4.1 Preparing Data using Tidyr and Stringr Packages

1. Introduction to `tidyr`

`tidyr` helps to organize and clean data so that it is easy to analyze. It focuses on transforming data into a "tidy" format, where:

- Each variable is a column.

- Each observation is a row.

- Each value is a single cell.

1.1 Installing and Loading `tidyr`

```
# Install the package if you haven't already
# install.packages("tidyr")

# Load tidyr
library(tidyr)
```

1.2 Common Functions in `tidyr`

Reshaping Data, Long and Wide Formats:

Long format data is common in various real-life scenarios, especially when dealing with time series, repeated measures, or categorical data. Here are a few examples:

Real-Life Examples of Long Format Data:

1. **Healthcare Data:**
    - **Example:** Patient data often comes in long format when tracking vital signs, medication doses, or symptoms over time. Each row might represent a single measurement for a specific patient at a particular time point.
    - **Why:** This format makes it easier to analyze trends, apply statistical models, and visualize changes over time.

Patient_ID	Date	Measurement	Value
001	2024-08-01	BloodPressure	120/80

Patient_ID	Date	Measurement	Value
001	2024-08-02	BloodPressure	130/85
002	2024-08-01	HeartRate	70
002	2024-08-02	HeartRate	72

2. **Survey Data:**
   - **Example:** When analyzing survey responses, each respondent might have multiple answers for different questions. Instead of having separate columns for each question, long format organizes it by question and response.
   - **Why:** Long format is useful for summarizing, visualizing, and performing statistical tests across different questions or groups of respondents.

Respondent	Question	Response
101	Q1	Yes
101	Q2	No
102	Q1	No
102	Q2	Yes

3. **Time Series Data:**
   - **Example:** Financial data often tracks metrics like stock prices, sales, or revenue over time. Each entry in the long format represents a single observation at a specific time point.
   - **Why:** This format is essential for time series analysis, forecasting, and modeling temporal trends.

Date	Company	Metric	Value
2024-08-01	A	Revenue	1000
2024-08-01	B	Revenue	1500
2024-08-02	A	Revenue	1100
2024-08-02	B	Revenue	1600

Why Are R Functions Like `pivot_longer()` and `pivot_wider()` Important?

1. **Data Preparation for Analysis:**
   - Many statistical models, especially those for repeated measures, mixed-effects models, or time series analysis, require data in long format.
   - Visualization tools like `ggplot2` in R often prefer data in long format for plotting.
2. **Flexibility in Data Transformation:**

- Having the ability to switch between wide and long formats allows you to adapt your data structure to the needs of different analyses, making your workflow more efficient.
- `pivot_longer()` and `pivot_wider()` automate this process, saving time and reducing the potential for manual errors.

3. **Interoperability:**
   - Different tools and libraries might expect data in different formats. By converting between wide and long formats, you can ensure compatibility across tools, whether you're doing machine learning, statistical analysis, or data visualization.

1.3 pivot_longer()

Converts data from wide to long format.

Converts wide data into long data. It's useful when you want to convert several columns into key-value pairs.

```
# Example: Converting sales data from wide to long format

library(tidyr)

# Original wide data frame
wide_data <- data.frame(
  Student = c("Alice", "Bob", "Carol"),
  Math_Score = c(85, 90, 75),
  English_Score = c(78, 88, 82),
  Science_Score = c(92, 85, 80),
  History_Score = c(88, 90, 78),
  Art_Score = c(79, 86, 85),
  Music_Score = c(84, 90, 83),
  PE_Score = c(91, 88, 82)
)

print(wide_data)
```

```
  Student Math_Score English_Score Science_Score History_Score Art_Score
1   Alice         85            78            92            88        79
2     Bob         90            88            85            90        86
3   Carol         75            82            80            78        85
  Music_Score PE_Score
1          84       91
2          90       88
3          83       82
```

```
# Convert to long format
long_data <- pivot_longer(
  wide_data,
  cols = starts_with("Math_Score"):starts_with("PE_Score"),
  names_to = "Course",
  values_to = "Score"
```

```
)

# Print long format data
print(long_data)
```

```
# A tibble: 21 × 3
   Student Course        Score
   <chr>   <chr>         <dbl>
 1 Alice   Math_Score       85
 2 Alice   English_Score    78
 3 Alice   Science_Score    92
 4 Alice   History_Score    88
 5 Alice   Art_Score        79
 6 Alice   Music_Score      84
 7 Alice   PE_Score         91
 8 Bob     Math_Score       90
 9 Bob     English_Score    88
10 Bob     Science_Score    85
# ℹ 11 more rows
```

