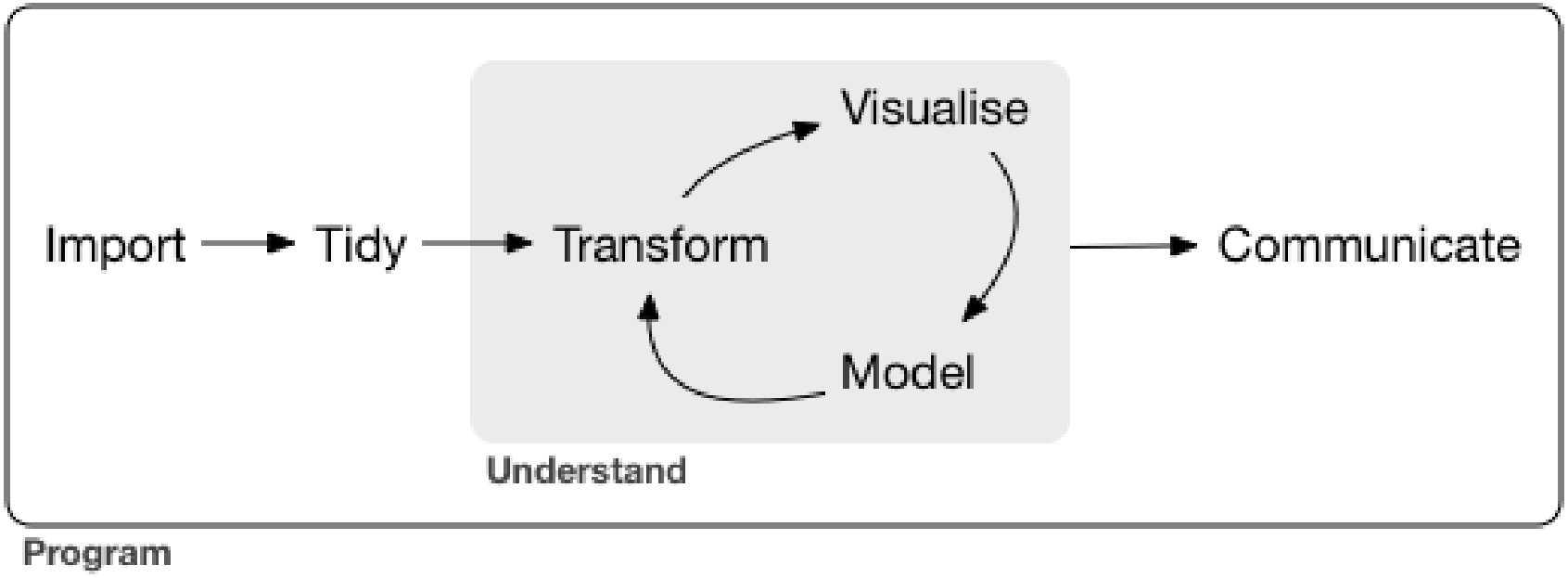
Data Wrangling

Guy J. Abel

Tidyverse

- Some of the functions in `base` can be un-intuitive and slow when dealing with big data sets.
  - For a long time manipulations of data frames in R was done with functions like `with`, `subset`, `merge`, `transform`, `split`, `reshape`.
  - Some of these are really painful to understand (check out `?reshape`)
  - They were also fairly slow and prone to crashing R if you used the wrong values.
- The tidyverse collection of packages are becoming the preferred way of have handling data in R.
- The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.
- Tidyverse packages were created in recent years by Hadley Wickham (Chief Scientist at RStudio) and colleagues.

Tidyverse


Visualise
Import
Tidy
Transform
Communicate
Model
Understand
Program

Tidyverse

- The `tidyverse` package does not have any functions for importing, manipulating or visualizing data.
  - Is a wrapper to load six other packages for these purposes.
  - When the tidyverse package is installed, so are these five other packages as well as their dependencies and some other related packages, such as `readxl`

```
> library(tidyverse)
Loading tidyverse: ggplot2
Loading tidyverse: tibble
Loading tidyverse: tidyr
Loading tidyverse: readr
Loading tidyverse: purrr
Loading tidyverse: dplyr
Conflicts with tidy packages ----------------------------------------------
filter(): dplyr, stats
lag():    dplyr, stats
```

Tidy Data

- There are many ways to organize data frames.
- However, R likes data if
  - Variables in the data set have their own column
  - Observation are placed in its own row
  - Values are placed in its own cell
- This is what Wickham calls "tidy" data.
- Most functions in R (and Stata, SPSS and SAS) work with data in this format.
  - R is a vectorized programming language, the machinery behind R is optimized to work with vectors.
  - When data is in tidy format R coding is more straightforward and data manipulations are faster.

Tidy Data Philosophy

- Raw data is rarely in a tidy format.
- Tidying data can take much longer than the analysis.
- Learning the grammar of data manipulations to format data to a tidy format.
  - Saves a lot of time.
  - Reduces the steps in managing non-tidy data. These tend to multiply quickly for larger data sets.
  - Reduces the risk of making an errors (through less steps).
  - Provides a consistency to analysis, making extensions far easier.

Reshaping Data

- The `tidyr` package helps get data into a tidy format.
- At its heart is two functions that can alter the layout of data sets:
  - `gather()`
  - `spread()`.
- For each of these, the concept of a keys and values is essential.
- A keys and values are a simple way to record information:
  - key explains what the information describes
  - value contains the actual information.
- When data is in a tidy format, the values for each variable need their own (single) column.

Reshaping Data

gather(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor_key = FALSE**)**

Gather moves column names into a key column, gathering the column values into a single value column.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

gather(table4a, `1999`, `2000`, key = "year", value = "cases")

spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL**)**

Spread moves the unique values of a key column into the column names, spreading the values of a value column across the new columns that result.

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

key value

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

spread(table2, type, count)

Gather

- The `gather()` function takes values over multiple columns and gathering them to form a database with fewer columns.
- It to has four main arguments for
  - `data`: your data frame.
  - `key`: the name of a new column describing the columns to be gathered
  - `value`: the name of the new column describing the values in the columns to be gathered
  - `...`: variable names (or column numbers) of columns to be gathered.
- Convert to the key variable to numeric using the `convert = TRUE` argument (default `convert = FALSE`)

Keys and Values

- To demonstrate we will use data from the UN on past total fertility rates (TFR) in all countries.
- Inspect the `WPP2015_FERT_F04_TOTAL_FERTILITY.xls` file. What are the values here in the `ESTIMATES` sheet? What are the keys?

```
file.show("./data/WPP2015_FERT_F04_TOTAL_FERTILITY.xls")
```

```
> library(readxl)
> #read in the data and tidy up (more later on %>%, rename filter and select)
> df1 <- read_excel("./data/WPP2015_FERT_F04_TOTAL_FERTILITY.xls",
+                   sheet = "ESTIMATES", skip = 16) %>%
+   rename(name = "Major area, region, country or area *",
+          code = "Country code") %>%
+   filter(!(Notes %in% letters), code != 948) %>%
+   select(-Index, -Notes, -Variant)
```

Keys and Values

```
> df1
# A tibble: 230 x 15
                name  code `1950-1955` `1955-1960` `1960-1965` `1965-1970` `1970-1975`
               <chr> <dbl>        <dbl>        <dbl>        <dbl>        <dbl>        <dbl>
 1           WORLD   900     4.961571     4.898665     5.024379     4.922202     4.475478
 2          AFRICA   903     6.599479     6.640245     6.702032     6.671123     6.667982
 3 Eastern Africa   910     7.012207     7.006985     7.074612     7.088212     7.129931
 4         Burundi   108     6.801000     6.857000     7.071000     7.268000     7.343000
 5         Comoros   174     6.000000     6.601000     6.909000     7.050000     7.050000
 6        Djibouti   262     6.312000     6.387400     6.547000     6.707000     6.845000
 7         Eritrea   232     6.965000     6.965000     6.815000     6.699000     6.620000
 8        Ethiopia   231     7.169600     6.902300     6.897200     6.869100     7.103800
 9           Kenya   404     7.481000     7.785000     8.065000     8.110000     7.990000
10      Madagascar   450     7.300000     7.300000     7.300000     7.300000     7.300000
# ... with 220 more rows, and 8 more variables: `1975-1980` <dbl>,
#   `1980-1985` <dbl>, `1985-1990` <dbl>, `1990-1995` <dbl>, `1995-2000` <dbl>,
#   `2000-2005` <dbl>, `2005-2010` <dbl>, `2010-2015` <dbl>
```

