming data frames: filtering rows, choosing columns, adding new columns, joining tables, and summarizing.

dynamic data Data that are continually updated, so the same request may return different data later. Static data are a fixed snapshot.

edge effect Features near the edge of a study area get too few neighbors because the data stop at the boundary.

Editor The RStudio pane (top left) where you write and save scripts and R Markdown files. Code here can be kept and run again.

ellipsoid A sphere flattened at the poles: the smooth mathematical shape used to model the Earth, defined by its semi-major and semi-minor axes.

emerging hot spot analysis Running a hot spot statistic, such as Gi*, for every time step and classifying each location's trend: new, intensifying, persistent, or diminishing.

endpoint The web address of one dataset or service in an API, such as `https://api.census.gov/data/2022/acs/acs`

entity A real-world object or thing we want to record, such as a tree, a building, or a lake.

Environment The RStudio pane (top right) that lists the variables and data currently in memory.

environmental justice The fair sharing of environmental burdens, such as pollution, and benefits, such as parks, across communities, whatever their race or income.

EPSG code A number that identifies a coordinate reference system in a public registry, such as 4326 for WGS84 latitude and longitude.

equal intervals A classification method that makes every class cover the same range of values.

equal-area Describes a projection that preserves area, such as Albers or Mollweide. No projection is both equal-area and conformal.

Equator The circle halfway between the North and South Poles: `0`° latitude.

equidistant Describes a projection that preserves distance from one point, or along chosen lines.

Euclidean distance Straight-line distance, from the Pythagorean theorem. Use it only with projected coordinates.

exploratory data analysis (EDA) Learning what is in a dataset, with summaries and plots, before trying to explain or predict anything with it.

exploratory spatial data analysis (ESDA) EDA that also asks how values vary across space, using maps, spatial autocorrelation, and clustering.

extent The area a map or a layer covers, from its westernmost to easternmost and southernmost to northernmost points.

extrapolation Using a model to predict outside the range of the data it was fitted to. The relationship may not hold there.

F distribution The values of *F* you'd expect if the null hypothesis were true. Its shape depends on two degrees of freedom. `pf()` in R.

F statistic ANOVA's test statistic: the variance between groups divided by the variance within groups. Near 1 if the groups don't differ.

feature One spatial object in a layer of data, such as one tree, one sidewalk segment, or one building. In an sf object, each row is a feature.

feature service A layer published on an ArcGIS server whose features and attributes can be requested through its REST endpoint.

FGDC The Federal Geographic Data Committee, whose metadata standard US federal agencies follow. Internationally, the standard is ISO 19115.

file path The location of a file on your computer, such as `data/counties.csv`. When you knit, paths are read relative to the folder the `.Rmd` file is saved in.

FIPS code A standard numeric code for a state or county (Federal Information Processing Standards). North Carolina is `37`.

first law of geography Waldo Tobler's statement that "everything is related to everything else, but near things are more related than distant things." It is the idea behind spatial autocorrelation.

flattening How much an ellipsoid is squashed: (a – b) / a, where a and b are the semi-major and semi-minor axes. About 1/298 for the Earth.

flexdashboard An R package that lays out an R Markdown file as a dashboard of columns or rows of panels, with an optional sidebar.

for loop A loop that runs its block once for each item in a vector, such as `for (f in files) { ... }`, then stops.

full join A join that keeps every row of both tables, with NA wherever there is no match. `full_join()` in dplyr.

function Code that takes inputs (arguments), does a job, and returns an output, such as `mean(temps)`.

generalization Simplifying features when a map shows a place at a smaller scale, for example drawing a city as a point instead of an area.

GeoAI Artificial intelligence methods, such as deep learning, applied to spatial data, for example to find buildings in satellite images.

geocoding Turning an address or place name into coordinates that can be mapped.

geodesy The science of measuring the Earth's size, shape, orientation, and gravity. Every coordinate rests on a geodetic model of the Earth.

geographic variation The change in a phenomenon from place to place, such as income from county to county.

GEOID The unique ID of a Census area, made by stringing together the FIPS codes of each level, such as `37135` for Orange County, NC. Keep GEOIDs as text, and join tables on them.

geoid (shape of the Earth) The lumpy shape of the Earth defined by gravity, roughly where mean sea level would be. Ellipsoids are smooth approximations of it.

