

R for Stata Users

02: Data Manipulation

Purushottam Mohanty

05 June 2021

overview

Data manipulation or data cleaning is an important aspect of any project and it is important to be well acquainted with all the tools provided by R to facilitate an easy transition from Stata to R.

While Stata has specific commands for specific data cleaning operations, R is more versatile and often the same operation can be performed in multiple ways through different packages.

In this slide deck I use `dplyr` and `tidyr` packages to perform data manipulation operations. The same tasks can also be performed using `baseR` functions but they are often more complicated and cumbersome.

I also provide equivalent commands in Stata where possible to help easily understand R functions.

tidyverse

The `tidyverse` package in R is a composite of many different data manipulation, functional programming and data visualization packages. Check documentation about `tidyverse`, `dplyr` and `tidyr` for more information.

```
# only loads the primary tidyverse packages
```

```
library(tidyverse)
```

```
tidyverse_packages()
```

```
## [1] "broom"          "cli"            "crayon"         "dbplyr"
## [5] "dplyr"          "dtplyr"         "forcats"        "googledrive"
## [9] "googlesheets4" "ggplot2"        "haven"          "hms"
## [13] "httr"           "jsonlite"       "lubridate"      "magrittr"
## [17] "modelr"         "pillar"         "purrr"          "readr"
## [21] "readxl"         "reprex"         "rlang"          "rstudioapi"
## [25] "rvest"          "stringr"        "tibble"         "tidyr"
## [29] "xml2"           "tidyverse"
```

Packages like `ggplot2` for data visualization, `forcats` for handling categorical data and `stringr` for text manipulation enable so many possibilities. Packages within tidyverse can be separately loaded as well.

pipe %>% operator

Many different packages in R, including `tidyr` and `dplyr` follow the pipe `%>%` operator syntax which makes way for clean looking code and saves considerable time by not having to specify the dataframe name everytime.

both lines are equivalent

```
df %>% filter(!continent == 'Europe') %>% group_by(continent, year) %>%  
  summarize(mean_gdppc = mean(gdpPercap))  
summarise(group_by(filter(df, !continent == 'Europe'), continent, year), mean_gdppc =
```

The first line can be read as, specifying the dataframe, filtering the rows and then grouping based on column names and the summarizing the `gdpPercap` variable. As you can see the first line is easier to read and logical in nature. With complicated and lengthy code, `%>%` operator becomes extremely handy.

dplyr | tidyr

create new columns (1/n)

In R, new columns can be created using `dplyr::mutate()` function.

```
df %>%  
  mutate(pop_mn = pop / 1000000)
```

Note that `mutate()` creates a new column if dataframe `df` doesn't have a column with the specified namespace (i.e. `pop_mn`) or overwrites the existing column. So, `mutate()` is a substitute for Stata commands `generate` and `replace` depending on the namespace provided.

```
# Stata equivalent  
generate pop_mn = pop / 1000000  
replace pop_mn = pop / 1000000
```

create new column (2/n)

The original dataframe imported from the `gapminder` package.

```
df
```

```
## # A tibble: 1,704 x 6
##   country      continent  year lifeExp      pop gdpPercap
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.
## 2 Afghanistan Asia      1957   30.3  9240934    821.
## 3 Afghanistan Asia      1962   32.0 10267083    853.
## 4 Afghanistan Asia      1967   34.0 11537966    836.
## 5 Afghanistan Asia      1972   36.1 13079460    740.
## 6 Afghanistan Asia      1977   38.4 14880372    786.
## 7 Afghanistan Asia      1982   39.9 12881816    978.
## 8 Afghanistan Asia      1987   40.8 13867957    852.
## 9 Afghanistan Asia      1992   41.7 16317921    649.
## 10 Afghanistan Asia      1997   41.8 22227415    635.
## # ... with 1,694 more rows
```

create new column (3/n)

New column `pop_mn` has been created.

```
df %>%  
  mutate(pop_mn = pop / 1000000)
```

```
## # A tibble: 1,704 x 7  
##   country      continent  year lifeExp      pop gdpPercap pop_mn  
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>  <dbl>  
## 1 Afghanistan Asia      1952   28.8  8425333    779.    8.43  
## 2 Afghanistan Asia      1957   30.3  9240934    821.    9.24  
## 3 Afghanistan Asia      1962   32.0 10267083    853.   10.3  
## 4 Afghanistan Asia      1967   34.0 11537966    836.   11.5  
## 5 Afghanistan Asia      1972   36.1 13079460    740.   13.1  
## 6 Afghanistan Asia      1977   38.4 14880372    786.   14.9  
## 7 Afghanistan Asia      1982   39.9 12881816    978.   12.9  
## 8 Afghanistan Asia      1987   40.8 13867957    852.   13.9  
## 9 Afghanistan Asia      1992   41.7 16317921    649.   16.3  
## 10 Afghanistan Asia      1997   41.8 22227415    635.   22.2  
## # ... with 1,694 more rows
```


create new column (4/n)

