12. tidyr Deep Dive

Advanced reshaping, separating, and rectangling with tidyr

Dr. Paul Schmidt

To install and load all packages used in this chapter, run the following code:

```
for (pkg in c("tidyverse")) {
  if (!require(pkg, character.only = TRUE)) install.packages(pkg)
}

library(tidyverse)
```

Introduction

In Chapter 1 (Combining Tables) we introduced `pivot_longer()` and `pivot_wider()` for basic reshaping between wide and long formats. Those two functions are workhorses of everyday data wrangling, but they only scratch the surface of what tidyr can do. Real-world data frequently arrives in formats that demand more sophisticated transformations: column names that encode multiple variables, cells that contain composite values, list-columns from APIs or nested computations, and implicit missing values that must be made explicit before analysis.

This chapter covers the full breadth of tidyr's reshaping toolkit. We start with advanced pivoting patterns — regex-based column name parsing, the powerful `.value` sentinel, and multi-column pivots. From there we move to the `separate_*()` family for splitting columns, the `unnest_*()` and `hoist()` functions for working with nested data, and finally the missing-value utilities `complete()`, `fill()`, and `replace_na()`. By the end, you will be comfortable tackling even the messiest data layouts.

Beyond Basic Pivoting

The basic usage of `pivot_longer()` involves selecting columns, naming the new "names" column, and naming the new "values" column. This is sufficient when each column name represents a single variable. However, many real datasets encode multiple pieces of information in their column names. The tidyr package provides several arguments to handle these cases elegantly.

The WHO Tuberculosis Dataset

The `tidyr::who` dataset is a canonical example of messy real-world data. It contains tuberculosis case counts reported by the World Health Organization, where column names like `new_sp_m014` encode three variables at once: the diagnosis method (`sp` = positive pulmonary smear), the sex (`m` = male), and the age group (`014` = 0-14 years).

```
glimpse(who)
```

```
Rows: 7,240
Columns: 60
$ country      <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan…
$ iso2         <chr> "AF", "AF", "AF", "AF", "AF", "AF", "AF", "AF", "AF", "AF…
$ iso3         <chr> "AFG", "AFG", "AFG", "AFG", "AFG", "AFG", "AFG", "AFG", "…
$ year         <dbl> 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 198…
$ new_sp_m014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_m1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_m2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_m3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_m4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_m5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_m65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_f014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_f1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_f2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_f3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_f4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_f5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sp_f65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_m014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_m1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_m2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_m3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_m4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_m5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_m65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_f014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_f1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_f2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_f3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_f4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_f5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_sn_f65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_m014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_m1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_m2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_m3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_m4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_m5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_m65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_f014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_f1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_f2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
```

```
$ new_ep_f3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_f4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_f5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ new_ep_f65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_m014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_m1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_m2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_m3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_m4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_m5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_m65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_f014  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_f1524 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_f2534 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_f3544 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_f4554 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_f5564 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
$ newrel_f65   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N…
```

The first four columns (`country`, `iso2`, `iso3`, `year`) are already tidy. The remaining 56 columns all follow the pattern `new_<method>_<sex><age>` and need to be gathered into a long format with separate columns for each encoded variable.

names_pattern — Regex-Based Column Parsing

The `names_pattern` argument accepts a regular expression with capture groups (parentheses). Each group maps to one of the names specified in `names_to`. This is ideal when column names follow a structured pattern but cannot be split by a simple delimiter alone.

```
who_tidy <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_to = c("diagnosis", "sex", "age_group"),
    names_pattern = "new_?(.*)_(.)(.*)",
    values_to = "cases",
    values_drop_na = TRUE
  )

who_tidy
```

```
# A tibble: 76,046 × 8
   country     iso2  iso3   year diagnosis sex   age_group cases
   <chr>       <chr> <chr> <dbl> <chr>     <chr> <chr>     <dbl>
 1 Afghanistan AF    AFG    1997 sp        m     014           0
 2 Afghanistan AF    AFG    1997 sp        m     1524         10
 3 Afghanistan AF    AFG    1997 sp        m     2534          6
 4 Afghanistan AF    AFG    1997 sp        m     3544          3
 5 Afghanistan AF    AFG    1997 sp        m     4554          5
 6 Afghanistan AF    AFG    1997 sp        m     5564          2
 7 Afghanistan AF    AFG    1997 sp        m     65            0
 8 Afghanistan AF    AFG    1997 sp        f     014           5
 9 Afghanistan AF    AFG    1997 sp        f     1524         38
10 Afghanistan AF    AFG    1997 sp        f     2534         36
# i 76,036 more rows
```

The regex `"new_?(.*)_(.)(.*)"` works as follows: `new_?` matches the literal prefix “new” with an optional underscore (some columns use `newrel` instead of `new_rel`). The first capture group `(.*)` grabs the diagnosis method (e.g., `sp`, `sn`, `ep`, `rel`). The second

group `(.)` captures a single character for sex (`m` or `f`). The third group `(.*)` captures the age range (e.g., `014`, `1524`, `65`).

