1.4.1 Preparing Data using Tidyr Package

The tidyr package in R is part of the tidyverse, a collection of R packages designed for data science. tidyr focuses on tidying data, meaning transforming it into a consistent format that makes it easy to analyze. The package simplifies the process of reshaping data, ensuring that it is structured in a way that is compatible with other tidyverse packages like `dplyr` and `ggplot2`.

Data Tidying:

The primary goal of tidyr is to help organize data into a tidy format: each variable forms a column, each observation forms a row, and each type of observational unit forms a table.

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
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.

pivot_longer()

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

```r
# 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
```

```r
# 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_Sco
  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
```

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>
```

Handling Missing Values:

- **drop_na()**: Removes rows with missing values.

```
          # Example: Dropping rows with missing values in a customer dataset

          customer_data <- data.frame(
            customer_id = c(1, 2, 3, 4),
            region = c("North", "South", NA, "West"),
            sales = c(5000, 7000, NA, 8500)
          )

          clean_customer_data <- drop_na(customer_data)
          clean_customer_data
```

```
  customer_id region sales
1           1  North  5000
2           2  South  7000
3           4   West  8500
```

- **replace_na()**: Replaces missing values with a specified value.

```
          # Example: Replacing missing values in customer data with defaults

          filled_customer_data <- replace_na(customer_data, list(region = "Unknown", sales = 0
          filled_customer_data
```

```
  customer_id  region sales
1           1   North  5000
2           2   South  7000
3           3 Unknown     0
4           4    West  8500
```

Splitting and Uniting Columns:

- **separate()**: Splits a single column into multiple columns based on a delimiter.

```
          # Example: Separating full name into first and last name

          employee_data <- data.frame(
            full_name = c("John Doe", "Jane Smith")
          )

          separated_employee_data <- separate(employee_data, full_name, into = c("first_name",
          separated_employee_data
```

```
  first_name last_name
1       John       Doe
2       Jane     Smith
```

- **unite()**: Combines multiple columns into a single column.

```
# Example: Uniting first and last names into full name

united_employee_data <- unite(separated_employee_data, "full_name", first_name, last
united_employee_data
```

```
   full_name
1   John Doe
2 Jane Smith
```

Creating New Variables with extract():

extract(): Extracts specific parts of a string from a column and stores them as new columns.

```
# Example: Extracting department code from an employee ID
employee_id_data <- data.frame(
  employee_id = c("HR-001", "IT-002", "FIN-003")
)

extracted_department_data <- extract(employee_id_data, employee_id, into = c("department
extracted_department_data
```

```
  department  id
1         HR 001
2         IT 002
3        FIN 003
```

Another example.

```
# extract info from hiring email using regex
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
```

```
library(tidyr)
library(stringr)
```

Sample emails

```
emails <- c( "Dear Alice, Welcome to our company! We are excited to have you on board. Y
             "Hello Bob, Congratulations on your new position! Employee Number - 67890 |
             "Hi Carol, We're thrilled to welcome you to the team! Emp#: 54321 | Pay: 75
```

Function to extract employee details

```
library(stringr)
library(dplyr)

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

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

# 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
```

The function is designed to extract specific details (Employee ID, Salary, and Appointment Date) from a series of emails, which contain this information in different formats. These details are then organized into a data frame for easier analysis.

Function Breakdown:

1. **Defining the Function:**

```
extract_details_tidyr <- function(email)
```

The function `extract_details_tidyr` takes a single argument `email`, which is expected to be a string (or character vector) containing the content of an email.

2. **Extracting Employee ID:**

```
emp_id <- str_extract(email, "(?<=EmpNo: |Employee Number - |Emp#: )\\d+")
```

- **`str_extract`:** This function from the `stringr` package is used to find and return the first match of a pattern in a string.
- **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.

3. **Extracting Salary:**

```
salary <- str_extract(email, "(?<=Salary: \\$|Salary - \\$|Pay: |Pay: \\$|Pay - \\$)\\d+
```

- **Regex Explanation:**
  - `(?<=Salary: \\$|Salary - \\$|Pay: |Pay: \\$|Pay - \\$)`: This is another lookbehind assertion, searching for different possible ways the salary could be introduced in the text.
  - `\\d+`: Matches the salary amount, which is a number.
- This line extracts the numeric salary amount from the email after identifying the correct prefix (like `Salary: $`, `Pay:`, etc.).

4. **Extracting Appointment Date:**

```
date <- str_extract(email, "(?<=Appointment Date: |Start Date: |Hired on: )\\d{4}[-/.]\\
```

- **Regex Explanation:**
  - `(?<=Appointment Date: |Start Date: |Hired on: )`: This is a lookbehind assertion for the possible date prefixes.
  - `\\d{4}[-/.]\\d{2}[-/.]\\d{2}`: This pattern matches a date in the format `YYYY-MM-DD`, `YYYY/MM/DD`, or `YYYY.MM.DD`.
- This line identifies and extracts the date of appointment/start from the email text.

5. **Creating a Data Frame:**

```
data.frame(EmployeeID = emp_id, Salary = salary, AppointmentDate = date, stringsAsFactor
```

- This line takes the extracted Employee ID, Salary, and Appointment Date and combines them into a data frame.
- **`stringsAsFactors = FALSE`:** Ensures that the extracted values remain as character strings instead of being converted into factors (a categorical data type in R).

6. **Applying the Function to All Emails:**

```
results <- lapply(emails, extract_details_tidyr)
```

- **`lapply`:** This function applies the `extract_details_tidyr` function to each element in the `emails` vector, which contains all the emails.
- **`results`:** This stores the output of applying the function to each email, resulting in a list of data frames.

7. **Combining Results:**

```
final_results <- bind_rows(results)
```

- **`bind_rows`:** This function from the `dplyr` package combines the list of data frames into a single data frame where each row corresponds to one email's extracted details.

Output:

The final output is a data frame with three columns: `EmployeeID`, `Salary`, and `AppointmentDate`. Each row corresponds to the extracted information from one email.