GeoJSON An open, plain-text format for vector features, written in JSON. Coordinates are longitude first, then latitude, in WGS84.

geom Short for geometric object: the marks a ggplot draws to show the data, such as points (`geom_point()`), bars (`geom_bar()`), or boxes (`geom_boxplot()`).

geometry column The column of an sf object that holds each feature's shape: a point, a line, or a polygon.

geospatial foundation model A large AI model pretrained on huge image archives that can be fine-tuned for many tasks, such as mapping floods, with little labeled data.

GeoTIFF The standard file format for raster data: a TIFF image with its location and coordinate system stored inside the file.

Getis-Ord Gi* A local statistic that gives each observation a z-score: large positive for hot spots, large negative for cold spots.

ggplot2 The tidyverse package for making plots and maps by adding layers.

GISP Geographic Information Systems Professional: a certification from the GIS Certification Institute for experienced GIS practitioners.

global environment Your main R workspace, where the variables you create at the Console or in a script live. Functions don't change it unless you assign their result.

global statistic A statistic that gives one value for a whole study area, such as Moran's I. It doesn't say where clusters are.

grammar of graphics The idea that any chart can be described with the same small set of parts: data, aesthetic mappings, geoms, scales, facets, and coordinates. ggplot2 is built on it.

graticule The network of parallels and meridians drawn on a map or globe.

great circle distance The shortest distance between two points along the Earth's curved surface. Use it for longitude and latitude.

grouped data frame A data frame made by `group_by()`. It looks the same, but later verbs



work on each group separately until you call `ungroup()`.

haversine formula A formula for great circle distance that stays accurate for very short distances.

header The first line of a data file, when it holds the column names.

help page A function's documentation, opened with `?name` or `help(name)`. Every help page has a Description, Usage, Arguments, and Examples.

hex code A six-character code for a color, such as `#3182bd`: two characters each for its red, green, and blue.

histogram A chart that divides a variable's range into bins and draws a bar for how many values fall in each. `geom_histogram()` in ggplot2.

hotspot A cluster of places with unusually high values, such as a group of neighborhoods with many burglaries. Finding hotspots is a common goal of spatial analysis.

htmlwidgets R functions, such as those in leaflet, DT, and plotly, that produce JavaScript, so their interactivity runs in the browser with no R.

HTTP The protocol web browsers and servers use to talk. A `GET` request reads data; a `POST` request sends data.

httr2 An R package for building, sending, and reading web (HTTP) requests, used to call an API by hand.

if statement Code that runs a block only when a test is TRUE: `if (x < 5) { ... }`. Add `else { ... }` for when it is FALSE.

image classification Assigning each pixel of an image a land cover class, such as forest, water, or developed land.

independence The assumption, built into most standard statistics, that knowing one observation tells you nothing about another. Data about places usually break it, because nearby places tend to be alike.

independent variable A variable used to explain the dependent variable, *x*. It goes on the right of the ~.

indexing Selecting part of a vector or data frame with square brackets: by position (`vect[3]`), by a condition (`vect[vect > 10]`), or by name (`vect["Fozzie"]`).

infinite loop A loop that never ends because its test never becomes FALSE. Stop it with Esc or RStudio's red stop sign.

inner join A join that keeps only the rows whose key appears in both tables. `inner_join()` in dplyr.

input (in Shiny) A control, such as a drop-down menu or checkboxes, that collects a reader's choice. Its value is read as `input$id`.

integer The data type for whole numbers. R stores a typed number as numeric unless you add an `L`, as in `21L`. Shown as `<int>` by `glimpse()`.

integrated development environment (IDE) A program that puts everything for writing code in one window: an editor, a place to run the code, and tools to inspect the results.

interactive map A web map you can pan, zoom, and click to see each feature's attributes. tmap makes one in view mode.

intercept The predicted value of *y* when every *x* is 0: where the regression line crosses the y axis.

interpreted language A language, like R, that runs code one line at a time: an interpreter reads each line, translates it, and runs it before moving on.

interquartile range (IQR) Q3 minus Q1: the spread of the middle half of the data. Outliers barely affect it. `IQR()` in R.

inverse distance weights Neighbor weights of 1 / *d*^*x*, so a neighbor's weight falls as its distance grows.

isochrone A line, or area, of equal travel time from a place, such as everywhere within a 15-minute drive of a hospital.

Jenks natural breaks A way of dividing values into map classes that puts the breaks where the gaps between values are largest, so similar values share a color.

join Combining two tables by matching rows on a shared key column, such as attaching Census data to boundaries by `GEOID` with `left_join()`.

JSON JavaScript Object Notation: structured text that most web APIs send back, which R can turn into lists or data frames.

k nearest neighbors A neighbor definition: each feature's neighbors are the *k* features closest to it. Every feature gets exactly *k*.

key The column whose values are matched to line up rows in a join, such as a student ID or a county FIPS code.