Notice how `values_drop_na = TRUE` removes rows where no cases were reported. This is particularly useful here because the WHO data is sparse — many country-year-method combinations have no observations.

```
who_tidy %>%
  count(diagnosis, sex) %>%
  pivot_wider(names_from = sex, values_from = n)
```

```
# A tibble: 4 × 3
  diagnosis     f     m
  <chr>     <int> <int>
1 ep         7143  7161
2 rel        1290  1290
3 sn         7152  7190
4 sp        22363 22457
```

names_sep — Delimiter-Based Splitting

When column names use a consistent delimiter, `names_sep` offers a simpler alternative to `names_pattern`. It splits each column name at the specified delimiter and distributes the pieces across multiple `names_to` entries.

```
# Synthetic example: columns named metric_year
sales_wide <- tibble(
  store = c("North", "South", "East"),
  revenue_2022 = c(450, 380, 510),
  revenue_2023 = c(480, 410, 530),
  cost_2022 = c(200, 180, 250),
  cost_2023 = c(210, 190, 260)
)

sales_wide
```

```
# A tibble: 3 × 5
  store revenue_2022 revenue_2023 cost_2022 cost_2023
  <chr>        <dbl>        <dbl>     <dbl>     <dbl>
1 North          450          480       200       210
2 South          380          410       180       190
3 East           510          530       250       260
```

```
sales_wide %>%
  pivot_longer(
    cols = -store,
    names_to = c("metric", "year"),
    names_sep = "_",
    values_to = "amount"
  )
```

```
# A tibble: 12 × 4
   store metric  year  amount
   <chr> <chr>   <chr>  <dbl>
 1 North revenue 2022     450
 2 North revenue 2023     480
 3 North cost    2022     200
 4 North cost    2023     210
 5 South revenue 2022     380
 6 South revenue 2023     410
 7 South cost    2022     180
```

```
 8 South cost    2023     190
 9 East  revenue 2022     510
10 East  revenue 2023     530
11 East  cost    2022     250
12 East  cost    2023     260
```

The `names_sep = "_"` argument splits each column name at the underscore, assigning the first part to `metric` and the second to `year`. This is cleaner and more readable than an equivalent regex pattern.

Exercise: Reshape the WHO Data

The `tidyr::who` dataset contains tuberculosis counts with column names encoding diagnosis method, sex, and age group. Reshape this dataset to long format using `pivot_longer()` with `names_pattern`, then answer the following questions:

a) How many unique diagnosis methods are in the data?

b) Which country reported the highest total number of cases across all years, methods, and demographics?

c) Create a summary showing total cases by sex for the year 2010. Which sex had more reported cases?

Solution

```
# Reshape first
who_long <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_to = c("diagnosis", "sex", "age_group"),
    names_pattern = "new_?(.*)_(.)(.*)",
    values_to = "cases",
    values_drop_na = TRUE
  )

# a) Unique diagnosis methods
who_long %>%
  distinct(diagnosis)
```

```
# A tibble: 4 × 1
  diagnosis
  <chr>
1 sp
2 sn
3 ep
4 rel
```

```
# b) Country with highest total cases
who_long %>%
  group_by(country) %>%
  summarise(total = sum(cases, na.rm = TRUE)) %>%
  arrange(desc(total)) %>%
  head(5)
```

```
# A tibble: 5 × 2
  country        total
  <chr>          <dbl>
1 China        8389839
2 India        7098552
3 South Africa 3010272
4 Indonesia    2909925
5 Bangladesh   1524034
```

```
# c) Cases by sex in 2010
who_long %>%
  filter(year == 2010) %>%
  group_by(sex) %>%
  summarise(total_cases = sum(cases, na.rm = TRUE))
```

```
# A tibble: 2 × 2
  sex   total_cases
  <chr>       <dbl>
1 f         1479295
2 m         2507516
```

Separating and Uniting Columns

Sometimes the issue is not the number of columns but the content within them. A single column may contain two variables glued together (e.g., “cases/population” in `table3`), or conversely, a single variable may be split across multiple columns (e.g., century and year in `table5`). The tidyr package provides modern functions for both directions.

separate_wider_delim() — Split by Delimiter

The function `separate_wider_delim()` splits a string column into multiple new columns at a specified delimiter. It is the modern replacement for the older `separate()` function, which is now superseded but still widely encountered in existing code.

```
table3
```

```
# A tibble: 6 × 3
  country      year rate
  <chr>       <dbl> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

The `rate` column in `table3` contains both `cases` and `population` separated by a `/`. We can split this into two separate numeric columns:

```
table3 %>%
  separate_wider_delim(
    cols = rate,
    delim = "/",
    names = c("cases", "population")
  ) %>%
  mutate(
    cases = as.integer(cases),
    population = as.integer(population),
    rate_per_100k = cases / population * 100000
  )
```

```
# A tibble: 6 × 5
  country      year  cases population rate_per_100k
  <chr>       <dbl>  <int>      <int>         <dbl>
1 Afghanistan  1999    745   19987071          3.73
2 Afghanistan  2000   2666   20595360         12.9
3 Brazil       1999  37737  172006362         21.9
4 Brazil       2000  80488  174504898         46.1
5 China        1999 212258 1272915272         16.7
6 China        2000 213766 1280428583         16.7
```

Note that `separate_wider_delim()` produces character columns by default, so we convert to integer explicitly. This explicit conversion is a deliberate design choice — it forces the analyst to confirm the expected data type rather than relying on automatic guessing.

separate_wider_regex() — Split by Regex

For more complex patterns, `separate_wider_regex()` uses named capture groups to extract specific parts of a string. Each capture group defines a new column.