Let's overwrite `pop_mn` where `pop_mn` is log of population. Note how the old `pop_mn` column is overwritten with log population values. Also, see how the `%>%` function allows us to perform multiple operations on the same dataframe.

```
df %>%  
  mutate(pop_mn = pop / 1000000) %>%  
  mutate(pop_mn = log(pop))
```

```
## # A tibble: 1,704 x 7
```

```
##   country      continent  year lifeExp      pop gdpPercap pop_mn  
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>  <dbl>  
## 1 Afghanistan Asia      1952   28.8  8425333    779.   15.9  
## 2 Afghanistan Asia      1957   30.3  9240934    821.   16.0  
## 3 Afghanistan Asia      1962   32.0 10267083    853.   16.1  
## 4 Afghanistan Asia      1967   34.0 11537966    836.   16.3  
## 5 Afghanistan Asia      1972   36.1 13079460    740.   16.4  
## 6 Afghanistan Asia      1977   38.4 14880372    786.   16.5  
## 7 Afghanistan Asia      1982   39.9 12881816    978.   16.4  
## 8 Afghanistan Asia      1987   40.8 13867957    852.   16.4  
## 9 Afghanistan Asia      1992   41.7 16317921    649.   16.6  
## 10 Afghanistan Asia      1997   41.8 22227415    635.   16.9  
## # ... with 1,694 more rows
```

filter rows (1/n)

In R rows can be filtered using `dplyr::filter()` function.

```
df %>%  
  filter(continent = "Europe")
```

This keeps only those rows for which continent is Europe. Similarly, for keeping all rows outside continent Europe one can use the "!=" logical operation. Multiple conditions can also be specified using,

```
df %>%  
  filter(continent %in% c("Europe", "Africa"))  
df %>%  
  filter(continent = "Europe" | continent = "Africa")
```

Both lines above are equivalent. Note how the `%in%` syntax makes code much more concise and readable.

```
# Stata equivalent  
keep if continent = "Europe"  
keep if continent = "Europe" | continent = "Africa"
```

filter rows (2/n)

Only keeps the rows where continent is Europe.

```
df %>%  
  filter(continent = "Europe")
```

```
## # A tibble: 360 x 6  
##   country continent  year lifeExp      pop gdpPercap  
##   <fct>    <fct>    <int>  <dbl>   <int>    <dbl>  
## 1 Albania Europe    1952   55.2 1282697   1601.  
## 2 Albania Europe    1957   59.3 1476505   1942.  
## 3 Albania Europe    1962   64.8 1728137   2313.  
## 4 Albania Europe    1967   66.2 1984060   2760.  
## 5 Albania Europe    1972   67.7 2263554   3313.  
## 6 Albania Europe    1977   68.9 2509048   3533.  
## 7 Albania Europe    1982   70.4 2780097   3631.  
## 8 Albania Europe    1987   72    3075321   3739.  
## 9 Albania Europe    1992   71.6 3326498   2497.  
## 10 Albania Europe    1997   73.0 3428038   3193.  
## # ... with 350 more rows
```

filter rows (3/n)

Keeps all rows where continent is either Europe or Africa. (check no. of rows)

```
df %>%  
  filter(continent %in% c("Africa", "Europe"))
```

```
## # A tibble: 984 x 6  
##   country continent  year lifeExp      pop gdpPercap  
##   <fct>    <fct>    <int>  <dbl>   <int>    <dbl>  
## 1 Albania Europe    1952   55.2 1282697   1601.  
## 2 Albania Europe    1957   59.3 1476505   1942.  
## 3 Albania Europe    1962   64.8 1728137   2313.  
## 4 Albania Europe    1967   66.2 1984060   2760.  
## 5 Albania Europe    1972   67.7 2263554   3313.  
## 6 Albania Europe    1977   68.9 2509048   3533.  
## 7 Albania Europe    1982   70.4 2780097   3631.  
## 8 Albania Europe    1987   72    3075321   3739.  
## 9 Albania Europe    1992   71.6 3326498   2497.  
## 10 Albania Europe    1997   73.0 3428038   3193.  
## # ... with 974 more rows
```

filter columns (1/n)

In R, columns can be filtered using the `dplyr::select()` function.

```
df %>%  
  select(continent)  
df %>%  
  select(country, continent)
```

Similarly, columns can be dropped using a `-` sign before the column name

```
df %>%  
  select(-country, -continent)
```

Unlike Stata, for R both keeping and dropping columns is done using the same function. Also, one can use `select()` to order columns with or without dropping columns using the `select(columnA, columnB, everything())` syntax. The `everything()` function selects all columns not specified in `select()`.

```
# Stata equivalent  
keep country continent  
drop country continent
```

filter columns (2/n)

```
df %>%  
  dplyr::select(continent, country)
```

```
## # A tibble: 1,704 x 2  
##   continent country  
##   <fct>      <fct>  
## 1 Asia      Afghanistan  
## 2 Asia      Afghanistan  
## 3 Asia      Afghanistan  
## 4 Asia      Afghanistan  
## 5 Asia      Afghanistan  
## 6 Asia      Afghanistan  
## 7 Asia      Afghanistan  
## 8 Asia      Afghanistan  
## 9 Asia      Afghanistan  
## 10 Asia     Afghanistan  
## # ... with 1,694 more rows
```