KML Keyhole Markup Language: the XML-based format behind Google Earth. A `.kmz` file is a zipped KML file.

knit To build an R Markdown file into a finished document (HTML, PDF, or Word). Knitting reruns every code chunk from the top in a fresh R session.

Kruskal-Wallis test A test of whether three or more groups differ, using ranks instead of values, so it doesn't need normal data. `kruskal.test()` in R.

label (in leaflet) Text that appears when the reader hovers over a feature on a leaflet map.

latitude The angle north or south of the Equator, from `0`° to `90`°. Lines of equal latitude are parallels.

layer One piece of a plot or map, added with + and drawn on top of the layers before it. In GIS, a layer holds one theme of features, such as roads or soils.

layer ID The number at the end of an ArcGIS layer's URL, such as the 2 in `.../MapServer/2`, that picks one layer out of a service.

leaflet An R package (and JavaScript library) for interactive web maps, built in layers joined with the pipe.

left join A join that keeps every row of the first (left) table, with NA where the second table has no match. `left_join()` in dplyr.

legend The key that explains what the colors, sizes, or shapes on a plot or map mean. ggplot2 and tmap build one for every mapped aesthetic.

library The folder on your computer where installed packages are stored. You install packages into a library, then load them with `library()`.

lidar An active sensor that measures distance with laser pulses, to build detailed 3D models of terrain, buildings, and trees.

line of best fit The straight line that comes closest to the points in a scatterplot: a regression's estimate of *y* for each *x*.

line segment The straight piece of a line between two vertices.

linear model A model in which *y* is a straight-line function of the *x* variables. `lm()` in R.

linked views Maps, charts, and tables that respond to each other, so filtering or selecting in one updates the others.

local environment The private workspace R makes each time a function runs. Its arguments and variables live there and vanish when it finishes.

local Moran's I Moran's I calculated for each observation, to find clusters and outliers. `localmoran()` in spdep.

local statistic A statistic calculated for each observation from its neighbors, such as Gi*, so its results can be mapped.

location intelligence The business use of spatial data and analysis, for site selection, routing, market areas, and real estate.

location–allocation A method that chooses which of several candidate sites to use, such as for fire stations, so that they minimize travel or serve the most people.

logical The data type for `TRUE` and `FALSE` values, written in capitals with no quotation marks.

logical operator An operator that combines `TRUE` and `FALSE` values: & (AND, both must be true), | (OR, either can be true), and `!` (NOT).

logical vector A vector of `TRUE` and `FALSE` values, such as the result of `vect > 10`. Inside square brackets it keeps the `TRUE` positions.

longitude The angle east or west of the Prime Meridian, from `0`° to `180`°. Lines of equal



longitude are meridians.

loop variable The variable in a `for` loop that takes the next item of the vector on each pass, such as `f` in `for (f in files)`.

Manhattan distance Distance along a grid of streets: the horizontal plus the vertical difference, never shorter than the straight line.

many-to-many A relationship where rows on both sides can match many rows, such as students and classes. Joins like this multiply rows.

many-to-one A relationship where many rows match one row in the other table, such as cities and their state.

map projection A method for flattening the curved graticule onto a flat map. Every projection distorts shape, area, distance, or direction.

map scale The ratio of a distance on the map to the same distance on the ground. A large-scale map (1:10,000) shows a small area in detail; a small-scale map (1:10,000,000) shows a large area with little detail.

mapping an aesthetic Tying an aesthetic to a column, inside `aes()`, as in `aes(color = origin)`. The look of each mark then comes from the data, and ggplot2 adds a legend.

margin of error (MOE) How far a sample estimate might be from the true value. ACS margins of error use a 90% confidence level: there is a 90% chance the true value lies within the estimate plus or minus the MOE.

Markdown A simple way of formatting plain text: # for headings, `*italic*`, `**bold**`, and `-` for list items.

masking When two loaded packages have a function with the same name, the one loaded later hides (masks) the other. Reach the hidden one with `package::function()`.

matrix A grid of values in R with rows and columns, all of the same type, made with `matrix()`. A small raster can be held in one.

MaxRecordCount The most features an ArcGIS server returns for one request. arcpullr asks for larger layers in batches.

mean The average: the sum of the values divided by how many there are. A few very large or small values can pull it a long way.

mean center The average x coordinate and average y coordinate of a set of features. Like the mean, outliers pull it.

mean center of population The point where a flat map of the country would balance if every resident weighed the same.

mean square A sum of squares divided by its degrees of freedom: a variance. *F* is the ratio of two mean squares.

median The middle value when values are sorted, so half are above and half below. Unlike counts, medians can't be added across areas.

