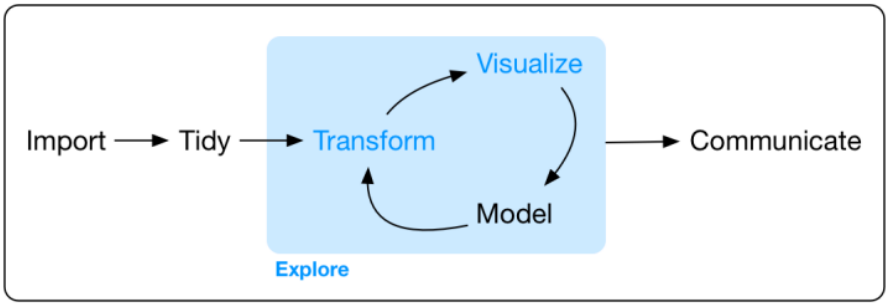
UNIT II R PROGRAMMING FOR DATA SCIENCE

Explore – Data Visualization with ggplot2 – Data Transformation with dplyr – Exploratory Data Analysis, Wrangle – Tibble with tibble – Data Import with readr – Tidy Data with tidyr – Relational Data with dplyr – Strings with stringr – Factors with forcats – Dates and Times with lubridate, Program – Pipes with magrittr – Functions – Vectors – Iteration with purr, Model – Model basics with modelr – Model Building – Many Models with Purrr and broom, Communicate – R Markdown – Graphics for Communication with ggplot2 – R Markdown Formats.

Explore



- Data Exploration:
  a. Look at your data, generate hypotheses, test them, and repeat.
  b. Aim to generate many promising leads for deeper exploration.
- Visualization:
  a. Start with R programming to create elegant and informative plots.
  b. Learn the basic structure of a ggplot2 plot and techniques for turning data into plots.
- Transformation:
  a. Learn key verbs to:
    i. Select important variables.
    ii. Filter out key observations.
    iii. Create new variables.
    iv. Compute summaries.
- Combining Techniques:
  a. Combine visualization and transformation with curiosity and skepticism to ask and answer interesting questions about data.
- Modeling:
  a. Important for the exploratory process but requires more data wrangling and programming skills.
- R Workflow:
  a. Learn good practices for writing and organizing R code.

These practices will help you stay organized for real projects.

Data Visualization with ggplot2

- ggplot2 is an elegant and versatile R package for data visualization, implementing the grammar of graphics.
- Quote by John Tukey: "The simple graph has brought more information to the data analyst's mind than any other device."
- ggplot2 is part of the tidyverse; load it with library(tidyverse). Install the tidyverse package if not already installed: install.packages("tidyverse").
- A layered grammar of graphics: A new template for a layered grammar of graphics in ggplot2 takes seven parameters, but typically only requires the data, mappings, and geom function due to useful defaults.

```
ggplot (data = <DATA> ) +
 <GEOM_FUNCTION> (mapping = aes( <MAPPINGS> ), stat = <STAT> , position = <POSITION> ) +
 <COORDINATE_FUNCTION> +
 <FACET_FUNCTION> +
 <SCALE_FUNCTION> +
 <THEME_FUNCTION>
```

- The <*DATA*> is used to load the datasets to work with (eg. *mpg* is a dataset which is available in *tidyverse*, more information about the dataset using `?mpg`.). In ggplot2, it is primarily involve importing and preparing data for visualization.
- Geometry <*GEOM_FUNCTION*> s in ggplot2 dictate the type of geometric objects used to represent data visually in a plot. These functions determine how data points are rendered and displayed. Some common <*GEOM_FUNCTION*> s include:

Function	Description	Common aes Parameters
geom_bar()	Creates bar charts (count).	x, fill, color
geom_col()	Creates bar charts (pre-summarized data).	x, y, fill, color
geom_point()	Creates scatterplots.	x, y, color, size, shape, alpha
geom_jitter()	Scatterplots with jitter to reduce overplotting.	x, y, color, size, shape, alpha, width, height
geom_smooth()	Adds smoothed conditional means (e.g., loess, lm).	x, y, color, linetype, size, method, se

- In ggplot2, AES (Aesthetic) Mapping, aes(<*MAPPINGS*>), refers to the process of mapping variables in your dataset to aesthetic attributes of your plot, such as x and y coordinates, colors, shapes, sizes, and more. AES mapping allows you to visually represent different aspects of your data by assigning data variables to these aesthetics.

aes Parameter	Description	Possible Values
x	The variable mapped to the x-axis.	Any numeric, categorical, or date-time variable.
y	The variable mapped to the y-axis.	Any numeric or date-time variable.
fill	The color used to fill bars or areas (e.g., in bar charts).	Any categorical variable; colors can be specified as named colors, hex codes (e.g., "#FF5733"), etc.
color	The color of points, lines, or borders of bars.	Any categorical variable; colors can be specified as named colors, hex codes, etc.
size	The size of points or lines.	Any numeric variable; can specify a fixed size (e.g., size = 2). Default is 1 for points, 0.5 for lines.
shape	The shape of points.	Any categorical variable; can specify fixed shapes (e.g., shape = 1 for circles, shape = 2 for triangles). Shape values range from 0 to 25. Default is 16 (solid circle).

alpha	The transparency level of points, bars, or lines.	Any numeric variable; values range from 0 (completely transparent) to 1 (completely opaque). Default is 1 (fully opaque).
width	The amount of horizontal jitter for geom_jitter().	Any numeric value specifying the jitter width (e.g., width = 0.1). Default is 0.4.
height	The amount of vertical jitter for geom_jitter().	Any numeric value specifying the jitter height (e.g., height = 0.1). Default is 0.
linetype	The type of line (e.g., solid, dashed) for geom_smooth().	Any categorical variable; fixed linetypes include "solid", "dashed", "dotted", "dotdash", "longdash", "twodash". Default is "solid".
method	The smoothing method for geom_smooth().	Fixed values like "lm" (linear model), "glm" (generalized linear model), "loess" (local regression), "gam" (generalized additive model). Default is "auto" (chooses best method).
se	Logical value indicating whether to display the confidence interval around the smooth.	TRUE or FALSE. Default is TRUE.

- The <*STAT*>s in ggplot2 determine how data should be summarized and transformed before visualizing. They compute statistical summaries or transformations that are then plotted using <GEOM_FUNCTION>s. Common <*STAT*>s include:

STAT Function	Description	Parameters
stat_summary()	Summarizes data into a statistical summary (e.g., mean, median) and displays as a summary bar in a plot.	**fun.data:** Function to summarize data; **fun.args:** Additional arguments for the summary function; **geom:** Geometric object to use ("point", "bar", "line", "errorbar", etc.); **position:** Position adjustment for overlapping objects ("identity", "dodge", "jitter", etc.); **width:** Width of the summary bar; **na.rm**; **show.legend**
stat_count()	Counts the number of observations for each x-axis category and displays as bar height in a plot.	**weight:** Weight variable; **na.rm:** Whether to remove missing values; **show.legend:** Whether to show legend for this layer.

- <POSITION>s in ggplot2 control the positioning of graphical elements to handle overlapping data or to create multi-panel displays. They adjust how layers are placed relative to each other. Common <POSITION>s include:

POSITION Function	Description	Parameters
position_identity()	Leaves the data as-is. Objects are placed exactly at the coordinates given by the data.	None
position_dodge()	Adjusts the positions of objects so that they are placed side by side instead of overlapping.	width: The width of the dodge (default is 0.9).
position_fill()	Scales the height of stacked objects so that they fill the plot area.	**vjust:** Vertical adjustment (default is 1).
position_jitter()	Adds random noise to the positions of objects to reduce overlap.	**width:** Amount of horizontal jitter (default is 0.4); **height:** Amount of vertical jitter (default is 0).

- <COORDINATE_FUNCTION>s in ggplot2 specify the coordinate system used to map data onto the plot. They define how data coordinates are transformed or projected onto the plot area. Common <COORDINATE_FUNCTION>s include:

COORDINATE FUNCTION	Description	Parameters

coord_flip()	Flips the x and y coordinates. Useful for horizontal bar plots.	None
coord_polar()	Projects data into a polar coordinate system. Useful for pie charts and circular bar plots.	**theta:** Variable to map angle ("x" or "y"); **start:** Offset of starting point; **direction:** Clockwise or counterclockwise.
coord_quickmap()	Provides an approximate map projection with correct aspect ratio. Useful for simple geographic visualizations.	**xlim:** Limits for x-axis; **ylim:** Limits for y-axis.

- <FACET_FUNCTION>s in ggplot2 facilitate the creation of multiple plots (facets) based on one or more categorical variables. They split data into subsets and display them in separate panels. Common <FACET_FUNCTION>s include:

FACET FUNCTION	Description	Parameters
facet_wrap()	Creates a series of plots wrapped into a specified number of rows or columns.	**facets:** A formula defining faceting variables (e.g., ~ var1); **nrow:** Number of rows; **ncol:** Number of columns; **scales:** Whether scales are fixed or free ("fixed", "free", "free_x", "free_y"); **shrink:** Whether to shrink axes to fit data.
facet_grid()	Creates a grid of plots defined by one or two categorical variables, creating rows and columns.	**rows:** A formula defining row faceting variables (e.g., vars(var1)); **cols:** A formula defining column faceting variables (e.g., vars(var2)); **scales:** Whether scales are fixed or free ("fixed", "free", "free_x", "free_y"); **space:** Whether space is fixed or free ("fixed", "free", "free_x", "free_y").
facet_null()	No faceting; all data is plotted in a single panel. This is the default.	None

- <SCALE_FUNCTION>s in ggplot2 control the mapping between data values and aesthetic attributes such as colors, shapes, and sizes. These functions allow for extensive customization of how data is represented visually.
- <THEME_FUNCTION>s in ggplot2 customize the overall appearance of a plot, including elements like text, grid lines, background, and other visual components. These functions allow users to create plots with specific stylistic choices to enhance readability and presentation.
- A common problem found here is miss placement of + at front. Be careful in placing that and should always at last.

Data Transformation with dplyr

1. filter() Function:

Description	**Parameters**	**Syntax Template**
Subsets rows using column values, with conditions combined using logical operators. It retains only those rows where the conditions evaluate to `TRUE`.	data (data frame or tibble), ... (logical conditions)	filter(data, condition1, condition2, ...)