filter columns (3/n)

```
df %>%  
  dplyr::select(-continent, -country)
```

```
## # A tibble: 1,704 x 4  
##   year lifeExp      pop gdpPercap  
##   <int> <dbl>    <int>    <dbl>  
## 1  1952   28.8  8425333    779.  
## 2  1957   30.3  9240934    821.  
## 3  1962   32.0 10267083    853.  
## 4  1967   34.0 11537966    836.  
## 5  1972   36.1 13079460    740.  
## 6  1977   38.4 14880372    786.  
## 7  1982   39.9 12881816    978.  
## 8  1987   40.8 13867957    852.  
## 9  1992   41.7 16317921    649.  
## 10 1997   41.8 22227415    635.  
## # ... with 1,694 more rows
```

order rows (1/n)

In R, rows can be ordered in ascending or descending order using the `dplyr::arrange()` function.

```
df %>%  
  arrange(year, gdpPercap) # ascending order  
df %>%  
  arrange(desc(year), desc(gdpPercap)) # descending order
```

```
# Stata equivalent  
sort year gdpPercap  
gsort -year -gdpPercap # using gtools
```


order rows (2/n)

```
df %>%  
  arrange(year, gdpPercap) # ascending order
```

```
## # A tibble: 1,704 x 6  
##   country      continent  year lifeExp      pop gdpPercap  
##   <fct>      <fct>    <int>  <dbl>    <int>    <dbl>  
## 1 Lesotho      Africa    1952   42.1    748747    299.  
## 2 Guinea-Bissau Africa    1952   32.5    580653    300.  
## 3 Eritrea      Africa    1952   35.9    1438760   329.  
## 4 Myanmar      Asia      1952   36.3    20092996  331  
## 5 Burundi      Africa    1952   39.0    2445618   339.  
## 6 Ethiopia      Africa    1952   34.1    20860941  362.  
## 7 Cambodia      Asia      1952   39.4    4693836   368.  
## 8 Malawi        Africa    1952   36.3    2917802   369.  
## 9 Equatorial Guinea Africa    1952   34.5     216964   376.  
## 10 China         Asia      1952   44     556263527 400.  
## # ... with 1,694 more rows
```

order rows (3/n)

```
df %>%  
  arrange(desc(year), desc(gdpPercap)) # descending order
```

```
## # A tibble: 1,704 x 6  
##   country      continent  year lifeExp      pop gdpPercap  
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>  
## 1 Norway      Europe    2007   80.2    4627926   49357.  
## 2 Kuwait      Asia     2007   77.6    2505559   47307.  
## 3 Singapore   Asia     2007   80.0    4553009   47143.  
## 4 United States Americas  2007   78.2  301139947   42952.  
## 5 Ireland     Europe    2007   78.9    4109086   40676.  
## 6 Hong Kong, China Asia     2007   82.2    6980412   39725.  
## 7 Switzerland Europe    2007   81.7    7554661   37506.  
## 8 Netherlands Europe    2007   79.8   16570613   36798.  
## 9 Canada      Americas  2007   80.7   33390141   36319.  
## 10 Iceland    Europe    2007   81.8    301931    36181.  
## # ... with 1,694 more rows
```

distinct (1/n)

In R, duplicates can be removed from a dataframe using `dplyr::distinct()` function. The `.keep_all = T` option ensures that all columns are kept after removal of the duplicates.

```
df %>%  
  distinct(.keep_all = T)
```

Duplicates can also be removed from a particular column using,

```
df %>%  
  distinct(country, .keep_all = T)
```

The `distinct()` can also be used to view unique values for a column(s).

```
df %>%  
  distinct(country) # all countries present in df
```

```
# Stata equivalent  
duplicates drop # for all rows  
duplicates drop country, force # for a column  
duplicates report country
```

distinct (2/n)

```
df %>%  
  distinct(.keep_all = T)
```

```
## # A tibble: 1,704 x 6  
##   country      continent  year lifeExp      pop gdpPercap  
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>  
## 1 Afghanistan Asia      1952   28.8  8425333    779.  
## 2 Afghanistan Asia      1957   30.3  9240934    821.  
## 3 Afghanistan Asia      1962   32.0 10267083    853.  
## 4 Afghanistan Asia      1967   34.0 11537966    836.  
## 5 Afghanistan Asia      1972   36.1 13079460    740.  
## 6 Afghanistan Asia      1977   38.4 14880372    786.  
## 7 Afghanistan Asia      1982   39.9 12881816    978.  
## 8 Afghanistan Asia      1987   40.8 13867957    852.  
## 9 Afghanistan Asia      1992   41.7 16317921    649.  
## 10 Afghanistan Asia      1997   41.8 22227415    635.  
## # ... with 1,694 more rows
```