Mercator projection A conformal cylindrical projection, made for navigation, that greatly enlarges areas near the poles.

meridian A line of longitude. Meridians run from pole to pole and meet at the poles.

metadata Data about data: who made a dataset, when, how, and what each part means. In a YAML header, it describes the document.

microdata Records for individual people or households (with identifying details removed), rather than totals for areas. The ACS publishes microdata as PUMS.

missing value A value that is unknown, written NA ("not available") in R. Most calculations with NA give NA; test for it with `is.na()`.

mode The most common value. The only measure of center that works for categorical data.

modifiable areal unit problem (MAUP) The way results can change when the same individual data are grouped into different areas, either larger or smaller ones or the same size with different boundaries.

Monte Carlo test A test that shuffles values across locations many times and asks how often the shuffles beat the observed statistic.

Moran's I A global measure of spatial autocorrelation, from –1 (dispersed) through 0 (random)

to +1 (clustered), with a p-value.

multimodal Having several peaks.

multivariate regression A regression with more than one independent variable. Each coefficient is the change in *y* with the others held constant.

NAD27 The North American Datum of 1927, based on the Clarke 1866 ellipsoid and a single origin point in Kansas.

NAD83 The North American Datum of 1983, based on the GRS80 ellipsoid and maintained through a network of permanent GPS stations (CORS).

named argument An argument given with its name, such as `digits = 1` in `round(x, digits = 1)`.

natural join An inner join on every column the two tables share by name. Handy but fragile, so name your key instead.

NC OneMap North Carolina's statewide GIS data portal, https://www.nconemap.gov/, with layers from state and local agencies.

NDVI Normalized difference vegetation index: (NIR – Red) / (NIR + Red). It ranges from –1 to 1, and dense green vegetation is high.

negative index An index with a minus sign, such as `vect[-3]`, which drops that position and returns the rest.

neighbor list spdep's compact way of storing neighbors (class `nb`): each observation's neighbors, without all the zeros of a matrix.

neighborhood (in spatial analysis) The zone of influence around a feature. Features inside it are its neighbors, defined by a distance, the nearest features, or shared borders.

neighborhood weight matrix An $n \times n$ table, **W**, whose entry w_{ij} records the neighbor relationship between observations *i* and *j*.

nesting When every smaller area lies entirely inside exactly one larger area, as blocks do inside block groups, tracts, and counties. Nested areas can be added up without double-counting anyone.

network analysis Analysis of movement along a network, such as roads, to measure travel distance or time, or how easily people can reach a place.

network distance Distance along the lines of a network, such as streets, adding up the lengths of the segments on the route.

no data A flag for raster cells with no measurement. In R these cells become NA.

node The start or end point of a line. Lines in a network, such as streets, connect at nodes.

nominal dollars Dollar amounts as reported in their own year, not adjusted for inflation. Adjust them before comparing across years.

normal distribution A symmetric, bell-shaped distribution. About 68% of values fall within one standard deviation of the mean, and 95% within two.

north arrow A symbol on a map that shows which direction is north. In tmap, `tm_compass()`.

null hypothesis The claim a test starts from: that there is no difference or no relationship. A small p-value is evidence against it.

numeric The data type for numbers that can have a decimal part, such as 103.21. Also called *double*, and shown as `<dbl>` by `glimpse()`.

nycflights13 An R package of data on every flight that left New York City's three airports in 2013, including the `flights` data frame.

observation One thing that was measured or recorded, such as a student, a county, or a flight. In a data frame, each row is an observation.

one-to-many A relationship where one row matches many rows in the other table, such as a state and its cities.

one-to-one A relationship where each row matches at most one row in the other table, such as states and governors.

open science Doing research so others can check and reuse it: open access papers, open data, open methods, and open-source software.

ordinary least squares The usual way to fit a regression line: choose the line that makes the sum of squared residuals as small as possible.

origin The corner of a raster, usually the lower left, whose coordinates fix where the whole grid sits.

outlier A value far from all the others. It may be real or an error, so check it before you report it.

overlay analysis Combining thematic layers to find places that meet several conditions at once, such as wetlands on one soil type near a road.

p-hacking Re-running an analysis in different ways until a p-value falls below 0.05, which produces false findings.

p-value The probability of a result at least as extreme as yours if the null hypothesis were true. It is not the chance the null hypothesis is true.