Operator	**Usage**	**Example Statement**
>	Filters rows where the column's value is greater than a specified value.	filter(flights, dep_delay > 60)
<	Filters rows where the column's value is less than a specified value.	filter(flights, arr_delay < 30)
>=	Filters rows where the column's value is greater than or equal to a specified value.	filter(flights, dep_delay >= 60)

<=	Filters rows where the column's value is less than or equal to a specified value.	filter(flights, arr_delay <= 30)
==	Filters rows where the column's value is equal to a specified value.	filter(flights, month == 1)
!=	Filters rows where the column's value is not equal to a specified value.	filter(flights, carrier != "UA")
&	Filters rows where both conditions are true. Conditions are combined using &.	filter(flights, month == 1 & day == 1)
|	Filters rows where at least one of the conditions is true. Conditions are combined using `	filter(flights, month == 1 | day == 1)
!	Filters rows where the condition is not true. The condition is negated using !.	filter(flights, !is.na(dep_time))
Multiple	Filters rows using a combination of conditions and operators. Conditions can be nested and combined.	`filter(flights, month == 1 & day == 1, dep_delay > 60)

2. select() Function:

Description	Parameters	Syntax Template
A function in `dplyr` is used selects columns from a data frame or tibble. Allows for renaming columns, selecting columns by name, position, or helper functions, and dropping columns.	data (data frame or tibble), *condi(s)* (column names, positions, or helper functions to select or drop columns)	select(data, *condi(s)*)

Function	Description	Example Statement
starts_with()	Selects columns whose names start with a specified string.	select(flights, starts_with("dep"))
ends_with()	Selects columns whose names end with a specified string.	select(flights, ends_with("time"))
contains()	Selects columns whose names contain a specified string.	select(flights, contains("delay"))
matches()	Selects columns whose names match a regular expression.	select(flights, matches(".*time.*"))
everything()	Selects all columns. Can be used to reorder columns by placing it at a specific location.	select(flights, year, month, day, everything())
Column Range :	Selects a range of columns by name.	select(flights, year:day)
num_range()	num_range(prefix, range) where prefix is the string prefix of the column names and range is the sequence of numbers to match. Selects columns that have names following a numeric sequence pattern. This is useful for selecting columns like x1, x2, x3, etc.	select(flights, num_range("hour", 1:3))
Exclude -	Excludes specified columns from selection.	select(flights, -year, -month)

3. mutate() Function:

Description	Parameters	Syntax
A function in dplyr used to add new variables or transform existing variables in a data frame or tibble. It allows for the creation of new columns based on existing data or complex calculations.	data (data frame or tibble), condi(s) (new variable definitions or transformations)	mutate(data, condi(s))

Function	Description	Example Statement
+, -, *, /, ^	Supports arithmetic operations: addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^).	mutate(flights, total_delay = dep_delay + arr_delay)
%/%, %%	Supports modular arithmetic: integer division (%/%) and modulo (%%).	mutate(flights, hours = air_time %/% 60, minutes = air_time %% 60)

log(), log2(), log10()	Supports logarithmic functions: natural logarithm (log()), base-2 logarithm (log2()), and base-10 logarithm (log10()).	mutate(flights, log_air_time = log(air_time))
lead(), lag()	Functions to access subsequent (lead()) or previous (lag()) values in a vector.	mutate(flights, next_dep_delay = lead(dep_delay))
cumsum(), cumprod(), cummin(), cummax(), cummean()	Cumulative functions: cumulative sum (cumsum()), cumulative product (cumprod()), cumulative minimum (cummin()), cumulative maximum (cummax()), and cumulative mean (cummean()).	mutate(flights, cum_dep_delay = cumsum(dep_delay))

4. summarize() Function:

Description	Parameters	Syntax
A function in dplyr used to create a summary of a data frame or tibble by applying summary functions to variables. Often used with group_by() to perform operations within groups.	data (data frame or tibble), condi(s) (summary functions and their target variables)	summarize(data, condi(s))

Function	Description	Example Statement
mean()	Calculates the mean (average) of a numeric variable. Can handle missing values with na.rm = TRUE.	summarize(flights, mean_dep_delay = mean(dep_delay, na.rm = TRUE))
sum()	Calculates the sum of a numeric variable. Can handle missing values with na.rm = TRUE.	summarize(flights, total_dep_delay = sum(dep_delay, na.rm = TRUE))
median()	Calculates the median of a numeric variable.	summarize(flights, median_dep_delay = median(dep_delay, na.rm = TRUE))
min()	Calculates the minimum value of a numeric variable. Can handle missing values with na.rm = TRUE.	summarize(flights, min_dep_delay = min(dep_delay, na.rm = TRUE))
max()	Calculates the maximum value of a numeric variable. Can handle missing values with na.rm = TRUE.	summarize(flights, max_dep_delay = max(dep_delay, na.rm = TRUE))
n()	Counts the number of observations.	summarize(flights, count = n())
n_distinct()	Counts the number of distinct values in a variable.	summarize(flights, unique_carriers = n_distinct(carrier))
sd()	Calculates the standard deviation of a numeric variable. Can handle missing values with na.rm = TRUE.	summarize(flights, sd_dep_delay = sd(dep_delay, na.rm = TRUE))
var()	Calculates the variance of a numeric variable. Can handle missing values with na.rm = TRUE.	summarize(flights, var_dep_delay = var(dep_delay, na.rm = TRUE))

arrange() Function:

Description	Parameters	Syntax
A function in dplyr used to reorder rows of a data frame or tibble according to one or more variables. It sorts the data in ascending order by default, but can also sort in descending order if specified. When used within grouped data frames defined by `group_by()`, it maintains order within each group.	data (data frame or tibble), condi(s) (variables or expressions to arrange by, use desc() for descending order)	arrange(data, condi(s))

Function	Description	Example Statement
desc()	Reorders rows in descending order. Used within arrange() to sort variables in descending order.	arrange(flights, desc(dep_time))
ascending	The default sorting order in arrange(), reorders rows in ascending order.	arrange(flights, dep_time)

arrange()	Reorders rows of a data frame or tibble according to one or more variables.	arrange(flights, dep_time, desc(arr_delay))

Exploratory Data Analysis

- Exploratory Data Analysis (EDA) involves generating questions about your data, searching for answers through visualization, transformation, and modeling, and using the insights gained to refine or generate new questions; it is a flexible, iterative, and inquisitive process rather than a formal set of rules.
- It combines `dplyr` and `ggplot2` for interactive analysis, focusing on cleaning data and answering questions effectively.
- Asking a variety of questions is crucial in EDA to uncover insights, understand variation within variables, and explore covariation between variables.
- In data analysis, a variable is a measurable quantity or property, a value is its measured state, an observation is a set of measurements under similar conditions (often called a data point), and tabular data is a set of these values organized with variables in columns and observations in rows, where tidy data places each value in its own cell.
- Variation in variables refers to changes in values across measurements, whether continuous or categorical, and visualizing distributions involves using bar charts for categorical variables and histograms for continuous variables to understand their frequency patterns.

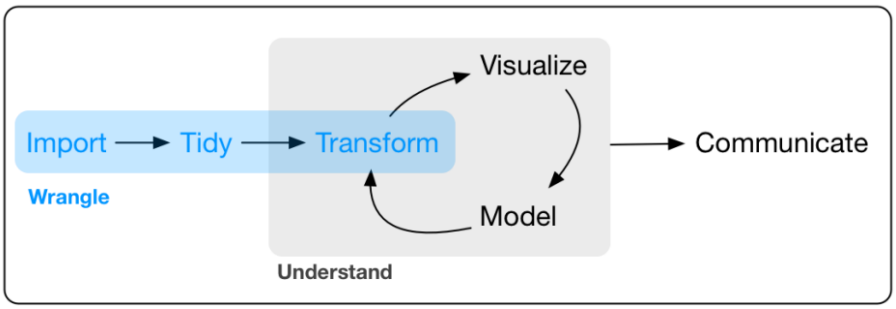
Function	Description	Common aes Parameters
geom_histogram()	Creates histograms to visualize the distribution of a continuous variable.	x (variable for x-axis), fill (fill color), color (outline color), binwidth (width of bins), bins (number of bins)
geom_freqpoly()	Creates frequency polygons to visualize the distribution of a continuous variable.	x, color (line color), linetype (type of line), size (line size), binwidth, bins
geom_boxplot()	Creates box plots to visualize the distribution and identify outliers in a continuous variable.	x, y (variable for y-axis), fill, color
geom_count()	Creates bubble charts where the size of the bubble represents the count of observations at each x, y position.	x, y, color, size (bubble size)
geom_tile()	Creates heatmaps to visualize the relationship between two continuous variables.	x, y, fill, color
geom_bin2d()	Creates 2D binning heatmaps for large datasets to visualize the relationship between two continuous variables.	x, y, fill
geom_hex()	Creates hexbin plots for large datasets to visualize the relationship between two continuous variables.	x, y, fill

Function	Description	Example Statement
count()	Tallies the number of observations in each group. Often used with group_by().	count(flights, carrier)
cut_width()	Divides a continuous variable into intervals of a specified width. Often used within mutate() to create bins.	mutate(flights, dep_delay_bin = cut_width(dep_delay, 10))
coord_cartesian()	Adjusts the Cartesian coordinates of a plot, including zooming in on a specific area.	ggplot(flights, aes(dep_delay)) + geom_histogram() + coord_cartesian(xlim = c(0, 100))
between()	Tests if values lie within a specified range. Often used within mutate() to create logical vectors.	mutate(flights, on_time = between(dep_delay, 0, 10))

ifelse()	Vectorized conditional function, used within mutate() to create new columns based on conditions.	mutate(flights, delay_flag = ifelse(dep_delay > 0, "Delayed", "On Time"))
reorder()	Reorders the levels of a factor variable based on the values of another variable.	mutate(flights, carrier = reorder(carrier, dep_delay))
add_residuals()	Adds residuals from a model to the data. Often used within mutate() to analyze model fit.	add_residuals(flights, lm(dep_delay ~ distance))
lm()	Fits a linear model to the data. Can be used with mutate() to add predicted values or residuals.	mutate(flights, lm_fit = lm(dep_delay ~ distance))

- Analyzing typical values in charts helps identify common and uncommon data points, prompting further investigation into unexpected patterns using geom_histogram(), geom_freqpoly(), and geom_boxplot().
- Identifying outliers is crucial for understanding potential errors or unique phenomena, best visualized with geom_boxplot().
- Handling unusual values involves replacing them with NA using mutate() and ifelse(), preserving valid data.
- Missing values are excluded from ggplot2 plots by default, with warnings issued, and the impact can be analyzed using is.na().
- Covariation describes how multiple variables change together, visualized using geom_freqpoly() for continuous and categorical variables, and geom_point() for continuous variables.
- Boxplots for comparison across categories using geom_boxplot() illustrate distribution differences, including median values, spread, and outliers.
- Analyzing two categorical variables can be done with geom_count() or geom_tile() after computing counts to highlight correlations.
- Large datasets can be handled using geom_bin2d() and geom_hex() to reduce overplotting by binning points into 2D space.
- Binned data visualization with cut_width() and geom_boxplot() allows comparison across categories.
- Patterns and models are understood using linear regression with lm() to predict one variable from another and analyze residuals.

Wrangling



- Data wrangling in R involves preparing data for analysis and visualization.
- Key skills include understanding tibbles, importing data from various formats, and organizing data into tidy formats.

- Specialized tasks cover handling categorical data, using regular expressions for string manipulation, and managing dates and times.
- Effective data wrangling ensures data is structured optimally for downstream analysis and modeling tasks in R.