```r
# Example: measurement strings like "12.5cm" or "3.2kg"
measurements <- tibble(
  id = 1:4,
  reading = c("12.5cm", "3.2kg", "8.0cm", "1.7kg")
)

measurements %>%
  separate_wider_regex(
    cols = reading,
    patterns = c(
      value = "[0-9.]+",
      unit = "[a-z]+"
    )
  ) %>%
  mutate(value = as.numeric(value))
```

```
# A tibble: 4 × 3
     id value unit
  <int> <dbl> <chr>
1     1  12.5 cm
2     2   3.2 kg
3     3   8   cm
4     4   1.7 kg
```

Each named element in the `patterns` vector defines a capture group. Unnamed elements serve as separators that are consumed but not stored.

separate_longer_delim() — One Row per Element

While `separate_wider_delim()` creates new columns, `separate_longer_delim()` creates new rows — one for each element after splitting. This is useful when a cell contains a delimited list.

```r
# Survey data where respondents selected multiple options
survey <- tibble(
  respondent = c("A", "B", "C"),
  hobbies = c("reading;hiking", "cooking;reading;painting", "hiking")
)

survey
```

```
# A tibble: 3 × 2
  respondent hobbies
  <chr>      <chr>
1 A          reading;hiking
2 B          cooking;reading;painting
3 C          hiking
```

```r
survey %>%
  separate_longer_delim(cols = hobbies, delim = ";")
```

```
# A tibble: 6 × 2
  respondent hobbies
  <chr>      <chr>
1 A          reading
2 A          hiking
3 B          cooking
4 B          reading
5 B          painting
6 C          hiking
```

unite() — Combining Columns

The inverse operation — combining multiple columns into one — is handled by `unite()`.

The `table5` dataset stores the century and year in separate columns:

```
table5
```

```
# A tibble: 6 × 4
  country     century year  rate
  <chr>       <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
2 Afghanistan 20      00    2666/20595360
3 Brazil      19      99    37737/172006362
4 Brazil      20      00    80488/174504898
5 China       19      99    212258/1272915272
6 China       20      00    213766/1280428583
```

We can combine them into a proper year column:

```
table5 %>%
  unite(col = "year", century, year, sep = "")
```

```
# A tibble: 6 × 3
  country     year  rate
  <chr>       <chr> <chr>
1 Afghanistan 1999  745/19987071
2 Afghanistan 2000  2666/20595360
3 Brazil      1999  37737/172006362
4 Brazil      2000  80488/174504898
5 China       1999  212258/1272915272
6 China       2000  213766/1280428583
```

By default, `unite()` joins the values with an underscore; we override this with `sep = ""` to get `1999` instead of `19_99`.

> **i Legacy Function: separate()**
>
> The older `separate()` function is still commonly found in code written before tidyr 1.3.0. It combines the roles of `separate_wider_delim()` and `separate_wider_regex()` but with a less explicit interface. While it continues to work, the newer functions are recommended for new code because they make the intended operation clearer and provide better error messages.

Exercise: Split and Combine Columns

a) Take `table3` and split the `rate` column into `cases` and `population` using `separate_wider_delim()`. Then calculate the rate as cases per 10,000 population.

b) Take `table5` and unite `century` and `year` into a single `year` column. Convert it to integer.

c) Given the following tibble, split the `id_code` column into `department`, `level`, and `sequence` using `separate_wider_regex()`:

```
records <- tibble(
  id_code = c("HR-Senior-042", "IT-Junior-118", "FIN-Mid-007"),
  name = c("Alice", "Bob", "Carol")
)
```

Solution

```r
# a) Split table3 rate column
table3 %>%
  separate_wider_delim(
    cols = rate,
    delim = "/",
    names = c("cases", "population")
  ) %>%
  mutate(
    cases = as.integer(cases),
    population = as.integer(population),
    rate_per_10k = cases / population * 10000
  )
```

```
# A tibble: 6 × 5
  country      year  cases population rate_per_10k
  <chr>       <dbl>  <int>      <int>        <dbl>
1 Afghanistan  1999    745   19987071        0.373
2 Afghanistan  2000   2666   20595360        1.29
3 Brazil       1999  37737  172006362        2.19
4 Brazil       2000  80488  174504898        4.61
5 China        1999 212258 1272915272        1.67
6 China        2000 213766 1280428583        1.67
```

```r
# b) Unite table5 columns
table5 %>%
  unite(col = "year", century, year, sep = "") %>%
  mutate(year = as.integer(year))
```

```
# A tibble: 6 × 3
  country      year rate
  <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

```r
# c) Regex-based separation
records <- tibble(
  id_code = c("HR-Senior-042", "IT-Junior-118", "FIN-Mid-007"),
  name = c("Alice", "Bob", "Carol")
)

records %>%
  separate_wider_regex(
    cols = id_code,
    patterns = c(
      department = "[A-Z]+",
      "-",
      level = "[A-Za-z]+",
      "-",
      sequence = "[0-9]+"
    )
  )
```

```
# A tibble: 3 × 4
  department level  sequence name
  <chr>      <chr>  <chr>    <chr>
1 HR         Senior 042      Alice
2 IT         Junior 118      Bob
3 FIN        Mid    007      Carol
```

Complex Pivoting Patterns

The most powerful feature of `pivot_longer()` is arguably the `.value` sentinel. It allows column names to encode both the future column name and a grouping variable — keeping related columns paired during the pivot.