Gather

```
> library(tidyverse)
> # reshape
> df2 <- gather(data = df1, key = period, value = tfr, 3:15)
> df2
# A tibble: 2,990 x 4
             name  code    period      tfr
            <chr> <dbl>     <chr>    <dbl>
 1           WORLD   900 1950-1955 4.961571
 2          AFRICA   903 1950-1955 6.599479
 3 Eastern Africa   910 1950-1955 7.012207
 4         Burundi   108 1950-1955 6.801000
 5         Comoros   174 1950-1955 6.000000
 6        Djibouti   262 1950-1955 6.312000
 7         Eritrea   232 1950-1955 6.965000
 8        Ethiopia   231 1950-1955 7.169600
 9           Kenya   404 1950-1955 7.481000
10      Madagascar   450 1950-1955 7.300000
# ... with 2,980 more rows
```

Gather

- You can use a variety of methods to select the part of the data set you want to collapse
- These all give the same data set as before

```
> # from column `1950-1955` to column `2010-2015` (and every column inbetween)
> df2 <- gather(data = df1, key = period, value = tfr, `1950-1955`:`2010-2015`)
> # columns containing "-" in their name
> df2 <- gather(data = df1, key = period, value = tfr, contains("-"))
> # without the name and code columns
> df2 <- gather(data = df1, key = period, value = tfr, -name, -code)
> # without the columns from name to column code (and every column inbetween)
> df2 <- gather(data = df1, key = period, value = tfr, -(name:code))
```

Exercise 1 (ex41.R)

```
# 0.  Clear your workspace and set your working directory to your data folder in the


##
##
##
# 1.  Load the tidyverse and readxl packages


# 2. Uncomment and run the following code (will cover these functions later in the c
# d1 <- read_excel("WPP2015_POP_F13_A_OLD_AGE_DEPENDENCY_RATIO_1564.xls", sheet = "E
#   select(-Index, -Notes, -Variant) %>%
#   rename(name = `Major area, region, country or area *`,
#          code = `Country code`)
# d1
# 3. Take a look at d1 and the ESTIMATES sheet of WPP2015_POP_F13_A_OLD_AGE_DEPENDEN
#    a) Q: What are the values for when we tidy the data?
#       A:
#    b) Q: What are the key variables for when we tidy the data?
#       A:
#    c) Modify d1 to
#       i) a tidy format with dimensions 3,374 x 4
#       ii) with the key variable converted to a suitable mode

##
```

Spread

- Where `gather()` collects values from many columns to one, `spread()` does the opposite, taking values in one columns and spreading them to form a database with more columns, based on the variables in the data.
- It will take excessively long data sets with too many rows and widen them towards a tidy format
- It has three principle arguments, similar to `gather`
  - `data`: your data frame.
  - `key`: the name of the column whose values will be used as column headers
  - `value`: the name of the column whose values will population the cells.
- Notice
  - In `gather()` the names for `key` and `value` we had to make up.
  - In `spread()` the names for `key` and `value` are already in the data.

Spread

- To illustrate we use data from the World Bank Development database. It contains measures of inequality and poverty for all countries.
  - What do we need to think about when we import the data
  - What are the values in the data? What are the key variable(s)?

```
file.show("./data/wb_ginipov.csv")
```

```
> df3 <- read_csv(file = "./data/wb_ginipov.csv", skip = 3, na = "..") %>%
+   select(-`Series Name`)
> df3
# A tibble: 3,480 x 6
   `Series Code`     Country `Country Code`  Year `Year Code` Value
           <chr>       <chr>          <chr> <int>       <chr> <dbl>
 1   SI.POV.GINI Afghanistan            AFG  2006      YR2006    NA
 2   SI.POV.GINI Afghanistan            AFG  2007      YR2007    NA
 3   SI.POV.GINI Afghanistan            AFG  2008      YR2008    NA
 4   SI.POV.GINI Afghanistan            AFG  2009      YR2009    NA
 5   SI.POV.GINI Afghanistan            AFG  2010      YR2010    NA
 6   SI.POV.GINI Afghanistan            AFG  2011      YR2011    NA
 7   SI.POV.GINI Afghanistan            AFG  2012      YR2012    NA
 8   SI.POV.GINI Afghanistan            AFG  2013      YR2013    NA
 9   SI.POV.GINI Afghanistan            AFG  2014      YR2014    NA
10   SI.POV.GINI Afghanistan            AFG  2015      YR2015    NA
# ... with 3,470 more rows
```

Spread

```
> # create a tidy data frame.
> df4 <- spread(data = df3, key = `Series Code`, value = Value)
> df4
# A tibble: 1,740 x 6
       Country `Country Code`  Year `Year Code` SI.POV.GINI SI.POV.NAGP
 *       <chr>          <chr> <int>       <chr>       <dbl>       <dbl>
 1 Afghanistan            AFG  2006      YR2006          NA          NA
 2 Afghanistan            AFG  2007      YR2007          NA         7.9
 3 Afghanistan            AFG  2008      YR2008          NA          NA
 4 Afghanistan            AFG  2009      YR2009          NA          NA
 5 Afghanistan            AFG  2010      YR2010          NA          NA
 6 Afghanistan            AFG  2011      YR2011          NA         8.4
 7 Afghanistan            AFG  2012      YR2012          NA          NA
 8 Afghanistan            AFG  2013      YR2013          NA          NA
 9 Afghanistan            AFG  2014      YR2014          NA          NA
10 Afghanistan            AFG  2015      YR2015          NA          NA
# ... with 1,730 more rows
```

Split and Combine

- The `separate()` and `unite()` functions help you split and combine cells to place a single, complete value in each cell.
- The `separate_rows()` function helps tidy a variable containing observations with multiple delimited values

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ...**)**

Separate each cell in a column to make several columns.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174
C	1999	212K	1T
C	2000	213K	1T

separate_rows(table3, rate, into = c("cases", "pop"))

unite(data, col, ..., sep = "_", remove = TRUE**)**

Collapse cells across several columns to make a single column.

table5

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

→

country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

unite(table5, century, year, col = "year", sep = "")

separate_rows(data, ..., sep = "[^[:alnum:].]+", convert = FALSE**)**

Separate each cell in a column to make several rows. Also **separate_rows_()**.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M
C	1999	212K
C	1999	1T
C	2000	213K
C	2000	1T

separate_rows(table3, rate)

Separate

- The `separate()` function splits a character column into multiple columns based on pattern.
- For example we can work create new columns for the start and end year in the tfr data
- Notice by default these turn into `character` vectors

```
> separate(df2, period, into = c("year0", "year5"), sep = "-")
# A tibble: 2,990 x 5
             name  code year0 year5      tfr
 *           <chr> <dbl> <chr> <chr>    <dbl>
 1          WORLD   900  1950  1955 4.961571
 2         AFRICA   903  1950  1955 6.599479
 3 Eastern Africa   910  1950  1955 7.012207
 4        Burundi   108  1950  1955 6.801000
 5        Comoros   174  1950  1955 6.000000
 6       Djibouti   262  1950  1955 6.312000
 7        Eritrea   232  1950  1955 6.965000
 8       Ethiopia   231  1950  1955 7.169600
 9          Kenya   404  1950  1955 7.481000
10     Madagascar   450  1950  1955 7.300000
# ... with 2,980 more rows
```

Seperate

- We can convert to numeric using the `convert` argument
- Keep the original column using `remove`

```
> separate(df2, period, into = c("year0", "year5"), sep = "-",
+          convert = TRUE, remove = FALSE)
# A tibble: 2,990 x 6
              name  code    period year0 year5      tfr
 *           <chr> <dbl>     <chr> <int> <int>    <dbl>
 1          WORLD   900 1950-1955  1950  1955 4.961571
 2         AFRICA   903 1950-1955  1950  1955 6.599479
 3 Eastern Africa   910 1950-1955  1950  1955 7.012207
 4        Burundi   108 1950-1955  1950  1955 6.801000
 5        Comoros   174 1950-1955  1950  1955 6.000000
 6       Djibouti   262 1950-1955  1950  1955 6.312000
 7        Eritrea   232 1950-1955  1950  1955 6.965000
 8       Ethiopia   231 1950-1955  1950  1955 7.169600
 9          Kenya   404 1950-1955  1950  1955 7.481000
10     Madagascar   450 1950-1955  1950  1955 7.300000
# ... with 2,980 more rows
```

