Political Economy of Development:

Intro to tidyverse

Eduardo Montero

April 21, 2023

Outline

1. R:Data Wrangling with tidyverse:
    - Introduction & Motivation for using tidyverse functions
    - Pipes and Piping coding
    - Replacing R base functions
    - Functions on Groups: group_by

R: Data Wrangling with tidyverse: Introduction

tidyverse

tidyverse is a collection of packages for data manipulation. The tools within are more intuitive, and extremely useful.

It is built to be fast, highly expressive, and open-minded about how your data is stored.

tidyverse

The packages are installed as part of the the tidyverse meta-package:

tidyverse

The packages are installed as part of the the tidyverse meta-package:

```
# Load Tidyverse:
install.packages("tidyverse", repos="https://cloud.r-project.org")

## Installing package into
'/Users/emontero/Library/R/arm64/4.1/library'
## (as 'lib' is unspecified)
## also installing the dependencies 'lifecycle', 'tidyselect',
'vctrs', 'scales', 'gargle', 'googledrive', 'timechange',
'vroom', 'cpp11', 'broom', 'conflicted', 'cli', 'dbplyr',
'dplyr', 'dtplyr', 'forcats', 'ggplot2', 'googlesheets4',
'haven', 'hms', 'httr', 'jsonlite', 'lubridate', 'magrittr',
'modelr', 'pillar', 'purrr', 'ragg', 'readr', 'readxl', 'reprex',
'rlang', 'rstudioapi', 'rvest', 'stringr', 'tibble', 'tidyr',
'xml2'

##
##   There are binary versions available but the source
##   versions are later:
##         binary source needs_compilation
```

tidyverse

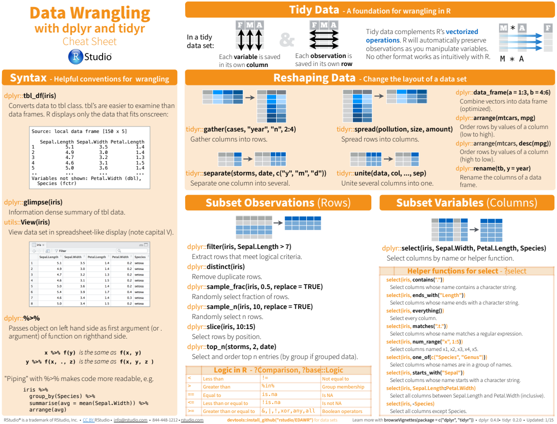
It is meant to improve on the R base functions.

For example, replaces: subset(), apply(), tapply(), aggregate(), split(), etc.

In addition, instead of relying on for() loops a lot, it provides other ways to iterate over rows or groups of rows or variables in a data frame.

tidyverse Cheat Sheet

R Studio provides an excellent cheat sheet for using it, highly recommended:



Data Wrangling
with dplyr and tidyr
Cheat Sheet

RStudio

Syntax - Helpful conventions for wrangling

dplyr::**tbl_df(iris)**

Converts data to tbl class. tbl's are easier to examine than data frames. R displays only the data that fits onscreen:

```
Source: local data frame [150 x 5]

   Sepal.Length Sepal.Width Petal.Length
1           5.1         3.5          1.4
2           4.9         3.0          1.4
3           4.7         3.2          1.3
4           4.6         3.1          1.5
5           5.0         3.6          1.4
..          ...         ...          ...
Variables not shown: Petal.Width (dbl),
  Species (fctr)
```

dplyr::**glimpse(iris)**

Information dense summary of tbl data.

utils::**View(iris)**

View data set in spreadsheet-like display (note capital V).

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa

dplyr::**%>%**

Passes object on left hand side as first argument (or . argument) of function on righthand side.

`x %>% f(y)` *is the same as* `f(x, y)`

`y %>% f(x, ., z)` *is the same as* `f(x, y, z )`

"Piping" with %>% makes code more readable, e.g.

```
iris %>%
  group_by(Species) %>%
  summarise(avg = mean(Sepal.Width)) %>%
  arrange(avg)
```

Tidy Data - A foundation for wrangling in R

In a tidy data set:

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

Each **observation** is saved in its own **row**

Tidy data complements R's **vectorized operations**. R will automatically preserve observations as you manipulate variables. No other format works as intuitively with R.