distinct (3/n)

```
df %>%  
  distinct(country, .keep_all = T)
```

```
## # A tibble: 142 x 6  
##   country      continent  year lifeExp      pop gdpPercap  
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>  
## 1 Afghanistan Asia      1952   28.8  8425333    779.  
## 2 Albania     Europe   1952   55.2  1282697   1601.  
## 3 Algeria     Africa   1952   43.1  9279525   2449.  
## 4 Angola      Africa   1952   30.0  4232095   3521.  
## 5 Argentina   Americas 1952   62.5  17876956   5911.  
## 6 Australia   Oceania   1952   69.1  8691212  10040.  
## 7 Austria     Europe    1952   66.8  6927772   6137.  
## 8 Bahrain     Asia      1952   50.9   120447   9867.  
## 9 Bangladesh  Asia      1952   37.5  46886859    684.  
## 10 Belgium    Europe    1952    68   8730405   8343.  
## # ... with 132 more rows
```

distinct (4/n)

```
df %>%  
  distinct(country)
```

```
## # A tibble: 142 x 1  
##   country  
##   <fct>  
## 1 Afghanistan  
## 2 Albania  
## 3 Algeria  
## 4 Angola  
## 5 Argentina  
## 6 Australia  
## 7 Austria  
## 8 Bahrain  
## 9 Bangladesh  
## 10 Belgium  
## # ... with 132 more rows
```

summarize (1/n)

Summarize operates with `mutate()` at the backend and creates a new dataframe with specified columns based on the statistics specified. Note that `summarize()` and `summarise()` are equivalent to each other and can be interchanged.

```
df %>%  
  summarize(mean_pop = mean(pop), median_gdppc = median(gdpPercap))
```

Summarise can be performed across multiple columns in combination with the `across()` function.

```
df %>%  
  summarize(across(c("pop", "gdpPercap"), mean))
```

```
# Stata equivalent  
collapse (mean) pop (median) gdpPercap
```

summarize (2/n)

```
df %>%  
  summarize(mean_pop = mean(pop), median_gdppc = median(gdpPercap))
```

```
## # A tibble: 1 x 2  
##   mean_pop median_gdppc  
##   <dbl>      <dbl>  
## 1 29601212.      3532.
```

```
df %>%  
  summarize(across(c("pop", "gdpPercap"), mean))
```

```
## # A tibble: 1 x 2  
##   pop gdpPercap  
##   <dbl>      <dbl>  
## 1 29601212.      7215.
```

```
# Stata equivalent
```

```
bysort continent year: egen mean_pop = mean(pop)  
bysort continent year: egen median_gdppc = median(gdpPercap)  
collapse (mean) mean_pop median_gdppc, by(continent year)
```


group operations (1/n)

Grouped row operations can be performed using the `dplyr::group_by()` function.

```
df %>%  
  group_by(continent, year) %>% # grouping variables  
  mutate(mean_pop = mean(pop)) # group wise operation to perform
```

The `dplyr` pipe operation implies that dataset is grouped as long as a separate `ungroup()` function is provided. It's a healthy practice to provide `ungroup()` function after the end of the grouped operation to avoid confusion.

```
df %>%  
  group_by(continent, year) %>% # grouping variables  
  mutate(mean_pop = mean(pop)) %>% # group wise operation to perform  
  ungroup() %>% # dataframe is now ungrouped  
  mutate(mean_gdppc = mean(gdpPercap)) # ungrouped operation
```

```
# Stata equivalent  
bysort continent year: egen mean_pop = mean(pop)  
egen mean_gdppc = mean(gdpPercap)
```

group operations (2/n)

```
df %>%  
  group_by(continent, year) %>% # grouping variables  
  summarize(mean_pop = mean(pop), mean_gdppc = mean(gdpPercap)) # group wise operation
```

```
## # A tibble: 60 x 4  
## # Groups:   continent [5]  
##   continent  year mean_pop mean_gdppc  
##   <fct>      <int>     <dbl>     <dbl>  
## 1 Africa    1952  4570010.    1253.  
## 2 Africa    1957  5093033.    1385.  
## 3 Africa    1962  5702247.    1598.  
## 4 Africa    1967  6447875.    2050.  
## 5 Africa    1972  7305376.    2340.  
## 6 Africa    1977  8328097.    2586.  
....
```

```
# Stata equivalent
```

```
bysort continent year: egen mean_pop = mean(pop)  
bysort continent year: egen mean_gdppc = mean(gdpPercap)  
collapse (mean) mean_pop mean_gdppc, by(continent year)
```

reshape - long to wide (1/n)

Dataframes can be transformed from long to wide using the `tidyr::pivot_wider()` function. Here's how the long dataframe looks.

```
df
```

```
## # A tibble: 1,704 x 6
##   country      continent  year lifeExp      pop gdpPercap
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.
## 2 Afghanistan Asia      1957   30.3  9240934    821.
## 3 Afghanistan Asia      1962   32.0 10267083    853.
## 4 Afghanistan Asia      1967   34.0 11537966    836.
## 5 Afghanistan Asia      1972   36.1 13079460    740.
## 6 Afghanistan Asia      1977   38.4 14880372    786.
## 7 Afghanistan Asia      1982   39.9 12881816    978.
## 8 Afghanistan Asia      1987   40.8 13867957    852.
## 9 Afghanistan Asia      1992   41.7 16317921    649.
## 10 Afghanistan Asia      1997   41.8 22227415    635.
## # ... with 1,694 more rows
```

reshape - long to wide (2/n)

