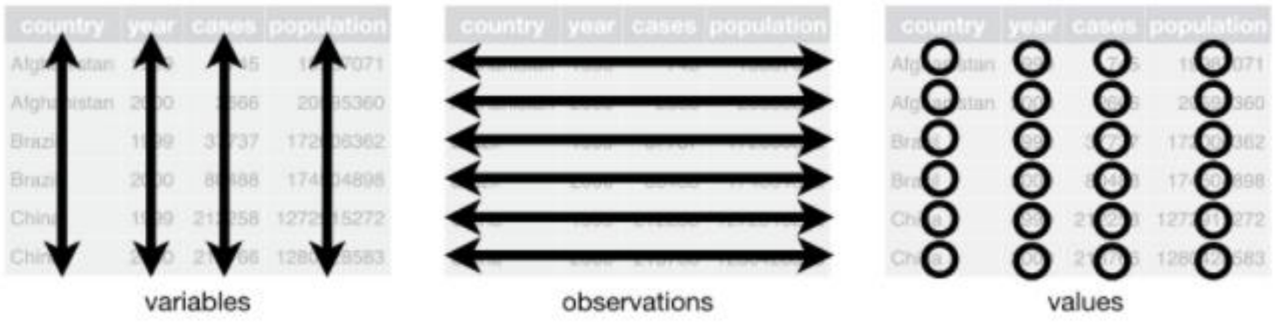
Data Wrangling - tidyr

“Tidy datasets are all alike, but every messy dataset is messy in its own way.” — Hadley Wickham

Tidy data:

- **Every column is variable**
- **Every row is an observation.**
- **Every cell is a single value.**



https://r4ds.had.co.nz/tidy-data.html

Why tidyr?

- If you have a consistent data structure, it’s easier to learn the tools that work with it.

- Most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

- dplyr, ggplot2, and all the other packages in the tidyverse are designed to work with tidy data.

https://r4ds.had.co.nz/tidy-data.html

Example Data

Representation of the same data in multiple ways

```
table1
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <int>  <int>      <int>
#> 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
```

```
table2
#> # A tibble: 12 x 4
#>   country      year type            count
#>   <chr>       <int> <chr>           <int>
#> 1 Afghanistan  1999 cases             745
#> 2 Afghanistan  1999 population   19987071
#> 3 Afghanistan  2000 cases            2666
#> 4 Afghanistan  2000 population   20595360
#> 5 Brazil       1999 cases           37737
#> 6 Brazil       1999 population  172006362
#> # … with 6 more rows
```

```
table3
#> # A tibble: 6 x 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
```

```
table4a  # cases
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
```

```
table4b  # population
#> # A tibble: 3 x 3
#>   country         `1999`     `2000`
#> * <chr>            <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2 Brazil       172006362  174504898
#> 3 China       1272915272 1280428583
```

Pivoting

- **One variable might be spread across multiple columns. (Use Pivot_longer())**

```
table4a  # cases
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
```

```
table4b  # population
#> # A tibble: 3 x 3
#>   country         `1999`     `2000`
#> * <chr>            <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2 Brazil       172006362  174504898
#> 3 China       1272915272 1280428583
```

- **One observation might be scattered across multiple rows. (Use Pivot_wider())**

https://r4ds.had.co.nz/tidy-data.html

Apply Pivot_longer() to table4a

```
table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
#> # A tibble: 6 x 3
#>   country     year   cases
#>   <chr>       <chr>  <int>
#> 1 Afghanistan 1999     745
#> 2 Afghanistan 2000    2666
#> 3 Brazil      1999   37737
#> 4 Brazil      2000   80488
#> 5 China       1999  212258
#> 6 China       2000  213766
```

https://r4ds.had.co.nz/tidy-data.html

Apply Pivot_longer() and combine data

```
tidy4a <- table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
tidy4b <- table4b %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")
left_join(tidy4a, tidy4b)
#> Joining, by = c("country", "year")
#> # A tibble: 6 x 4
#>   country     year   cases population
#>   <chr>       <chr>  <int>      <int>
#> 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
```

```
table4a  # cases
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
```

```
table4b  # population
#> # A tibble: 3 x 3
#>   country         `1999`     `2000`
#> * <chr>            <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2 Brazil       172006362  174504898
#> 3 China       1272915272 1280428583
```

https://r4ds.had.co.nz/tidy-data.html

Pivoting

- One variable might be spread across multiple columns. (Use Pivot_longer())