Reshaping Data - Change the layout of a data set

tidyr::**gather(cases, "year", "n", 2:4)**

Gather columns into rows.

tidyr::**spread(pollution, size, amount)**

Spread rows into columns.

tidyr::**separate(storms, date, c("y", "m", "d"))**

Separate one column into several.

tidyr::**unite(data, col, ..., sep)**

Unite several columns into one.

dplyr::**data_frame(a = 1:3, b = 4:6)**

Combine vectors into data frame (optimized).

dplyr::**arrange(mtcars, mpg)**

Order rows by values of a column (low to high).

dplyr::**arrange(mtcars, desc(mpg))**

Order rows by values of a column (high to low).

dplyr::**rename(tb, y = year)**

Rename the columns of a data frame.

Subset Observations (Rows)

dplyr::**filter(iris, Sepal.Length > 7)**

Extract rows that meet logical criteria.

dplyr::**distinct(iris)**

Remove duplicate rows.

dplyr::**sample_frac(iris, 0.5, replace = TRUE)**

Randomly select fraction of rows.

dplyr::**sample_n(iris, 10, replace = TRUE)**

Randomly select n rows.

dplyr::**slice(iris, 10:15)**

Select rows by position.

dplyr::**top_n(storms, 2, date)**

Select and order top n entries (by group if grouped data).

Logic in R - ?Comparison, ?base::Logic			
<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&,|,!,xor,any,all	Boolean operators

Subset Variables (Columns)

dplyr::**select(iris, Sepal.Width, Petal.Length, Species)**

Select columns by name or helper function.

Helper functions for select - ?select

select(iris, **contains(".")**)
Select columns whose name contains a character string.

select(iris, **ends_with("Length")**)
Select columns whose name ends with a character string.

select(iris, **everything()**)
Select every column.

select(iris, **matches(".t.")**)
Select columns whose name matches a regular expression.

select(iris, **num_range("x", 1:5)**)
Select columns named x1, x2, x3, x4, x5.

select(iris, **one_of(c("Species", "Genus"))**)
Select columns whose names are in a group of names.

select(iris, **starts_with("Sepal")**)
Select columns whose name starts with a character string.

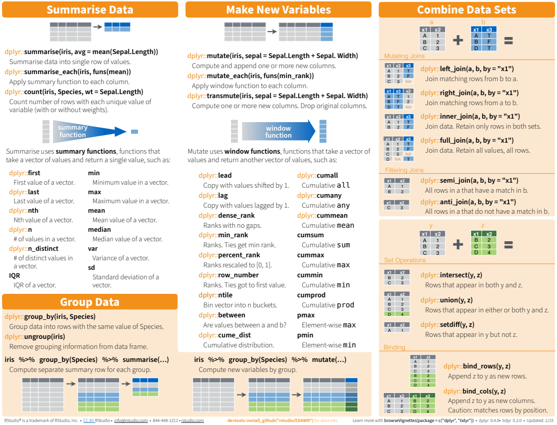
select(iris, Sepal.Length**:**Petal.Width)
Select all columns between Sepal.Length and Petal.Width (inclusive).

select(iris, -Species)
Select all columns except Species.

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com devtools::install_github("rstudio/EDAWR") for data sets Learn more with **browseVignettes(package = c("dplyr", "tidyr"))** • dplyr 0.4.0• tidyr 0.2.0 • Updated: 1/15

tidyverse Cheat Sheet II

First a quick detour into pipes and “piping” to write code, and then we will explore many useful functions:



Summarise Data

dplyr::**summarise(iris, avg = mean(Sepal.Length))**
Summarise data into single row of values.

dplyr::**summarise_each(iris, funs(mean))**
Apply summary function to each column.

dplyr::**count(iris, Species, wt = Sepal.Length)**
Count number of rows with each unique value of variable (with or without weights).

summary function

Summarise uses **summary functions**, functions that take a vector of values and return a single value, such as:

dplyr::**first**
First value of a vector.

dplyr::**last**
Last value of a vector.

dplyr::**nth**
Nth value of a vector.

dplyr::**n**
# of values in a vector.

dplyr::**n_distinct**
# of distinct values in a vector.

IQR
IQR of a vector.

min
Minimum value in a vector.

max
Maximum value in a vector.

mean
Mean value of a vector.

median
Median value of a vector.

var
Variance of a vector.

sd
Standard deviation of a vector.

Group Data

dplyr::**group_by(iris, Species)**
Group data into rows with the same value of Species.

dplyr::**ungroup(iris)**
Remove grouping information from data frame.

iris %>% group_by(Species) %>% summarise(…)
Compute separate summary row for each group.