Expand and Complete

- The `expand()` and `complete()` functions help you quickly create tables with combinations of values.

```
> # create data
> d0 <- data_frame(year = c(2010, 2000, 2015), alpha3 = c("CHN", "KOR", "JPN"))
> d0
# A tibble: 3 x 2
   year alpha3
  <dbl>  <chr>
1  2010    CHN
2  2000    KOR
3  2015    JPN
```

Expand and Complete

- Expand can save you lots of time and mistakes when you want all combinations.

```
> # expand for all combinations
> expand(d0, year, alpha3)
# A tibble: 9 x 2
   year alpha3
  <dbl>  <chr>
1  2000    CHN
2  2000    JPN
3  2000    KOR
4  2010    CHN
5  2010    JPN
6  2010    KOR
7  2015    CHN
8  2015    JPN
9  2015    KOR
```

Expand and Complete

- The `full_seq()` function in `tidyr` helps complete data frames based on existing columns

```
> # complete
> complete(d0, alpha3, year = full_seq(x = year, period = 5))
# A tibble: 12 x 2
   alpha3  year
    <chr> <dbl>
 1    CHN  2000
 2    CHN  2005
 3    CHN  2010
 4    CHN  2015
 5    JPN  2000
 6    JPN  2005
 7    JPN  2010
 8    JPN  2015
 9    KOR  2000
10    KOR  2005
11    KOR  2010
12    KOR  2015
```

Missing Values

- The `drop_na()` function filters out rows with one or more missing values
- The `fill()` function replaces missing values going up or down a named column.
- The `replace_na()` function replaces missing values given in a names `list`

drop_na(data, ...**)**

Drop rows containing NA's in … columns.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
D	3

drop_na(x, x2)

fill(data, ..., .direction = c("down", "up")**)**

Fill in NA's in … columns with most recent non-NA values.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	1
C	1
D	3
E	3

fill(x, x2)

replace_na(data, replace = list(), ...**)**

Replace NA's by column.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	2
C	2
D	3
E	2

replace_na(x,list(x2 = 2))

Missing Values

```
> # create data x
> d0 <- data_frame(x1 = LETTERS[1:5], x2 = c(1, NA, NA, 3, NA))
> d0
# A tibble: 5 x 2
     x1    x2
  <chr> <dbl>
1     A     1
2     B    NA
3     C    NA
4     D     3
5     E    NA
> # drop_na
> drop_na(d0)
# A tibble: 2 x 2
     x1    x2
  <chr> <dbl>
1     A     1
2     D     3
```

Missing Values

```
> d0
# A tibble: 5 x 2
     x1    x2
  <chr> <dbl>
1     A     1
2     B    NA
3     C    NA
4     D     3
5     E    NA
> # fill
> fill(d0, x2)
# A tibble: 5 x 2
     x1    x2
  <chr> <dbl>
1     A     1
2     B     1
3     C     1
4     D     3
5     E     3
```

Missing Values

```
> d0
# A tibble: 5 x 2
     x1    x2
  <chr> <dbl>
1     A     1
2     B    NA
3     C    NA
4     D     3
5     E    NA
> # separate rows
> replace_na(d0, list(x2 = 2))
# A tibble: 5 x 2
     x1    x2
  <chr> <dbl>
1     A     1
2     B     2
3     C     2
4     D     3
5     E     2
```

tidyr

Function	Description
`gather()`	Gather columns into key-value pairs
`spread()`	Spread a key-value pair across multiple columns.
`seperate()`	Separate one column into multiple columns.
`unite()`	Unite multiple columns into one.
`separate_rows()`	Separate a collapsed column into multiple rows
`fill()`	Fill in missing values.
`drop_na()`	Drop rows containing missing values
`replace_na()`	Replace missing values

Exercise 2 (ex42.R)

```
# 0.  a) Check your working directory is in the course folder

#     b) Load the data and packages by sourcing the solution file for ex41.R

##
##
##
# 1.  Modify d2 to drop the rows with missing values

# 2.  Modify d2 to separate the age column so that
#     a) two new columns for the first and last age in each group are created
#     b) the original age column is not removed
#     c) the new columns are integer values

# 3.  Modify d3 to drop the rows with missing values

# 4.  Uncomment the following code to display part of the policy data base in d3
# d3 %>%
#   filter(issue == "Measures on irregular immigration") %>%
#   select(name, issue, policy)
# 5.  a) Modify d3 using the separate_rows() function to create new rows for each po
#     (Hint: use sep = "," and the final data frame should be 7,796 rows)
#     b) Check they are working by re-running the R code from the previous question.
```

dplyr

- Once your data is in a tidy format, more often than not we will want to make some transformations.
- The `dplyr` package is becoming increasingly popular for data manipulation.
  - Built by Hadley Wickham and Romain Francois
  - Fast, highly expressive, and open-minded about how your data is stored.
  - Originated out of `plyr`, which works for many different inputs (e.g., `arrays`, `data.frames`, `lists`)
  - `dplyr` is focused on data frames only.

dplyr Verbs

- There are six `dplyr` functions that you will use to do the vast majority of data manipulations:
  - `select()` to pick variables by their names
  - `filter()` to pick observations by their values
  - `arrange()` to reorder the rows
  - `mutate()` to create new variables with functions of existing variables
  - `summarise()` to collapse many values down to a single summary
- These can all be used in conjunction with `group_by()` (no. 6) which changes the scope of each function from operating on the entire data set to operating on it group-by-group.
- These six functions provide the verbs for a language of data manipulation.

dplyr Verbs

- All verbs work similarly:
  - The first argument is a data frame.
  - The subsequent arguments describe what to do with the data frame.
  - You no longer have to to columns in the data frame directly with $.
  - Will output a new data frame.
- These properties make it easy to chain together multiple simple steps to achieve a complex result.

Extract Variables

- The `select()` function can be used to subset the variables or columns of a data set.

```
> # for example
> select(df2, name, period, tfr)
# A tibble: 2,990 x 3
             name    period      tfr
            <chr>     <chr>    <dbl>
 1          WORLD 1950-1955 4.961571
 2         AFRICA 1950-1955 6.599479
 3 Eastern Africa 1950-1955 7.012207
 4        Burundi 1950-1955 6.801000
 5        Comoros 1950-1955 6.000000
 6       Djibouti 1950-1955 6.312000
 7        Eritrea 1950-1955 6.965000
 8       Ethiopia 1950-1955 7.169600
 9          Kenya 1950-1955 7.481000
10     Madagascar 1950-1955 7.300000
# ... with 2,980 more rows
```

Extract Variables

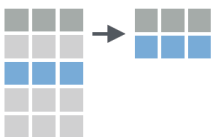
- As demonstrated with `gather()` we can use standard indexing methods in R:
  - `:` for sequence
  - `-` to drop columns
- These also work with names and column numbers
- There are helpers functions to aid selections, e.g.
  - `contains(match)`
  - `starts_with(match)`

Extract Variables

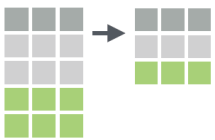
```
> # these are all the same...
> # select(df2, name, period:tfr)
> # select(df2, -2)
> # select(df2, -starts_with("c"))
> # select(df2, -contains("c"))
> select(df2, -code)
# A tibble: 2,990 x 3
             name    period      tfr
            <chr>     <chr>    <dbl>
 1          WORLD 1950-1955 4.961571
 2         AFRICA 1950-1955 6.599479
 3 Eastern Africa 1950-1955 7.012207
 4        Burundi 1950-1955 6.801000
 5        Comoros 1950-1955 6.000000
 6       Djibouti 1950-1955 6.312000
 7        Eritrea 1950-1955 6.965000
 8       Ethiopia 1950-1955 7.169600
 9          Kenya 1950-1955 7.481000
10     Madagascar 1950-1955 7.300000
# ... with 2,980 more rows
```

Extract Cases

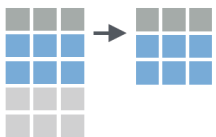
- The `filter()` function returns rows with matching conditions.
- The `distinct()` function returns only unique/distinct rows
- The `slice()` function returns the observations requested.
- The `top_n()` returns the first `n` observations. Similar to `head()`



filter(.data, ...**)**
Extract rows that meet logical criteria.



distinct(.data, ..., .keep_all = FALSE**)**
Remove rows with duplicate values.



slice(.data, ...**)**
Select rows by position.

top_n(*x, n, wt***)**
Select and order top n entries (by group if grouped data).