The .value Sentinel

Consider a dataset where columns come in logical pairs: a measured value and a count, recorded for each year.

```
experiment <- tibble(
  site = c("A", "B", "C"),
  value_2020 = c(3.2, 4.1, 2.8),
  count_2020 = c(10L, 15L, 8L),
  value_2021 = c(3.5, 4.0, 3.1),
  count_2021 = c(12L, 14L, 9L),
  value_2022 = c(3.8, 3.9, 3.4),
  count_2022 = c(11L, 16L, 10L)
)

experiment
```

```
# A tibble: 3 × 7
  site  value_2020 count_2020 value_2021 count_2021 value_2022 count_2022
  <chr>      <dbl>      <int>      <dbl>      <int>      <dbl>      <int>
1 A            3.2         10        3.5         12        3.8         11
2 B            4.1         15        4           14        3.9         16
3 C            2.8          8        3.1          9        3.4         10
```

A naive `pivot_longer()` would mix values and counts into a single column, losing the distinction between them. Instead, we use `.value` in the `names_to` argument to tell tidyr that part of each column name should become the new column name:

```
experiment %>%
  pivot_longer(
    cols = -site,
    names_to = c(".value", "year"),
    names_sep = "_"
  )
```

```
# A tibble: 9 × 4
  site  year  value count
  <chr> <chr> <dbl> <int>
1 A     2020    3.2    10
2 A     2021    3.5    12
3 A     2022    3.8    11
4 B     2020    4.1    15
5 B     2021    4      14
6 B     2022    3.9    16
7 C     2020    2.8     8
8 C     2021    3.1     9
9 C     2022    3.4    10
```

The `.value` sentinel tells `pivot_longer()` that the first part of each column name (before `_`) defines which output column receives the data, while the second part (after `_`) goes into the `year` column. The result has separate `value` and `count` columns — exactly the pairing we need.

Multi-Column Pivoting with Pre/Post Data

This pattern is especially common in longitudinal or pre/post study designs. Suppose we have subjects measured on two outcomes at two time points:

```
study <- tibble(
  subject = 1:4,
  score_pre = c(72, 85, 68, 91),
  score_post = c(78, 88, 75, 93),
  time_pre = c(45, 38, 52, 33),
  time_post = c(40, 35, 48, 31)
)

study
```

```
# A tibble: 4 × 5
  subject score_pre score_post time_pre time_post
    <int>     <dbl>      <dbl>    <dbl>     <dbl>
1       1        72         78       45        40
2       2        85         88       38        35
3       3        68         75       52        48
4       4        91         93       33        31
```

Using `.value` with `names_sep`, we pivot so that each subject has two rows (pre and post) while keeping `score` and `time` as separate columns:

```
study %>%
  pivot_longer(
    cols = -subject,
    names_to = c(".value", "period"),
    names_sep = "_"
  )
```

```
# A tibble: 8 × 4
  subject period score  time
    <int> <chr>  <dbl> <dbl>
1       1 pre       72    45
2       1 post      78    40
3       2 pre       85    38
4       2 post      88    35
5       3 pre       68    52
6       3 post      75    48
7       4 pre       91    33
8       4 post      93    31
```

Combining .value with names_pattern

When column names have more complex structures, `.value` can be combined with `names_pattern` for full control:

```
# Blood pressure data: columns like bp_sys_visit1, bp_dia_visit1, ...
bp_data <- tibble(
  patient = c("P01", "P02", "P03"),
  bp_sys_visit1 = c(120, 135, 118),
  bp_dia_visit1 = c(80, 88, 76),
  bp_sys_visit2 = c(118, 130, 122),
  bp_dia_visit2 = c(78, 85, 80)
)

bp_data
```

```
# A tibble: 3 × 5
  patient bp_sys_visit1 bp_dia_visit1 bp_sys_visit2 bp_dia_visit2
  <chr>           <dbl>         <dbl>         <dbl>         <dbl>
1 P01               120            80           118            78
2 P02               135            88           130            85
3 P03               118            76           122            80
```

```
bp_data %>%
  pivot_longer(
    cols = -patient,
    names_to = c(".value", "visit"),
    names_pattern = "bp_(.+)_(visit\\d)"
  )
```

```
# A tibble: 6 × 4
  patient visit    sys   dia
  <chr>   <chr>  <dbl> <dbl>
1 P01     visit1   120    80
2 P01     visit2   118    78
3 P02     visit1   135    88
4 P02     visit2   130    85
5 P03     visit1   118    76
6 P03     visit2   122    80
```

The regex `"bp_(.+)_(visit\\d)"` has two capture groups. The first group (mapped to `.value`) captures `sys` or `dia`, which become column names. The second group captures `visit1` or `visit2`, which populates the `visit` column.

Exercise: Multi-Column Pivot

Given the following clinical trial data, pivot to long format so that each row represents one assessment, with `score` and `duration` remaining as separate columns. The result should have columns: `patient`, `assessment`, `score`, and `duration`.

```
trial <- tibble(
  patient = c("P1", "P2", "P3"),
  score_baseline = c(45, 52, 38),
  score_week4 = c(40, 48, 35),
  score_week8 = c(35, 42, 30),
  duration_baseline = c(120, 95, 140),
  duration_week4 = c(110, 90, 130),
  duration_week8 = c(100, 85, 125)
)
```

Hint: Use `.value` in `names_to` with an appropriate `names_sep`.