Make New Variables

dplyr::**mutate(iris, sepal = Sepal.Length + Sepal. Width)**
Compute and append one or more new columns.

dplyr::**mutate_each(iris, funs(min_rank))**
Apply window function to each column.

dplyr::**transmute(iris, sepal = Sepal.Length + Sepal. Width)**
Compute one or more new columns. Drop original columns.

window function

Mutate uses **window functions**, functions that take a vector of values and return another vector of values, such as:

dplyr::**lead**
Copy with values shifted by 1.

dplyr::**lag**
Copy with values lagged by 1.

dplyr::**dense_rank**
Ranks with no gaps.

dplyr::**min_rank**
Ranks. Ties get min rank.

dplyr::**percent_rank**
Ranks rescaled to [0, 1].

dplyr::**row_number**
Ranks. Ties got to first value.

dplyr::**ntile**
Bin vector into n buckets.

dplyr::**between**
Are values between a and b?

dplyr::**cume_dist**
Cumulative distribution.

dplyr::**cumall**
Cumulative `all`

dplyr::**cumany**
Cumulative `any`

dplyr::**cummean**
Cumulative `mean`

cumsum
Cumulative `sum`

cummax
Cumulative `max`

cummin
Cumulative `min`

cumprod
Cumulative `prod`

pmax
Element-wise `max`

pmin
Element-wise `min`

iris %>% group_by(Species) %>% mutate(…)
Compute new variables by group.

Combine Data Sets

a

x1	x2
A	1
B	2
C	3

+

b

x1	x3
A	T
B	F
D	T

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::**left_join(a, b, by = "x1")**
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::**right_join(a, b, by = "x1")**
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::**inner_join(a, b, by = "x1")**
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::**full_join(a, b, by = "x1")**
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

dplyr::**semi_join(a, b, by = "x1")**
All rows in a that have a match in b.

x1	x2
C	3

dplyr::**anti_join(a, b, by = "x1")**
All rows in a that do not have a match in b.

y

x1	x2
A	1
B	2
C	3

+

z

x1	x2
B	2
C	3
D	4

=

Set Operations

x1	x2
B	2
C	3

dplyr::**intersect(y, z)**
Rows that appear in both y and z.

x1	x2
A	1
B	2
C	3
D	4

dplyr::**union(y, z)**
Rows that appear in either or both y and z.

x1	x2
A	1

dplyr::**setdiff(y, z)**
Rows that appear in y but not z.

Binding

x1	x2
A	1
B	2
C	3
B	2
C	3
D	4

dplyr::**bind_rows(y, z)**
Append z to y as new rows.

x1	x2	x1	x2
A	1	B	2
B	2	C	3
C	3	D	4

dplyr::**bind_cols(y, z)**
Append z to y as new columns.
Caution: matches rows by position.

devtools::install_github("rstudio/EDAWR") for data sets

Learn more with **browseVignettes(package = c("dplyr", "tidyr"))** • dplyr 0.4.0• tidyr 0.2.0 • Updated: 1/15

R: Data Wrangling with tidyverse: Pipes

Pipes & Piping: More “Readable” Code

One huge advantage of tidyverse functions is that they can be used with pipe notation

piping makes (1) R coding more intuitive, and (2) dramatically improves code readability

- Evaluates function calls as “chains”: goes “outside-in” (left-to-right) rather than “inside-out” (right-to-left)

Piping: More "Readable" Code

To see what piping is and how it improves coding readability:

- Let's go through a simple example using filter from tidyverse
- (filter is the tidyverse improvement of the subset function from R base functions)
- We will first load the gapminder dataset: data on life expectancy, GDP per capita, and population by country

```r
# Load the gapminder data:
gapminder.data <- read.csv(file = "./gapminder.csv")
```

Piping: More "Readable" Code

- Explore the gapminder dataset: data on life expectancy, GDP per capita, and population by country

```
# Check out the gapminder data:
head(gapminder.data)

##       country continent year lifeExp      pop gdpPercap
## 1 Afghanistan      Asia 1952  28.801  8425333  779.4453
## 2 Afghanistan      Asia 1957  30.332  9240934  820.8530
## 3 Afghanistan      Asia 1962  31.997 10267083  853.1007
## 4 Afghanistan      Asia 1967  34.020 11537966  836.1971
## 5 Afghanistan      Asia 1972  36.088 13079460  739.9811
## 6 Afghanistan      Asia 1977  38.438 14880372  786.1134

  # From the gapminder library
```

Piping: Example

filter replaces subset:

filter() takes logical expressions and returns the rows for which all are TRUE.

Piping: Example

filter replaces subset:

filter() takes logical expressions and returns the rows for which all are TRUE.