Transforming from long to wide.

```
df_wide = df %>%  
  pivot_wider(names_from = year, values_from = c("lifeExp", "pop", "gdpPercap"))  
df_wide
```

```
## # A tibble: 142 x 38  
##   country      continent lifeExp_1952 lifeExp_1957 lifeExp_1962 lifeExp_1967  
##   <fct>        <fct>         <dbl>         <dbl>         <dbl>         <dbl>  
## 1 Afghanistan Asia           28.8           30.3           32.0           34.0  
## 2 Albania      Europe           55.2           59.3           64.8           66.2  
## 3 Algeria      Africa           43.1           45.7           48.3           51.4  
## 4 Angola       Africa           30.0           32.0           34             36.0  
## 5 Argentina    Americas         62.5           64.4           65.1           65.6  
## 6 Australia    Oceania          69.1           70.3           70.9           71.1  
## 7 Austria      Europe           66.8           67.5           69.5           70.1  
## ...
```

```
# Stata equivalent
```

```
reshape wide lifeExp pop gdpPercap, i(continent country) j(year)
```

reshape - wide to long (3/n)

Dataframes can be transformed from wide to long using the `tidyr::pivot_longer()` function. We can transform the dataframe created earlier by,

```
df_wide %>%  
  pivot_longer(cols = c(-continent, -country), names_to = c("type", "year"),  
               names_sep = "_", values_to = "values")
```

```
## # A tibble: 5,112 x 5  
##   country      continent type    year  values  
##   <fct>        <fct>    <chr>  <chr>  <dbl>  
## 1 Afghanistan Asia      lifeExp 1952    28.8  
## 2 Afghanistan Asia      lifeExp 1957    30.3  
## 3 Afghanistan Asia      lifeExp 1962    32.0  
## 4 Afghanistan Asia      lifeExp 1967    34.0  
## 5 Afghanistan Asia      lifeExp 1972    36.1  
## 6 Afghanistan Asia      lifeExp 1977    38.4  
## 7 Afghanistan Asia      lifeExp 1982    39.9  
## ....
```

```
# Stata equivalent
```

```
reshape long lifeExp pop gdpPercap, i(continent country) j(year)
```

reshape - wide to long (4/n)

You might have noticed that the long data doesn't look like the original long data. The original data was partially long, so we'll have to convert it back to wide.

```
df_wide %>% # wide data
  pivot_longer(cols = c(-continent, -country), names_to = c("type", "year"),
               names_sep = "_", values_to = "values") %>% # long data
  pivot_wider(names_from = "type", values_from = "values") # original data
```

```
## # A tibble: 1,704 x 6
##   country      continent year  lifeExp      pop gdpPercap
##   <fct>        <fct>    <chr>   <dbl>    <dbl>    <dbl>
## 1 Afghanistan Asia      1952    28.8  8425333    779.
## 2 Afghanistan Asia      1957    30.3  9240934    821.
## 3 Afghanistan Asia      1962    32.0 10267083    853.
## 4 Afghanistan Asia      1967    34.0 11537966    836.
## 5 Afghanistan Asia      1972    36.1 13079460    740.
## 6 Afghanistan Asia      1977    38.4 14880372    786.
## 7 Afghanistan Asia      1982    39.9 12881816    978.
## 8 Afghanistan Asia      1987    40.8 13867957    852.
## 9 Afghanistan Asia      1992    41.7 16317921    649.
## 10 Afghanistan Asia      1997    41.8 22227415    635.
## # ... with 1,694 more rows
```

merging dataframes (1/n)

Two dataframes can be joined using using keys with the `*_join()` suite of functions. Dataframe X and Y can be merged using,

1. `inner_join(X, Y, by = "COL_NAME")` - includes all rows of X and Y.
2. `left_join(X, Y, by = "COL_NAME")` - includes all rows of X
3. `right_join(X, Y, by = "COL_NAME")` - includes all rows of Y
4. `full_join(X, Y, by = "COL_NAME")` - includes all rows of X or Y

Stata equivalent

```
merge 1:1 COL_NAME using `Y`
```

```
merge m:1 COL_NAME using `Y`
```

```
merge 1:m COL_NAME using `Y`
```

```
merge m:m COL_NAME using `Y`
```

```
keep if _merge = 3 # includes all rows of X and Y
```

merging dataframes (2/n)