Solution

```
trial <- tibble(
  patient = c("P1", "P2", "P3"),
  score_baseline = c(45, 52, 38),
  score_week4 = c(40, 48, 35),
  score_week8 = c(35, 42, 30),
  duration_baseline = c(120, 95, 140),
  duration_week4 = c(110, 90, 130),
  duration_week8 = c(100, 85, 125)
)

trial %>%
  pivot_longer(
    cols = -patient,
    names_to = c(".value", "assessment"),
    names_sep = "_"
  )
```

```
# A tibble: 9 × 4
  patient assessment score duration
  <chr>   <chr>      <dbl>    <dbl>
1 P1      baseline      45      120
2 P1      week4         40      110
3 P1      week8         35      100
4 P2      baseline      52       95
5 P2      week4         48       90
6 P2      week8         42       85
7 P3      baseline      38      140
8 P3      week4         35      130
9 P3      week8         30      125
```

Handling Nested Data

Modern data pipelines frequently produce list-columns — columns where each cell contains not a single value but a list, a data frame, or another complex object. This happens naturally when reading JSON data, when using `tidyr::nest()`, or when working with APIs. The tidyr package provides three functions for turning these nested structures into rectangular (tidy) data.

unnest_longer() — List-Column to Rows

When each element of a list-column contains a vector of values, `unnest_longer()` creates one row per element, duplicating the other columns as needed.

```
# Each patient has multiple measurements
patients <- tibble(
  patient_id = c("P01", "P02", "P03"),
  systolic = list(
    c(120, 125, 118),
    c(135, 132),
    c(140, 138, 136, 133)
  )
)

patients
```

```
# A tibble: 3 × 2
  patient_id systolic
  <chr>      <list>
1 P01        <dbl [3]>
2 P02        <dbl [2]>
3 P03        <dbl [4]>
```

```
patients %>%
  unnest_longer(systolic)
```

```
# A tibble: 9 × 2
  patient_id systolic
  <chr>         <dbl>
1 P01             120
2 P01             125
3 P01             118
4 P02             135
5 P02             132
6 P03             140
7 P03             138
8 P03             136
9 P03             133
```

This is conceptually similar to `separate_longer_delim()` but works on list-columns rather than delimited strings.

unnest_wider() — List-Column to Columns

When each element of a list-column is a named list (like a JSON object), `unnest_wider()` creates one column per named element.

```
# Metadata stored as named lists
samples <- tibble(
  sample_id = c("S1", "S2", "S3"),
  metadata = list(
```

```
    list(method = "PCR", concentration = 2.5, quality = "high"),
    list(method = "ELISA", concentration = 1.8, quality = "medium"),
    list(method = "PCR", concentration = 3.1, quality = "high")
  )
)

samples
```

```
# A tibble: 3 × 2
  sample_id metadata
  <chr>     <list>
1 S1        <named list [3]>
2 S2        <named list [3]>
3 S3        <named list [3]>
```

```
samples %>%
  unnest_wider(metadata)
```

```
# A tibble: 3 × 4
  sample_id method concentration quality
  <chr>     <chr>          <dbl> <chr>
1 S1        PCR              2.5 high
2 S2        ELISA            1.8 medium
3 S3        PCR              3.1 high
```

Each named element in the list becomes its own column. This is a very common pattern when working with JSON data parsed into R.

hoist() — Selective Extraction

When list-columns contain deeply nested structures and you only need specific fields, `hoist()` provides a targeted extraction. It reaches into each list element and pulls out only the fields you specify, leaving the rest in the original list-column.

```
# Complex nested data — imagine this came from a JSON API
experiments <- tibble(
  exp_id = c("E1", "E2"),
  results = list(
    list(
      temperature = 37.2,
      duration_h = 24,
      reagents = list(name = "Buffer A", lot = "L001"),
      notes = "Standard run"
    ),
    list(
      temperature = 37.5,
      duration_h = 48,
      reagents = list(name = "Buffer B", lot = "L002"),
      notes = "Extended incubation"
    )
  )
)

experiments %>%
  hoist(
    results,
    temperature = "temperature",
    reagent_name = list("reagents", "name")
  )
```

```
# A tibble: 2 × 4
  exp_id temperature reagent_name results
  <chr>        <dbl> <chr>        <list>
```

```
1 E1            37.2 Buffer A     <named list [3]>
2 E2            37.5 Buffer B     <named list [3]>
```

The path `list("reagents", "name")` reaches into the nested `reagents` sublist to extract the `name` field. The remaining unextracted elements stay in the `results` column. This selective approach is more efficient and readable than a full `unnest_wider()` followed by column selection when you only need a few fields from a complex structure.

Exercise: Rectangle a List-Column

Given the following tibble with nested lab results, perform the following tasks:

```
lab_data <- tibble(
  lab = c("Lab A", "Lab B", "Lab C"),
  results = list(
    list(ph = 7.2, conductivity = 450, method = "automated"),
    list(ph = 6.8, conductivity = 380, method = "manual"),
    list(ph = 7.5, conductivity = 510, method = "automated")
  ),
  replicates = list(c(7.1, 7.3, 7.2), c(6.9, 6.7), c(7.4, 7.5, 7.6, 7.5))
)
```

a) Use `hoist()` to extract only `ph` and `method` from the `results` column.

b) Use `unnest_wider()` on the `results` column instead. How does the output differ from `hoist()`?

c) Use `unnest_longer()` on the `replicates` column to get one row per replicate measurement.