We can “filter” the dataset to only include countries with life expectancy less than 29:

Without Piping:

```
# Filter the data
gapminder.data.under29 <- filter(gapminder.data, lifeExp < 29)

## Error in filter(gapminder.data, lifeExp < 29):  object
'lifeExp' not found
```

Piping: Use filter() to subset data row-wise

Filter to countries with life expectancy less than 29 using the filter function with piping:

Piping uses $\% > \%$ to pass previous objects into functions, going “left-to-right” (or in “chains”)

With Piping:

```
# Filter the data and pipe it:
gapminder.data.under29 <- gapminder.data %>%  filter(lifeExp < 29)

## Error in gapminder.data %>% filter(lifeExp < 29):  could not
find function "%>%"

  # Code evaluates ""left-to-right", more readable!
```

Piping: Use filter() to subset data row-wise

Piping is particularly helpful for lines of code with many different function calls. Let's see an example:

- Note: Example below uses mutate to create a variable:

```
## Filter data, create a variable using mutate, and filter again:

# Without Piping:
gap.without.pipe <-
  filter(
    mutate(
      filter(gapminder.data, lifeExp < 29), filter_sample = 1),
    lifeExp > 20)

## Error in mutate(filter(gapminder.data, lifeExp < 29),
filter_sample = 1):  could not find function "mutate"

                  # have to go "inside-out" to understand this...
# With Piping:
gap.with.pipe <- gapminder.data %>%  filter( lifeExp < 29) %>%
                  mutate(filter_sample = 1) %>%  filter(lifeExp > 20)

## Error in gapminder.data %>% filter(lifeExp < 29) %>%
```

Piping: Another Example

Piping can now also be used with many base functions

Thus, you can use pipes for almost all data preparation now

Piping: Another Example

Another example of readability with pipes using basic functions:

Compute the logarithm of some x, calculate lagged differences, compute the exponential function, and round the result:

```
## Piping Example
# Initialize `x`
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)
# No Pipes:
round(exp(diff(log(x))), 1)

## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1

# With Pipes
x %>% log() %>%
    diff() %>%
    exp() %>%
    round(1)

## Error in x %>% log() %>% diff() %>% exp() %>% round(1):  could
not find function "%>%"
```

Piping: Conclusion

Piping is super useful. Makes R coding even more intuitive and code more readable

Highly recommend using it! We will see many examples using it today

RStudio also has shortcuts to add %>% on a line:

- On Mac: “Shift + Command + M”
- On Windows: “Shift+ Control + M”

Piping: Another Example

Piping Practice: Take the x vector below and using piping:

1. Take the exponent of all x values (exp())
2. Take the square root of the result (sqrt())
3. Find the sum of the result (sum())

```r
## Piping Practice:
# Initialize `x`
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)
```

Piping: Another Example

Piping Practice: Take the x vector below and using piping:

1. Take the exponent of all x values (exp())
2. Take the square root of the result (sqrt())
3. Find the sum of the result (sum())

```
## Piping Practice:
# One Solution:
x %>% exp() %>% sqrt() %>% sum()

## Error in x %>% exp() %>% sqrt() %>% sum():  could not find
function "%>%"
```

R: Back to tidyverse: Basics

What do tidyverse functions return: tibbles

Calling tidyverse functions – e.g. filter – often return an object called a tibble

A tibble is very similar to a data.frame object, it just optimized to be faster for tidyverse functions.

What do tidyverse functions return: tibbles

For example, we can filter the data to select only Rwanda and Afghanistan:

Option 1: Listing two logical statements. Returns a tibble:

```
# Filter and see Result:
head(
  gapminder.data %>% filter(country == "Rwanda" | country == "Afghanist
  ## Here, we do not "store" the output in another object (using <- nor
)

## Error in gapminder.data %>% filter(country == "Rwanda" |
country == "Afghanistan"):  could not find function "%>%"
```

What do tidyverse functions return: tibbles

Option 2: Using %in% – checks if an item is “in” another vector or list:

- Convenient alternative to typing out multiple different statements, especially for long lists

```
# Filter using %in% and see result:
head(
  gapminder.data %>% filter(country %in% c("Rwanda", "Afghanistan"))
)

## Error in gapminder.data %>% filter(country %in% c("Rwanda",
"Afghanistan")):  could not find function "%>%"
```

Use select() on variables or columns

Other useful functions:

Use select to select only year and lifeExp columns from the data:

Use select() on variables or columns

Other useful functions:

Use select to select only year and lifeExp columns from the data:

```
head(
  gapminder.data %>% select(year, lifeExp)
)

## Error in gapminder.data %>% select(year, lifeExp):  could not
find function "%>%"
```

Use mutate() to add new variables

Imagine we wanted to recover each country's total GDP.

mutate() is a function that defines and inserts new variables into the dataset

One big advantage: You can refer to existing variables by name!

