# 12. tidyr-Vertiefung

Fortgeschrittenes Reshaping, Trennen und Rectangling mit tidyr

Dr. Paul Schmidt

Um alle in diesem Kapitel verwendeten Pakete zu installieren und zu laden, kann man folgenden Code ausführen:

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

library(tidyverse)
```

## Einleitung

In Kapitel 1 (Tabellen kombinieren) haben wir `pivot_longer()` und `pivot_wider()` für grundlegendes Reshaping zwischen Wide- und Long-Format vorgestellt. Diese beiden Funktionen sind Arbeitstiere des alltäglichen Data Wrangling, aber sie kratzen nur an der Oberfläche dessen, was tidyr leisten kann. Reale Daten kommen häufig in Formaten an, die anspruchsvollere Transformationen erfordern: Spaltennamen, die mehrere Variablen kodieren, Zellen mit zusammengesetzten Werten, List-Columns aus APIs oder verschachtelten Berechnungen, und implizite fehlende Werte, die vor der Analyse explizit gemacht werden müssen.

Dieses Kapitel behandelt die gesamte Bandbreite von tidyrs Reshaping-Toolkit. Wir beginnen mit fortgeschrittenen Pivoting-Mustern — Regex-basiertes Parsen von Spaltennamen, der mächtige `.value`-Sentinel und Multi-Column-Pivots. Von dort gehen wir zur `separate_*()`-Familie zum Aufteilen von Spalten, den `unnest_*()`- und `hoist()`-Funktionen für verschachtelte Daten, und schließlich zu den Missing-Value-Werkzeugen `complete()`, `fill()` und `replace_na()`. Am Ende des Kapitels ist man in der Lage, auch die unordentlichsten Datenstrukturen souverän zu bearbeiten.

# Über einfaches Pivoting hinaus

Die grundlegende Verwendung von `pivot_longer()` umfasst das Auswählen von Spalten, das Benennen der neuen “Names”-Spalte und das Benennen der neuen “Values”-Spalte. Das reicht aus, wenn jeder Spaltenname eine einzelne Variable repräsentiert. Viele reale Datensätze kodieren jedoch mehrere Informationen in ihren Spaltennamen. Das tidyr-Paket bietet dafür mehrere Argumente, um diese Fälle elegant zu behandeln.

## Der WHO-Tuberkulose-Datensatz

Der `tidyr::who`-Datensatz ist ein kanonisches Beispiel für unordentliche reale Daten. Er enthält von der Weltgesundheitsorganisation gemeldete Tuberkulose-Fallzahlen, wobei Spaltennamen wie `new_sp_m014` drei Variablen gleichzeitig kodieren: die Diagnosemethode (`sp` = positiver pulmonaler Abstrich), das Geschlecht (`m` = männlich) und die Altersgruppe (`014` = 0–14 Jahre).

```
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…
```

Die ersten vier Spalten (`country`, `iso2`, `iso3`, `year`) sind bereits tidy. Die verbleibenden 56 Spalten folgen alle dem Muster `new_<method>_<sex><age>` und müssen in ein Long-Format mit separaten Spalten für jede kodierte Variable überführt werden.

## names_pattern — Regex-basiertes Spalten-Parsen

Das Argument `names_pattern` akzeptiert einen regulären Ausdruck mit Capture Groups (Klammern). Jede Gruppe wird einer der in `names_to` angegebenen Bezeichnungen zugeordnet. Das ist ideal, wenn Spaltennamen einem strukturierten Muster folgen, sich aber nicht allein durch einen einfachen Delimiter aufteilen lassen.

```r
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
```

Der Regex `"new_?(.*)_(.)(.*)"` funktioniert folgendermaßen: `new_?` matcht das Literal-Präfix „new” mit einem optionalen Unterstrich (manche Spalten verwenden `newrel` statt

`new_rel`). Die erste Capture Group `(.*)` erfasst die Diagnosemethode (z.B. `sp`, `sn`, `ep`, `rel`). Die zweite Gruppe `(.)` erfasst ein einzelnes Zeichen für das Geschlecht (`m` oder `f`). Die dritte Gruppe `(.*)` erfasst den Altersbereich (z.B. `014`, `1524`, `65`).

Man beachte, dass `values_drop_na = TRUE` Zeilen entfernt, in denen keine Fälle gemeldet wurden. Das ist hier besonders nützlich, da die WHO-Daten dünn besetzt sind — viele Land-Jahr-Methoden-Kombinationen haben keine Beobachtungen.