package A bundle of code, written by other users, that adds new functions to R. Install it once with `install.packages()`, then load it in each session with `library()`.

palette function In leaflet, a function, made by `colorBin()`, `colorNumeric()`, `colorQuantile()`, or `colorFactor()`, that turns data values into colors. Use the same one for the map and its legend.

panel One box in a flexdashboard, started by a ### heading, that holds one map, chart, table, or value box.

parallel A line of latitude. Parallels run east–west, parallel to the Equator.

parameter A `name=value` setting added to an API request after the ?, such as `for=county:*`, that says what you want.

passive sensor A sensor that records sunlight reflected from a surface, or heat the surface gives off, such as a camera or multispectral scanner.

percentile The value below which a given share of the values fall: a quarter of values are below the 25th percentile. `quantile()` in R.

physical distance Separation measured across the Earth's surface, in meters, kilometers, or miles.

pipe The operator |>, which passes the result on its left as the first argument of the function on its right. Read it as "and then."

polygon A closed shape that represents an area, such as a county, a lake, or a census tract.

popup A box of text that appears when the reader clicks a feature on an interactive map.

portal A website where an organization publishes GIS data for people to search, preview, and download, such as NC OneMap.

post-hoc test A test run after an ANOVA to find which pairs of groups differ, such as Tukey's HSD.

PostGIS An extension that adds spatial data types and spatial queries, such as spatial joins, to the PostgreSQL database.

Prime Meridian The meridian through Greenwich, England, chosen in 1884 as 0° longitude.

program A set of lines and blocks of code that work together to do a whole job. R runs a program from top to bottom, so order matters.

purrr The tidyverse package for repeating a task over many inputs, such as downloading data for several years.

qualitative palette Distinct colors with no order, one per category, used for text columns such as species names.

quartile The values that split sorted data into four groups of equal size. Q1 is the 25th percentile and Q3 the 75th.

Quarto The newer successor to R Markdown, from the same developers. Quarto files end in `.qmd`. This book is written in Quarto.

queen contiguity Polygons are neighbors if they share an edge or just a corner. On a grid, each cell has eight. spdep's default.

R Markdown A document format (`.Rmd`) that mixes formatted text with chunks of R code. Knitting it runs the code and builds a finished report.

R script A plain file of R code, ending in `.R`. Anything that isn't code must be written as a comment.

R session One run of R, from start to finish. A new session starts with an empty Environment and no packages loaded; knitting always starts a fresh one.

R-squared The share of the variation in *y* that a model explains, from 0 to 1.

radar An active sensor that sends out microwaves. It sees through clouds and at night, so it is used to map floods during storms.

radiometric resolution How finely a sensor records differences in brightness.

range The largest value minus the smallest. `range()` in R returns both.

raster Spatial data stored as a grid of cells (pixels), each holding a value, such as a satellite image or an elevation model. In R, the **terra** package handles rasters.

reactive expression Shiny code, made with `reactive()`, that reruns automatically when the inputs it uses change. Call it like a function.

recycling R repeating a shorter vector until it matches the length of a longer one, so `vect[c(TRUE, FALSE)]` keeps every other element.

redistricting data (PL 94-171) The first 2020 census product: basic population, race, voting-age, and housing counts down to the block, used to redraw voting districts.

regression A statistical model of how one variable (such as home price) changes with one or more others (such as distance to downtown).

remote sensing Collecting information about Earth without touching it, from sensors on satellites, aircraft, or drones. It produces most of the world's raster data.

render function A Shiny function, such as `renderLeaflet()` or `renderDT()`, that draws an output and redraws it when the inputs or reactives it uses change.

replicability When a new study, with new data, answering the same question gets consistent results.

replication crisis The discovery, in the 2010s, that many published findings fail to hold up when other researchers repeat the studies.

reproducibility The ability for anyone to rerun an analysis from its code and data and get the same result.

research question The specific, answerable question a project sets out to answer, usually with a place and a time.

residual An observation's actual value minus the value a model predicts for it. Map them to see where a model fits poorly.

resolution How much detail a raster holds, set by its cell size: smaller cells mean higher resolution, and more data.

REST The most common style of web API: each dataset has its own URL, you ask for data with ordinary web requests, and the answer usually comes back as JSON.