Let's use the approval ratings of US Presidents from `datasets::presidents`, which is pre-installed in R and merge it with the `gapminder::gapminder` data. The data is a quarterly time-series data so we perform some basic manipulation first.

```
df_presidents = data.frame(year = time(datasets::presidents),
                           rating = as.matrix(datasets::presidents)) %>%
  mutate(year = substr(year,1,4)) %>%
  group_by(year) %>%
  summarise(approval_rating = mean(rating, na.rm=T)) # rating by year
```

```
df_presidents
```

```
## # A tibble: 30 x 2
##   year approval_rating
##   <chr>          <dbl>
## 1 1945           81.3
## 2 1946           47
## 3 1947           51
## 4 1948           37.5
## 5 1949           58.5
## 6 1950           41.8
## 7 1951           28.8
```

```
....
```


merging dataframes (3/n)

```
df %>% mutate(year = as.character(year)) %>%  
  left_join(df_presidents, by = "year")
```

```
## # A tibble: 1,704 x 7  
##   country      continent year  lifeExp      pop gdpPercap approval_rating  
##   <fct>        <fct>    <chr>   <dbl>    <int>    <dbl>         <dbl>  
## 1 Afghanistan Asia      1952    28.8  8425333    779.         29.7  
## 2 Afghanistan Asia      1957    30.3  9240934    821.         65.2  
## 3 Afghanistan Asia      1962    32.0 10267083    853.         71.5  
## 4 Afghanistan Asia      1967    34.0 11537966    836.          45  
## 5 Afghanistan Asia      1972    36.1 13079460    740.          55  
## 6 Afghanistan Asia      1977    38.4 14880372    786.          NA  
## 7 Afghanistan Asia      1982    39.9 12881816    978.          NA  
....
```

Since the `datasets::presidents` only contained data from 1945 to 1974, merged values after 1974 are missing (coded as `NA`).

```
# Stata equivalent  
tostring year, replace  
merge 1:1 year using `df_presidents`  
keep if _merge = 1 | _merge = 2 # same as left_join()
```

appending rows

Dataframes in R can be appended using the `dplyr::bind_rows()` function. The dataframes are appended based on column names however unlike many other R functions, it allows appending dataframes even when dataframes have different number of columns. In this regard, it works almost exactly like Stata's `append` command.

```
df %>%  
  filter(continent == "Europe") %>%  
  bind_rows(df) # any dataframe can be provided
```

R objects are also often appended using the `baseR::rbind()` function. It also works similarly but the column length of the two dataframes must be identical. It can be used through the `rbind(DATAFRAME1, DATAFRAME2)` syntax.

```
# Stata equivalent  
append using `Y`
```

rename columns

There are a number of ways to rename columns in R. The easiest way is through the `dplyr::rename()` function which can be combined with the `%>%` operator.

```
df %>%  
  rename(country_name = country)
```

Similarly, multiple columns can be renamed using the following syntax,

```
df %>%  
  rename(country_name = country, continent_name = continent)
```

Columns can also be renamed using the `baseR::names()` function.

```
names(df)[1] = "country"
```

```
# Stata equivalent  
rename (country continent) (country_name continent_name)
```

data.table

intro

Until now we've largely been using `dplyr` and `tidyr` (i.e. `tidyverse`) packages along with `baseR` where necessary. `data.table` is another data manipulation or wrangling package that displays the true power of R.

Detailed documentation about `data.table` can be found at [CRAN](#) and further examples can be found at the `data.table` [vignette](#).

Unique advantages of `data.table` can be listed down as,

1. insane speed (see next slide)
2. concise code
3. no dependency (stable code)
4. memory efficient

speed insanity

```
pacman::p_load(data.table, microbenchmark)

collapse_dplyr = function() {
  storms %>%
    group_by(name, year, month, day) %>%
    summarize(wind = mean(wind), pressure = mean(pressure), category = first(category))
}

storms_dt = as.data.table(storms)
collapse_dt = function() {
  storms_dt[, .(wind = mean(wind), pressure = mean(pressure), category = first(category)),
    by = .(name, year, month, day)]
}

microbenchmark(collapse_dplyr(), collapse_dt(), times = 10)

## Unit: milliseconds
##           expr           min           lq         mean        median           uq          max
## collapse_dplyr() 68.837375 72.594876 75.892371 74.361729 78.874334 85.295334
##   collapse_dt()  1.858126  1.897376  3.041051  1.984417  2.419501  9.049793
## ...
```

Check the computation time difference between `data.table` and `dplyr`.

data.table class

As mentioned earlier, every object in R has its own class and functions can be applied to that object depending on the class of the object.

`tidyverse` converts dataframes into a tibble (`tbl_df` class) in order to operate on them. Similarly, `data.table` functions can be applied to an object of `data.table` class. There are few ways to do that.

1. `data.table(x = 1:20, y = 50:70)` - to create data.table from scratch
2. `as.data.table()` - convert existing dataframes or tibbles to data.table
3. `setDT()` - reference existing dataframe as data.table. It does not create a copy of the data unlike `as.data.table()` hence need not be assigned to a new object.
4. `fread("./example.csv")` - imports csv file extremely fast as a data.table object.

operations

The `data.table` function follows the following syntax `DT[i, j, by]` where **i** refers to row-wise operations, **j** refers to column wise-operations and **by** refers to group based operations.