We can also explicitly specify the columns

```
# Convert to long format using specific column names
long_data <- pivot_longer(
  wide_data,
  cols = c(Math_Score, English_Score, Science_Score, History_Score, Art_Score, Music_Score, PE_Sc
  names_to = "Course",
  values_to = "Score"
)

# Print long format data
print(long_data)
```

```
# A tibble: 21 × 3
   Student Course        Score
   <chr>   <chr>         <dbl>
 1 Alice   Math_Score       85
 2 Alice   English_Score    78
 3 Alice   Science_Score    92
 4 Alice   History_Score    88
 5 Alice   Art_Score        79
 6 Alice   Music_Score      84
 7 Alice   PE_Score         91
 8 Bob     Math_Score       90
 9 Bob     English_Score    88
10 Bob     Science_Score    85
# ℹ 11 more rows
```

1.4 pivot_wider():

Converts long data into wide data. It's the inverse of pivot_longer().

```
# Example: Converting long sales data back to wide format

wide_data_again <- pivot_wider(
  long_data,
  names_from = Course,
  values_from = Score
)

# Print wide format data
print(wide_data_again)
```

```
# A tibble: 3 × 8
  Student Math_Score English_Score Science_Score History_Score Art_Score
  <chr>        <dbl>         <dbl>         <dbl>         <dbl>     <dbl>
1 Alice           85            78            92            88        79
2 Bob             90            88            85            90        86
3 Carol           75            82            80            78        85
# ℹ 2 more variables: Music_Score <dbl>, PE_Score <dbl>
```

1.5 separate()

Splits one column into multiple columns.

Example: Split a column containing "Date-Time" into separate "Date" and "Time" columns.

```
# Example dataset
data <- data.frame(
  ID = 1:3,
  DateTime = c("2024-11-01 10:30", "2024-11-02 14:45", "2024-11-03 18:00")
)

# Separate DateTime into Date and Time
separated_data <- data %>%
  separate(DateTime, into = c("Date", "Time"), sep = " ")

print(separated_data)
```

```
  ID       Date  Time
1  1 2024-11-01 10:30
2  2 2024-11-02 14:45
3  3 2024-11-03 18:00
```

Output:

```
  ID       Date   Time
1  1 2024-11-01 10:30
```

```
2  2 2024-11-02 14:45
3  3 2024-11-03 18:00
```

1.6 unite()

Combines multiple columns into one column.

Example: Combine “First” and “Last” name columns.

```r
# Example dataset
data <- data.frame(
  First = c("John", "Jane", "Jake"),
  Last = c("Doe", "Smith", "Johnson")
)

# Unite First and Last into FullName
united_data <- data %>%
  unite("FullName", First, Last, sep = " ")

print(united_data)
```

```
      FullName
1     John Doe
2   Jane Smith
3 Jake Johnson
```

Output:

```
         FullName
1       John Doe
2     Jane Smith
3 Jake Johnson
```

1.7 drop_na()

Removes rows with missing values.

Example: Drop rows where any value is missing.

```r
# Example dataset
data <- data.frame(
  Name = c("Alice", "Bob", NA),
  Age = c(25, 30, 35)
)

# Drop rows with NA
clean_data <- data %>%
  drop_na()
```

```
print(clean_data)
```

```
   Name Age
1 Alice  25
2   Bob  30
```

Output:

```
   Name Age
1 Alice  25
2   Bob  30
```

1.8 fill()

Fills missing values with the last non-missing value.

