Session 2a

The Data Carpentry (amended and delivered by Chris Oldnall)

October 25th 2023

Usage and Adaptation of Data Carpentry Materials

Objectives of the session:

- Describe the purpose of an R package and the **`dplyr`** package.
- Select certain columns in a dataframe with the **`dplyr`** function `select`.
- Select certain rows in a dataframe according to filtering conditions with the **`dplyr`** function `filter`.
- Link the output of one **`dplyr`** function to the input of another function with the 'pipe' operator %>%.
- Add new columns to a dataframe that are functions of existing columns with `mutate`.
- Use the split-apply-combine concept for data analysis.
- Use `summarize`, `group_by`, and `count` to split a dataframe into groups of observations, apply a summary statistics for each group, and then combine the results.

- Describe the concept of a wide and a long table format and for which purpose those formats are useful.
- Describe the roles of variable names and their associated values when a table is reshaped.
- Reshape a dataframe from long to wide format and back with the `pivot_wider` and `pivot_longer` commands from the **`tidyr`** package.
- Export a dataframe to a csv file.

Questions to be able to answer:

- How can I select specific rows and/or columns from a dataframe?
- How can I combine multiple commands into a single command?
- How can I create new columns or remove existing columns from a dataframe?
- How can I reformat a dataframe to meet my needs?

What is dplyr

`dplyr` is a package for making tabular data wrangling easier by using a limited set of functions that can be combined to extract and summarize insights from your data.

Like **`readr`**, **`dplyr`** is a part of the tidyverse. These packages were loaded in R's memory when we called `library(tidyverse)` earlier.

Note

The packages in the tidyverse, namely `dplyr`, `tidyr` and `ggplot2` accept both the British (e.g. *summarise*) and American (e.g. *summarize*) spelling variants of different function and option names. For this lesson, we utilize the American spellings of different functions; however, feel free to use the regional variant for where you are teaching.

Learning `dplyr`

To make sure everyone will use the same dataset for this lesson, we'll read again the SAFI dataset that we downloaded earlier.

```
library(tidyverse)
interviews <- read_csv("https://raw.githubusercontent.com/c
```

Functions in dplyr

We're going to learn some of the most common **`dplyr`** functions:

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `mutate()`: create new columns by using information from other columns
- `group_by()` and `summarize()`: create summary statistics on grouped data
- `arrange()`: sort results
- `count()`: count discrete values

Selecting columns and filtering rows

To select columns of a dataframe, use `select()`. The first argument to this function is the dataframe (`interviews`), and the subsequent arguments are the columns to keep, separated by commas. Alternatively, if you are selecting columns adjacent to each other, you can use a `:` to select a range of columns, read as "select columns from ____ to ____." You may have done something similar in the past using subsetting. `select()` is essentially doing the same thing as subsetting, using a package (`dplyr`) instead of R's base functions.

Selecting example

```
# to select columns throughout the dataframe
select(interviews, village, no_membrs, months_lack_food)
# to select a series of connected columns
select(interviews, village:respondent_wall_type)
```

Filter example

To choose rows based on specific criteria, we can use the `filter()` function. The argument after the dataframe is the condition we want our final dataframe to adhere to (e.g. village name is Chirodzo):

```
# filters observations where village name is "Chirodzo"
filter(interviews, village == "Chirodzo")
```

```
## # A tibble: 39 x 14
##    key_ID village  interview_date       no_membrs years_l
##     <dbl> <chr>    <dttm>                   <dbl>     <db
##  1      8 Chirodzo 2016-11-16 00:00:00         12
##  2      9 Chirodzo 2016-11-16 00:00:00          8
##  3     10 Chirodzo 2016-12-16 00:00:00         12
##  4     34 Chirodzo 2016-11-17 00:00:00          8
##  5     35 Chirodzo 2016-11-17 00:00:00          5
##  6     36 Chirodzo 2016-11-17 00:00:00          6
##  7     37 Chirodzo 2016-11-17 00:00:00          3
##  8     43 Chirodzo 2016-11-17 00:00:00          7
```

Filter example cont.

We can also specify multiple conditions within the `filter()` function. We can combine conditions using either "and" or "or" statements. In an "and" statement, an observation (row) must meet **every** criteria to be included in the resulting dataframe. To form "and" statements within dplyr, we can pass our desired conditions as arguments in the `filter()` function, separated by commas:

```
# filters observations with "and" operator (comma)
# output dataframe satisfies ALL specified conditions
filter(interviews, village == "Chirodzo",
                   rooms > 1,
                   no_meals > 2)
```

```
## # A tibble: 10 x 14
##    key_ID village  interview_date       no_membrs years_l
##     <dbl> <chr>    <dttm>                   <dbl>     <db
##  1     10 Chirodzo 2016-12-16 00:00:00         12
##  2     49 Chirodzo 2016-11-16 00:00:00          6
##  3     52 Chirodzo 2016-11-16 00:00:00         11
```

Filter example cont.