Tibble with tibble

Function	Description	Parameters	Example Statement
as_tibble()	Converts an existing data frame or matrix into a tibble.	- x: An object to be converted to a tibble (e.g., data frame, matrix).	tb <- as_tibble(df)
tibble()	Creates a new tibble.	- ...: Named arguments to create columns in the tibble. - .name_repair: Controls the behavior of column name repair.	tb <- tibble(name = c("Alice", "Bob"), age = c(25, 30))
tribble()	Creates a tibble in a row-wise manner.	- ...: Column names and values specified in a row-wise format. - ~: Used to indicate column names.	tb <- tribble(~name, ~age, "Alice", 25, "Bob", 30)
print() (for tibble)	Prints the tibble to the console with formatting.	- x: The tibble to be printed. - n: Number of rows to print. - width: Width of the printed output.	print(tb, n = 10, width = Inf)
$ (subsetting)	Accesses a column in a tibble or data frame by name.	- name: The name of the column to access.	tb$name
[[(subsetting)	Accesses a column in a tibble or data frame by name or index.	- i: The index or name of the column to access.	tb[[1]] or tb[["name"]]
class(as.data.frame(tb))	Converts a tibble to a data frame and returns its class.	- tb: The tibble to be converted.	class(as.data.frame(tb))

- Tibbles Overview: Tibbles are an enhanced version of R data frames, designed for modern workflows with features like preserving variable types and names unchanged.
- Printing and Display: Tibbles are printed to show only the first 10 rows and columns that fit on the screen by default. This ensures readability for large datasets, with columns labeled by type.
- Creating Tibbles: Use tibble() to create new tibbles and as_tibble() to convert existing data frames into tibbles, preserving the original structure and data integrity.
- Subsetting Tibbles: Access specific columns using $ for name-based extraction or [[for name or index-based extraction. Tibbles do not support partial matching with [, ensuring consistent behavior.
- Interacting with Older Code: Convert tibbles back to data frames with as.data.frame() for compatibility with older R functions that expect traditional data frames.

Data Import with readr

Function	Description	Parameters	Example Statement
read_csv()	Reads comma-delimited files into a tibble.	- file: Path to the file. - col_types: Column types. - locale: Locale settings.	df <- read_csv("data.csv")
read_csv2()	Reads semicolon-separated files, common in some countries.	- file: Path to the file. - col_types: Column types. - locale: Locale settings.	df <- read_csv2("data.csv")

read_tsv()	Reads tab-delimited files into a tibble.	- file: Path to the file. - col_types: Column types. - locale: Locale settings.	df <- read_tsv("data.tsv")
read_delim()	Reads files with any delimiter into a tibble.	- file: Path to the file. - delim: Delimiter character. - col_types: Column types.	`df <- read_delim("data.txt", delim = "
read_fwf()	Reads fixed-width files into a tibble.	- file: Path to the file. - fwf_widths(): Specify field widths. - fwf_positions(): Specify positions.	df <- read_fwf("data.fwf", fwf_widths(c(5, 10)))
fwf_widths()	Defines the widths of fields for reading fixed-width files.	- widths: A numeric vector specifying the widths of each column.	fwf_widths(c(5, 10))
fwf_positions()	Defines the positions of fields for reading fixed-width files.	- start: Starting positions of columns. - end: Ending positions of columns.	fwf_positions(start = c(1, 6), end = c(5, 15))
read_table()	Reads tabular data with columns separated by white space into a tibble.	- file: Path to the file. - col_types: Column types. - locale: Locale settings.	df <- read_table("data.txt")
read_log()	Reads Apache style log files into a tibble.	- file: Path to the log file. - col_types: Column types. - locale: Locale settings.	df <- read_log("access.log")
charToRaw()	Converts a character vector to raw vector.	- x: A character vector to convert.	raw_data <- charToRaw("example")
guess_encoding()	Guesses the encoding of a file based on its content.	- file: Path to the file.	encoding <- guess_encoding("data.csv")
guess_parser()	Guesses the parser to use based on the type of data in a column.	- x: A character vector to analyze.	parser <- guess_parser(c("1", "2", "3"))
parse_guess()	Parses data based on the guessed type from guess_parser().	- x: A character vector to parse.	parsed_data <- parse_guess(c("1", "2", "3"))
problems()	Identifies issues with reading data, such as parsing errors or missing values.	- x: The data read by a read function.	issues <- problems(df)
reader_example()	Provides an example of reading a file with the specified reader function.	- file: Path to the file.	example <- reader_example("data.csv")
write_csv()	Writes a tibble to a comma-delimited file.	- x: The tibble to write. - file: Path to the file.	write_csv(df, "data.csv")
write_tsv()	Writes a tibble to a tab-delimited file.	- x: The tibble to write. - file: Path to the file.	write_tsv(df, "data.tsv")
write_excel_csv()	Writes a tibble to a comma-delimited file compatible with Excel.	- x: The tibble to write. - file: Path to the file.	write_excel_csv(df, "data.csv")
write_rds()	Writes a tibble to an RDS file.	- x: The tibble to write. - file: Path to the file.	write_rds(df, "data.rds")
read_rds()	Reads an RDS file into a tibble.	- file: Path to the RDS file.	df <- read_rds("data.rds")
write_feather()	Writes a tibble to a Feather file.	- x: The tibble to write. - file: Path to the file.	write_feather(df, "data.feather")

read_feather()	Reads a Feather file into a tibble.	- file: Path to the Feather file.	df <- read_feather("data.feather")
parse_logical()	Parses logical values (TRUE/FALSE) from character strings.	- x: Character vector with logical values.	parsed_logical <- parse_logical(c("TRUE", "FALSE"))
parse_integer()	Parses integer values from character strings.	- x: Character vector with integer values.	parsed_integer <- parse_integer(c("1", "2", "3"))
parse_date()	Parses date values from character strings using a specified format.	- x: Character vector with date values. - format: Format of the date string.	parsed_date <- parse_date(c("2024-01-01"), format = "%Y-%m-%d")
parse_time()	Parses time values from character strings.	- x: Character vector with time values. - format: Format of the time string.	parsed_time <- parse_time(c("12:00", "23:59"), format = "%H:%M")
parse_datetime()	Parses date-time values from character strings.	- x: Character vector with date-time values. - format: Format of the date-time string.	parsed_datetime <- parse_datetime(c("2024-01-01 12:00", "2024-01-02 23:59"), format = "%Y-%m-%d %H:%M")
parse_double()	Parses double precision numeric values from character strings.	- x: Character vector with double values.	parsed_double <- parse_double(c("1.23", "4.56"))
parse_number()	Parses numeric values from character strings, allowing for locale-specific formats.	- x: Character vector with numeric values. - locale(): Locale settings for number formats.	parsed_number <- parse_number(c("1,234.56", "7.89"), locale = locale(decimal_mark = ","))
parse_character()	Parses character strings from input data.	- x: Input data to be parsed as character strings.	parsed_character <- parse_character(c("text1", "text2"))
parse_factor()	Parses character data into factors, converting strings to factor levels.	- x: Character vector to be converted into factors. - levels: Factor levels.	parsed_factor <- parse_factor(c("low", "medium", "high"), levels = c("low", "medium", "high"))
locale()	Specifies locale settings for parsing data, such as number formats and date-time formats.	- decimal_mark: Character used for decimal points. - date_format: Format for dates. - time_format: Format for times.	loc <- locale(decimal_mark = ",", date_format = "%d/%m/%Y")

- Introduction: readr simplifies data import into R, supporting various file formats with functions like read_csv(), read_tsv(), and read_fwf().
- Core Functions: Utilize read_csv(), read_csv2(), read_tsv(), read_delim(), and read_fwf() for importing different file types.
- Handling File Variations: Manage files with missing column names using col_names, handle missing values with na, and use skip and comment for metadata.
- Parsing Functions: Convert data types with parse_logical(), parse_integer(), parse_double(), parse_number(), parse_character(), parse_factor(), parse_date(), parse_time(), parse_datetime(), and customize parsing with locale().
- Column Type Guessing: Employ guess_parser() and parse_guess() to infer and parse column types.
- Handling Parsing Challenges: Use problems() for error diagnostics, adjust guess_max for better type inference, and use col_types for troubleshooting.

- Exporting Data: Export with write_csv(), write_tsv(), write_rds(), and write_feather() to ensure proper encoding and type preservation.
- Additional Resources: Utilize haven, readxl, DBI, and refer to the R data import/export manual and rio package for extended data handling capabilities.

Tidy Data with tidyr

Function	Description	Parameters	Example Statement
gather()	Converts wide data to long format by collapsing multiple columns into key-value pairs.	- data: Data to reshape. - key: Name of the key column. - value: Name of the value column. - ...: Columns to gather.	gather(df, key = "variable", value = "value", col1:col3)
spread()	Converts long data to wide format by spreading key-value pairs into multiple columns.	- data: Data to reshape. - key: Column containing new column names. - value: Column containing values to fill the new columns.	spread(df, key = "variable", value = "value")
separate()	Splits a single column into multiple columns based on a delimiter.	- data: Data to transform. - col: Name of the column to split. - into: Names of the new columns. - sep: Delimiter for splitting.	separate(df, col = "name", into = c("first_name", "last_name"), sep = " ")
unite()	Combines multiple columns into a single column with a specified separator.	- data: Data to transform. - col: Name of the new column. - ...: Columns to unite. - sep: Separator for the combined values.	unite(df, col = "full_name", first_name, last_name, sep = " ")
complete()	Fills in missing combinations of data to make the dataset complete.	- data: Data to complete. - ...: Variables to check for missing combinations.	complete(df, variable1, variable2)
fill()	Replaces missing values in a column by propagating non-missing values forward or backward.	- data: Data to transform. - ...: Columns to fill. - direction: Direction to fill (forward or backward).	fill(df, column_name, .direction = "down")

- Introduction to Tidy Data: Tidy data, as defined by Hadley Wickham, follows principles where each variable is a column, each observation is a row, and each cell contains a single value. This format enhances data manipulation and analysis efficiency in R.
- Benefits and Compatibility: Adopting tidy data principles streamlines processes within the tidyverse, including packages like dplyr and ggplot2, allowing more focus on analysis rather than data wrangling.
- Using tidyr: The tidyr package helps transform messy data into tidy formats with functions like gather(), spread(), separate(), unite(), complete(), and fill(). These functions facilitate data restructuring and handling of missing values.
- Handling Missing Values: Missing values can be explicit (NA) or implicit (absent). Use spread() to make implicit values explicit and gather() with na.rm = TRUE to manage explicit missing values. The complete() function ensures all variable combinations are present, and fill() propagates previous values forward.

- Examples and Applications: Practical applications of tidy data include using datasets like table1 for computing rates and summarizing data. This demonstrates how tidy data principles can be applied to real-world data scenarios.
- Challenges with Messy Data: Messy datasets often require transformation to adhere to tidy data rules for effective analysis. Not all data fits neatly into tidy structures, and in some cases, non-tidy formats may be preferred for performance or specialized conventions.