Example: Fill down missing values.

```
# Example dataset
data <- data.frame(
  Group = c("A", NA, NA, "B", NA),
  Value = c(10, 20, 30, 40, 50)
)

# Fill missing Group values
filled_data <- data %>%
  fill(Group, .direction = "down")

print(filled_data)
```

```
  Group Value
1     A    10
2     A    20
3     A    30
4     B    40
5     B    50
```

Output:

```
  Group Value
1     A    10
2     A    20
3     A    30
4     B    40
5     B    50
```

1.9 replace_na()

Replaces missing values with a specified value.

Example: Replace missing values in “Score” with 0.

```
# Example dataset
data <- data.frame(
  Name = c("Tom", "Jerry", "Spike"),
  Score = c(95, NA, 88)
)

# Replace NA with 0
replaced_data <- data %>%
  replace_na(list(Score = 0))

print(replaced_data)
```

```
   Name Score
1   Tom    95
2 Jerry     0
3 Spike    88
```

1.10 Combining `tidyr` with dplyr

You can combine `tidyr` with `dplyr` for powerful data manipulation.

Example: Tidy data and calculate summary statistics.

```
# Load the necessary library
library(tidyr)
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
# Example data
data <- data.frame(
  Product = c("A", "B", "C"),
  `year_2020` = c(500, 300, 200),
  `year_2021` = c(600, 400, 300),
  `year_2022` = c(700, 500, 400)
)
```

```
# View the original data
print("Original Data:")
```

```
[1] "Original Data:"
```

```
print(data)
```

```
  Product year_2020 year_2021 year_2022
1       A       500       600       700
2       B       300       400       500
3       C       200       300       400
```

```
# Use pivot_longer to reshape the data to a long format
long_data <- data %>%
  pivot_longer(
    cols = starts_with("year_20"),  # Specify columns to pivot (years in this case)
    names_to = "Year",         # Create a 'Year' column from the column names
    values_to = "Sales"        # Create a 'Sales' column from the values
  )

print(long_data)
```

```
# A tibble: 9 × 3
  Product Year      Sales
  <chr>   <chr>     <dbl>
1 A       year_2020   500
2 A       year_2021   600
3 A       year_2022   700
4 B       year_2020   300
5 B       year_2021   400
6 B       year_2022   500
7 C       year_2020   200
8 C       year_2021   300
9 C       year_2022   400
```

```
# Aggregate total sales per product
aggregated_data <- long_data %>%
  group_by(Product) %>%       # Group by product
  summarise(Total_Sales = sum(Sales))  # Summarize to calculate total sales

# View the aggregated data
print("Aggregated Data:")
```

```
[1] "Aggregated Data:"
```

```
print(aggregated_data)
```

```
# A tibble: 3 × 2
  Product Total_Sales
  <chr>         <dbl>
1 A              1800
2 B              1200
3 C               900
```

Summary Table of Functions

Function	Purpose
`pivot_longer`	Convert wide data to long format
`pivot_wider`	Convert long data to wide format
`separate`	Split one column into multiple columns
`unite`	Combine multiple columns into one column
`drop_na`	Remove rows with missing values
`fill`	Fill missing values with previous/next one
`replace_na`	Replace missing values with specific value

---

2. Regular Expressions with `stringr` in R

Introduction

Regular expressions (regex) are patterns for finding, extracting, or replacing text. The `stringr` package in R simplifies regex usage.

In regex, special symbols and characters have specific meanings. To use these symbols literally, you need to **escape** them with a backslash `\`. In R, because `\` itself is a special character, you write **double backslashes** `\\` for regex.

Tutorial: Regular Expressions with `stringr` in R

Introduction to Regular Expressions

Regular expressions (regex) are patterns used to match and manipulate text. They are incredibly powerful for tasks such as data cleaning, validation, and text analysis. In R, the `stringr` package from the `tidyverse` simplifies working with strings and regex.