Solution

```
lab_data <- tibble(
  lab = c("Lab A", "Lab B", "Lab C"),
  results = list(
    list(ph = 7.2, conductivity = 450, method = "automated"),
    list(ph = 6.8, conductivity = 380, method = "manual"),
    list(ph = 7.5, conductivity = 510, method = "automated")
  ),
  replicates = list(c(7.1, 7.3, 7.2), c(6.9, 6.7), c(7.4, 7.5, 7.6, 7.5))
)

# a) hoist: extract only ph and method
lab_data %>%
  hoist(results, ph = "ph", method = "method")
```

```
# A tibble: 3 × 5
  lab      ph method    results          replicates
  <chr> <dbl> <chr>     <list>           <list>
1 Lab A   7.2 automated <named list [1]> <dbl [3]>
2 Lab B   6.8 manual    <named list [1]> <dbl [2]>
3 Lab C   7.5 automated <named list [1]> <dbl [4]>
```

```
# Note: results column remains, containing the leftover 'conductivity'

# b) unnest_wider: extract all fields
lab_data %>%
  unnest_wider(results)
```

```
# A tibble: 3 × 5
  lab      ph conductivity method    replicates
  <chr> <dbl>        <dbl> <chr>     <list>
1 Lab A   7.2          450 automated <dbl [3]>
2 Lab B   6.8          380 manual    <dbl [2]>
3 Lab C   7.5          510 automated <dbl [4]>
```

```
# Note: results column is replaced by ph, conductivity, and method

# c) unnest_longer: one row per replicate
lab_data %>%
  unnest_longer(replicates)
```

```
# A tibble: 9 × 3
  lab   results          replicates
  <chr> <list>                <dbl>
1 Lab A <named list [3]>        7.1
2 Lab A <named list [3]>        7.3
3 Lab A <named list [3]>        7.2
4 Lab B <named list [3]>        6.9
5 Lab B <named list [3]>        6.7
6 Lab C <named list [3]>        7.4
7 Lab C <named list [3]>        7.5
8 Lab C <named list [3]>        7.6
9 Lab C <named list [3]>        7.5
```

Missing Values in Reshaping

Reshaping operations frequently create or reveal missing values. A dataset may appear complete in wide format but develop gaps when pivoted to long format, or vice versa. The tidyr package provides three complementary functions for managing missing values in the context of data reshaping.

complete() — Make Implicit Missing Values Explicit

Datasets often have “implicit” missing values — combinations of variables that simply do not appear in the data rather than being recorded as `NA`. The `complete()` function generates all combinations of the specified variables and fills in `NA` where data is absent.

```
# Experiment with some missing observations
growth <- tibble(
  treatment = c("Control", "Control", "Low", "Low", "High"),
  time_point = c(1, 2, 1, 2, 2),
  biomass = c(2.3, 2.8, 3.1, 3.9, 5.2)
)

growth
```

```
# A tibble: 5 × 3
  treatment time_point biomass
  <chr>          <dbl>   <dbl>
1 Control            1     2.3
2 Control            2     2.8
3 Low                1     3.1
4 Low                2     3.9
5 High               2     5.2
```

```
# High at time 1 is implicitly missing
growth %>%
  complete(treatment, time_point)
```

```
# A tibble: 6 × 3
  treatment time_point biomass
  <chr>          <dbl>   <dbl>
1 Control            1     2.3
2 Control            2     2.8
3 High               1    NA
4 High               2     5.2
5 Low                1     3.1
6 Low                2     3.9
```

Now the missing `High` treatment at time point 1 is explicitly shown as `NA`. This is important for analyses that require balanced designs or for visualizations where missing points should be apparent.

You can also supply fill values to replace the generated `NA`s:

```
growth %>%
  complete(treatment, time_point, fill = list(biomass = 0))
```

```
# A tibble: 6 × 3
  treatment time_point biomass
  <chr>          <dbl>   <dbl>
1 Control            1     2.3
2 Control            2     2.8
3 High               1     0
```

```
4 High               2     5.2
5 Low                1     3.1
6 Low                2     3.9
```

fill() — Fill Down or Up

The `fill()` function propagates the last non-missing value forward (down) or backward (up). This is particularly useful for data where group headers appear only once and subsequent rows inherit the same value.

```
# Data where group labels appear only in the first row of each group
report <- tibble(
  department = c("Sales", NA, NA, "Engineering", NA),
  employee = c("Alice", "Bob", "Carol", "Dave", "Eve"),
  salary = c(55000, 52000, 58000, 72000, 68000)
)

report
```

```
# A tibble: 5 × 3
  department  employee salary
  <chr>       <chr>     <dbl>
1 Sales       Alice     55000
2 <NA>        Bob       52000
3 <NA>        Carol     58000
4 Engineering Dave      72000
5 <NA>        Eve       68000
```

```
report %>%
  fill(department, .direction = "down")
```

```
# A tibble: 5 × 3
  department  employee salary
  <chr>       <chr>     <dbl>
1 Sales       Alice     55000
2 Sales       Bob       52000
3 Sales       Carol     58000
4 Engineering Dave      72000
5 Engineering Eve       68000
```

The `.direction` argument controls the fill direction: `"down"` (default), `"up"`, `"downup"` (down first, then up), or `"updown"` (up first, then down).