Relational Data with dplyr

Function	Description	Parameters	Example Statement
inner_join()	Joins two datasets by matching rows based on a common key.	- x: First dataset. - y: Second dataset. - by: Columns to join on.	inner_join(df1, df2, by = "id")
left_join()	Merges two datasets, including all rows from the left dataset and matched rows from the right dataset.	- x: Left dataset. - y: Right dataset. - by: Columns to join on.	left_join(df1, df2, by = "id")
right_join()	Merges two datasets, including all rows from the right dataset and matched rows from the left dataset.	- x: Left dataset. - y: Right dataset. - by: Columns to join on.	right_join(df1, df2, by = "id")
full_join()	Merges two datasets, including all rows from both datasets, with missing values filled as NA.	- x: Left dataset. - y: Right dataset. - by: Columns to join on.	full_join(df1, df2, by = "id")
semi_join()	Returns all rows from the left dataset that have a match in the right dataset, without duplicating columns from the right dataset.	- x: Left dataset. - y: Right dataset. - by: Columns to join on.	semi_join(df1, df2, by = "id")
anti_join()	Returns all rows from the left dataset that do not have a match in the right dataset.	- x: Left dataset. - y: Right dataset. - by: Columns to join on.	anti_join(df1, df2, by = "id")
intersect()	Finds common rows between two datasets based on specified columns.	- x: First dataset. - y: Second dataset. - by: Columns to compare.	intersect(df1, df2, by = "id")
union()	Combines rows from two datasets, removing duplicates.	- x: First dataset. - y: Second dataset. - by: Columns to combine.	union(df1, df2)
setdiff()	Returns rows in the first dataset that are not present in the second dataset.	- x: First dataset. - y: Second dataset. - by: Columns to compare.	setdiff(df1, df2)

- Introduction to Relational Data: Relational data involves multiple interconnected tables. Understanding these relationships is key to effective data analysis and integrity.
- Fundamentals and Keys: Relationships are formed between tables using primary and foreign keys. Primary keys uniquely identify rows within a table, while foreign keys reference rows in other tables.
- Verbs for Relational Data: Dplyr's verbs for relational data include mutating joins (left_join(), right_join(), full_join()), which add variables based on matching keys, and filtering joins (semi_join(), anti_join()) for selecting observations based on matches.
- Handling Missing Keys: Use mutate() and row_number() to add surrogate keys when primary keys are missing. For joins, ensure that primary keys in both tables are complete and correct.

- Join Types and Operations:
  - Inner Join: inner_join() matches rows with equal keys from both datasets.
  - Outer Joins: left_join(), right_join(), full_join() include all rows from one or both datasets, filling unmatched rows with NA.
  - Mutating Joins: Add variables from one table to another based on matching keys.
- Set Operations: Use intersect(), union(), and setdiff() to compare datasets:
  - intersect(x, y) finds common rows.
  - union(x, y) combines rows from both datasets, removing duplicates.
  - setdiff(x, y) identifies rows in x not present in y.
- Visualizing Joins: Venn diagrams can illustrate the results of different join types, showing which rows are retained or excluded.
- Join Problems and Solutions: Identify primary key variables, handle missing values, and use anti_join() to check key matches. Avoid relying solely on row counts for verifying joins, especially when keys are not unique.

Strings with stringr

Function	Description	Parameters	Example Statement
singlequotes	Denotes a single quote character.	None	'single quote'
doublequotes	Denotes a double quote character.	None	"double quote"
escape characters	Represents special characters in strings (e.g., \n for newline).	None	"line1\nline2"
write_lines()	Writes lines of text to a file.	- x: Character vector. - path: File path.	write_lines(c("line1", "line2"), "file.txt")
c()	Combines values into a vector.	- ...: Values to combine.	c("a", "b", "c")
str_length()	Computes the length of strings.	- string: Input character vector.	str_length("text")
str_c()	Concatenates strings.	- ...: Character vectors to concatenate. - sep: Separator string.	str_c("a", "b", sep = "-")
str_sub()	Extracts substrings from strings.	- string: Input character vector. - start: Start position. - end: End position.	str_sub("string", 1, 3)
str_to_lower()	Converts strings to lowercase.	- string: Input character vector.	str_to_lower("STRING")
str_to_upper()	Converts strings to uppercase.	- string: Input character vector.	str_to_upper("string")
str_sort()	Sorts strings alphabetically.	- string: Input character vector. - locale: Locale for sorting.	str_sort(c("b", "a", "c"))
str_wrap()	Wraps strings to a specified width.	- string: Input character vector. - width: Maximum line width.	str_wrap("long text", width = 10)
str_trim()	Trims whitespace from strings.	- string: Input character vector. - side: Side to trim ("both", "left", "right").	str_trim(" text ")

str_view()	Displays the first match of a pattern in a string.	- string: Input character vector. - pattern: Regular expression.	str_view("abc", "a")
str_view_all()	Displays all matches of a pattern in a string.	- string: Input character vector. - pattern: Regular expression.	str_view_all("abc", "a")
anchors-^	Matches the start of a string.	None	^abc
anchors-$	Matches the end of a string.	None	abc$
character classes-\d	Matches any digit.	None	\d
character classes-\s	Matches any whitespace character.	None	\s
character classes-[abc]	Matches any character in the set.	None	[abc]
character classes-[^abc]	Matches any character not in the set.	None	[^abc]
repetitions-?	Matches 0 or 1 occurrence of the preceding element.	None	a?
repetitions-+	Matches 1 or more occurrences of the preceding element.	None	a+
repetitions-*	Matches 0 or more occurrences of the preceding element.	None	a*
repetitions-{n}	Matches exactly n occurrences of the preceding element.	- n: Number of occurrences.	a{2}
repetitions-{n,}	Matches at least n occurrences of the preceding element.	- n: Minimum number of occurrences.	a{2,}
repetitions-{,m}	Matches up to m occurrences of the preceding element.	- m: Maximum number of occurrences.	a{,3}
repetitions-{n,m}	Matches between n and m occurrences of the preceding element.	- n: Minimum number of occurrences. - m: Maximum number of occurrences.	a{2,3}

- String manipulation in R using the stringr package, focusing on regular expressions (regexps) for handling unstructured or semi-structured data. Regexps, initially complex, become intuitive with practice.
- Basics: Strings can be created with single (') or double (") quotes, with escape sequences (\" for double quotes and \' for single quotes) used for embedding quotes. Use write_lines() to view raw string contents without escape sequences. Determine string length with str_length(), and concatenate using str_c(), which handles missing values with str_replace_na().
- Substrings and Case Transformation: Extract substrings using str_sub(), specifying start and end positions. Modify strings directly with str_sub()<-. Change case with str_to_lower(), str_to_upper(), and str_to_title(), using locale-specific rules.
- Regular Expressions and Pattern Matching: Visualize pattern matches with str_view() and str_view_all(). Basic patterns include exact matches, wildcards (.), and escaped characters (\). Anchor patterns at the start (^) or end ($) of strings for precise matching.
- Character Classes and Alternation: Utilize character classes (\d for digits, \s for whitespace, [abc] for specific characters, [^abc] for exclusions) and alternation (|). Group patterns with parentheses for precedence.

- Repetition and Grouping: Control repetitions with operators: ? (0 or 1 time), + (1 or more times), * (0 or more times), {n} (exactly n times), {n,} (n or more times), {,m} (up to m times), {n,m} (between n and m times). Use parentheses for grouping and backreferences (\1, \2).
- String Detection, Extraction, and Replacement: Detect matches with str_detect(). Count matches using str_count(). Extract matches with str_extract() and str_extract_all(). Grouped matches can be obtained using str_match(). Replace matches with str_replace() and str_replace_all().
- Splitting and Locating Strings: Split strings with str_split(). Locate match positions using str_locate() and str_locate_all().
- Advanced Patterns: Use regex() for standard expressions, fixed() for exact byte matches, and coll() for locale-aware comparisons. Enable multiline mode, comments, and dotall mode in regex with appropriate options (multiline, comments, dotall). Use ignore_case for case-insensitive matching.
- Utility Functions: Search for objects with matching names using apropos(). List files in a directory with dir().

Factors with forcats

Function	Description	Parameters	Example Statement
factor()	Creates a factor (categorical variable) from a character vector.	- x: A character vector. - levels: An optional vector of the unique values.	factor(x = c("a", "b", "a"))
parse_factor()	Parses a character vector into a factor with specified levels.	- x: A character vector. - levels: The levels of the factor. - ordered: Logical value for order.	parse_factor(x = c("a", "b"), levels = c("a", "b", "c"))
unique()	Extracts the unique levels of a factor.	- x: A factor vector.	unique(factor(x = c("a", "b", "a")))
fct_inorder()	Reorders factor levels by their first appearance in the data.	- f: A factor vector.	fct_inorder(factor(x = c("a", "b", "a")))
levels()	Retrieves or sets the levels of a factor.	- x: A factor vector.	levels(factor(x = c("a", "b", "a")))
fct_reorder()	Reorders factor levels by the values of another variable.	- f: A factor vector. - x: A numeric vector. - fun: Function to summarize within levels.	fct_reorder(factor(x = c("a", "b", "a")), c(2, 1, 3))
fct_relevel()	Relevels a factor by placing specified levels at the front.	- f: A factor vector. - ...: Levels to move to the front.	fct_relevel(factor(x = c("a", "b", "a")), "b")
fct_reorder2()	Reorders factor levels by two variables.	- f: A factor vector. - x: A numeric vector. - y: A numeric vector. - fun: Function to summarize.	fct_reorder2(factor(x = c("a", "b", "a")), c(2, 1, 3), c(4, 5, 6))
fct_infreq()	Reorders factor levels by frequency of appearance.	- f: A factor vector.	fct_infreq(factor(x = c("a", "b", "a")))
fct_rev()	Reverses the order of factor levels.	- f: A factor vector.	fct_rev(factor(x = c("a", "b", "a")))
fct_recode()	Recodes factor levels into new values.	- f: A factor vector. - ...: A sequence of named character vectors specifying old and new levels.	fct_recode(factor(x = c("a", "b", "a")), "new_a" = "a")

fct_collapse()	Collapses multiple factor levels into one.	- f: A factor vector. - ...: A sequence of named character vectors specifying levels to collapse.	fct_collapse(factor(x = c("a", "b", "a")), new = c("a", "b"))
fct_lump()	Lumps together all small groups into "Other".	- f: A factor vector. - n: The number of levels to preserve. - other_level: Name for the lumped level.	fct_lump(factor(x = c("a", "b", "a")), n = 1)

- Introduction: Factors in R handle categorical variables with a fixed set of values, allowing non-alphabetical ordering of character vectors. They prevent typos and ensure valid, consistent data representation.
- Prerequisites: Load the forcats package from the tidyverse to manage categorical variables effectively with functions tailored for factors.
- Creating Factors: Use factor() to create factors from character vectors, ensuring a fixed set of levels.
- Handling Invalid Values: Convert invalid values to NA using factor() or generate errors with parse_factor().
- Setting Levels: By default, factor levels are in alphabetical order. Use unique() to set levels based on the order of first appearance or specify custom levels directly.
- Reordering Factors: Modify factor levels for visualization using functions like fct_reorder(), fct_reorder2(), fct_inorder(), and fct_infreq() to reorder based on various criteria.
- Reversing and Recoding Levels: Reverse the order of factor levels with fct_rev() or change factor level names using fct_recode().
- Collapsing and Lumping Levels: Combine multiple levels into one with fct_collapse() or lump less frequent levels into "Other" with fct_lump().
- Visualizing Factors: Create bar charts and other visualizations, ensuring all levels are displayed using scale_x_discrete(drop = FALSE).