Extract Cases

- The `filter()` function in `dplyr` is a popular tool for creating subsets.
- Takes logical expressions and returns the rows for which all are `TRUE`

```
> filter(df2, period == "2010-2015", name == "China")
# A tibble: 1 x 4
   name  code    period   tfr
  <chr> <dbl>     <chr> <dbl>
1 China   156 2010-2015  1.55
```

Extract Cases

- The %in% is useful for matching multiple cases in the same variable.

```
> # can also use `<` `>` `>=` `<=` `!=`
> filter(df2, name == "China", tfr < 2)
# A tibble: 4 x 4
   name  code    period   tfr
  <chr> <dbl>     <chr> <dbl>
1 China   156 1995-2000  1.48
2 China   156 2000-2005  1.50
3 China   156 2005-2010  1.53
4 China   156 2010-2015  1.55
>
> # multiple matches on the same variable we can also use %in%
> filter(df2, period == "2010-2015", code %in% c(156, 410))
# A tibble: 2 x 4
                name  code    period    tfr
               <chr> <dbl>     <chr>  <dbl>
1              China   156 2010-2015 1.5500
2 Republic of Korea   410 2010-2015 1.2557
```

Pipelines

- The pipeline operator (%>%) from the `magrittr` package by Stefan Bache is part of the tidyverse and has two big benefits
  - You can tie together R commands and thus avoid nesting code.
  - The syntax leads to code that is much easier to write and explain.
- The pipe operator takes whatever is before the pipe and places into and drops it in as the first argument of the next function call.
- When you see the pipe operator, %>%, read it as 'then'.

```
> # for example
> df2 %>%
+   filter(name == "China", tfr < 2) %>%
+   select(-code)
# A tibble: 4 x 3
   name    period   tfr
  <chr>     <chr> <dbl>
1 China 1995-2000  1.48
2 China 2000-2005  1.50
3 China 2005-2010  1.53
4 China 2010-2015  1.55
> # same as
> # select(filter(.data = df2, name == "China", tfr < 2), -code)
```

Pipelines

- The pipe operator takes whatever is before the pipe and places into and drops it in as the first argument of the next function call.
- You can still specify other (second, third, etc.) arguments (i.e. any but the first) in each function.
- For example to return only the last 3 rows of China's data in `df2`, we could use
  - `tail(filter(.data = df2, name == "China"), n = 3)`
  - or with pipes:

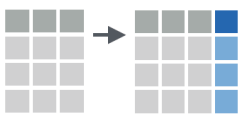
```
> df2 %>%
+   filter(name == "China") %>%
+   tail(n = 3)
# A tibble: 3 x 4
   name  code    period   tfr
  <chr> <dbl>     <chr> <dbl>
1 China   156 2000-2005  1.50
2 China   156 2005-2010  1.53
3 China   156 2010-2015  1.55
```

Exercise 3 (ex43.R)

```
# 0.  a) Check your working directory is in the course folder

#     b) Souce the solution file for ex42.R

##
##
##
# 1.  Display the observations in d1 from Austria

# 2.  Display the distinct observations in d1 of each area name and code (241 rows)

# 3.  Display the observations in d1 using pipes (%>%)
#     a) from Austria
#     b) the name, year and odr variables only


# 4.  Display the observations in d2 using pipes
#     a) from Philippines in 2015
#     b) the name, age and pop variables only


# 5.  Display the observations in d2 using pipes
#     a) from Australia in 2000 over 65
#     b) the name, age and pop variables only
```

Data Creation

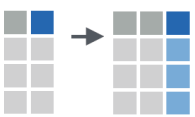
- The `mutate()` function for adding new or replacing existing columns.
- The `transmutate()` function for dropping other columns
- The `rename()` function for changing column names
- The `mutate_all()` modifies all columns
- The `mutate_if()` modifies all columns selected



mutate(.data, …**)**
Compute new column(s).

transmute(.data, …**)**
Compute new column(s), drop others.

rename(.data, …**)**
Rename columns.



mutate_at(.tbl, .cols, .funs, ...**)**
Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for select().

mutate_if(.tbl, .predicate, .funs, ...**)**
Apply funs to all columns of one type. Use with **funs().**

mutate_all(.tbl, .funs, ...**)**
Apply funs to every column. Use with **funs()**.

Adding New Variables

- The `mutate()` function is used when you want to create a new column.
  - Can overwrite new columns if you want to replace their values.
  - Can create new columns, for example to code regions or countries (where country code greater than or equal to 900)

```
> df2 %>%
+   mutate(tfr = round(tfr,1),
+          location_type = ifelse(test = code >= 900, yes = "region", no = "nation"
# A tibble: 2,990 x 5
             name  code    period   tfr location_type
            <chr> <dbl>     <chr> <dbl>         <chr>
 1          WORLD   900 1950-1955   5.0        region
 2         AFRICA   903 1950-1955   6.6        region
 3 Eastern Africa   910 1950-1955   7.0        region
 4        Burundi   108 1950-1955   6.8        nation
 5        Comoros   174 1950-1955   6.0        nation
 6       Djibouti   262 1950-1955   6.3        nation
 7        Eritrea   232 1950-1955   7.0        nation
 8       Ethiopia   231 1950-1955   7.2        nation
 9          Kenya   404 1950-1955   7.5        nation
10     Madagascar   450 1950-1955   7.3        nation
# ... with 2,980 more rows
```

Rename

- If you want to rename the column, rather than create a new column and delete the old one there is a `rename()` function
- Nicer than `colnames(df)[1] <- "newname"` that we were using before as we can use it with pipes.

```
> df2 %>%
+   rename(country_name = name)
# A tibble: 2,990 x 4
      country_name  code    period      tfr
             <chr> <dbl>     <chr>    <dbl>
 1           WORLD   900 1950-1955 4.961571
 2          AFRICA   903 1950-1955 6.599479
 3 Eastern Africa   910 1950-1955 7.012207
 4         Burundi   108 1950-1955 6.801000
 5         Comoros   174 1950-1955 6.000000
 6        Djibouti   262 1950-1955 6.312000
 7         Eritrea   232 1950-1955 6.965000
 8        Ethiopia   231 1950-1955 7.169600
 9           Kenya   404 1950-1955 7.481000
10      Madagascar   450 1950-1955 7.300000
# ... with 2,980 more rows
```

Data Creation

```
> df2 %>%
+   filter(name == "China", tfr < 2) %>%
+   select(code, tfr) %>%
+   mutate_all(funs(.*100))
# A tibble: 4 x 2
   code   tfr
  <dbl> <dbl>
1 15600   148
2 15600   150
3 15600   153
4 15600   155
>
> df2 %>%
+   filter(name == "China", tfr < 2) %>%
+   mutate_if(is.numeric, funs(.*100))
# A tibble: 4 x 4
   name  code    period   tfr
  <chr> <dbl>     <chr> <dbl>
1 China 15600 1995-2000   148
2 China 15600 2000-2005   150
3 China 15600 2005-2010   153
4 China 15600 2010-2015   155
```

Data Creation

```
> df2 %>%
+   filter(name == "China", tfr < 2) %>%
+   transmute(tfr = round(tfr,2))
# A tibble: 4 x 1
    tfr
  <dbl>
1  1.48
2  1.50
3  1.53
4  1.55
```