Installing and Loading `stringr`

If you don't have the `stringr` package installed, install it first:

```
# install.packages("stringr")
```

Load the package:

```
library(stringr)
```

Key Functions and Explicit Examples

2.1 Detect Patterns: `str_detect()`

Check if a pattern exists in each string.

```
text <- c("apple", "banana", "cherry")
str_detect(text, "an")
```

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

Explanation: - `"an"`: Matches the substring `"an"` literally. - The result is `TRUE` for "banana" because it contains `"an"`.

2.2 Extract Patterns: `str_extract()`

Extract the first match of a pattern.

```
text <- c("Born in 1990", "Graduated in 2005")
str_extract(text, "\\d{4}")
```

```
[1] "1990" "2005"
```

Explanation: - `\\d`: Matches a **digit** (0–9). The double backslash `\\` escapes the `\` so it works as part of the regex. - `{4}`: Matches exactly **four digits**. - Combined, `\\d{4}` looks for a 4-digit number (e.g., a year).

2.3 Extract All Matches: `str_extract_all()`

Extract multiple matches from a string.

```
text <- c("The price is $10.50 and $15.75.")
str_extract_all(text, "\\$\\d+\\.\\d{2}")
```

```
[[1]]
[1] "$10.50" "$15.75"
```

Explanation: - `\\$`: Matches the literal dollar sign `$`. The backslash escapes it. - `\\d+`: Matches one or more (`+`) digits. - `\\.\\d{2}`: Matches a literal dot (`\\.`) followed by exactly two digits (`\\d{2}`). - Combined, `\\$\\d+\\.\\d{2}` matches dollar amounts like `$10.50`.

2.4 Replace Patterns: `str_replace()`

Replace the first occurrence of a pattern.

```
text <- c("I love cats", "Cats are cute")
str_replace(text, "cats", "dogs")
```

```
[1] "I love dogs"   "Cats are cute"
```

Explanation: - `"cats"`: Matches the exact string `"cats"` (case-sensitive). - `"dogs"`: Replaces the first match of `"cats"` with `"dogs"`.

2.5 Replace All Patterns: `str_replace_all()`

Replace all occurrences of a pattern.

```
text <- c("Clean   this   text")
str_replace_all(text, "\\s+", " ")
```

```
[1] "Clean this text"
```

Explanation: - `\\s`: Matches any **whitespace** character (spaces, tabs, etc.). - `+`: Matches **one or more** occurrences of the preceding pattern. - `\\s+`: Matches one or more spaces. - `" "`: Replaces the matched spaces with a single space.

2.6 Count Matches: `str_count()`

Count occurrences of a pattern in each string.

```
text <- c("banana", "apple", "cherry")
str_count(text, "a")
```

```
[1] 3 1 0
```

Explanation: - `"a"`: Matches the letter `"a"`. - The function counts how many times `"a"` appears in each string.

2.7 Split Strings: `str_split()`

Split strings into parts based on a pattern.

```
text <- "apple,banana,cherry"
str_split(text, ",")
```

```
[[1]]
[1] "apple"  "banana" "cherry"
```

Explanation: - `","`: Matches a literal comma `,`. - Splits the string wherever a comma appears.

2.8 Real-World Applications

2.8.1. Cleaning Data: Remove Non-Numeric Characters

Clean messy phone numbers by keeping only digits.

```
phone_numbers <- c("(123) 456-7890", "123.456.7890")
cleaned <- str_replace_all(phone_numbers, "[^0-9]", "")
```

Explanation: - `[^0-9]`: Matches any character that is **not** a digit. - `[^...]`: Matches anything **not** inside the brackets. - `0-9`: Represents the range of digits from 0 to 9. - The pattern removes all non-digit characters.

2.8.2. Extracting Information: Find Years in Text

Extract years from sentences.

```
text <- c("Event in 2023", "Held in 1999")
years <- str_extract(text, "\\b\\d{4}\\b")
```

Explanation: - `\\b`: Matches a **word boundary**, ensuring the 4-digit number is a whole word. - `\\d{4}`: Matches a 4-digit number.