Dates and Times with lubridate

Function	Description	Parameters	Example Statement
today()	Returns the current date.	None	today()
now()	Returns the current date and time.	None	now()
ymd()	Converts a date string in "year-month-day" format to a Date object.	- x: Date string.	ymd("2023-07-01")
mdy()	Converts a date string in "month-day-year" format to a Date object.	- x: Date string.	mdy("07-01-2023")
dmy()	Converts a date string in "day-month-year" format to a Date object.	- x: Date string.	dmy("01-07-2023")
ymd_hms()	Converts a date-time string in "year-month-day hour:minute	- x: Date-time string.	ymd_hms("2023-07-01 12:00:00")
	" format to a POSIXct object.		
mdy_hm()	Converts a date-time string in "month-day-year hour	- x: Date-time string.	mdy_hm("07-01-2023 12:00")
	" format to a POSIXct object.		

make_datetime()	Creates a datetime object from separate year, month, day, hour, minute, and second components.	- year, month, day, hour, minute, second: Date-time components.	make_datetime(2023, 7, 1, 12, 0, 0)
make_datetime_100()	Creates a datetime object with fractional seconds.	- year, month, day, hour, minute, second: Date-time components.	make_datetime_100(2023, 7, 1, 12, 0, 0.5)
as_datetime()	Converts a Date or character object to a POSIXct object.	- x: Date or character object.	as_datetime("2023-07-01 12:00:00")
as_date()	Converts a POSIXct or character object to a Date object.	- x: POSIXct or character object.	as_date("2023-07-01")
year()	Extracts the year component from a Date or POSIXct object.	- x: Date or POSIXct object.	year(today())
month()	Extracts the month component from a Date or POSIXct object.	- x: Date or POSIXct object. - label: Logical, return abbreviated month name.	month(today(), label = TRUE)
mday()	Extracts the day of the month component from a Date or POSIXct object.	- x: Date or POSIXct object.	mday(today())
yday()	Extracts the day of the year component from a Date or POSIXct object.	- x: Date or POSIXct object.	yday(today())
wday()	Extracts the day of the week component from a Date or POSIXct object.	- x: Date or POSIXct object. - label: Logical, return abbreviated weekday name.	wday(today(), label = TRUE)
floor_date()	Rounds down to the nearest date-time unit.	- x: Date or POSIXct object. - unit: Unit to round to.	floor_date(now(), "month")
round_date()	Rounds to the nearest date-time unit.	- x: Date or POSIXct object. - unit: Unit to round to.	round_date(now(), "hour")
ceiling_date()	Rounds up to the nearest date-time unit.	- x: Date or POSIXct object. - unit: Unit to round to.	ceiling_date(now(), "week")
update()	Updates components of a Date or POSIXct object.	- x: Date or POSIXct object. - ...: Components to update (year, month, etc.).	update(now(), year = 2025)
as.duration()	Converts a period to a duration.	- x: Period object.	as.duration(days(5))
dseconds()	Creates a duration object representing a specified number of seconds.	- x: Number of seconds.	dseconds(60)

dminutes()	Creates a duration object representing a specified number of minutes.	- x: Number of minutes.	dminutes(60)
dhours()	Creates a duration object representing a specified number of hours.	- x: Number of hours.	dhours(24)
ddays()	Creates a duration object representing a specified number of days.	- x: Number of days.	ddays(7)
dweeks()	Creates a duration object representing a specified number of weeks.	- x: Number of weeks.	dweeks(2)
dyears()	Creates a duration object representing a specified number of years.	- x: Number of years.	dyears(1)
seconds()	Creates a period object representing a specified number of seconds.	- x: Number of seconds.	seconds(60)
minutes()	Creates a period object representing a specified number of minutes.	- x: Number of minutes.	minutes(60)
hours()	Creates a period object representing a specified number of hours.	- x: Number of hours.	hours(24)
days()	Creates a period object representing a specified number of days.	- x: Number of days.	days(7)
months()	Creates a period object representing a specified number of months.	- x: Number of months.	months(1)
weeks()	Creates a period object representing a specified number of weeks.	- x: Number of weeks.	weeks(2)
years()	Creates a period object representing a specified number of years.	- x: Number of years.	years(1)
Sys.timezone()	Returns the current system's timezone.	None	Sys.timezone()
length(olsonNames())	Returns the number of available timezones in the Olson database.	None	length(olsonNames())

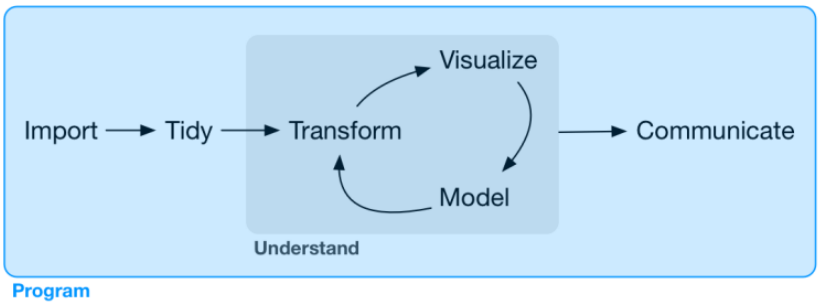
- Dates and Times in R: Dates and times in R are complex due to variations like leap years, daylight saving time (DST), and leap seconds, which complicate alignment with our calendar.
- Using lubridate: Use the lubridate package to simplify date-time manipulation, practicing with nycflights13, and loading tidyverse, lubridate, and nycflights13.
- Date/Time Types: R supports three types: date (<date>), time (<time>), and date-time (<dttm>), and it's best to use the simplest type needed to avoid unnecessary complexity.
- Current Date/Time Functions: Retrieve the current date using today() and the current date-time using now().
- Parsing Dates from Strings: Parse dates from strings using ymd(), mdy(), and dmy(), and parse date-times using functions like ymd_hms() and mdy_hm().

- Creating Dates/Times from Components: Create dates with make_date() and date-times with make_datetime() by combining individual components.
- Converting Date/Times: Convert between date-time and date using as_datetime() and as_date(), and handle Unix Epoch offsets similarly.
- Extracting Date/Time Components: Extract specific components of a date-time object with functions like year(), month(), mday(), yday(), wday(), hour(), minute(), and second().
- Rounding Date/Times: Round dates to the nearest specified unit using floor_date(), round_date(), and ceiling_date().
- Updating Date/Time Components: Set or update components of a date-time object using accessor functions or the update() function.
- Creating Durations and Periods: Create durations using functions like dseconds(), dminutes(), dhours(), ddays(), dweeks(), and dyears(), and create periods with seconds(), minutes(), hours(), days(), months(), weeks(), and years() which handle "human" times.
- Handling Time Zones: Retrieve the current system timezone using Sys.timezone() and check available timezones with length(olsonNames()).

Program

✦ Improving Programming Skills

- Programming and Data Science: Programming is essential for all data science work. You cannot perform data science purely in your head or with pencil and paper; you need to use a computer.
- Programming as Communication: Code communicates with the computer and other humans. Every project is collaborative, even if you are only working with your future self. Writing clear code is crucial for understanding and maintaining your work over time.
- Clear Code: Writing clear code is important so others, including your future self, can understand your analysis. Clear code makes it easier to revisit and modify your work later.
- Rewriting for Clarity: Rewriting your code is key to clarity, similar to writing prose. The first expression of your ideas is unlikely to be clear. Multiple rewrites enhance clarity, balancing immediate needs with long-term readability.



✦ Learning Programming Skills

- The Pipe (%>%): Learn about the pipe (%>%), how it works, what the alternatives are, and when not to use it. This skill helps in writing cleaner and more readable code.
- Writing Functions: Avoid repetitive copy-pasting by writing functions. Extract repeated code into functions for easy reuse. This practice reduces errors and inconsistencies in your code.
- R's Data Structures: Gain a solid grounding in R's data structures, including the four common atomic vectors and the three important S3 classes built on top of them. Understanding lists and data frames is essential for managing complex data in R.

- Tools for Iteration: Learn about tools for iteration, such as for loops and functional programming. These tools allow you to perform the same actions on different inputs efficiently, streamlining your code and making it more robust.

Pipes with magrittr

- Introduction: Pipes help chain multiple operations in a readable way by passing the result of one operation as input to the next. This guide explains the basics of using pipes and introduces other useful magrittr tools.
- Prerequisites: Pipes are provided by the `magrittr` package. Although tidyverse packages automatically load `%>%`, you can load `magrittr` explicitly if you need to focus on piping:
- Using the Pipe: The pipe (`%>%`) allows you to write sequences of operations more clearly. For example:

```
library(dplyr)
data <- mtcars %>%
filter(mpg > 20) %>%
arrange(desc(mpg)) %>%
select(mpg, cyl, hp)
```

- In this example, `mtcars` is filtered for `mpg > 20`, arranged in descending order of `mpg`, and then selected for specific columns. The `%>%` operator makes it easier to read and understand this sequence of operations.

✦ When to Avoid Pipes:
  - Long Pipelines: Avoid when pipelines exceed 10 steps, as readability decreases.
  - Multiple Inputs/Outputs: Complex operations with multiple inputs or outputs are not suited for pipes.
  - Complex Dependencies: Pipes are less effective when operations have complex dependency structures.

✦ Other magrittr Tools:
  - Tee Pipe (`%T>%`): Allows you to perform actions for side effects without ending the pipe. For example:

```
data <- mtcars %>%
filter(mpg > 20) %>%
{print(nrow(.))} %T>%
arrange(desc(mpg)) %>%
select(mpg, cyl, hp)
```

  Here, `print(nrow(.))` prints the number of rows before continuing with the rest of the pipeline.
  - Exploding Variables (`%$%`): Extracts variables for use in functions that don't directly operate on data frames:

```
mtcars %$%
cor(mpg, hp)
```

  This computes the correlation between `mpg` and `hp` directly from the variables in `mtcars`.
  - Assignment Operator (`%<>%`): Performs an inline transformation and assignment. Use with caution:

```
mtcars %<>%
filter(mpg > 20) %>%
arrange(desc(mpg))
```

Here, `mtcars` is updated in place with the filtered and arranged data. This can make code less clear by combining transformation and assignment.

Functions

- Functions in R are blocks of code designed to perform specific tasks. They help modularize code, making it reusable and easier to manage.
- Function Syntax: To define a function in R, use the `function` keyword. The basic syntax is:

```
function_name <- function(arg1, arg2) {
  # Code to execute
}
```

```
Example:
greet <- function(name = "User", greeting = "Hello") {
  message <- paste(greeting, name)
  return(message)
}
```

- Function Naming: Function names should be descriptive and follow conventions:
  - Use descriptive names (e.g., `calculate_mean`).
  - Use underscores or camelCase (e.g., `calculateMean`).
  - Avoid using R's reserved words or function names.
- Function Arguments and Naming: Arguments are inputs to functions. They can be required or optional, and their names should be clear and descriptive. From the above example, `name` and `greeting` are arguments with default values. You can call `greet()` with or without arguments.
- Function Return Types: Functions can return various types of values including numbers, strings, vectors, data frames, or lists. The `return()` function is optional; the last evaluated expression is returned if `return()` is not used. Form the above example it return the greeting message.
- Dot-Dot-Dot (`...`) for Functions: The `...` (ellipsis) allows functions to accept an arbitrary number of arguments, which can be useful for passing additional arguments to other functions or handling variable-length inputs.

```
Example:
combine_strings <- function(..., separator = " ") {
  strings <- c(...)
  combined <- paste(strings, collapse = separator)
  return(combined)
}
```

In this function, `...` collects all arguments into a vector, and `separator` specifies how to join them. You can use it like:

```
combine_strings("Hello", "world", "from", "R")
```

This will produce: `"Hello world from R"`.