```
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-basiertes Aufteilen

Wenn Spaltennamen einen konsistenten Delimiter verwenden, bietet `names_sep` eine einfachere Alternative zu `names_pattern`. Es teilt jeden Spaltennamen am angegebenen Delimiter auf und verteilt die Teile auf die mehreren `names_to`-Einträge.

```
# 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
```

Das Argument `names_sep = "_"` teilt jeden Spaltennamen am Unterstrich auf und weist den ersten Teil `metric` und den zweiten `year` zu. Das ist übersichtlicher und lesbarer als ein äquivalentes Regex-Muster.

**Übung: WHO-Daten umstrukturieren**

Der `tidyr::who`-Datensatz enthält Tuberkulose-Fallzahlen mit Spaltennamen, die Diagnosemethode, Geschlecht und Altersgruppe kodieren. Man strukturiere diesen Datensatz mit `pivot_longer()` und `names_pattern` ins Long-Format um und beantworte dann folgende Fragen:

a) Wie viele eindeutige Diagnosemethoden gibt es in den Daten?

b) Welches Land hat über alle Jahre, Methoden und Demographien hinweg die höchste Gesamtzahl an Fällen gemeldet?

c) Man erstelle eine Zusammenfassung der Gesamtfälle nach Geschlecht für das Jahr 2010. Welches Geschlecht hatte mehr gemeldete Fälle?

## Lösung

```
# 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
```

# Spalten trennen und vereinen

Manchmal liegt das Problem nicht in der Anzahl der Spalten, sondern im Inhalt innerhalb der Spalten. Eine einzelne Spalte kann zwei Variablen enthalten, die zusammengeklebt sind (z.B. „cases/population" in `table3`), oder umgekehrt kann eine einzelne Variable über mehrere Spalten verteilt sein (z.B. Jahrhundert und Jahr in `table5`). Das tidyr-Paket bietet moderne Funktionen für beide Richtungen.

## separate_wider_delim() — Trennen per Delimiter

Die Funktion `separate_wider_delim()` teilt eine String-Spalte an einem angegebenen Delimiter in mehrere neue Spalten auf. Sie ist der moderne Ersatz für die ältere Funktion `separate()`, die inzwischen abgelöst ist, aber in bestehendem Code noch häufig anzutreffen ist.

```
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
```

Die Spalte `rate` in `table3` enthält sowohl `cases` als auch `population`, getrennt durch einen `/`. Man kann dies in zwei separate numerische Spalten aufteilen:

```
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
```

Man beachte, dass `separate_wider_delim()` standardmäßig Character-Spalten erzeugt, weshalb man explizit in Integer konvertiert. Diese explizite Konvertierung ist eine bewusste Design-Entscheidung — sie zwingt dazu, den erwarteten Datentyp zu bestätigen, anstatt sich auf automatisches Raten zu verlassen.

## separate_wider_regex() — Trennen per Regex

Für komplexere Muster verwendet `separate_wider_regex()` benannte Capture Groups, um bestimmte Teile eines Strings zu extrahieren. Jede Capture Group definiert eine neue Spalte.

```
# 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
```

Jedes benannte Element im `patterns`-Vektor definiert eine Capture Group. Unbenannte Elemente dienen als Trennzeichen, die konsumiert, aber nicht gespeichert werden.

## separate_longer_delim() — Eine Zeile pro Element

Während `separate_wider_delim()` neue Spalten erzeugt, erzeugt `separate_longer_delim()` neue Zeilen — eine für jedes Element nach dem Aufteilen. Das ist nützlich, wenn eine Zelle eine durch Trennzeichen getrennte Liste enthält.

```
# 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
```

```
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() — Spalten zusammenführen

Die umgekehrte Operation — das Zusammenführen mehrerer Spalten zu einer — wird von `unite()` übernommen. Der Datensatz `table5` speichert Jahrhundert und Jahr in getrennten Spalten:

```
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
```

Man kann sie zu einer ordentlichen Jahresspalte zusammenführen:

```
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
```

Standardmäßig verbindet `unite()` die Werte mit einem Unterstrich; mit `sep = ""` überschreibt man dies, um `1999` statt `19_99` zu erhalten.

> **i Legacy-Funktion: separate()**
>
> Die ältere Funktion `separate()` ist in Code, der vor tidyr 1.3.0 geschrieben wurde, noch häufig anzutreffen. Sie vereint die Rollen von `separate_wider_delim()` und `separate_wider_regex()`, allerdings mit einer weniger expliziten Schnittstelle. Sie funktioniert weiterhin, aber für neuen Code werden die neueren Funktionen empfohlen, da sie die beabsichtigte Operation klarer machen und bessere Fehlermeldungen liefern.

Übung: Spalten trennen und zusammenführen

a) Man nehme `table3` und teile die Spalte `rate` mit `separate_wider_delim()` in `cases` und `population` auf. Anschließend berechne man die Rate als Fälle pro 10.000 Einwohner.

b) Man nehme `table5` und vereinige `century` und `year` zu einer einzelnen Spalte `year`. Diese konvertiere man in Integer.

c) Man teile im folgenden Tibble die Spalte `id_code` mit `separate_wider_regex()` in `department`, `level` und `sequence` auf:

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

## Lösung