Arranging Data

- The arrange function sorts data frames according the value of the cells.

```
> df2 %>%
+   filter(name == "China") %>%
+   arrange(tfr)
# A tibble: 13 x 4
    name  code    period    tfr
   <chr> <dbl>     <chr>  <dbl>
 1 China   156 1995-2000 1.4800
 2 China   156 2000-2005 1.5000
 3 China   156 2005-2010 1.5300
 4 China   156 2010-2015 1.5500
 5 China   156 1990-1995 2.0000
 6 China   156 1980-1985 2.5200
 7 China   156 1985-1990 2.7500
 8 China   156 1975-1980 3.0138
 9 China   156 1970-1975 4.8500
10 China   156 1955-1960 5.4760
11 China   156 1950-1955 6.1070
12 China   156 1960-1965 6.1500
13 China   156 1965-1970 6.3000
```

Arranging Data

- Can arrange in descending order using `desc`

```
> df2 %>%
+   filter(name == "China") %>%
+   arrange(desc(tfr))
# A tibble: 13 x 4
    name  code    period    tfr
   <chr> <dbl>     <chr>  <dbl>
 1 China   156 1965-1970 6.3000
 2 China   156 1960-1965 6.1500
 3 China   156 1950-1955 6.1070
 4 China   156 1955-1960 5.4760
 5 China   156 1970-1975 4.8500
 6 China   156 1975-1980 3.0138
 7 China   156 1985-1990 2.7500
 8 China   156 1980-1985 2.5200
 9 China   156 1990-1995 2.0000
10 China   156 2010-2015 1.5500
11 China   156 2005-2010 1.5300
12 China   156 2000-2005 1.5000
13 China   156 1995-2000 1.4800
```

Arranging Data

- If the column being sorted is a `character` will use alphabetical order.
- This is where `factor` come in useful.
- Consider the UN population data by age group...

```
> df5 <- read_excel("./data/WPP2015_POP_F07_1_POPULATION_BY_AGE_BOTH_SEXES.xls",
+                   sheet = "ESTIMATES", skip = 16) %>%
+   rename(name = `Major area, region, country or area *`,
+          code = `Country code`,
+          year = `Reference date (as of 1 July)`) %>%
+   filter(!(Notes %in% letters), code != 948) %>%
+   select(-Index, -Variant, -Notes) %>%
+   gather(key = age_grp, value = pop, -name, -code, -year)
```

Arranging Data

```
> df5
# A tibble: 70,840 x 5
    name  code  year age_grp      pop
   <chr> <dbl> <dbl>   <chr>    <dbl>
 1 WORLD   900  1950     0-4 337431.9
 2 WORLD   900  1955     0-4 402845.0
 3 WORLD   900  1960     0-4 430565.1
 4 WORLD   900  1965     0-4 477798.0
 5 WORLD   900  1970     0-4 522640.9
 6 WORLD   900  1975     0-4 543224.9
 7 WORLD   900  1980     0-4 546894.2
 8 WORLD   900  1985     0-4 589594.1
 9 WORLD   900  1990     0-4 643186.9
10 WORLD   900  1995     0-4 625929.7
# ... with 70,830 more rows
```

Arranging Data

- Consider sorting the China data by age
- Gives an unfriendly order

```
> df5 %>%
+   filter(name == "China", year == 2015) %>%
+   arrange(age_grp)
# A tibble: 22 x 5
    name  code  year age_grp        pop
   <chr> <dbl> <dbl>   <chr>      <dbl>
 1 China   156  2015     0-4  83185.944
 2 China   156  2015   10-14  75291.908
 3 China   156  2015    100+     47.657
 4 China   156  2015   15-19  78930.129
 5 China   156  2015   20-24 106138.612
 6 China   156  2015   25-29 129085.568
 7 China   156  2015   30-34  99374.546
 8 China   156  2015   35-39  95219.098
 9 China   156  2015   40-44 119092.005
10 China   156  2015   45-49 123525.404
# ... with 12 more rows
```

Arranging Data

- The age groups need to be in a more logical order, for example how they appear in the data frame:

```
> unique(df5$age_grp)
 [1] "0-4"   "5-9"   "10-14" "15-19" "20-24" "25-29" "30-34" "35-39" "40-44" "45-49
[11] "50-54" "55-59" "60-64" "65-69" "70-74" "75-79" "80+"   "80-84" "85-89" "90-94
[21] "95-99" "100+"
```

Arranging Data

- We can use mutate to change the age_grp column to a factor as:

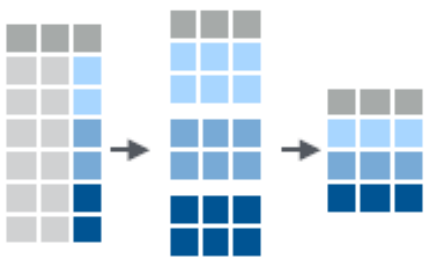
```
> df5 <- mutate(df5, age_grp = factor(age_grp, levels = unique(age_grp)))
>
> # now arrange works well...
> df5 %>%
+   filter(name == "China", year == 2015) %>%
+   arrange(age_grp)
# A tibble: 22 x 5
    name  code  year age_grp       pop
   <chr> <dbl> <dbl>  <fctr>     <dbl>
 1 China   156  2015     0-4  83185.94
 2 China   156  2015     5-9  78637.40
 3 China   156  2015   10-14  75291.91
 4 China   156  2015   15-19  78930.13
 5 China   156  2015   20-24 106138.61
 6 China   156  2015   25-29 129085.57
 7 China   156  2015   30-34  99374.55
 8 China   156  2015   35-39  95219.10
 9 China   156  2015   40-44 119092.01
10 China   156  2015   45-49 123525.40
# ... with 12 more rows
```

Exercise 4 (ex44.R)

```
# 0.  a) Check your working directory is in the course folder

#     b) Souce the solution file for ex43.R

##
##
##
# 1. Modify d1 to add a `location_type` variable with value 2 if region and 1 if cou

# 2. Display the observations in d1 using pipes
#    a) from countries in 2015
#    b) sorted in descending order based on old age dependency ratio




# 3. Modify d2 to
#    a) add location_type variable as above
#    b) change population from thousands to integer (Hint: * 1000)
#    c) change age to factor with levels in age order (Hint: use the unique() functi



# 4. Display the levels of the age variable
```

Grouped

- Group data adds some extra information to the tibble which is helpful when we want to do some extra calculations.
- Changes unit of analysis from the complete data set to groups.
- We can use it with
  - `mutate()` to create new columns to create summary variables
  - `summarise()` to create a new data frame of summarizing each group.



group_by(.data, ..., add = FALSE)
Returns copy of table grouped by …

ungroup(x, ...)
Returns ungrouped copy of table.

Grouped Mutate

```
> # original data
> df2
# A tibble: 2,990 x 4
             name  code    period      tfr
            <chr> <dbl>     <chr>    <dbl>
 1          WORLD   900 1950-1955 4.961571
 2         AFRICA   903 1950-1955 6.599479
 3 Eastern Africa   910 1950-1955 7.012207
 4        Burundi   108 1950-1955 6.801000
 5        Comoros   174 1950-1955 6.000000
 6       Djibouti   262 1950-1955 6.312000
 7        Eritrea   232 1950-1955 6.965000
 8       Ethiopia   231 1950-1955 7.169600
 9          Kenya   404 1950-1955 7.481000
10     Madagascar   450 1950-1955 7.300000
# ... with 2,980 more rows
```

Grouped Mutate

```
> # add columns for the all time mean tfr, max tfr and lagged tfr
> df2 %>%
+   filter(name %in% c("China", "Japan")) %>%
+   group_by(name) %>%
+   mutate(mean_tfr = mean(tfr),
+          max_tfr = max(tfr),
+          lag_tfr = lag(tfr))
# A tibble: 26 x 7
# Groups:   name [2]
    name  code    period   tfr mean_tfr max_tfr lag_tfr
   <chr> <dbl>     <chr> <dbl>    <dbl>   <dbl>   <dbl>
 1 China   156 1950-1955 6.107 3.478985   6.300      NA
 2 Japan   392 1950-1955 2.999 1.801446   2.999      NA
 3 China   156 1955-1960 5.476 3.478985   6.300   6.107
 4 Japan   392 1955-1960 2.155 1.801446   2.999   2.999
 5 China   156 1960-1965 6.150 3.478985   6.300   5.476
 6 Japan   392 1960-1965 1.986 1.801446   2.999   2.155
 7 China   156 1965-1970 6.300 3.478985   6.300   6.150
 8 Japan   392 1965-1970 2.020 1.801446   2.999   1.986
 9 China   156 1970-1975 4.850 3.478985   6.300   6.300
10 Japan   392 1970-1975 2.134 1.801446   2.999   2.020
# ... with 16 more rows
```