- When to Create Functions:
  - Reusability: Create functions to avoid code duplication. If you perform the same set of operations multiple times, encapsulate them in a function.
  - Complexity: Use functions to simplify complex code by breaking it into smaller, manageable parts.

- Clarity: Functions enhance code readability by giving descriptive names to blocks of operations.
- Testing and Debugging: Functions make it easier to test and debug specific parts of your code in isolation.

Vectors

- Vectors are the most basic data structures in R, used to store a sequence of elements of the same type.
- Types of Vectors:
  - Atomic Vectors: Store elements of a single type. Types include: Logical (`TRUE`, `FALSE`), Integer (Whole numbers e.g., `1L`, `2L`), Double (Real numbers e.g., `3.14`, `2.718`), Character (Strings e.g., `"hello"`, `"world"`), complex and raw.

```
logical_vec <- c(TRUE, FALSE, TRUE, FALSE)
logical_vec[2]  # FALSE
logical_vec[c(1, 3)]  # TRUE TRUE
named_vec <- c(a = 1, b = 2, c = 3)
named_vec["b"]  # 2
```

  - Lists: Can store elements of different types, including other lists.

```
my_list <- list(name = "Alice", age = 30, scores = c(85, 90, 88))
```

    Access list elements using `$` or double square brackets `[[]]`.

```
my_list$name  # "Alice"
my_list[[2]]  # 30
my_list$scores[1]  # 85
```

- Properties of Vectors:
  - Type: Can be determined using `typeof()`.
  - Length: Number of elements in a vector, found with `length()`.
  - Size: Total memory used, can be checked with `pryr::object_size()`.
- Missing Values:
  - Logical Missing Values: `NA` (not available)
  - Integer Missing Values: `NA_integer_`
  - Character Missing Values: `NA_character_`
  - Double Missing Values:`NA_real_`
- Type Conversions: Convert between types using `as.*` functions.

```
as_integer <- as.integer(c(1.5, 2.5)) # Converts to integers
as_logical <- as.logical(c(1, 0, 2))  # Converts to logical
as_double <- as.double(c(1, 2, 3))    # Converts to doubles
as_character <- as.character(c(1, 2, 3)) # Converts to characters
```

- Testing Types: Test the type of an object using `is.*` functions.

```
is_logical(logical_vec)  # TRUE
is_integer(integer_vec)  # TRUE
is_double(double_vec)    # TRUE
is_numeric(double_vec)   # TRUE (numeric includes double)
is_character(character_vec)  # TRUE
is_atomic(character_vec)   # TRUE
is_list(my_list)         # TRUE
is_vector(character_vec)   # TRUE (all lists and atomic vectors are vectors)
```

```
is_scalar_atomic(1L)      # TRUE (scalar atomic is a single atomic vector)
```

Iteration with purr

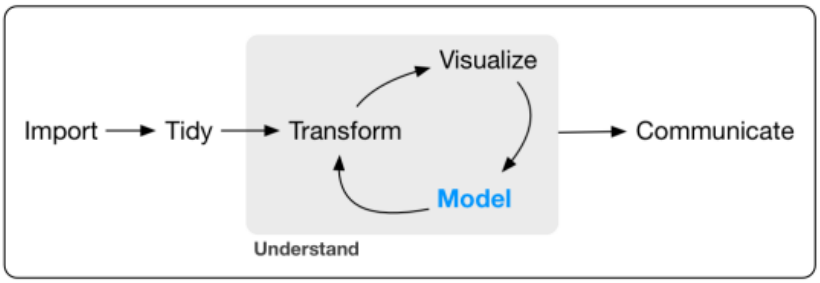
Function	Description	Parameters	Example Statement
lapply()	Applies a function over a list and returns a list.	X (list), FUN (function)	lapply(list(1, 2, 3), sqrt)
sapply()	Applies a function over a list and returns a simplified result.	X (list), FUN (function)	sapply(list(1, 2, 3), sqrt)
vapply()	Applies a function over a list and returns a result of a specified type.	X (list), FUN (function), FUN.VALUE (type)	vapply(list(1, 2, 3), sqrt, numeric(1))
map()	Applies a function over a list or vector and returns a list.	.x (list/vector), .f (function)	map(1:3, sqrt)
map_lgl()	Applies a function over a list or vector and returns a logical vector.	.x (list/vector), .f (function)	map_lgl(1:3, is.numeric)
map_int()	Applies a function over a list or vector and returns an integer vector.	.x (list/vector), .f (function)	map_int(1:3, length)
map_dbl()	Applies a function over a list or vector and returns a double vector.	.x (list/vector), .f (function)	map_dbl(1:3, sqrt)
map_chr()	Applies a function over a list or vector and returns a character vector.	.x (list/vector), .f (function)	map_chr(1:3, as.character)
map2()	Applies a function to pairs of elements from two lists or vectors.	.x, .y (lists/vectors), .f (function)	map2(1:3, 4:6, ~ .x + .y)
pmap()	Applies a function to the elements of multiple lists or vectors.	.l (lists/vectors), .f (function)	pmap(list(1:3, 4:6, 7:9), sum)
walk()	Applies a function over a list or vector for its side effects.	.x (list/vector), .f (function)	walk(1:3, print)
walk2()	Applies a function to pairs of elements from two lists for side effects.	.x, .y (lists/vectors), .f (function)	walk2(1:3, 4:6, ~ print(.x + .y))
pwalk()	Applies a function to the elements of multiple lists for side effects.	.l (lists/vectors), .f (function)	pwalk(list(1:3, 4:6), print)
detect()	Finds the first element matching a predicate.	.x (list/vector), .p (predicate function)	detect(1:10, ~ .x > 5)
detect_index()	Finds the index of the first element matching a predicate.	.x (list/vector), .p (predicate function)	detect_index(1:10, ~ .x > 5)
head_while()	Returns elements from the start of a list while a predicate is true.	.x (list/vector), .p (predicate function)	head_while(1:10, ~ .x < 5)
tail_while()	Returns elements from the end of a list while a predicate is true.	.x (list/vector), .p (predicate function)	tail_while(1:10, ~ .x > 5)
some()	Checks if at least one element matches a predicate.	.x (list/vector), .p (predicate function)	some(1:10, ~ .x > 5)
every()	Checks if all elements match a predicate.	.x (list/vector), .p (predicate function)	every(1:10, ~ .x > 0)
reduce()	Reduces a list or vector to a single value using a binary function.	.x (list/vector), .f (binary function)	reduce(1:10, +`)
accumulate()	Accumulates results of a binary function over a list or vector.	.x (list/vector), .f (binary function)	accumulate(1:10, +`)
keep()	Keeps elements matching a predicate.	.x (list/vector), .p (predicate function)	keep(1:10, ~ .x > 5)
discard()	Discards elements matching a predicate.	.x (list/vector), .p (predicate function)	discard(1:10, ~ .x > 5)

safely()	Captures errors and returns a list with result and error.	.f (function)	safe_log <- safely(log); safe_log("a")
possibly()	Returns a default value if an error occurs.	.f (function), otherwise (default value)	possible_log <- possibly(log, NA); possible_log("a")
quietly()	Captures output, messages, and warnings, along with the result.	.f (function)	quiet_log <- quietly(log); quiet_log("a")
invoke_map()	Invokes a function from a list of functions.	.f (functions list), ... (arguments)	invoke_map(list(mean, sd), list(1:10))

- Base R Functions (lapply(), sapply(), vapply()): These functions are fundamental for applying operations over lists and vectors, with lapply() returning lists, sapply() simplifying the result to a vector or matrix, and vapply() ensuring a specific return type for consistency.
- Core purrr Mapping Functions (map(), map_lgl(), map_int(), map_dbl(), map_chr()): These functions extend the functionality of base R's apply functions by providing a consistent interface for iterating over lists and vectors, with variants to return logical, integer, double, or character vectors.
- Advanced Mapping (map2(), pmap()): map2() and pmap() allow you to apply a function over multiple lists or vectors simultaneously, making it easy to handle more complex iteration scenarios with multiple inputs.
- Side Effects (walk(), walk2(), pwalk()): These functions are used when you need to perform actions for their side effects (like printing or plotting) rather than returning values. They work similarly to the map() functions but do not return a result.
- Predicate Functions (detect(), detect_index(), head_while(), tail_while(), some(), every()): These functions help in searching and validating elements within lists or vectors based on certain conditions, such as finding the first match or checking if all elements meet a criteria.
- Reduction and Accumulation (reduce(), accumulate()): reduce() is used to iteratively combine elements of a list or vector into a single value using a binary function, whereas accumulate() returns intermediate results, useful for tracking the progression of the reduction.
- Error Handling (safely(), possibly(), quietly()): These functions provide robust mechanisms for handling errors in functional operations. safely() captures errors without stopping execution, possibly() provides default values on errors, and quietly() captures all outputs, messages, and warnings for better debugging.
- Function Invocation (invoke_map()): invoke_map() allows invoking functions from a list of functions, useful for dynamically applying different functions to data, making it highly versatile for complex functional programming tasks.

Model

- Use programming tools to create and understand various models, focusing on exploring data rather than confirming hypotheses or formal statistical inference.
- Models simplify data by highlighting significant patterns and ignoring random noise, aiding in data analysis and intuitive understanding.



- Discuss predictive models that generate predictions rather than those exploring data relationships.
- Emphasize qualitative model evaluation and skepticism over quantitative assessment.
- Employ multiple simple models to explore complex datasets, using linear models and simple simulated data to gain insights.
- Use models to highlight known patterns in real data and conduct exploratory data analysis (EDA) to identify data issues before modeling.
- For hypothesis confirmation, partition data into:
    - 60% for training (exploration) to experiment with various models.
    - 20% for comparison (query) to manually compare models.
    - 20% for final testing (test) to evaluate the chosen model once.
- Proper data partitioning ensures the separation of exploration and confirmation, maintaining the integrity of hypothesis testing.

Model basics with modelr

Function	Description	Parameters	Example Statement
data_grid()	Creates a data frame with all combinations of the given variables.	.data (data frame), ... (variables)	data_grid(mtcars, cyl, vs)
add_predictions()	Adds a column of model predictions to a data frame.	.data (data frame), model, var (name)	add_predictions(mtcars, model)
add_residuals()	Adds a column of residuals to a data frame.	.data (data frame), model, var (name)	add_residuals(mtcars, model)
model_matrix()	Creates a model matrix from a data frame.	.data (data frame), ... (variables)	model_matrix(mtcars, mpg ~ cyl + wt)
seq_range()	Creates a sequence of values within the range of a variable.	x (variable), n (number of values)	seq_range(mtcars$mpg, 10)
glm()	Fits a generalized linear model.	formula, data, family	glm(mpg ~ hp + wt, data = mtcars, family = gaussian())
gam()	Fits a generalized additive model.	formula, data	gam(mpg ~ s(hp) + wt, data = mtcars)
glmnet()	Fits a generalized linear model via penalized maximum likelihood.	x, y, alpha, lambda	glmnet(as.matrix(mtcars[, -1]), mtcars$mpg)
rlm()	Fits a robust linear model.	formula, data	rlm(mpg ~ hp + wt, data = mtcars)
rpart()	Fits a recursive partitioning and regression trees model.	formula, data	rpart(mpg ~ hp + wt, data = mtcars)

- Model Objectives: The primary goal of a model is to provide a concise, low-dimensional summary of a dataset by partitioning it into patterns and residuals. This approach helps in

uncovering both prominent and subtle trends. Initially, simulated datasets are used to grasp modeling fundamentals before applying techniques to real-world data.