```
# 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
```

```
# 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
```

```
# 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
```

# Komplexe Pivoting-Muster

Das wohl mächtigste Feature von `pivot_longer()` ist der `.value`-Sentinel. Er ermöglicht es, dass Spaltennamen sowohl den zukünftigen Spaltennamen als auch eine Gruppierungsvariable kodieren — zusammengehörige Spalten bleiben beim Pivoting als Paar erhalten.

## Der .value-Sentinel

Man betrachte einen Datensatz, in dem Spalten in logischen Paaren auftreten: ein Messwert und eine Anzahl, jeweils für jedes Jahr erfasst.

```
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
```

Ein naives `pivot_longer()` würde Werte und Anzahlen in eine einzige Spalte mischen und die Unterscheidung zwischen ihnen verlieren. Stattdessen verwendet man `.value` im `names_to`-Argument, um tidyr mitzuteilen, dass ein Teil des Spaltennamens zum neuen Spaltennamen werden soll:

```
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
```

Der `.value`-Sentinel teilt `pivot_longer()` mit, dass der erste Teil jedes Spaltennamens (vor `_`) bestimmt, welche Ausgabespalte die Daten erhält, während der zweite Teil (nach `_`) in

die Spalte `year` fließt. Das Ergebnis hat separate Spalten `value` und `count` — genau die Paarung, die man braucht.

## Multi-Column-Pivoting mit Pre/Post-Daten

Dieses Muster ist besonders häufig bei longitudinalen oder Pre/Post-Studiendesigns. Angenommen, man hat Probanden, die bei zwei Zielgrößen zu zwei Zeitpunkten gemessen wurden:

```
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
```

Mit `.value` und `names_sep` pivotiert man so, dass jeder Proband zwei Zeilen hat (pre und post), während `score` und `time` als separate Spalten erhalten bleiben:

```
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
```

## .value mit names_pattern kombinieren

Wenn Spaltennamen komplexere Strukturen haben, kann `.value` mit `names_pattern` für volle Kontrolle kombiniert werden:

```
# 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
```

Der Regex `"bp_(.+)_(visit\\d)"` hat zwei Capture Groups. Die erste Gruppe (auf `.value` gemappt) erfasst `sys` oder `dia`, die zu Spaltennamen werden. Die zweite Gruppe erfasst `visit1` oder `visit2`, was die Spalte `visit` befüllt.

### Übung: Multi-Column-Pivot

Man pivotiere die folgenden klinischen Studiendaten ins Long-Format, sodass jede Zeile eine Bewertung darstellt, wobei `score` und `duration` als separate Spalten erhalten bleiben. Das Ergebnis sollte die Spalten `patient`, `assessment`, `score` und `duration` haben.

```
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)
)
```

Hinweis: Man verwende `.value` in `names_to` mit einem passenden `names_sep`.

**Lösung**

```
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
```

# Verschachtelte Daten verarbeiten

Moderne Daten-Pipelines erzeugen häufig List-Columns — Spalten, in denen jede Zelle nicht einen einzelnen Wert enthält, sondern eine Liste, einen Data Frame oder ein anderes komplexes Objekt. Dies passiert natürlicherweise beim Einlesen von JSON-Daten, bei der Verwendung von `tidyr::nest()` oder bei der Arbeit mit APIs. Das tidyr-Paket bietet drei Funktionen, um diese verschachtelten Strukturen in rechteckige (tidy) Daten zu überführen.

## unnest_longer() — List-Column zu Zeilen

Wenn jedes Element einer List-Column einen Vektor von Werten enthält, erzeugt `unnest_longer()` eine Zeile pro Element und dupliziert die anderen Spalten entsprechend.

```
# 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
```

Das ist konzeptionell ähnlich wie `separate_longer_delim()`, funktioniert aber mit List-Columns statt mit durch Trennzeichen getrennten Strings.

## unnest_wider() — List-Column zu Spalten

Wenn jedes Element einer List-Column eine benannte Liste ist (wie ein JSON-Objekt), erzeugt `unnest_wider()` eine Spalte pro benanntem 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
```

Jedes benannte Element in der Liste wird zu einer eigenen Spalte. Das ist ein sehr häufiges Muster bei der Arbeit mit in R geparsten JSON-Daten.