Grouped Summary

- The `summarise()` (or `summarize()`) function collapses a data frame to a single row:
- Paired with `group_by()` it becomes very powerful.

summarise() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

Counts

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NA's

Location

mean() - mean, also **mean(!is.na())**
median() - median

Logicals

mean() - Proportion of TRUE's
sum() - # of TRUE's

Position/Order

dplyr::**first()** - first value
dplyr::**last()** - last value
dplyr::**nth()** - value in nth location of vector

Grouped Summary

```
> # original data
> df5
# A tibble: 70,840 x 5
    name  code  year age_grp      pop
   <chr> <dbl> <dbl>  <fctr>    <dbl>
 1 WORLD   900  1950     0-4 337431.9
 2 WORLD   900  1955     0-4 402845.0
 3 WORLD   900  1960     0-4 430565.1
 4 WORLD   900  1965     0-4 477798.0
 5 WORLD   900  1970     0-4 522640.9
 6 WORLD   900  1975     0-4 543224.9
 7 WORLD   900  1980     0-4 546894.2
 8 WORLD   900  1985     0-4 589594.1
 9 WORLD   900  1990     0-4 643186.9
10 WORLD   900  1995     0-4 625929.7
# ... with 70,830 more rows
```

Grouped Summary

```
> # summarise over all age groups for China and Japan
> df5 %>%
+   filter(year == 2015, name %in% c("China", "Japan"))  %>%
+   summarise(pop_sum = sum(pop, na.rm = TRUE))
# A tibble: 1 x 1
  pop_sum
    <dbl>
1 1502622
>
> # summarise over all age groups for each of China and Japan
> df5 %>%
+   filter(year == 2015, name %in% c("China", "Japan"))  %>%
+   drop_na() %>%
+   group_by(name) %>%
+   summarise(pop_sum = sum(pop),
+             n = n(),
+             under5 = first(pop))
# A tibble: 2 x 4
   name    pop_sum     n    under5
  <chr>     <dbl> <int>     <dbl>
1 China 1376048.9    21 83185.944
2 Japan  126573.5    21  5269.038
```

Multiple Groups

- Produce group summaries on multiple sub groups.

```
> df5 %>%
+   filter(code < 900) %>%
+   group_by(name, year) %>%
+   summarise(pop_sum = sum(pop, na.rm = TRUE))
# A tibble: 2,814 x 3
# Groups:   name [?]
          name  year   pop_sum
         <chr> <dbl>     <dbl>
 1 Afghanistan  1950  7752.118
 2 Afghanistan  1955  8270.024
 3 Afghanistan  1960  8994.793
 4 Afghanistan  1965  9935.358
 5 Afghanistan  1970 11121.097
 6 Afghanistan  1975 12582.954
 7 Afghanistan  1980 13211.412
 8 Afghanistan  1985 11630.498
 9 Afghanistan  1990 12067.570
10 Afghanistan  1995 16772.522
# ... with 2,804 more rows
```

Exercise 5 (ex45.R)

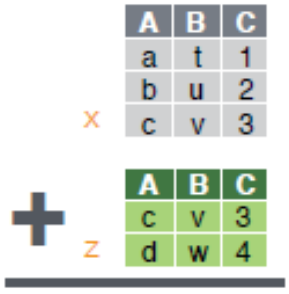
```
# 0.  a) Check your working directory is in the course folder

#     b) Souce the solution file for ex44.R

##
##
##
# 1. Display the observations in d1 using pipes
#    a) from Philippines, Indonesia and Malaysia
#    b) grouped by name
#    c) with a new max_odr variable for the maximum age dependency ratio in each ove
#    d) with a new lag_odr variable for the old age dependency ratio five years prev
#    e) with a new lead_odr variable for the old age dependency ratio five years ahe
#       (Hint: the lead() function does the opposite to the lag() function)









# 2. Create a new data set d6 from d2 that
#    a) filters to countries (Hint: location_type 1)
#    b) groups by name and year
#    c) summarises the population in each area and year (over age groups)
```

Relational Data

- Data analysis rarely involves only a single table of data.
- Typically you want to combine data to answer the questions that you are interested in.
- Data joins can involve
  - Joining data with different variables (columns)
  - Joining data with different observations (rows)

Combine Cases

- The `bind_rows()` will fill `NA` values for columns that are missing from one of the data sets.
- The `intersect()`, `union()` and `setdiff()` functions look at matching rules for observations (rows).



Use **bind_rows()** to paste tables below each other as they are.

DF	A	B	C
x	a	t	1
x	b	u	2
x	c	v	3
z	c	v	3
z	d	w	4

bind_rows(..., .id = NULL)
Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured)

A	B	C
c	v	3

intersect(x, y, ...)
Rows that appear in both x and z.

A	B	C
a	t	1
b	u	2

setdiff(x, y, ...)
Rows that appear in x but not z.

A	B	C
a	t	1
b	u	2
c	v	3
d	w	4

union(x, y, ...)
Rows that appear in x or z. (Duplicates removed). **union_all()** retains duplicates.

Combine Cases

```
> # creating some demonstration data frames
> d1 <- data_frame(A = c("a", "b", "c"), B = c("t", "u", "v"), C = 1:3)
> d2 <- data_frame(A = c("a", "b", "d"), B = c("t", "u", "w"), D = 3:1)
> d3 <- data_frame(A = c("c", "d"), B = c("v", "w"), C = 3:4)
> d4 <- rename(d2, AA = A, BB = B)
> d5 <- data_frame(A = c("a", "c", "c", "c"), D = 4:1)
```

Combine Cases

```
> d1
# A tibble: 3 x 3
      A     B     C
  <chr> <chr> <int>
1     a     t     1
2     b     u     2
3     c     v     3
> d2
# A tibble: 3 x 3
      A     B     D
  <chr> <chr> <int>
1     a     t     3
2     b     u     2
3     d     w     1
> bind_rows(d1, d2)
# A tibble: 6 x 4
      A     B     C     D
  <chr> <chr> <int> <int>
1     a     t     1    NA
2     b     u     2    NA
3     c     v     3    NA
4     a     t    NA     3
5     b     u    NA     2
6     d     w    NA     1
```

Combine Cases

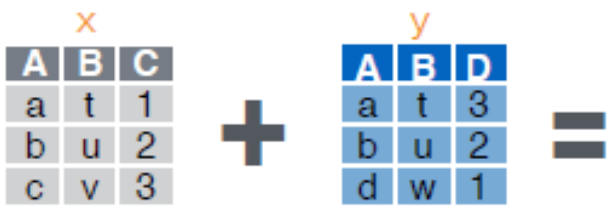
```
> d1
# A tibble: 3 x 3
      A     B     C
  <chr> <chr> <int>
1     a     t     1
2     b     u     2
3     c     v     3
> d3
# A tibble: 2 x 3
      A     B     C
  <chr> <chr> <int>
1     c     v     3
2     d     w     4
> bind_rows(d1, d3)
# A tibble: 5 x 3
      A     B     C
  <chr> <chr> <int>
1     a     t     1
2     b     u     2
3     c     v     3
4     c     v     3
5     d     w     4
```

Exercise 6 (ex46.R)

```
# 0.  a) Check your working directory is in the course folder

#     b) Souce the solution file for ex45.R

##
##
##
# 1. Using WPP2015_POP_F07_1_POPULATION_BY_AGE_BOTH_SEXES.xls on future population s
#    a) the 1st sheet and save as r1
#    b) the 2st sheet and save as r2
#    c) print r1
#    d) print r2




# 2. Bind the rows of d8 and d9 and call the result d10

# 3. a) Q: How many rows in r1
#       A:
#    b) Q: How many rows in r2
#       A:
#    b) Q: How many rows in r2
#       A:
```

Combine Variables

- The `bind_cols()` function is the simplest way to combine variables
  - The order of observations (rows) in both data MUST BE THE SAME
  - Can be rather dangerous, better to use joins (next)



Use **bind_cols()** to paste tables beside each other as they are.