- Components of a Model: Model Family Definition: This involves specifying a generic pattern to capture, such as linear or quadratic relationships. The pattern is expressed with equations where variables are known, and parameters are adjusted to capture different patterns.
- Model Fitting: This step finds the model from the family that best approximates the data, resulting in a specific model. It is important to note that a fitted model is an approximation rather than the absolute truth.
- Nature of Models: As George Box famously stated, "All models are wrong, but some are useful." This means that while no model can perfectly represent real-world systems, well-chosen models can provide valuable insights and approximations.
- Data Grid and Predictions: Functions such as data_grid(), add_predictions(), and add_residuals() are essential for generating data frames that help in modeling. They facilitate the creation of new datasets with combinations of variables, predictions, and residuals, enhancing the model analysis process.
- Model Fitting and Optimization: Various model fitting functions, including glm() for generalized linear models, gam() for generalized additive models, glmnet() for penalized regression models, rlm() for robust linear models, and rpart() for decision trees, are used to apply different modeling techniques. Systematic methods such as grid searches and numerical optimization (e.g., optim()) refine model fitting to find optimal parameters.
- Model Matrices and Sequences: Creating model matrices using model_matrix() is crucial for preparing data for regression models. Additionally, functions like seq_range() help in generating sequences of values within the range of variables, aiding in grid data creation and visualization.
- Visualization and Analysis: Visualizing predictions and residuals helps in understanding model performance. Tools like add_predictions() and add_residuals() are used to append these results to data frames, which can then be plotted to diagnose model fit. Residuals indicate model deficiencies and can be analyzed to improve the model.
- Formulas and Categorical Variables: R's formula notation supports specifying various predictors, including continuous, categorical, and interactions. Categorical variables are converted into binary indicators, and interactions between continuous and categorical variables are modeled using appropriate formula specifications.

Model Building: A Case Study on Diamond Prices

- We delve into real data to illustrate the process of building and refining a model. Our objective is to make implicit knowledge in the data explicit through quantitative models. We aim to balance predictive accuracy with interpretability. The case study on diamond prices demonstrates model building using tools from the `tidyverse` and `modelr` packages, with datasets from `ggplot2` and `nycflights13`.
- Prerequisites: We will use the following packages and datasets:

```
library(tidyverse)
library(modelr)
library(lubridate)
data("diamonds", package = "ggplot2")
data("flights", package = "nycflights13")
```

- Exploring Diamond Prices: Initial exploration of diamond prices against quality factors such as cut, color, and clarity revealed some unexpected trends:

```
ggplot(diamonds, aes(cut, price)) + geom_boxplot()
ggplot(diamonds, aes(color, price)) + geom_boxplot()
ggplot(diamonds, aes(clarity, price)) + geom_boxplot()
```

These plots suggested that lower-quality diamonds tend to have higher prices, contrary to expectations.

- Price and Carat: Carat weight significantly impacts diamond price. We visualized this relationship using hex plots:

```
ggplot(diamonds, aes(carat, price)) + geom_hex(bins = 50)
```

To clarify the pattern, we filtered and log-transformed the variables:

```
diamonds2 <- diamonds %>%
  filter(carat <= 2.5) %>%
  mutate(lprice = log2(price), lcarat = log2(carat))
```

- Linear Modeling: A simple linear model highlights the relationship between carat and price:

```
mod_diamond <- lm(lprice ~ lcarat, data = diamonds2)
diamonds2 <- diamonds2 %>%
  add_predictions(mod_diamond) %>%
  add_residuals(mod_diamond)
```

Visualizing the model and residuals confirmed that the strong linear pattern was effectively captured:

```
ggplot(diamonds2, aes(x = lcarat, y = lprice)) +
  geom_point() +
  geom_line(aes(y = pred), color = "red")
```

- Residual Analysis: Analyzing residuals provided clearer insights into the effects of cut, color, and clarity:

```
ggplot(diamonds2, aes(cut, resid)) + geom_boxplot()
ggplot(diamonds2, aes(color, resid)) + geom_boxplot()
ggplot(diamonds2, aes(clarity, resid)) + geom_boxplot()
```

- Building a Comprehensive Model: Incorporating additional predictors such as color, cut, and clarity improved our understanding of their combined effects on price:

```
mod_diamond2 <- lm(lprice ~ lcarat + color + cut + clarity, data = diamonds2)
diamonds2 <- diamonds2 %>%
  add_predictions(mod_diamond2) %>%
  add_residuals(mod_diamond2)
```

Analyzing residuals from this more complex model helped identify outliers and confirmed the improved fit:

```
ggplot(diamonds2, aes(x = lcarat, y = lprice)) +
  geom_point() +
  geom_line(aes(y = pred), color = "blue")
```

- Functions Utilized in Model Building:

1. data_grid(): Creates a grid of data for prediction.

```
grid <- diamonds2 %>%
  data_grid(cut, color, clarity)
```

2. add_predictions(): Adds model predictions to the data frame.

```
grid <- grid %>%
  add_predictions(mod_diamond2)
```

3. add_residuals(): Adds residuals to the data frame.

```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond2)
```

4. model_matrix(): Creates a model matrix for more complex models.

```
model_matrix(diamonds2, lprice ~ lcarat + color + cut + clarity)
```

5. seq_range(): Generates sequences within the range of carat.

```
seq_range(diamonds2$lcarat, n = 100)
```

6. glm(): Fits generalized linear models.

```
mod_glm <- glm(lprice ~ lcarat + color + cut + clarity, data = diamonds2)
```

7. gam(): Fits generalized additive models.

```
library(mgcv)
mod_gam <- gam(lprice ~ s(lcarat) + color + cut + clarity, data = diamonds2)
```

8. glmnet(): Fits penalized regression models.

```
library(glmnet)
x <- model.matrix(lprice ~ lcarat + color + cut + clarity, data = diamonds2)
y <- diamonds2$lprice
mod_glmnet <- glmnet(x, y)
```

9. rlm(): Fits robust linear models.

```
library(MASS)
mod_rlm <- rlm(lprice ~ lcarat + color + cut + clarity, data = diamonds2)
```

10. rpart(): Fits decision tree models.

```
library(rpart)
mod_rpart <- rpart(lprice ~ lcarat + color + cut + clarity, data = diamonds2)
```

Conclusion: Model building is an iterative process involving the following steps:

1. Exploring Initial Data Relationships: Visualize data to identify patterns and anomalies.
2. Building and Refining Linear Models: Start with simple models and gradually introduce complexity.
3. Analyzing Residuals for Insights: Residual analysis helps diagnose model fit and identify influential factors.
4. Incorporating Additional Predictors: Add relevant variables to improve model accuracy and understanding.
5. Utilizing Specialized Functions: Use appropriate functions to create, fit, and analyze models, gaining deeper insights into the data.

Many Models with Purrr and Broom

- Introduction to Modeling and List-Columns: We explore the use of multiple simple models to gain deeper insights from complex datasets. This approach involves using list-columns within data frames to store various data structures, such as models generated from different subsets of data. The `broom` package is essential for transforming these models into tidy data, facilitating further analysis and visualization.
- Motivating Example with Gapminder Data: We start with an example using the Gapminder dataset, which tracks life expectancy, population, and GDP across different countries over time. Initial visualizations reveal global trends but also highlight the challenges in discerning more subtle country-specific patterns.
- Using Many Models to Understand Data: To address the complexity of global versus country-specific trends, we fit simple linear models to each country's data. This approach helps isolate local variations from global trends, making it easier to identify outliers and unique patterns within specific countries.

- Working with Nested Data Frames: To efficiently manage and analyze models for each country, we structure the data into nested data frames. Each row in the nested data frame represents a country, with a list-column containing its corresponding data over time. This structure simplifies the application of model-fitting functions across multiple subsets of data.
- Creating and Storing Models: Using `purrr`'s `map()` function, we apply a model-fitting function to each country's data within the nested structure. The resulting models are stored as list-columns within the data frame. This method ensures that related objects (data frames and models) are kept together, simplifying management and analysis tasks.
- Simple Example with `nest()` and `unnest()`:

1. Loading Necessary Libraries and Data:

```
library(tidyverse)
library(modelr)
library(broom)
library(gapminder)
```

2. Creating Nested Data Frame: We group the data by country and nest it.

```
nested_gap <- gapminder %>%
  group_by(country) %>%
  nest()
```

3. Fitting Models to Each Country's Data: We apply a linear model to each country's data to predict life expectancy based on the year.

```
nested_gap <- nested_gap %>%
  mutate(model = map(data, ~ lm(lifeExp ~ year, data = .)))
```

4. Extracting Model Summaries with Broom: We use the `broom` package's `glance()` function to extract model performance metrics.

```
nested_gap <- nested_gap %>%
  mutate(glance = map(model, glance))
```

5. Unnesting to Tidy Data Frame: The `unnest()` function from the `tidyr` package flattens the list-columns into a tidy data frame.

```
glance_data <- nested_gap %>%
  unnest(glance)
```

6. Visualizing Model Performance: We plot the R-squared values to evaluate model performance across countries.

```
ggplot(glance_data, aes(x = r.squared, y = country)) +
  geom_point() +
  theme_minimal() +
  labs(title = "Model Performance by Country",
       x = "R-squared",
       y = "Country")
```

- Evaluating Model Performance: After fitting the models, evaluating their performance is crucial. The `broom` package's `glance()` function extracts model performance metrics such as \(R^2\) and residuals. This step helps identify countries where the model fits poorly, indicating unique challenges or phenomena not captured by global trends.
- Visualizing and Analyzing Results: We conclude with visualizations and analyses based on the model outputs. Techniques include plotting residuals across countries to identify outliers and using faceting to explore regional patterns in model performance metrics. These insights enable a deeper understanding of global and local trends in life expectancy and other variables.
- Conclusion and Considerations: Working with nested data frames and list-columns is powerful but requires familiarity with modeling, data structures, and iterative processes. Revisiting these materials after gaining foundational knowledge in R and statistics is recommended. The practical utility of these techniques in data science workflows cannot be overstated, offering robust tools for detailed and insightful data analysis.