## hoist() — Selektive Extraktion

Wenn List-Columns tief verschachtelte Strukturen enthalten und man nur bestimmte Felder benötigt, bietet `hoist()` eine gezielte Extraktion. Es greift in jedes Listenelement hinein und zieht nur die angegebenen Felder heraus, während der Rest in der ursprünglichen List-Column verbleibt.

```
# 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]>
```

Der Pfad `list("reagents", "name")` greift in die verschachtelte Unterliste `reagents` hinein, um das Feld `name` zu extrahieren. Die verbleibenden nicht extrahierten Elemente bleiben in der Spalte `results`. Dieser selektive Ansatz ist effizienter und lesbarer als ein volles `unnest_wider()` gefolgt von Spaltenauswahl, wenn man nur wenige Felder aus einer komplexen Struktur benötigt.

### Übung: List-Column rectanglen

Man führe mit dem folgenden Tibble mit verschachtelten Laborergebnissen die folgenden Aufgaben durch:

```
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) Man verwende `hoist()`, um nur `ph` und `method` aus der Spalte `results` zu extrahieren.

b) Man verwende stattdessen `unnest_wider()` auf der Spalte `results`. Wie unterscheidet sich die Ausgabe von `hoist()`?

c) Man verwende `unnest_longer()` auf der Spalte `replicates`, um eine Zeile pro Replikat-Messung zu erhalten.

## Lösung

```
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
```

# Fehlende Werte beim Reshaping

Reshaping-Operationen erzeugen oder enthüllen häufig fehlende Werte. Ein Datensatz kann im Wide-Format vollständig erscheinen, aber beim Pivotieren ins Long-Format Lücken entwickeln, oder umgekehrt. Das tidyr-Paket bietet drei komplementäre Funktionen für den Umgang mit fehlenden Werten im Kontext der Datenumstrukturierung.

## complete() — Implizite fehlende Werte explizit machen

Datensätze haben oft „implizite" fehlende Werte — Kombinationen von Variablen, die einfach nicht in den Daten vorkommen, anstatt als `NA` erfasst zu sein. Die Funktion `complete()` erzeugt alle Kombinationen der angegebenen Variablen und füllt `NA` ein, wo Daten fehlen.

```
# 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
```

Jetzt wird die fehlende `High`-Behandlung zum Zeitpunkt 1 explizit als `NA` angezeigt. Das ist wichtig für Analysen, die balancierte Designs erfordern, oder für Visualisierungen, bei denen fehlende Punkte erkennbar sein sollten.

Man kann auch Füllwerte angeben, um die erzeugten `NA`s zu ersetzen:

```
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() — Nach unten oder oben auffüllen

Die Funktion `fill()` propagiert den letzten nicht-fehlenden Wert nach vorne (unten) oder nach hinten (oben). Das ist besonders nützlich für Daten, bei denen Gruppenüberschriften nur einmal erscheinen und nachfolgende Zeilen denselben Wert erben.

```
# 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
```

Das Argument `.direction` steuert die Füllrichtung: `"down"` (Standard), `"up"`, `"downup"` (erst nach unten, dann nach oben) oder `"updown"` (erst nach oben, dann nach unten).

> ! Wichtig
>
> Das Vorwärts-Auffüllen numerischer Messwerte (Last Observation Carried Forward, LOCF) ist eine Form der Imputation, die Verzerrungen einführen kann. Es ist angemessen für strukturelle Muster wie wiederholte Gruppenbezeichnungen, sollte aber bei tatsächlichen Messwerten mit Vorsicht eingesetzt werden. Für seriöse statistische Imputation empfehlen sich spezialisierte Pakete wie {mice} oder {missForest}.

## replace_na() — Gezielte NA-Ersetzung

Die Funktion `replace_na()` ersetzt `NA`-Werte durch angegebene Ersatzwerte, entweder über den gesamten Data Frame oder innerhalb eines `mutate()`-Aufrufs für einzelne Spalten.

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

# 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 Die richtige Missing-Value-Strategie wählen**

Diese drei Funktionen dienen unterschiedlichen Zwecken. Man verwendet `complete()`, wenn alle Kombinationen von Variablen explizit vorhanden sein müssen — das ist üblich vor Joins oder Plots. `fill()` verwendet man, wenn fehlende Werte einem strukturellen Muster folgen (z.B. wiederholte Gruppenbezeichnungen). `replace_na()` verwendet man, wenn man einen bestimmten Imputationswert im Sinn hat. Für seriöse statistische Imputation (Multiple Imputation, Predictive Mean Matching) sind spezialisierte Pakete wie {mice} oder {missForest} besser geeignet.