A	B	C	A	B	D
a	t	1	a	t	3
b	u	2	b	u	2
c	v	3	d	w	1

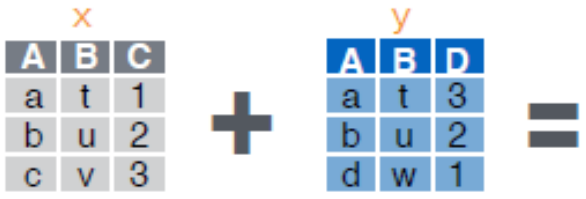
bind_cols(...)

Returns tables placed side by side as a single table.

BE SURE THAT ROWS ALIGN.

Combine Variables

- When combining variables we need to know:
  - Observations in which variables match up (link the data sets together)
  - These are called "key" variables, similar to what we discussed when tidying the data.



A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA

left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)

Join matching values from y to x.

A	B.x	C	B.y	D
a	t	1	t	3
b	u	2	u	2
c	v	3	NA	NA

Use **by = c("col1", "col2")** to specify the column(s) to match on.

left_join(x, y, by = "A")

A.x	B.x	C	A.y	B.y
a	t	1	d	w
b	u	2	b	u
c	v	3	a	t

Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.

left_join(x, y, by = c("C" = "D"))

Combine Variables

```
> d1
# A tibble: 3 x 3
      A     B     C
  <chr> <chr> <int>
1     a     t     1
2     b     u     2
3     c     v     3
> d2
# A tibble: 3 x 3
      A     B     D
  <chr> <chr> <int>
1     a     t     3
2     b     u     2
3     d     w     1
> # by default joins by common names
> left_join(d1, d2)
Joining, by = c("A", "B")
# A tibble: 3 x 4
      A     B     C     D
  <chr> <chr> <int> <int>
1     a     t     1     3
2     b     u     2     2
3     c     v     3    NA
```

Combine Variables

```
> d1
# A tibble: 3 x 3
      A     B     C
  <chr> <chr> <int>
1     a     t     1
2     b     u     2
3     c     v     3
> d2
# A tibble: 3 x 3
      A     B     D
  <chr> <chr> <int>
1     a     t     3
2     b     u     2
3     d     w     1
> # we can make this explicit
> left_join(d1, d2, by = c("A" = "A", "B" = "B"))
# A tibble: 3 x 4
      A     B     C     D
  <chr> <chr> <int> <int>
1     a     t     1     3
2     b     u     2     2
3     c     v     3    NA
```

Combine Variables

```
> d1
# A tibble: 3 x 3
      A     B     C
  <chr> <chr> <int>
1     a     t     1
2     b     u     2
3     c     v     3
> d4
# A tibble: 3 x 3
     AA    BB     D
  <chr> <chr> <int>
1     a     t     3
2     b     u     2
3     d     w     1
> # can join when names are not the same using the `by` argument
> left_join(d1, d4, by = c("A" = "AA", "B" = "BB"))
# A tibble: 3 x 4
      A     B     C     D
  <chr> <chr> <int> <int>
1     a     t     1     3
2     b     u     2     2
3     c     v     3    NA
```

Combine Variables

- When one table has duplicate observations in the key variable of `x` or `y`
  - Multiple matches will increase the number of rows
- When using `left_join()` this might not always be your intention.
  - Need to be careful with big data sets and check the number of rows `nrow()` of the new data set after the join and the original `x`.

Combine Variables

```
> d1
# A tibble: 3 x 3
      A     B     C
  <chr> <chr> <int>
1     a     t     1
2     b     u     2
3     c     v     3
> d5
# A tibble: 4 x 2
      A     D
  <chr> <int>
1     a     4
2     c     3
3     c     2
4     c     1
> left_join(d1, d5, by = "A")
# A tibble: 5 x 4
      A     B     C     D
  <chr> <chr> <int> <int>
1     a     t     1     4
2     b     u     2    NA
3     c     v     3     3
4     c     v     3     2
5     c     v     3     1
```

Combine Variables

- Usually `left_join()` function does the job
  - `right_join()` function switches the order of `x` and `y`. Mainly used with pipes.
  - `inner_join()` function is useful if you want to remove `NA` at the same time.
  - `full_join()` function is useful if you want to consider all possible combinations

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA

left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)
Join matching values from y to x.

A	B	C	D
a	t	1	3
b	u	2	2
d	w	NA	1

right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...)
Join matching values from x to y.

A	B	C	D
a	t	1	3
b	u	2	2

inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...)
Join data. Retain only rows with matches.

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA
d	w	NA	1

full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)
Join data. Retain all values, all rows.

Exercise 7 (ex47.R)

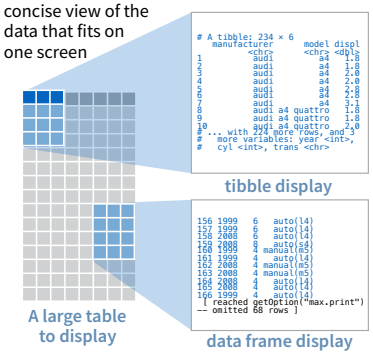
```
# 0.  a) Check your working directory is in the course folder

#     b) Souce the solution file for ex45.R

##
##
##
# 1. Create a new data set j1 that is a copy of d6

# 2. Create a new data set j2 that is a copy of d1, with only the code, year and odr

# 3. Modify j1 to attach the old age dependncy ratio obseravtions in j2 for the corr

# 4. View wpp.xlsx in the data folder. Then
#    a) read into R the 3rd sheet on sex ratios at birth and call it j3
#    b) modify j0 to attach the sex ratio obseravtions in j3 for the corresponding c


# 5. From the wpp.xlsx data.
#    a) read in the 4th sheet on total fertility rate and call it j4
#    b) read in the 5th sheet on infant mortality rates and call it j5
#    c) read in the 6th sheet on life expectancy and call it j6
#    d) read in the 7th sheet on net migration and call it j7
```

RStudio Cheatsheets

Tibbles - an enhanced data frame

The **tibble** package provides a new S3 class for storing tabular data, the tibble. Tibbles inherit the data frame class, but improve three behaviors:

- **Subsetting** - [always returns a new tibble, [[and $ always return a vector.
- **No partial matching** - You must use full column names when subsetting
- **Display** - When you print a tibble, R provides a concise view of the data that fits on one screen



- Control the default appearance with options:

  options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf**)**

- View full data set with **View()** or **glimpse()**
- Revert to data frame with **as.data.frame()**

CONSTRUCT A TIBBLE IN TWO WAYS

tibble(…**)**
Construct by columns.
tibble(x = 1:3, y = c("a", "b", "c"))

tribble(…**)**
Construct by rows.
tribble(~x, ~y, 1, "a", 2, "b", 3, "c")

Both make this tibble

```
A tibble: 3 × 2
      x     y
  <int> <chr>
1     1     a
2     2     b
3     3     c
```

as_tibble(x, …**)** Convert data frame to tibble.

enframe(x, name = "name", value = "value"**)**
Convert named vector to a tibble

is_tibble(x**)** Test whether x is a tibble.

Tidy Data with Tidyr

Tidy data is a way to organize tabular data. It provides a consistent data structure across packages.

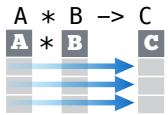
A table is tidy if:

Each **variable** is in its own **column**

Each **observation**, or **case**, is in its own **row**

Tidy data:

Makes variables easy to access as vectors



Preserves cases during vectorized operations

Reshape Data - change the layout of values in a table

Use **gather()** and **spread()** to reorganize the values of a table into a new layout.

gather(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor_key = FALSE**)**

Gather moves column names into a **key** column, gathering the column values into a single **value** column.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

gather(table4a, `1999`, `2000`, key = "year", value = "cases")

spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL**)**

Spread moves the unique values of a **key** column into the column names, spreading the values of a **value** column across the new columns.

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

key value

spread(table2, type, count)

Handle Missing Values

drop_na(data, ...**)**
Drop rows containing NA's in … columns.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
D	3

drop_na(x, x2)

fill(data, ..., .direction = c("down", "up"**)**
Fill in NA's in … columns with most recent non-NA values.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	1
C	1
D	3
E	3

fill(x, x2)

replace_na(data, replace = list(), ...**)**
Replace NA's by column.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	2
C	2
D	3
E	2

replace_na(x, list(x2 = 2))

Expand Tables - quickly create tables with combinations of values

complete(data, ..., fill = list()**)**
Adds to the data missing combinations of the values of the variables listed in …
complete(mtcars, cyl, gear, carb)

expand(data, ...**)**
Create new tibble with all possible combinations of the values of the variables listed in …
expand(mtcars, cyl, gear, carb)

Split Cells

Use these functions to split or combine cells into individual, isolated values.