- Do not have to use the $ operator and spell-out data object every time (e.g. gapminder$gdp)

Use mutate to create total GDP for each country:

Use mutate() to add new variables

Imagine we wanted to recover each country's total GDP.

mutate() is a function that defines and inserts new variables into the dataset

One big advantage: You can refer to existing variables by name!

- Do not have to use the $ operator and spell-out data object every time (e.g. gapminder$gdp)

Use mutate to create total GDP for each country:

```
## Using Mutate, with piping:
gapminder.data.mutated <- gapminder.data %>%
                          mutate(gdp = pop * gdpPercap)

## Error in gapminder.data %>% mutate(gdp = pop * gdpPercap):
could not find function "%>%"
```

Use arrange() to row-order data in a principled way

arrange() reorders the rows in a data frame.

Imagine you wanted this data ordered by year then country, as opposed to by country then year, in increasing order:

Use arrange() to row-order data in a principled way

arrange() reorders the rows in a data frame.

Imagine you wanted this data ordered by year then country, as opposed to by country then year, in increasing order:

```
# Arrange Data by year and country:
head(
  gapminder.data %>% arrange(year, country)
)

## Error in gapminder.data %>% arrange(year, country):  could not
find function "%>%"
```

Use arrange() to row-order data in a principled way

Or, if you want data sorted by life expectancy in a descending order:

Use arrange() to row-order data in a principled way

Or, if you want data sorted by life expectancy in a descending order:

```
head(
  gapminder.data %>%  arrange(desc(lifeExp))
)

## Error in gapminder.data %>% arrange(desc(lifeExp)):  could not
find function "%>%"
```

Use rename() to rename variables

Can use rename() to rename variables, very useful for cleaning data:

Can also use select():

```
## Using Rename -- Piping:
gapminder.data.renamed <- gapminder.data %>% rename(gdp_percap = gdpPer

## Error in gapminder.data %>% rename(gdp_percap = gdpPercap):
could not find function "%>%"

## Using Rename -- No Piping:
gapminder.data.renamed<- rename(gapminder.data, gdp_percap = gdpPercap)

## Error in rename(gapminder.data, gdp_percap = gdpPercap):
could not find function "rename"
```

tidyverse practice

Let's take a few minutes to do some practice with tidyverse

Exercise: Using the gapminder dataset and "piping", do the following:

1. Create a variable with the ratio of life expectancy to gdp per capita
2. Select only that variable you created, the country, and the year
3. Filter the data to only look at countries where this ratio is above 0.1 and the year is greater than 1995
4. Rename the variable created to be called ratio_practice
5. Arrange by ratio_practice in ascending order

Extra Exercise: Try keeping it to only one set of piping commands

tidyverse practice

One Possible Solution (using piping):

```r
gapminder.practice <- gapminder.data %>%
  # 1. Create the variable
  mutate(ratio_le_gdp = lifeExp/gdpPercap) %>%
  # 2. Select only year and ratio_le_gdp:
  select(year, country, ratio_le_gdp) %>%
  # 3. Filter to where ratio >2 and year > 1995:\
  filter(ratio_le_gdp > 0.1, year>1995) %>%
  # 4. Rename variable
  rename(ratio_practice = ratio_le_gdp) %>%
  # 5. Sort:
  arrange(ratio_practice)

## Error in gapminder.data %>% mutate(ratio_le_gdp =
lifeExp/gdpPercap) %>% :  could not find function "%>%"

# First Country:
gapminder.practice[1,]

## Error in eval(expr, envir, enclos):  object
'gapminder.practice' not found
```

R: Data Wrangling with tidyverse: group_by()

Group_by

tidyverse is extremely useful for questions like: “which country experienced the sharpest *5-year* drop in life expectancy?”

This is a question about “groups” of data: *a country* across intervals

- This type of data – many observations (countries) across time – is also known as panel data
- We will explore this type of data much more for difference-in-differences methods

Answering this question is difficult to do with base functions without using for() loops for example (or, apply())

These functions are not well suited for group questions; however, tidyverse is. This is very useful when preparing data.

Group_by

tidyverse offers powerful tools to solve this class of “grouped” problem:

Group_by

tidyverse offers powerful tools to solve this class of “grouped” problem:

- group_by() adds extra structure to your dataset – grouping information – which lays the groundwork for computations within the groups

Group_by

tidyverse offers powerful tools to solve this class of “grouped” problem:

- group_by() adds extra structure to your dataset – grouping information – which lays the groundwork for computations within the groups
- summarize() takes a dataset with n observations, computes requested summaries, and returns a dataset with 1 observation per group
- mutate() creates a variable for each group
- mutate() and summarize() will honor groups. So summarize() after using group_by() will turn n observation to m observations, where $m = numberofgroups$

group_by: Counting

Let's start with simple counting.