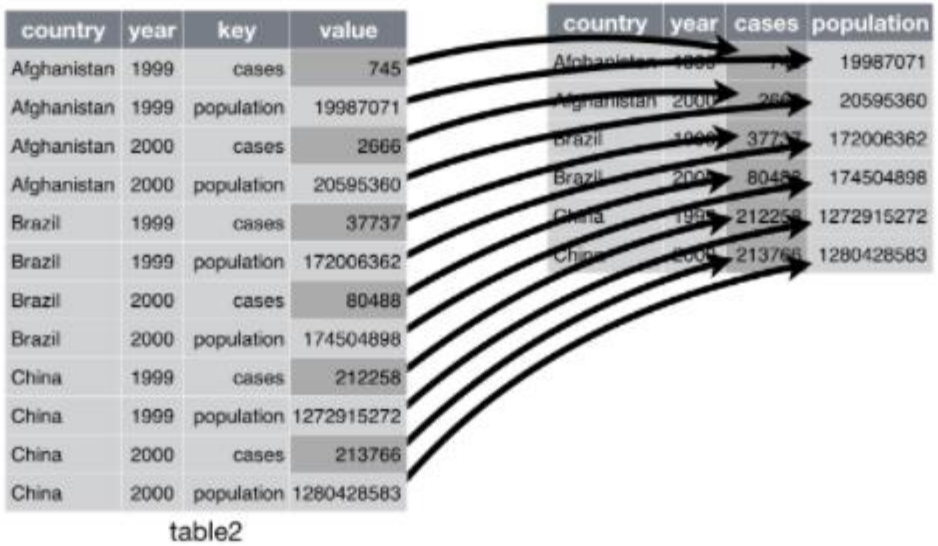
- **One observation might be scattered across multiple rows. (Use Pivot_wider())**

```
table2
#> # A tibble: 12 x 4
#>   country      year type           count
#>   <chr>       <int> <chr>          <int>
#> 1 Afghanistan  1999 cases            745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000 cases           2666
#> 4 Afghanistan  2000 population  20595360
#> 5 Brazil       1999 cases          37737
#> 6 Brazil       1999 population 172006362
#> # … with 6 more rows
```

https://r4ds.had.co.nz/tidy-data.html

Apply Pivot_wider()

```
table2 %>%
   pivot_wider(names_from = type, values_from = count)
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <int>  <int>      <int>
#> 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
```



https://r4ds.had.co.nz/tidy-data.html

Separating and uniting

- **One column contains two variables (Use separate())**

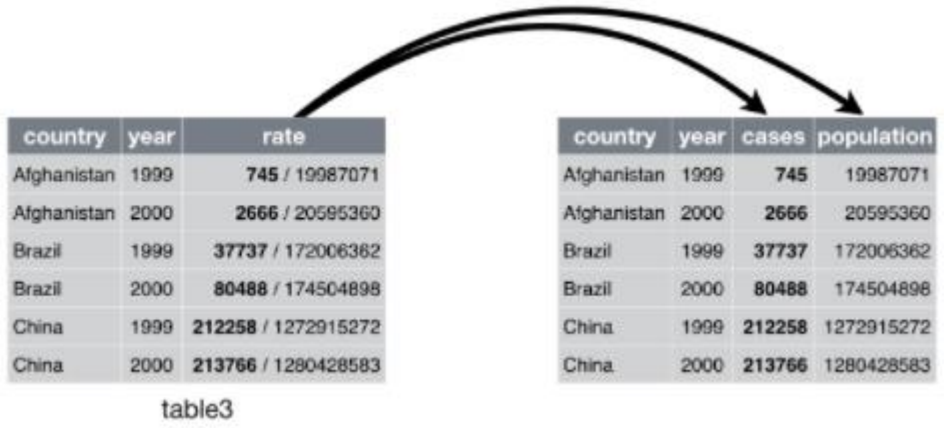
```
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil      1999 37737/172006362
#> 4 Brazil      2000 80488/174504898
#> 5 China       1999 212258/1272915272
#> 6 China       2000 213766/1280428583
```

- **Single variable is spread across multiple columns. (Use unite())**

https://r4ds.had.co.nz/tidy-data.html

Apply separate()

```
table3 %>%
  separate(rate, into = c("cases", "population"))
#> # A tibble: 6 x 4
#>   country      year cases  population
#>   <chr>        <int> <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
```



https://r4ds.had.co.nz/tidy-data.html

Separating and uniting

- One column contains two variables (Use separate())