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ...**)**

Separate each cell in a column to make several columns.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174
C	1999	212K	1T
C	2000	213K	1T

separate(table3, rate, into = c("cases", "pop"))

separate_rows(data, ..., sep = "[^[:alnum:].]+", convert = FALSE**)**

Separate each cell in a column to make several rows. Also **separate_rows_()**.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M
C	1999	212K
C	1999	1T
C	2000	213K
C	2000	1T

separate_rows(table3, rate)

unite(data, col, ..., sep = "_", remove = TRUE**)**

Collapse cells across several columns to make a single column.

table5

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

→

country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

unite(table5, century, year, col = "year", sep = "")

RStudio Cheatsheets

Data Transformation with dplyr : : CHEAT SHEET

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column**

&

Each **observation**, or **case**, is in its own **row**

pipes

x %>% f(y) becomes **f(x, y)**

Summarise Cases

These apply **summary functions** to columns to create a new table. Summary functions take vectors as input and return one value (see back).

summarise(.data, …)
Compute table of summaries. Also **summarise_**().
summarise(mtcars, avg = mean(mpg))

count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
count(iris, Species)

VARIATIONS

summarise_all() - Apply funs to every column.
summarise_at() - Apply funs to specific columns.
summarise_if() - Apply funs to all cols of one type.

Group Cases

Use **group_by()** to created a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

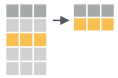
group_by(.data, ..., add = FALSE)
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

ungroup(x, …)
Returns ungrouped copy of table.
ungroup(g_iris)

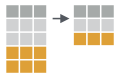
Manipulate Cases

EXTRACT CASES

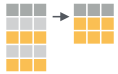
Row functions return a subset of rows as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.



filter(.data, …) Extract rows that meet logical criteria. Also **filter_()**. *filter(iris, Sepal.Length > 7)*

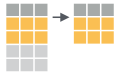


distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values. Also **distinct_()**. *distinct(iris, Species)*



sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows. *sample_n(iris, 10, replace = TRUE)*



slice(.data, …) Select rows by position. Also **slice_()**. *slice(iris, 10:15)*

top_n(x, n, wt) Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

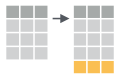
<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

See **?base::logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, …)
Order rows by values of a column (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES



add_row(.data, ..., .before = NULL, .after = NULL)
Add one or more rows to a table.
add_row(faithful, eruptions = 1, waiting = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.

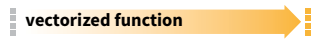
select(.data, …)
Extract columns by name. Also **select_if()**
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

contains(match)	**num_range**(prefix, range)	**:**, e.g. mpg:cyl
ends_with(match)	**one_of**(…)	**-**, e.g, -Species
matches(match)	**starts_with**(match)	

MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).



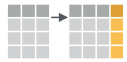
mutate(.data, …)
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

transmute(.data, …)
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

mutate_all(.tbl, .funs, …) Apply funs to every column. Use with **funs()**.
mutate_all(faithful, funs(log(.), log2(.)))

mutate_at(.tbl, .cols, .funs, …) Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for select().
mutate_at(iris, vars(-Species), funs(log(.)))

mutate_if(.tbl, .predicate, .funs, …)
Apply funs to all columns of one type.
Use with **funs()**.
mutate_if(iris, is.numeric, funs(log(.)))



add_column(.data, ..., .before = NULL, .after = NULL) Add new column(s).
add_column(mtcars, new = 1:32)

rename(.data, …) Rename columns.
rename(iris, Length = Sepal.Length)

RStudio

RStudio Cheatsheets

Vectorized Functions

TO USE WITH MUTATE ()

mutate() and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

vectorized function

OFFSETS

dplyr::**lag()** - Offset elements by 1
dplyr::**lead()** - Offset elements by -1

CUMULATIVE AGGREGATES

dplyr::**cumall()** - Cumulative all()
dplyr::**cumany()** - Cumulative any()
cummax() - Cumulative max()
dplyr::**cummean()** - Cumulative mean()
cummin() - Cumulative min()
cumprod() - Cumulative prod()
cumsum() - Cumulative sum()

RANKINGS

dplyr::**cume_dist()** - Proportion of all values <=
dplyr::**dense_rank()** - rank with ties = min, no gaps
dplyr::**min_rank()** - rank with ties = min
dplyr::**ntile()** - bins into n bins
dplyr::**percent_rank()** - min_rank scaled to [0,1]
dplyr::**row_number()** - rank with ties = "first"

MATH

+, - , *, /, ^, %/%, %% - arithmetic ops
log(), log2(), log10() - logs
<, <=, >, >=, !=, == - logical comparisons

MISC

dplyr::**between()** - x >= left & x <= right
dplyr::**case_when()** - multi-case if_else()
dplyr::**coalesce()** - first non-NA values by element across a set of vectors
dplyr::**if_else()** - element-wise if() + else()
dplyr::**na_if()** - replace specific values with NA
pmax() - element-wise max()
pmin() - element-wise min()
dplyr::**recode()** - Vectorized switch()
dplyr::**recode_factor()** - Vectorized switch() for factors

Summary Functions

TO USE WITH SUMMARISE ()

summarise() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

summary function

COUNTS

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NA's

LOCATION

mean() - mean, also **mean(!is.na())**
median() - median

LOGICALS

mean() - Proportion of TRUE's
sum() - # of TRUE's

POSITION/ORDER

dplyr::**first()** - first value
dplyr::last() - last value
dplyr::nth() - value in nth location of vector

RANK

quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

IQR() - Inter-Quartile Range
mad() - mean absolute deviation
sd() - standard deviation
var() - variance

Row Names

Tidy data does not use rownames, which store a variable outside of the columns. To work with the rownames, first move them into a column.

rownames_to_column()
Move row names into col.
a <- rownames_to_column(iris, var = "C")

column_to_rownames()
Move col in row names.
column_to_rownames(a, var = "C")

Also has **_rownames()**, **remove_rownames()**

Combine Tables

COMBINE VARIABLES

Use **bind_cols()** to paste tables beside each other as they are.

bind_cols(…) Returns tables placed side by side as a single table.
BE SURE THAT ROWS ALIGN.

Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…)
Join matching values from y to x.

right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…)
Join matching values from x to y.

inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…)
Join data. Retain only rows with matches.

full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…)
Join data. Retain all values, all rows.

Use **by = c("col1", "col2")** to specify the column(s) to match on.
left_join(x, y, by = "A")

Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.
left_join(x, y, by = c("C" = "D"))

Use **suffix** to specify suffix to give to duplicate column names.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

COMBINE CASES

Use **bind_rows()** to paste tables below each other as they are.

bind_rows(…, .id = NULL)
Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured)

intersect(x, y, …)
Rows that appear in both x and z.

setdiff(x, y, …)
Rows that appear in x but not z.

union(x, y, …)
Rows that appear in x or z. (Duplicates removed). union_all() retains duplicates.

Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

EXTRACT ROWS

Use a "**Filtering Join**" to filter one table against the rows of another.

semi_join(x, y, by = NULL, …)
Return rows of x that have a match in y.
USEFUL TO SEE WHAT WILL BE JOINED.

anti_join(x, y, by = NULL, …)
Return rows of x that do not have a match in y.
USEFUL TO SEE WHAT WILL NOT BE JOINED.

RStudio

Optional Assignment 4 (assign4.R)

```
##
## Assignment 4
##

# Take a look at the AnnualbyProvince.xlsx and china.csv files in the assignment fol
# You are going to write code to turn the data from AnnualbyProvince.xlsx to china.c
#
# 0) set your working directory to the folder of AnnualbyProvince.xlsx and load the




##
##
##
# 1) read in the data in the province sheet and save as d0

# 2) select the name, region and code column

# 3) read in the data from AnnualbyProvince.xlsx and save in objects d1, d2, d3 and
#    Hint: set skip to miss first few rows and n_max to only read for 31 rows after
#          to avoiding the rows with the notes and data comments
```