REST Services Directory The plain web page an ArcGIS server shows for each service and layer, listing its layer ID, geometry type, MaxRecordCount, and fields.

return value What a function gives back. In R, it is the value of the last line in the function's body.

right join A join that keeps every row of the second (right) table. `right_join()` in dplyr.

rook contiguity Polygons are neighbors only if they share an edge. On a grid, each cell has four.

row standardization Dividing each weight by its row's total so every row sums to 1. Then **W** times a variable gives the neighbors' average.

RStudio The integrated development environment (IDE) where you write and run R. R does the computing; RStudio is the workspace around it.

sample A subset of a population that is surveyed to estimate something about the whole population. Smaller samples give less certain estimates.

scale The level of detail at which you study something: addresses, neighborhoods, counties, or states. A pattern that is clear at one scale can vanish or reverse at another.

scale (in a plot or map) The rules that turn data values into visual values: which colors, which



classes, which axis breaks. Every mapped aesthetic has one, and functions such as `scale_fill_manual()` or `tm_scale_intervals()` change it.

scale bar A bar on a map that shows how distances on the map relate to distances on the ground.

scan statistic A cluster test that moves a window across a map (and often through time) to find areas with more cases than chance would produce. SaTScan is a widely used program for it.

scatterplot A plot of two continuous variables against each other, with one point per observation. In ggplot2, `geom_point()`.

scope Which variables a piece of code can see. A function's arguments are in its own scope, separate from variables outside it.

secant Describes a map surface that cuts through the globe, meeting it along two standard lines where the scale is true.

second-order neighbors The neighbors of a feature's neighbors.

selection helper A function that picks columns by name inside `select()`, such as `starts_with()`, `ends_with()`, and `contains()`.

self join A join of a table to itself, such as employees to their managers.

semi-major axis On an ellipsoid, the distance from the center to the Equator: about 6,378 km for the Earth.

semi-minor axis On an ellipsoid, the distance from the center to a pole: about 6,357 km for the Earth.

sequential palette Colors that run from light to dark, used for numbers so that bigger values look darker.

setting an aesthetic Giving every mark the same fixed value, outside `aes()`, as in `geom_point(color = "blue")`. It is a design choice, not a data choice.

sf "Simple features": the R package for spatial vector data. An sf object is a data frame with a geometry column holding each row's point, line, or polygon.

shapefile Esri's common format for vector data: a group of files with the same name (`.shp`, `.shx`, `.dbf`, `.prj`, and more) that must be kept together.

Shapiro-Wilk test A test of whether values, such as a model's residuals, are normally distributed. A large p-value means no evidence against normality.

SharedData The crosstalk object that wraps a data frame. Widgets built from SharedData objects with the same group are linked, row by row, through their key.

Shiny An R package for web apps that run R code on a server whenever the reader changes an input.

simple join A spatial join that attaches the attributes of one matching feature, such as the county each school is in.

skewed Lopsided: most values are bunched at one end, with a long tail of unusual values at the other, which pulls the mean away from the median.

slope How much *y* changes when *x* goes up by one unit: the coefficient of an independent variable.

small multiples A set of small charts, one per group, that share the same axes so they can be compared directly. ggplot2 calls this faceting.

snake_case A way of naming variables with lowercase words joined by underscores, as in `pop_chapel_hill`. The tidyverse style guide uses it.

social distance Separation measured by connection, similarity, or interaction, which a ruler can't measure.

source table In a join, the table that supplies the new columns: `y` in `st_join(x, y)`.

space-time cube A way to store space-time data: a map for every time step, stacked on top of each other, with x, y, and time as its three axes.

spatial autocorrelation The tendency for things that are close together to be more alike than things that are far apart, such as neighboring counties having similar incomes.

spatial cross-validation Testing a model by holding out whole regions, rather than random

observations, so the test data really are new places.

spatial data Data that have a location, such as campus buildings or rain gauges. Data with no location of their own, such as class grades, are non-spatial.

spatial data science The use of data that have a location attached to answer questions about the world. It combines what we study (the *spatial*), how we handle it (*data* skills), and how we ask and answer questions (*science*).

spatial descriptive statistics Statistics that describe the center and spread of a set of locations, such as the mean center and standard distance.

spatial epidemiology The study of how health outcomes, such as disease cases, vary across places, and why.

spatial error model A spatial regression model that allows the errors of neighboring observations to be correlated.

spatial heterogeneity The idea that relationships between variables change from place to place. A link that is strong in one city may be weak, or reversed, in another.

spatial join A join that matches features by their location, such as attaching to each point the polygon it falls inside. `st_join()` in R.

spatial lag The weighted average of an observation's neighbors' values, calculated with row-standardized weights. `lag.listw()` in spdep.