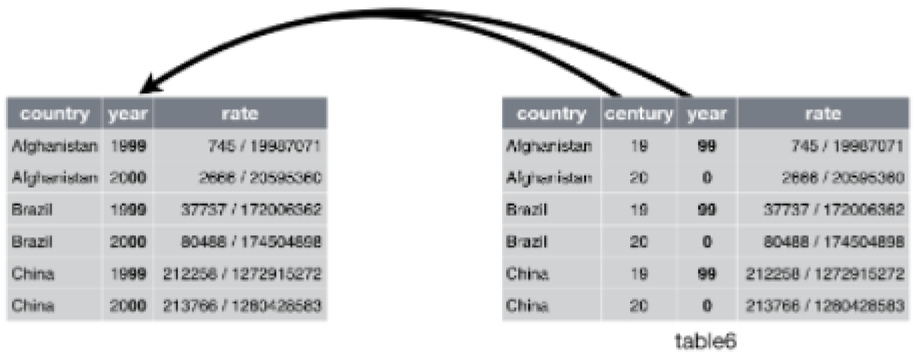
- **Single variable is spread across multiple columns. (Use unite())**

	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

https://r4ds.had.co.nz/tidy-data.html

Apply unite()

```
table5 %>%
  unite(new, century, year, sep = "")
#> # A tibble: 6 x 3
#>   country     new   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
```



https://r4ds.had.co.nz/tidy-data.html

Functional Programming

- For loops are quite verbose, and require quite a bit of bookkeeping code that is duplicated for every for loop.

- Functional programming offers tools to extract out this duplicated code, so each common for loop pattern gets its own function.

https://r4ds.had.co.nz/iteration.html

For loops

Random data

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

To calculate median

```
median(df$a)
#> [1] -0.2457625
median(df$b)
#> [1] -0.2873072
median(df$c)
#> [1] -0.05669771
median(df$d)
#> [1] 0.1442633
```

With for loops

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

https://r4ds.had.co.nz/iteration.html

Three components

The output: output <- vector("double", length(x)).
Allocate sufficient space for the output.
(If you grow the for loop at each iteration using c() (for example), your for loop will be very slow)

The sequence: i in seq_along(df).
what to loop over:
each run of the for loop will assign i to a different value from seq_along(df)

The body: output[[i]] <- median(df[[i]]).
This is the code that does the work.

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

https://r4ds.had.co.nz/iteration.html

For loops vs. functionals

Possible to wrap up for loops in a function

```
col_mean <- function(df) {
	output <- vector("double", ncol(df))
	for (i in seq_along(df)) {
		output[[i]] <- mean(df[[i]])
	}
	output
}
```

```
col_median <- function(df) {
	output <- vector("double", ncol(df))
	for (i in seq_along(df)) {
		output[[i]] <- median(df[[i]])
	}
	output
}
```

```
col_sd <- function(df) {
	output <- vector("double", ncol(df))
	for (i in seq_along(df)) {
		output[[i]] <- sd(df[[i]])
	}
	output
}
```

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
#> [1] -0.51850298  0.02779864  0.17295591 -0.61163819
col_summary(df, mean)
#> [1] -0.3260369  0.1356639  0.4291403 -0.2498034
```

https://r4ds.had.co.nz/iteration.html

The map function (purrr)

the purrr package provides a family of functions for looping patterns over a vector

- **map() makes a list**
- **map_lgl() makes a logical vector**
- **map_int() makes an integer vector**
- **map_dbl() makes a double vector**
- **map_chr() makes a character vector**

Alternatives: apply, lapply, etc

https://r4ds.had.co.nz/iteration.html

The map function (purrr)

```
df %>% map_dbl(mean)
#>          a          b          c          d
#> -0.3260369  0.1356639  0.4291403 -0.2498034
df %>% map_dbl(median)
#>           a           b           c           d
#> -0.51850298  0.02779864  0.17295591 -0.61163819
df %>% map_dbl(sd)
#>         a         b         c         d
#> 0.9214834 0.4848945 0.9816016 1.1563324
```

You can define a function in a map function

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df))
```

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

https://r4ds.had.co.nz/iteration.html

To extract a component

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>         4         6         8
#> 0.5086326 0.4645102 0.4229655
```

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
#>         4         6         8
#> 0.5086326 0.4645102 0.4229655
```

You can also use an integer to select elements by position

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
#> [1] 2 5 8
```

https://r4ds.had.co.nz/iteration.html