The `dplyr` equivalents of such operations are:

1. **i** - `filter()`, `arrange()`, `slice()`
2. **j** - `mutate()`, `select()`
3. **by** - `group_by()`, `add_count()`

For example,

```
dt_storms = as.data.table(storms)
dt_storms[status == "hurricane", mean(wind, na.rm=T), by = year]
```

```
# Stata equivalent
keep if status == "hurricane"
collapse (mean) wind, by(year)
```


row operations (1/n)

```
dt_storms[status = "hurricane",]
```

```
##           name year month day hour  lat  long   status category wind pressure
##    1: Caroline 1975     8  30    0 23.3 -94.2 hurricane         1    65      990
##    2: Caroline 1975     8  30    6 23.5 -94.9 hurricane         1    65      990
##    3: Caroline 1975     8  30   12 23.7 -95.6 hurricane         1    65      989
##    4: Caroline 1975     8  30   18 23.8 -96.3 hurricane         1    70      987
##    5: Caroline 1975     8  31    0 24.0 -97.0 hurricane         3   100      973
##    ---
## 3087:  Joaquin 2015    10   7    6 40.3 -51.5 hurricane         1    65      977
## 3088:    Kate 2015    11  11    0 33.1 -71.3 hurricane         1    65      990
## 3089:    Kate 2015    11  11    6 35.2 -67.6 hurricane         1    70      985
## 3090:    Kate 2015    11  11   12 36.2 -62.5 hurricane         1    75      980
## 3091:    Kate 2015    11  11   18 37.6 -58.2 hurricane         1    65      980
## ....
```

Note that `dt_storms[status = "hurricane"]` (without comma) is equivalent to the above operation.

```
# Stata equivalent
```

```
keep if status = "hurricane"
```

row operations (2/n)

```
dt_storms[status = "hurricane" & year > 2000 & category > 1,]
```

```
##      name year month day hour  lat  long  status category wind pressure
##  1:   Erin 2001     9   9    6 30.6 -61.3 hurricane        2    90      982
##  2:   Erin 2001     9   9   12 31.5 -62.2 hurricane        2    95      979
##  3:   Erin 2001     9   9   18 32.4 -62.8 hurricane        3   105      968
##  4:   Erin 2001     9  10    0 33.3 -63.3 hurricane        3   105      969
##  5:   Erin 2001     9  10    6 34.2 -64.1 hurricane        3   105      969
##  ---
## 649: Joaquin 2015    10   3   18 26.3 -71.0 hurricane        4   130      934
## 650: Joaquin 2015    10   4    0 27.4 -69.5 hurricane        4   115      941
## 651: Joaquin 2015    10   4    6 28.9 -68.3 hurricane        3   105      949
## 652: Joaquin 2015    10   4   12 30.4 -67.2 hurricane        2    95      956
## 653: Joaquin 2015    10   4   18 31.6 -66.5 hurricane        2    85      958
##  ....
```

Note that `dt_storms[status = "hurricane"]` (without comma) is equivalent to the above operation.

```
# Stata equivalent
```

```
keep if status = "hurricane" & year > 2000 & category > 1
```

column operations (1/n)

The `data.table` equivalent of `mutate()` is the `:=` operator which works with reference (i.e. it changes the original `data.table` so assignment is not required).

New columns can be created using the,

- `dt_storms[, wind_scaled := wind / 100]`: creates new column `wind_scaled` from existing column `wind`.
- `dt_storms[, wind := wind^2]`: changes existing column `wind`

Note that results are not printed while make such changes. In order to print results, one has to mention `[]` after the end of the operation. For example,

```
dt_storms[, wind := wind^2][]
```

Stata equivalent

```
generate wind_scaled = wind / 100
```

```
replace wind = wind^2
```

column operations (2/n)

Columns can be removed from a `data.table` using the following syntax,

```
dt_storms[, ts_diameter := NULL]
```

Note that for memory efficiency `data.table` functions work by reference implying that changes made to a copy of a data are reflected in the original data as well. For example,

```
dt_storms = as.data.table(storms)
dt_new = dt_storms
dt_new[, ts_diameter := NULL] # also drop ts_diameter from dt_storms
```

This can be avoided by creating a true copy of the original `data.table` using `data.table::copy()`. For example,

```
dt_storms = as.data.table(storms)
dt_new = copy(dt_storms)
```

```
# Stata equivalent
drop ts_diameter
```

column operations (3/n)

In order to operate on **multiple columns** at once `data.table` provides two different options which perform identically.

```
DT[, c("newCol1", "newCol2") := .(col1, col2)]  
DT[, ':= ' (newCol1=col1, newCol2=col2)]
```

Like `dplyr`, the `data.table` also allows the usage of the **pipe** `%>%` operator but a `.` needs to be added at the beginning of any operation.

```
DT %>%  
  .[, newCol1 := col1] %>%  
  .[, newCol2 := col2]
```

```
# Stata equivalent  
generate newCol1 = col1  
generate newCol2 = col2
```

column operations (4/n)

New data.table can be created from scratch using the `data.table()` function. For example,

```
dt = data.table(cities = c("London", "Paris"), rent_eur = c(1800, 1400))  
dt
```

```
##      cities rent_eur  
## 1: London      1800  
## 2:  Paris      1400
```

data.table works by reference so it changes original data.table without assignment.

```
dt[, rent_inr := rent_eur * 85] # references data.table (no assignment needed)  
dt
```

```
##      cities rent_eur rent_inr  
## 1: London      1800    153000  
## 2:  Paris      1400    119000
```

column operations (5/n)