# Alles zusammenführen

Zum Abschluss dieses Kapitels gehen wir eine End-to-End-Pipeline durch, die mehrere der oben behandelten Techniken kombiniert. Das Ziel ist, einen absichtlich unordentlichen Datensatz Schritt für Schritt in ein sauberes, analysetaugliches Format zu überführen.

## Der unordentliche Datensatz

Man stelle sich ein standortübergreifendes Feldexperiment vor, bei dem Messungen von zwei Zielgrößen (Ertrag und Feuchtigkeit) über drei Jahre erhoben wurden. Die Daten kommen in einem Wide-Format mit zusammengesetzten Spaltennamen an:

```
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>
```

Dieser Datensatz hat drei Probleme: (1) Ertrag und Feuchtigkeit für jedes Jahr befinden sich in separaten Spalten, (2) einige Standort-Behandlungs-Jahres-Kombinationen fehlen, und (3) der Standort Gamma hat nur Behandlung A.

## Schritt 1: Pivotieren mit .value

Zunächst strukturiert man die Daten um, sodass jede Zeile eine Standort-Behandlungs-Jahres-Kombination darstellt, während `yield` und `moisture` als separate Spalten erhalten bleiben:

```
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
```

## Schritt 2: Fehlende Kombinationen vervollständigen

Als Nächstes macht man alle implizit fehlenden Kombinationen explizit. Dem Standort Gamma fehlt Behandlung B vollständig:

```
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
```

## Schritt 3: Strukturelle Muster auffüllen

In diesem speziellen Beispiel könnte man entscheiden, dass die fehlende Gamma-B-Kombination als `NA` bleiben soll (sie wurde nie angelegt). Angenommen jedoch, die Spalte `site` wäre nur für die erste Zeile jeder Gruppe eingetragen worden — dann würde `fill()` die Werte wiederherstellen. Zur Demonstration wenden wir `fill()` auf die verbleibenden `NA`s in den Antwortvariablen an, indem wir innerhalb jeder Standort-Behandlungs-Gruppe den Wert des Vorjahres verwenden:

```
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
```

## Schritt 4: Abschließende Zusammenfassung

Mit den bereinigten Daten kann man nun Zusammenfassungen berechnen. Zum Beispiel den mittleren Ertrag und die mittlere Feuchtigkeit pro Behandlung über alle Standorte und Jahre:

```
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
```

Die vollständige Pipeline von den Rohdaten bis zur Zusammenfassung lässt sich als eine einzige Kette schreiben:

```
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
```

Diese Pipeline zeigt, wie tidyr-Funktionen sich natürlich mit dplyr zusammenfügen: Pivotieren zum Umstrukturieren, Complete zum Auffüllen von Lücken, Fill zum Propagieren von Werten, und dann Summarise für die Analyse. Jeder Schritt erzeugt ein gültiges Tibble, das unabhängig inspiziert werden kann, was die Transformation sowohl transparent als auch gut debuggbar macht.

# Zusammenfassung

Dieses Kapitel hat tidyrs vollständiges Reshaping-Toolkit über einfaches Pivoting hinaus behandelt.

> ℹ Wichtige Erkenntnisse

1. **Fortgeschrittenes Pivoting:** `names_sep` teilt Spaltennamen an einem Delimiter, `names_pattern` verwendet Regex-Capture-Groups für komplexe Muster, und der `.value`-Sentinel hält zusammengehörige Spalten beim Pivoting zusammen.
2. **Trennen und Vereinen:** `separate_wider_delim()` und `separate_wider_regex()` teilen Spalten in mehrere neue Spalten, `separate_longer_delim()` erzeugt neue Zeilen, und `unite()` führt Spalten zusammen.
3. **Verschachtelte Daten:** `unnest_longer()` expandiert List-Columns in Zeilen, `unnest_wider()` in Spalten, und `hoist()` extrahiert gezielt bestimmte Felder aus tief verschachtelten Strukturen.
4. **Fehlende Werte:** `complete()` macht implizite fehlende Werte explizit, `fill()` propagiert Werte nach unten oder oben (bei Messwerten mit Vorsicht verwenden), und `replace_na()` ersetzt durch bestimmte Werte.
5. **Best Practice:** Reshaping-Pipelines schrittweise aufbauen und jedes Zwischenergebnis inspizieren. So werden komplexe Transformationen transparent und gut debuggbar.

# Bibliography