spatial lag model A spatial regression model that includes the neighbors' values of y as an extra explanatory variable.

spatial pattern The arrangement of objects in space: clustered, random, or dispersed.

spatial resolution The size of one pixel on the ground, such as 30 m for Landsat.

spatial thinking Taking seriously how location connects observations: asking whether nearby values are alike, whether a pattern depends on the size of the areas, and whether a relationship holds everywhere.

spatiotemporal kriging Predicting values at places and times with no measurements, from measurements nearby in both space and time.

species distribution model A model, often a regression with environmental predictors, that predicts where a plant or animal can live.

spectral resolution How many wavelength bands a sensor records, and how narrow they are.

spectral signature The pattern of how much light a material reflects at each wavelength. Different surfaces, such as plants, soil, and water, have different signatures.

SQL Structured Query Language, the language of databases. ArcGIS `where` clauses use it: `fcounty = 'Orange'`.

stacked bar chart A bar chart where each bar is split into colored segments, one per group, so the bar's height is the total.

standard deviation A measure of how spread out values are around their mean, in the same units as the values. `sd()` in R.

standard distance The typical distance of features from their mean center: the spatial counterpart of the standard deviation, drawn as a circle.

standard error A measure of how much a sample estimate would vary from sample to sample. For the ACS, it is the margin of error divided by 1.645.

standard line A line where a map projection's surface touches the globe, so the scale there is exactly true.

State Plane Coordinate System A set of local projections for the US, with one or more zones per state, in feet or meters. North Carolina's zone is EPSG:32119.

statistic A number calculated from a sample that summarizes the data, such as a mean. It would change if you drew a different sample.

statistically significant By convention, a result with $p < 0.05$. Significant doesn't mean large or important.

sum of squares The squared distances of values from a mean, added up. ANOVA splits it into between-group and within-group parts.

summarized join A spatial join that aggregates all the matching features, such as counting



the schools in each county.

summary statistic One number that describes many values, such as their mean, median, maximum, or count. `summarize()` calculates them.

syntax The exact rules for writing code: spelling, capitalization, commas, quotation marks, and matching brackets.

t-test A test of whether the means of two groups differ. ANOVA extends it to three or more groups.

temporal resolution How often a sensor revisits the same place.

terra An R package for working with raster data.

test statistic A statistic that weighs evidence in a hypothesis test, such as *t* or *F*: how large a pattern is compared with random variation.

thematic layer A layer that holds one theme of features, such as roads, soils, or elevation. Layers that share a coordinate system stack into a map.

thematic map A map that shows how one theme, such as income, tree height, or building age, varies from place to place.

theme The settings for everything on a plot that isn't data: background, grid lines, fonts. Add one with a layer such as `theme_minimal()`.

Thiessen polygons Polygons drawn around a set of points so each holds the area closer to its point than to any other.

tidy data Data arranged so that each variable is a column, each observation is a row, and each value has its own cell.

tidycensus An R package that downloads US Census data through the Census API and returns it as a tidy data frame, with boundaries attached if you ask.

tidyverse A family of R packages (including **dplyr**, **ggplot2**, and **readr**) for reading, cleaning, reshaping, summarizing, and plotting data, all designed to work together.

tie Two or more rows with the same value in the column being sorted. `arrange()` breaks ties with the next column you name.

TIGER The Census Bureau's national digital map database of streets and boundaries, first used for the 1990 census. Census boundary files come from it.

tigris An R package that downloads Census boundary files (TIGER/Line shapefiles) as sf objects.

tmap An R package for making thematic maps quickly, with sensible defaults.

topological data structure A data structure that records how features connect and share boundaries, so a shared border is stored only once.

topology How features connect, such as which polygons touch, regardless of the distances between them.

travel time Separation measured in minutes along a network, using speeds, turns, and traffic, instead of meters.

truth table A table listing the result of a logical operator for every combination of `TRUE` and `FALSE` inputs.