Example: How many observations do we have per continent?

With tidyverse and piping:

```
# Piping: Num Obs per Continent
 gapminder.data %>% group_by(continent) %>%
  summarize(n=n())
```

```
## Error in gapminder.data %>% group_by(continent) %>%
summarize(n = n()):  could not find function "%>%"
```

group_by: Counting

Can also use the tally() function as a convenience function that knows to count rows.

It honors groups.

```
# Count Observations by continent using tally:
gapminder.data %>% group_by(continent) %>% tally()

## Error in gapminder.data %>% group_by(continent) %>% tally():
could not find function "%>%"
```

group_by: Counting

What if we wanted to add the number of unique countries for each continent?

You can compute multiple summaries inside summarize().

Use the n_distinct() function to count the number of distinct countries within each continent.

group_by: Counting

What if we wanted to add the number of unique countries for each continent?

You can compute multiple summaries inside summarize().

Use the n_distinct() function to count the number of distinct countries within each continent.

```
# Group by Continent, then find number of observations by continent, an
gapminder.data %>% group_by(continent) %>%
  summarize(n = n(),
            n_countries = n_distinct(country))

## Error in gapminder.data %>% group_by(continent) %>%
summarize(n = n(), :  could not find function "%>%"
```

group_by: Summarizing

Can apply many functions within summarize().

For example, classical statistical summaries, like mean(), median(), var(), sd(), mad(), IQR(), min(), and max()

These will be applied *for each group*:

```
# Group and Calculate Avergae Life Expectacy BY CONTINENT:
gapminder.grouped <- gapminder.data %>% group_by(continent)

## Error in gapminder.data %>% group_by(continent):  could not
find function "%>%"

summarize(gapminder.grouped,
          avg_lifeExp = mean(lifeExp))

## Error in summarize(gapminder.grouped, avg_lifeExp =
mean(lifeExp)):  could not find function "summarize"
```

tidyverse group practice

Let's take a few minutes to do some practice with tidyverse and groups:

Exercise: Using the gapminder dataset and "pipes", do the following:

1. Focus on Asian countries,
2. Find: What are the minimum and maximum life expectancies each year in Asia?

tidyverse group practice

Exercise: Using the gapminder dataset and “pipes”, do the following:

1. Focus on Asian countries,
2. Find: What are the minimum and maximum life expectancies seen *by year*?

```r
# filter to Asia, group by year, and find min and max life exp
gapminder.data %>%
  filter(continent == "Asia") %>%
  group_by(year) %>%
  summarize(min_lifeExp = min(lifeExp), max_lifeExp = max(lifeExp))

## Error in gapminder.data %>% filter(continent == "Asia") %>%
group_by(year) %>% :  could not find function "%>%"
```

group_by: Summarizing

summarize_at() applies the same summary function(s) to multiple variables.

For example, for computing average and median life expectancy **and** GDP per capita **by continent, by year**:

```
# Per continent-year, mean and median life exp and gdp per capita:
gapminder.data %>% group_by(continent, year) %>%
  summarize_at(vars(lifeExp, gdpPercap), list(mean, median))

## Error in gapminder.data %>% group_by(continent, year) %>%
summarize_at(vars(lifeExp, :  could not find function "%>%"
```

group_by: Mutate

summarize() collapses the data (n rows to 1 per group). But, sometimes, we want to keep all rows and compute within the rows and add group information to all rows.

We can do this using mutate() with group_by.

For example, we can make a new variable that is the *years of life expectancy gained (lost) relative to 1952, for each individual country*

We group by country and use mutate() to make a new variable

- In some ways, creates "mini-datasets" within our dataset (gapminder.data)

group_by: Mutate

Making a new variable that is the years of life expectancy gained (lost) relative to 1952, for each individual country.

The first() function extracts the first value from a vector. Notice that first() is operating on the vector of life expectancies within each country group.

```
gapminder.grouped <- group_by(gapminder.data, country)

## Error in group_by(gapminder.data, country):  could not find
function "group_by"

mutate(gapminder.grouped,
       lifeExp_gain = lifeExp - first(lifeExp))

## Error in mutate(gapminder.grouped, lifeExp_gain = lifeExp -
first(lifeExp)):  could not find function "mutate"
```

group_by: Window Functions

Window functions, such as mutate, also take n inputs and give back n outputs, but are even more flexible

Example: Examine the worst and best life expectancies in Asia over time, **and** retaining info about **which** country contributes these extreme values.