> ! Important
>
> Forward-filling numeric measurements (last observation carried forward, LOCF) is a form of imputation that can introduce bias. It is appropriate for structural patterns like repeated group labels, but should be used with caution for actual measured values. For serious statistical imputation, consider dedicated packages like {mice} or {missForest}.

replace_na() — Targeted NA Replacement

The `replace_na()` function replaces `NA` values with specified replacements, either across the whole data frame or within a `mutate()` call for individual columns.

```
# Using airquality, which has natural NAs
aq <- as_tibble(airquality)
```

```r
# Count NAs per column
aq %>%
  summarise(across(everything(), \(x) sum(is.na(x))))
```

```
# A tibble: 1 × 6
  Ozone Solar.R  Wind  Temp Month   Day
  <int>   <int> <int> <int> <int> <int>
1    37       7     0     0     0     0
```

```r
# Replace NAs in Ozone and Solar.R with column medians
aq %>%
  mutate(
    Ozone = replace_na(Ozone, as.integer(median(Ozone, na.rm = TRUE))),
    Solar.R = replace_na(Solar.R, as.integer(median(Solar.R, na.rm = TRUE)))
  ) %>%
  summarise(across(everything(), \(x) sum(is.na(x))))
```

```
# A tibble: 1 × 6
  Ozone Solar.R  Wind  Temp Month   Day
  <int>   <int> <int> <int> <int> <int>
1     0       0     0     0     0     0
```

ℹ Choosing the Right Missing-Value Strategy

These three functions serve different purposes. Use `complete()` when you need all combinations of variables to exist explicitly — this is common before joining or plotting. Use `fill()` when missing values follow a structural pattern (e.g., repeated group labels). Use `replace_na()` when you have a specific imputation value in mind. For serious statistical imputation (multiple imputation, predictive mean matching), dedicated packages like {mice} or {missForest} are more appropriate.

Putting It All Together

To close this chapter, we walk through an end-to-end pipeline that combines several of the techniques covered above. The goal is to take a deliberately messy dataset and transform it into a clean, analysis-ready format step by step.

The Messy Dataset

Imagine a multi-site field experiment where measurements of two response variables (yield and moisture) were taken across three years. The data arrives in a wide format with composite column names:

```
messy <- tibble(
  site = c("Alpha", "Alpha", "Beta", "Beta", "Gamma"),
  treatment = c("A", "B", "A", "B", "A"),
  yield_2021 = c(4.2, 5.1, NA, 4.8, 3.9),
  yield_2022 = c(4.5, 5.3, 4.0, 5.0, 4.1),
  yield_2023 = c(4.8, NA, 4.3, 5.2, 4.4),
  moisture_2021 = c(12.1, 11.5, NA, 11.8, 13.2),
  moisture_2022 = c(11.8, 11.2, 12.5, 11.6, 12.9),
  moisture_2023 = c(11.5, NA, 12.0, 11.3, 12.6)
)

messy
```

```
# A tibble: 5 × 8
  site  treatment yield_2021 yield_2022 yield_2023 moisture_2021 moisture_2022
  <chr> <chr>          <dbl>      <dbl>      <dbl>         <dbl>         <dbl>
1 Alpha A                4.2        4.5        4.8          12.1          11.8
2 Alpha B                5.1        5.3       NA            11.5          11.2
3 Beta  A               NA          4          4.3          NA            12.5
4 Beta  B                4.8        5          5.2          11.8          11.6
5 Gamma A                3.9        4.1        4.4          13.2          12.9
# i 1 more variable: moisture_2023 <dbl>
```

This dataset has three issues: (1) yield and moisture for each year are in separate columns, (2) some site-treatment-year combinations are missing, and (3) the Gamma site only has treatment A.

Step 1: Pivot with .value

First, we reshape so that each row represents one site-treatment-year combination, while keeping `yield` and `moisture` as separate columns:

```
step1 <- messy %>%
  pivot_longer(
    cols = -c(site, treatment),
    names_to = c(".value", "year"),
    names_sep = "_"
  ) %>%
  mutate(year = as.integer(year))

step1
```

```
# A tibble: 15 × 5
   site  treatment  year yield moisture
   <chr> <chr>     <int> <dbl>    <dbl>
 1 Alpha A          2021   4.2     12.1
 2 Alpha A          2022   4.5     11.8
 3 Alpha A          2023   4.8     11.5
 4 Alpha B          2021   5.1     11.5
```

```
 5 Alpha B          2022   5.3     11.2
 6 Alpha B          2023  NA       NA
 7 Beta  A          2021  NA       NA
 8 Beta  A          2022   4       12.5
 9 Beta  A          2023   4.3     12
10 Beta  B          2021   4.8     11.8
11 Beta  B          2022   5       11.6
12 Beta  B          2023   5.2     11.3
13 Gamma A          2021   3.9     13.2
14 Gamma A          2022   4.1     12.9
15 Gamma A          2023   4.4     12.6
```