Operations on a copy of data.table modifies the original data.table as well.

```
dt_new = dt
dt_new[, rent_eur := NULL]
dt # original data.table has changed
```

```
##      cities rent_inr
## 1: London   153000
## 2:  Paris   119000
```

Operations on true copy of data.table doesn't modify original data.table.

```
dt = data.table(cities = c("London", "Paris"), rent_eur = c(1800, 1400))
dt_new = copy(dt)
dt_new[, cities := NULL]
dt # original data.table unmodified
```

```
##      cities rent_eur
## 1: London     1800
## 2:  Paris     1400
```

grouped operation (1/n)

Grouped operation in `data.table` is straightforward and similar to `dplyr::group_by()` and Stata's `bysort` command. For example to obtain summary statistics by group,

```
dt_storms[, mean(wind, na.rm = TRUE), by = .(status,category)]
```

In order to explicitly specify the new column name,

```
dt_storms[, .(mean_wind = mean(wind, na.rm = TRUE)), by = .(status,category)]
```

In order to add mean values to a new column rather without summarizing (collapsing) the `data.table`, we can use,

```
dt_storms[, mean_wind := mean(wind, na.rm = TRUE), by = .(status,category)]
```

Stata equivalent

```
bysort status category: egen mean_wind = mean(wind)
```

```
collapse (mean) mean_wind, by(status category)
```

```
collapse (mean) wind, by(status category) # same as above 2 lines together
```


grouped operation (2/n)

```
dt_storms[, .(mean_wind = mean(wind, na.rm = TRUE)), by = .(status,category)]
```

```
##           status category mean_wind
## 1: tropical depression    -1  27.26916
## 2:      tropical storm     0  45.80037
## 3:      hurricane        1  70.91152
## 4:      hurricane        3 104.64187
## 5:      hurricane        2  89.43471
## 6:      hurricane        5 145.07353
## 7:      hurricane        4 121.55172
## 8:      tropical storm     1  70.00000
```

```
dt_storms[!status == "tropical storm", .(mean_wind = mean(wind, na.rm = TRUE)),
  by = .(status,category)]
```

```
##           status category mean_wind
## 1: tropical depression    -1  27.26916
## 2:      hurricane        1  70.91152
## 3:      hurricane        3 104.64187
## 4:      hurricane        2  89.43471
## 5:      hurricane        5 145.07353
## 6:      hurricane        4 121.55172
```

data.table features

data.table has a number of other functions or features that take data manipulation to the next level in terms of speed and ease.

- `data.table::setkey()` allows users to order the data.table on specified columns which makes any form of computation on that involving keys insanely fast. It can lead to 2-3x gains over regular data.table and 100-200x gains over dplyr operations.
- `melt()` and `cast()` to reshape data to wide or long format
- joining (merging) large data.table(s) incredibly fast using a combination of `on` option and `setkeys()` function.
- `fread()` to import large csv files lightning fast. It can also import files from a zipped file without unzipping it. (Insane!)

dplyr + data.table

dtplyr package

It is one of the most amazing packages out there which provides a data.table translation for dplyr code. It can be installed using `pacman::p_load(dtplyr)`.

In order to use dtplyr, the dataframe object has to be converted to lazy data.table class using,

```
gapminder_dtplyr = lazy_dt(gapminder::gapminder)
gapminder_dtplyr %>%
  filter(continent %in% c("Asia", "Europe")) %>%
  group_by(continent, year) %>%
  summarize(mean_pop = mean(pop, na.rm=T), mean_gdppc = mean(gdpPercap, na.rm=T))
```

dtplyr automatically translates dplyr syntax to data.table equivalent in the backend. It is an effective way to take advantage of data.table without putting any effort into learning its syntax.

tidyfast package

It provides tidying functions built on `data.table` that also accepts `%>%` operator. The package can be installed using `pacman::p_load(tidyfast)`.

Transformation from long to wide.

```
dt = as.data.table(gapminder::gapminder)
dt_wide = dt %>%
  dt_pivot_wider(names_from = year, values_from = c(lifeExp, pop, gdpPercap))
```

Transformation from wide to long

```
dt_wide %>%
  dt_pivot_longer(cols = c(-continent, -country), names_to = "type_year",
    values_to = "values") %>%
  .[, c("type", "year") := tstrsplit(type_year, "_", fixed=TRUE)] %>%
  .[, type_year := NULL] %>%
  dt_pivot_wider(names_from = type, values_from = values)
```

some important points

- R is versatile and provides different functions to perform the similar task. It is the user's job to decide which is the appropriate function to use.
- Do not be hell bent to fit every data manipulation exercise into a `dplyr` vs `data.table` problem. The correct way is to use both of them along with other packages depending on the requirement.
- Remember whatever issue you're trying to solve has more often than not already been solved. Your job is to find the right answer. In other words, google and stackoverflow is your answer to every problem.
- The easiest way to learn is to do. I believe it's better to straightaway jump into a task rather than read guides to no avail. Pick a task you already performed in Stata and try replicating in R.

thank you