2.8.3. Standardizing Text: Remove Punctuation

Remove punctuation and convert text to lowercase.

```
text <- c("Hello, World!", "Good Morning!")
cleaned <- str_replace_all(str_to_lower(text), "[[:punct:]]", "")
```

Explanation: - `str_to_lower`: Converts text to lowercase. - `[[:punct:]]`: Matches any punctuation character (e.g., `.`, `,`, `!`). - Replaces punctuation with an empty string.

2.8.4. Validating Data: Check Email Addresses

Check if emails follow a valid format.

```
emails <- c("user@example.com", "invalid_email")
valid <- str_detect(emails, "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$")
```

Explanation: - `^`: Matches the **start** of the string. - `[A-Za-z0-9._%+-]+`: Matches one or more valid characters for the username part of an email. - `@`: Matches the literal `@` symbol. - `[A-Za-z0-9.-]+`: Matches the domain name (letters, numbers, dots, or hyphens). - `\\.[A-Za-z]{2,}$`: Matches a dot followed by 2 or more letters, ending the string (`$`).

2.8.5 Information Extraction From Text (emails)

```
library(stringr)
library(dplyr)

emails <- c( "Dear Alice, Welcome to our company! We are excited to have you on board. Your EmpNo
             "Hello Bob, Congratulations on your new position! Employee Number - 67890 | Salary -
             "Hi Carol, We're thrilled to welcome you to the team! Emp#: 54321 | Pay: 75000 USD |

extract_details_tidyr <- function(email) {
  emp_id <- str_extract(email, "(?<=EmpNo: |Employee Number - |Emp#: )\\d+")
  salary <- str_extract(email, "(?<=Salary: \\$|Salary - \\$|Pay: |Pay: \\$|Pay - \\$)\\d+")
  date <- str_extract(email, "(?<=Appointment Date: |Start Date: |Hired on: )\\d{4}[-/.]\\d{2}[-/

  data.frame(EmployeeID = emp_id, Salary = salary, AppointmentDate = date, stringsAsFactors = FAL
}

# Apply the function to all emails
results <- lapply(emails, extract_details_tidyr)

# Combine the results into a single data frame
final_results <- bind_rows(results)
print(final_results)
```

```
  EmployeeID Salary AppointmentDate
1      12345  55000      2023-06-01
2      67890  65000      2023/07/15
3      54321  75000      2023.08.20
```

- **Regex Explanation:**
  - `(?<=EmpNo: |Employee Number - |Emp#: )`: This is a "lookbehind" assertion, meaning it looks for the text that follows any of these prefixes: `EmpNo:`, `Employee Number -`, or `Emp#:`.
  - `\\d+`: This matches one or more digits, representing the Employee ID.
- This line extracts the numeric Employee ID based on different possible prefixes found in the email.

2.9 Regex Reference Table

Special Characters in Regex

Pattern	Meaning	Example
`\\d`	Digit (0–9)	`\\d{4}` matches “2023”
`\\w`	Word character (letters, numbers)	`\\w+` matches “apple”
`\\s`	Whitespace	`\\s+` matches spaces
`[^...]`	Not any character in brackets	`[^0-9]` removes non-digits
`.`	Any character except newline	`a.b` matches “a_b”
`^` / `$`	Start / End of string	`^Hello` matches “Hello World”
`[abc]`	Any character inside brackets	`[aeiou]` matches vowels
`{n}` / `{n,m}`	Exact or range of repetitions	`\\d{2,4}` matches “23” or “2023”
`*`	Matches zero or more occurrences of the preceding element.	`ca*t` matches “ct”, “cat”, or “caaaat” in “catapult”
`+`	Matches one or more occurrences of the preceding element	ca+t matches “cat” or “caaaat”, but not “ct”
`?`	Matches zero or one occurrence of the preceding element	`colou?r` matches “color” or “colour”
`|`	Acts as a logical OR	
`\`	Escapes special characters	\\. matches a literal “.” in “file.txt”