Tukey's HSD Tukey's honestly significant difference test: compares every pair of groups after an ANOVA, with p-values adjusted for the number of comparisons.

two-step floating catchment area An accessibility measure that combines provider supply, population demand, and travel time, in two steps: one around each provider and one around each population location.

unique value A value counted once, however many times it repeats. `distinct()` keeps one row for each unique value, or each unique combination of values.

urban heat island An area of a city, usually paved and with few trees, that is hotter than its surroundings.

value box A dashboard panel that shows one headline number, with a caption and an icon.

variable In R, a named piece of memory that holds data, such as `months <- 12`. In a dataset, a measured characteristic, such as median income; each Census variable has a code.

variance A measure of spread: the standard deviation squared, so its units are squared too. `var()` in R.

vector A sequence of values that R stores together, made with `c()`, such as `c(71, 75, 68)`.

vector data Spatial data stored as points, lines, and polygons, each with a table of attributes. In R, the **sf** package handles vector data.

vector data model Storing geographic features as coordinates: points, lines (chains of vertices), and polygons, each with a row of attributes. It suits discrete objects with clear edges, such as roads and parcels.

vectorized Done to every element of a vector at once, without a loop. Arithmetic in R is vectorized, so `distance / air_time` divides row by row.

verb A dplyr function named for the job it does to a data frame, such as `filter()`, `select()`, `arrange()`, and `mutate()`.

vertex A point between the ends of a line where it changes direction. More vertices make a smoother line and a bigger file.

weighted average An average in which some values count more than others. Total distance divided by total air time weights each flight's speed by its air time.

weighted mean center A mean center in which each feature counts in proportion to an attribute, such as population.

Welch's ANOVA A version of ANOVA that doesn't assume the groups have equal variances. `oneway.test()` in R.

WGS84 The World Geodetic System of 1984, the datum GPS uses. GeoJSON and KML give coordinates as WGS84 longitude and latitude.

where clause An SQL filter, such as `"fcounty = 'Orange'"`, that the server applies before sending data back, so you download only what you need.

while loop A loop that repeats its block for as long as a test stays TRUE. Something in the block must change the test, or it never stops.

wrapper package An R package that hides the details of calling an API behind simple functions, such as **tidycensus** for the Census API.

YAML header The block at the top of an R Markdown file, between two lines of three dashes, that holds the document's title, author, date, and output format.

z-score How many standard deviations a value is from what's expected. Beyond ±1.96 is significant at the 0.05 level.

ZIP Code Tabulation Area (ZCTA) The Census Bureau's approximation of a ZIP code's area, built from census blocks. ZCTAs do not nest inside counties or tracts.

About the author



I'm **Dr. David Parr**, a Teaching Assistant Professor in the Department of Geography and the Environment at the University of North Carolina at Chapel Hill. I teach classes in spatial data science, GIS, programming, web mapping, spatial databases, cartography, spatial modeling and analysis, and data acquisition and integration.

I see my disciplines, Geographic Information Science (GIS) and spatial data science, as the intersections of Geography and Computer Science, the fields in which I have earned my degrees. I want students to be able to *think geographically*: to critically consider the dynamic spatial and temporal issues of a problem, and to consider how scale and distance affect our human and physical environments. I also want them to *think computationally*: to break problems into smaller components in a logical sequence, and to consider how a representation in a computer is related to the object in the real world. These skills, understanding spatial relations and logical problem solving, are critical in our discipline and in the wider world.

My teaching and research interests include volunteered geographic information, space-time spatial analysis and modeling, GIS and geography education, and GIS and society, including digital places and web GIS.

- **Email:** daveparr@unc.edu
- **Website:** daveparr.com
- **Office:** 317B Carolina Hall, UNC-Chapel Hill

About this book

This book was developed and written from several years of lectures for GEOG 215. It is a Quarto book: every chapter is a runnable document, so the tables, maps, and figures come from the R code printed beside them, and many code boxes can be edited and run right in your browser.

References

The books, articles, and documentation cited in this book.

Openshaw, Stan. 1984. *The Modifiable Areal Unit Problem*. Concepts and Techniques in Modern Geography 38. Geo Books.

Tobler, Waldo R. 1970. "A Computer Movie Simulating Urban Growth in the Detroit Region." *Economic Geography* 46: 234–40. https://doi.org/10.2307/143141.

U.S. Census Bureau. 2020. *Understanding and Using American Community Survey Data: What All Data Users Need to Know*. U.S. Government Publishing Office. https://www.census.gov/programs-surveys/acs/library/handbooks/general.html.

U.S. Census Bureau. n.d. *Centers of Population, 1790–2020*. https://www.census.gov/geographies/reference-files/time-series/geo/centers-population.html.

Walker, Kyle. 2023. *Analyzing US Census Data: Methods, Maps, and Models in R*. CRC Press. https://walker-data.com/census-r.

Walker, Kyle, and Matt Herman. n.d. *Tidycensus: Load US Census Boundary and Attribute Data as 'Tidyverse' and 'Sf'-Ready Data Frames*. https://walker-data.com/tidycensus.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 2nd ed. O'Reilly Media. https://r4ds.hadley.nz.