Let's first subset the data to be data from Asia and only keep year, country and life expectancy:

```
# Commands without piping:
asia <-
  select(
    group_by(
      filter(
        gapminder.data,
        continent == "Asia"),
      year),
    year,
    country,
    lifeExp)
```

group_by: Window Functions

Alternative, using piping:

```
# OR: with piping: Much easier to follow:
asia <- gapminder.data %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  group_by(year)

## Error in gapminder.data %>% filter(continent == "Asia") %>%
select(year, :  could not find function "%>%"

head(asia)

## Error in head(asia):  object ’asia’ not found
```

group_by: Window Functions

Apply window function: min_rank(). Since Asia is grouped by year, min_rank() operates within *mini-datasets*, each for a specific year.

Applied to the variable lifeExp, min_rank() returns the rank of each country's observed life expectancy for each year group.

```r
# Min and Max Rank Each Year:
asia <- asia %>% mutate(
               lowest_rank_eachyear = min_rank(lifeExp),
               highest_rank_eachyear = min_rank(desc(lifeExp)))

## Error in asia %>% mutate(lowest_rank_eachyear =
min_rank(lifeExp), highest_rank_eachyear =
min_rank(desc(lifeExp))):  could not find function "%>%"
```

group_by: Window Functions

Can now filter to the top and bottom countries for each year:

```
# Filtering to Min and Max Rank within Asia:
filter(asia,
       highest_rank_eachyear==1 | lowest_rank_eachyear==1) %>%
  arrange(year, country)

## Error in filter(asia, highest_rank_eachyear == 1 |
lowest_rank_eachyear == :  could not find function "%>%"
```

group_by: Window Functions

Can also use top_n() function if we only want the highest-rank observation:

```
gapminder.data %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  arrange(year) %>%
  group_by(year) %>%
  top_n(1, wt = desc(lifeExp))

## Error in gapminder.data %>% filter(continent == "Asia") %>%
select(year, :  could not find function "%>%"

  #top_n(1, wt = lifeExp)        ## gets the min
```

group_by: Final Question Exercise

Exercise: let's answer that "simple" question we started with: which country experienced the sharpest 5-year drop in life expectancy?

The Gapminder data only has data every five years, e.g. for 1952, 1957, etc. So this really means looking at life expectancy changes between adjacent time points.

group_by: Final Question Exercise

Exercise: let's answer that "simple" question we started with: which country experienced the sharpest 5-year drop in life expectancy?

The Gapminder data only has data every five years, e.g. for 1952, 1957, etc. So this really means looking at life expectancy changes between adjacent time points.

Hint: Start by creating the change in expectancy (using lag "window" function with mutate).

group_by: Final Question Exercise

Possible Solution: let's answer that "simple" question: which country experienced the sharpest 5-year drop in life expectancy?

Hint: Start by creating the change in expectancy (using lag "window" function with mutate).

Dense code on next slide, might not want to do so much manipulation in one fell swoop. But, to do for you: break the code into pieces, starting at the top, and inspect the intermediate results.

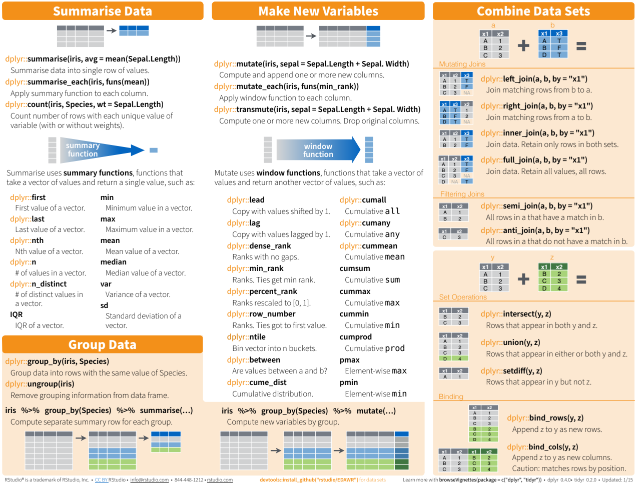
group_by: Final Question

```
gapminder.data %>%
  # Sort by country and year:
  arrange(country, year) %>%
  # Keep only variables of interest:
  select(country, year, continent, lifeExp) %>%
  # Group by country (since we are interested on "within" country cha
  group_by(country) %>%
  # Create variable = (lifeExp in year i) - (lifeExp in year i - 1)
  mutate(le_delta = lifeExp - lag(lifeExp)) %>%
  # within country, retain the worst lifeExp change = smallest or mos
  summarize(worst_le_delta = min(le_delta, na.rm = TRUE)) %>%
  # within country, retain the row with the lowest worst_le_delta
  top_n(1, wt = desc(worst_le_delta)) %>%
  # Sort data:
  arrange(worst_le_delta)
```

```
## Error in gapminder.data %>% arrange(country, year) %>%
select(country, : could not find function "%>%"
```