Communicate Effectively with R Markdown and Graphics

- R Markdown for Integration: R Markdown allows for the seamless integration of prose, code, and results within a single document, facilitating comprehensive and clear communication.
  - Combining Elements: Use R Markdown to combine narrative text, executable R code, and the results of that code, ensuring a cohesive and easily navigable document.
  - Modes of Use:
    - Notebook Mode: This mode is ideal for sharing detailed analyses with fellow analysts. It includes all the steps, code, and intermediate results, making it perfect for collaborative work and in-depth examination.
    - Report Mode: Tailored for presenting final results to decision-makers. This mode focuses on key insights and conclusions with minimal code, ensuring clarity and conciseness for non-technical stakeholders.
  - Flexibility: Create documents that serve dual purposes, allowing you to switch between detailed and summarized views as needed. This flexibility ensures that the same document can cater to both technical and non-technical audiences.
- Creating Effective Expository Graphics: Transforming exploratory graphics into polished, expository graphics helps clearly convey your findings to a wider audience.
  - Purpose: Ensure your graphics communicate the intended message clearly and effectively, making complex data easily understandable.
  - Key Elements:
    - Clarity: Ensure that your graphics are easy to understand at a glance. Avoid unnecessary complexity to prevent confusion.
    - Annotation: Use labels, titles, and captions to provide context and guide the viewer through the data. Effective annotation makes your graphics more informative and accessible.
    - Design Principles: Apply design principles such as alignment, color usage, and visual hierarchy. These principles enhance the readability and impact of your graphics, making them more professional and compelling.
- Expanding Output Formats with R Markdown: R Markdown's versatility allows you to generate a variety of output formats, catering to different audience needs and contexts.
  - Versatility: Beyond traditional documents, R Markdown can produce a range of dynamic and interactive formats.
    - Dashboards: Create interactive dashboards that allow users to explore data and insights in real-time. Dashboards are particularly useful for presenting dynamic data and facilitating user engagement.
    - Websites: Develop informative and engaging websites to share your analyses with a broader audience. Websites can host interactive content and provide a platform for wider dissemination.
    - Books: Compile comprehensive analyses and tutorials into book formats. These can serve educational purposes or provide detailed documentation for complex projects.
  - Customization: Tailor the appearance and functionality of these outputs to suit the needs of different audiences and contexts. Customization ensures that your content is relevant and engaging for its intended users.
- Systematic Recording with Analysis Notebooks: Maintaining a detailed record of your analysis process is crucial for learning, improvement, and reproducibility.

- Documentation: Use analysis notebooks to keep a thorough record of your analysis process, including code, results, and reflections on what worked and what didn't. Detailed documentation supports transparency and accountability in your work.
- Learning and Improvement: Track your progress and identify patterns in successes and failures. Analysis notebooks help you reflect on your methods and continuously refine your analytical approach.
- Reproducibility: Ensure that your analyses are reproducible by keeping clear and organized records. Reproducibility facilitates collaboration and future reference, making it easier for others (or yourself) to understand and build upon your work.
- By leveraging R Markdown for integration, creating effective expository graphics, expanding output formats, and systematically recording your analysis, you can significantly enhance the clarity, impact, and reproducibility of your data communication.

R Markdown

- R Markdown serves as a unified authoring framework that integrates code, results, and commentary, making it an ideal tool for data science. It ensures full reproducibility and supports various output formats, including PDFs, Word files, and slideshows.
- Uses of R Markdown: R Markdown is versatile in its applications. It helps in communicating with decision makers by focusing on conclusions rather than the code. It facilitates collaboration among data scientists by sharing both conclusions and the underlying code. Additionally, it acts as a lab notebook within a data science environment, capturing actions and thoughts for thorough documentation.
- Essential Resources: To maximize the utility of R Markdown, several resources are available. Cheat sheets can be accessed within the RStudio IDE under Help → Cheatsheets. Key references include the R Markdown Cheat Sheet and the R Markdown Reference Guide. Online resources such as the RStudio Cheat Sheets provide additional support and guidance.
- Prerequisites: The primary prerequisite for using R Markdown is the `rmarkdown` package, which is automatically installed and loaded by RStudio when necessary. This package enables the functionality required to create and render R Markdown documents.
- Basics: An R Markdown file (.Rmd) consists of a YAML header, code chunks, and text formatted with Markdown. The notebook interface allows for the execution of code and displays the results inline, making it easier to follow and understand. The knitting process combines text, code, and results into a final report through the `knitr` and `pandoc` tools.
- Text Formatting with Markdown: Markdown provides various formatting options to enhance the readability and structure of the text. This includes italic and bold text, inline code formatting, superscripts, and subscripts. Headers are created using `#` for main headings and additional `#` symbols for sub-headings. Lists can be bulleted or numbered, and both hyperlinks and images can be embedded within the document. Simple tables with headers and cells can also be created using Markdown syntax.
- Code Chunks: Code chunks in R Markdown allow for the inclusion and execution of R code within the document. These chunks can be inserted using a keyboard shortcut (Cmd/Ctrl-Alt-I), an insert button, or manual delimiters. Running chunks can be done with the Cmd/Ctrl-Shift-Enter shortcut. Various chunk options are available to control the evaluation, inclusion, visibility of code, and handling of messages, warnings, and errors.

- Tables and Figures: For customizing table formatting, the `knitr::kable` function can be used. Caching options in R Markdown help save chunk outputs to avoid re-running expensive computations. This includes enabling caching, tracking dependencies, and monitoring external file changes that might affect the results.
- Global Options: The `knitr::opts_chunk$set()` function allows users to change default chunk options for the entire document. This can be useful for setting global behaviors such as hiding code by default.
- Inline Code: R Markdown supports embedding R code directly into text, allowing for dynamic updates of content based on code execution. For example, `We have data about `r nrow(diamonds)` diamonds.` dynamically inserts the number of rows in the diamonds dataset.
- Troubleshooting: When encountering issues, recreating problems in an interactive session can help identify the cause. Ensuring the working directory is set correctly is also important. Systematic checks using `print()` and `str()` functions can assist in diagnosing issues within the code.
- YAML Header: The YAML header in an R Markdown document controls various settings, including document parameters and bibliography. Parameters can be set to allow for re-rendering reports with different inputs. Bibliography files and CSL styles enable automatic citation generation, streamlining the documentation process.

Graphics for Communication with ggplot2

- Adding Titles and Labels: Adding titles and labels to your plots enhances clarity and context. The labs() function is used to add titles, subtitles, and captions to ggplot2 plots.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    title = "Fuel efficiency generally decreases with engine size",
    subtitle = "Two seaters (sports cars) are an exception because of their light weight",
    caption = "Data from fueleconomy.gov"
  )
```

- Axis and Legend Titles: To make plots more informative, replace short variable names with detailed descriptions and units using labs().

```
labs(
  x = "Engine displacement (L)",
  y = "Highway fuel economy (mpg)",
  colour = "Car type"
)
```

- Mathematical Expressions in Labels: For mathematical equations in axis labels, use the quote() function.

```
labs(
  x = quote(sum(x[i] ^ 2, i == 1, n)),
  y = quote(alpha + beta + frac(delta, theta))
)
```

- Annotations:
  1. Adding Text Labels: Add textual labels to plots using geom_text() or geom_label().

```
geom_text(aes(label = model), data = best_in_class)
```

2. Adjusting Label Positions: Adjust label positions using nudge_y and avoid overlap with ggrepel::geom_label_repel().

```
ggrepel::geom_label_repel(aes(label = model), data = best_in_class)
```

3. Direct Labeling: Replace legends with direct labels on the plot using ggrepel::geom_label_repel() and disable legends with theme(legend.position = "none").

```
ggrepel::geom_label_repel(aes(label = class), data = class_avg, size = 6, label.size = 0, segment.color = NA) +
theme(legend.position = "none")
```

4. Adding Single Labels: Create a single label and place it at specific plot locations using summarize() or tibble().

```
tibble(
  displ = Inf,
  hwy = Inf,
  label = "Increasing engine size is \nrelated to decreasing fuel economy."
)
```

- Scales:
  1. Continuous Position Scales: Control x- and y-scales using functions like scale_x_continuous(), scale_y_continuous(), xlim(), ylim(), scale_x_log10(), and scale_y_log10().

```
xlim(1, 10)
```

  2. Color Scales: Manually set colors with scale_color_manual() or use predefined palettes with scale_color_brewer() and viridis::scale_color_viridis().

```
scale_color_manual(values = c("2seater" = "red", "compact" = "blue", ...))
```

- Legends:
  1. Modifying Legends: Change legend titles, order, or remove legends using guides().

```
guides(color = guide_legend(title = "Car type", reverse = TRUE))
```

  2. Removing Legends: Disable all legends with theme(legend.position = "none").

```
theme(legend.position = "none")
```

- Themes:
  1. Changing Themes: Change the overall appearance of the plot using theme_set().

```
theme_set(theme_minimal())
```

  2. Customizing Themes: Modify existing themes using theme() to control elements such as text size, background colors, and axis text.

```
theme(
  text = element_text(size = 16),
  panel.background = element_rect(fill = "black"),
  plot.background = element_rect(fill = "black"),
  axis.text = element_text(color = "white"),
  axis.title = element_text(color = "white")
)
```

- Saving Plots:Using ggsave(), save plots in various formats like PNG, PDF, and SVG by specifying the filename, width, height, and resolution.

```
ggsave("plot.png")
```

R Markdown Formats

- R Markdown is a versatile tool for creating a variety of output formats from a single source document. Here's an overview of the different types of outputs and how to set and customize them.
- Setting Output Format: You can set the output format in two ways:
  1. Permanently, by modifying the YAML header:

```
title: "Viridis Demo"
output: html_document
```

  2. Transiently, by calling rmarkdown::render():

```
rmarkdown::render("diamond-sizes.Rmd", output_format = "word_document")
```

  RStudio's knit button renders a file to the first format listed in its output field. For additional formats, use the drop-down menu beside the knit button.
- Output Options: Each output format is associated with an R function. Check the parameters you can set using the help documentation for the function:

```
?rmarkdown::html_document
```

  1. Override default parameters by expanding the output field in the YAML header:

```
output:
  html_document:
    toc: true
    toc_float: true
```

  2. Render to multiple outputs by supplying a list of formats:

```
output:
  html_document:
    toc: true
    toc_float: true
  pdf_document: default
```

- Document Types:
  - html_document: Produces an HTML document.
  - pdf_document: Produces a PDF using LaTeX.
  - word_document: Produces a Microsoft Word document (.docx).
  - odt_document: Produces an OpenDocument Text document (.odt).
  - rtf_document: Produces a Rich Text Format document (.rtf).
  - md_document: Produces a Markdown document.
  - github_document: Produces a Markdown document tailored for GitHub.
- Notebooks: Notebooks (html_notebook) are similar to html_document but designed for collaboration. They contain the fully rendered output and the full source code, making them useful for sharing analyses with colleagues.
- Presentations: R Markdown can produce various presentation formats:
  - ioslides_presentation: HTML presentation with ioslides.
  - slidy_presentation: HTML presentation with W3C Slidy.
  - beamer_presentation: PDF presentation with LaTeX Beamer.
  - revealjs::revealjs_presentation: HTML presentation with reveal.js.
  - rmdshower: Wrapper around the shower presentation engine.
- Dashboards: Dashboards are created using the flexdashboard package, organizing content into pages, columns, and rows:

```
title: "Diamonds distribution dashboard"
output: flexdashboard::flex_dashboard
```

- Interactivity: Any HTML format can include interactive components using htmlwidgets or Shiny.
  - htmlwidgets: Functions that produce interactive HTML visualizations. Examples include leaflet for maps, dygraphs for time series, and DT for tables.
  - Shiny: Allows creating interactivity using R code, requiring a Shiny server to run online.
- Websites: R Markdown can generate a complete website by organizing .Rmd files in a directory with index.Rmd as the home page. Use _site.yml for navigation:

```
name: "my-website"
navbar:
  title: "My Website"
  left:
    - text: "Home"
      href: index.html
    - text: "Viridis Colors"
      href: 1-example.html
    - text: "Terrain Colors"
      href: 3-inline.html
```

- Other Formats: Additional packages provide more output formats:
  - bookdown: For writing books.
  - prettydoc: For lightweight documents with attractive themes.
  - rticles: For formats tailored for specific scientific journals.