Step 2: Complete Missing Combinations

Next, we make all implicit missing combinations explicit. The Gamma site is missing treatment B entirely:

```
step2 <- step1 %>%
  complete(site, treatment, year)

step2
```

```
# A tibble: 18 × 5
   site  treatment  year yield moisture
   <chr> <chr>     <int> <dbl>    <dbl>
 1 Alpha A          2021   4.2     12.1
 2 Alpha A          2022   4.5     11.8
 3 Alpha A          2023   4.8     11.5
 4 Alpha B          2021   5.1     11.5
 5 Alpha B          2022   5.3     11.2
 6 Alpha B          2023  NA       NA
 7 Beta  A          2021  NA       NA
 8 Beta  A          2022   4       12.5
 9 Beta  A          2023   4.3     12
10 Beta  B          2021   4.8     11.8
11 Beta  B          2022   5       11.6
12 Beta  B          2023   5.2     11.3
13 Gamma A          2021   3.9     13.2
14 Gamma A          2022   4.1     12.9
15 Gamma A          2023   4.4     12.6
16 Gamma B          2021  NA       NA
17 Gamma B          2022  NA       NA
18 Gamma B          2023  NA       NA
```

Step 3: Fill Structural Patterns

In this particular example, we might decide that the missing Gamma-B combination should remain as `NA` (it was never planted). However, suppose the `site` column had been entered only for the first row of each group — then `fill()` would restore the values. For demonstration, we apply `fill()` on the remaining `NA`s in the response columns by filling within each site-treatment group using the previous year's value:

```
step3 <- step2 %>%
  group_by(site, treatment) %>%
  fill(yield, moisture, .direction = "down") %>%
  ungroup()

step3
```

```
# A tibble: 18 × 5
   site  treatment  year yield moisture
```

```
   <chr> <chr>     <int> <dbl>    <dbl>
 1 Alpha A          2021   4.2     12.1
 2 Alpha A          2022   4.5     11.8
 3 Alpha A          2023   4.8     11.5
 4 Alpha B          2021   5.1     11.5
 5 Alpha B          2022   5.3     11.2
 6 Alpha B          2023   5.3     11.2
 7 Beta  A          2021  NA       NA
 8 Beta  A          2022   4       12.5
 9 Beta  A          2023   4.3     12
10 Beta  B          2021   4.8     11.8
11 Beta  B          2022   5       11.6
12 Beta  B          2023   5.2     11.3
13 Gamma A          2021   3.9     13.2
14 Gamma A          2022   4.1     12.9
15 Gamma A          2023   4.4     12.6
16 Gamma B          2021  NA       NA
17 Gamma B          2022  NA       NA
18 Gamma B          2023  NA       NA
```

Step 4: Final Summary

With the cleaned data, we can now compute summaries. For instance, the mean yield and moisture per treatment across all sites and years:

```
step3 %>%
  group_by(treatment) %>%
  summarise(
    across(c(yield, moisture), \(x) mean(x, na.rm = TRUE)),
    n_obs = sum(!is.na(yield))
  )
```

```
# A tibble: 2 × 4
  treatment yield moisture n_obs
  <chr>     <dbl>    <dbl> <int>
1 A          4.28     12.3     1
2 B          5.12     11.4     1
```

The full pipeline from raw data to summary can be written as a single chain:

```
messy %>%
  pivot_longer(
    cols = -c(site, treatment),
    names_to = c(".value", "year"),
    names_sep = "_"
  ) %>%
  mutate(year = as.integer(year)) %>%
  complete(site, treatment, year) %>%
  group_by(site, treatment) %>%
  fill(yield, moisture, .direction = "down") %>%
  ungroup() %>%
  group_by(treatment) %>%
  summarise(
    across(c(yield, moisture), \(x) mean(x, na.rm = TRUE)),
    n_obs = sum(!is.na(yield))
  )
```

```
# A tibble: 2 × 4
  treatment yield moisture n_obs
  <chr>     <dbl>    <dbl> <int>
1 A          4.28     12.3     1
2 B          5.12     11.4     1
```

This pipeline demonstrates how tidyr functions compose naturally with dplyr: pivot to restructure, complete to fill gaps, fill to propagate values, and then summarise for analysis. Each step produces a valid tibble that can be inspected independently, making the transformation both transparent and debuggable.

Summary

This chapter explored tidyr's full reshaping toolkit beyond basic pivoting.

> **i** Key Takeaways
>
> 1. **Advanced Pivoting:** `names_sep` splits column names at a delimiter, `names_pattern` uses regex capture groups for complex patterns, and the `.value` sentinel keeps paired columns together during pivoting.
> 2. **Separating and Uniting:** `separate_wider_delim()` and `separate_wider_regex()` split columns into multiple new columns, `separate_longer_delim()` creates new rows, and `unite()` combines columns.
> 3. **Nested Data:** `unnest_longer()` expands list-columns into rows, `unnest_wider()` expands them into columns, and `hoist()` selectively extracts specific fields from deeply nested structures.
> 4. **Missing Values:** `complete()` makes implicit missing values explicit, `fill()` propagates values down or up (use with caution for measured data), and `replace_na()` substitutes specific values.
> 5. **Best Practice:** Build reshaping pipelines step by step, inspecting each intermediate result. This makes complex transformations transparent and debuggable.

Bibliography