Remember: tidyverse Cheat sheet

Highly recommended cheat sheet, hopefully it makes more sense now (and will keep making more sense):



Summarise Data

dplyr::**summarise(iris, avg = mean(Sepal.Length))**
Summarise data into single row of values.

dplyr::**summarise_each(iris, funs(mean))**
Apply summary function to each column.

dplyr::**count(iris, Species, wt = Sepal.Length)**
Count number of rows with each unique value of variable (with or without weights).

summary function

Summarise uses **summary functions**, functions that take a vector of values and return a single value, such as:

dplyr::**first**
First value of a vector.

dplyr::**last**
Last value of a vector.

dplyr::**nth**
Nth value of a vector.

dplyr::**n**
# of values in a vector.

dplyr::**n_distinct**
# of distinct values in a vector.

IQR
IQR of a vector.

min
Minimum value in a vector.

max
Maximum value in a vector.

mean
Mean value of a vector.

median
Median value of a vector.

var
Variance of a vector.

sd
Standard deviation of a vector.

Group Data

dplyr::**group_by(iris, Species)**
Group data into rows with the same value of Species.

dplyr::**ungroup(iris)**
Remove grouping information from data frame.

iris %>% group_by(Species) %>% summarise(…)
Compute separate summary row for each group.

Make New Variables

dplyr::**mutate(iris, sepal = Sepal.Length + Sepal. Width)**
Compute and append one or more new columns.

dplyr::**mutate_each(iris, funs(min_rank))**
Apply window function to each column.

dplyr::**transmute(iris, sepal = Sepal.Length + Sepal. Width)**
Compute one or more new columns. Drop original columns.

Mutate uses **window functions**, functions that take a vector of values and return another vector of values, such as:

dplyr::**lead**
Copy with values shifted by 1.

dplyr::**lag**
Copy with values lagged by 1.

dplyr::**dense_rank**
Ranks with no gaps.

dplyr::**min_rank**
Ranks. Ties get min rank.

dplyr::**percent_rank**
Ranks rescaled to [0, 1].

dplyr::**row_number**
Ranks. Ties got to first value.

dplyr::**ntile**
Bin vector into n buckets.

dplyr::**between**
Are values between a and b?

dplyr::**cume_dist**
Cumulative distribution.

dplyr::**cumall**
Cumulative `all`

dplyr::**cumany**
Cumulative `any`

dplyr::**cummean**
Cumulative `mean`

cumsum
Cumulative `sum`

cummax
Cumulative `max`

cummin
Cumulative `min`

cumprod
Cumulative `prod`

pmax
Element-wise `max`

pmin
Element-wise `min`

iris %>% group_by(Species) %>% mutate(…)
Compute new variables by group.

Combine Data Sets

a

x1	x2
A	1
B	2
C	3

+

b

x1	x3
A	T
B	F
D	T

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::**left_join(a, b, by = "x1")**
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::**right_join(a, b, by = "x1")**
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::**inner_join(a, b, by = "x1")**
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::**full_join(a, b, by = "x1")**
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

dplyr::**semi_join(a, b, by = "x1")**
All rows in a that have a match in b.

x1	x2
C	3

dplyr::**anti_join(a, b, by = "x1")**
All rows in a that do not have a match in b.

y

x1	x2
A	1
B	2
C	3

+

z

x1	x2
B	2
C	3
D	4

=

Set Operations

x1	x2
B	2
C	3

dplyr::**intersect(y, z)**
Rows that appear in both y and z.

x1	x2
A	1
B	2
C	3
D	4

dplyr::**union(y, z)**
Rows that appear in either or both y and z.

x1	x2
A	1

dplyr::**setdiff(y, z)**
Rows that appear in y but not z.

Binding

x1	x2
A	1
B	2
C	3
B	2
C	3
D	4

dplyr::**bind_rows(y, z)**
Append z to y as new rows.

x1	x2	x1	x2
A	1	B	2
B	2	C	3
C	3	D	4

dplyr::**bind_cols(y, z)**
Append z to y as new columns.
Caution: matches rows by position.

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

devtools::install_github("rstudio/EDAWR") for data sets

Learn more with **browseVignettes(package = c("dplyr", "tidyr"))** • dplyr 0.4.0• tidyr 0.2.0 • Updated: 1/15

Next class

Next class:

- Development Through Foreign Aid I
- R: Fixed Effects and Difference-in-Differences in R