We can also form "and" statements with the & operator instead of commas:

```
# filters observations with "&" logical operator
# output dataframe satisfies ALL specified conditions
filter(interviews, village == "Chirodzo" &
                   rooms > 1 &
                   no_meals > 2)
```

```
## # A tibble: 10 x 14
##    key_ID village  interview_date       no_membrs years_l
##     <dbl> <chr>    <dttm>                   <dbl>     <db
##  1     10 Chirodzo 2016-12-16 00:00:00         12
##  2     49 Chirodzo 2016-11-16 00:00:00          6
##  3     52 Chirodzo 2016-11-16 00:00:00         11
##  4     56 Chirodzo 2016-11-16 00:00:00         12
##  5     65 Chirodzo 2016-11-16 00:00:00          8
##  6     66 Chirodzo 2016-11-16 00:00:00         10
##  7     67 Chirodzo 2016-11-16 00:00:00          5
```

Filter example cont.

In an "or" statement, observations must meet *at least one* of the specified conditions. To form "or" statements we use the logical operator for "or," which is the vertical bar (|):

```
# filters observations with "|" logical operator
# output dataframe satisfies AT LEAST ONE of the specified
filter(interviews, village == "Chirodzo" | village == "Ruac
```

```
## # A tibble: 88 x 14
##    key_ID village  interview_date       no_membrs years_
##     <dbl> <chr>    <dttm>                   <dbl>     <db
##  1      8 Chirodzo 2016-11-16 00:00:00         12
##  2      9 Chirodzo 2016-11-16 00:00:00          8
##  3     10 Chirodzo 2016-12-16 00:00:00         12
##  4     23 Ruaca    2016-11-21 00:00:00         10
##  5     24 Ruaca    2016-11-21 00:00:00          6
##  6     25 Ruaca    2016-11-21 00:00:00         11
##  7     26 Ruaca    2016-11-21 00:00:00          3
##  8     27 Ruaca    2016-11-21 00:00:00          7
```

Multiple steps

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

With intermediate steps, you create a temporary dataframe and use that as input to the next function, like this:

```
interviews2 <- filter(interviews, village == "Chirodzo")
interviews_ch <- select(interviews2, village:respondent_wal
```

This is readable, but can clutter up your workspace with lots of objects that you have to name individually. With multiple steps, that can be hard to keep track of.

Nesting

You can also nest functions (i.e. one function inside of another), like this:

```
interviews_ch <- select(filter(interviews, village == "Chi
                               village:respondent_wall_type)
```

Pipes

The last option, *pipes* '%>%', are a recent addition to R. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. You can access this pipe function with:

- Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```
interviews %>%
    filter(village == "Chirodzo") %>%
    select(village:respondent_wall_type)
```

```
## # A tibble: 39 x 5
##    village  interview_date      no_membrs years_liv resp
##    <chr>    <dttm>                  <dbl>     <dbl> <chr
##  1 Chirodzo 2016-11-16 00:00:00        12        70 burn
##  2 Chirodzo 2016-11-16 00:00:00         8         6 burn
##  3 Chirodzo 2016-12-16 00:00:00        12        23 burn
##  4 Chirodzo 2016-11-17 00:00:00         8        18 burn
##  5 Chirodzo 2016-11-17 00:00:00         5        45 mudd
```

Assigning Alongside Piping

If we want to create a new object with this smaller version of the data, we can assign it a new name:

```
interviews_ch <- interviews %>%
    filter(village == "Chirodzo") %>%
    select(village:respondent_wall_type)
```

Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or to find the ratio of values in two columns. For this we'll use `mutate()`.

We might be interested in the ratio of number of household members to rooms used for sleeping (i.e. avg number of people per room):

```
interviews %>%
    mutate(people_per_room = no_membrs / rooms)
```

```
## # A tibble: 131 x 15
##    key_ID village  interview_date       no_membrs years_l
##     <dbl> <chr>    <dttm>                   <dbl>     <db
##  1      1 God      2016-11-17 00:00:00          3
##  2      2 God      2016-11-17 00:00:00          7
##  3      3 God      2016-11-17 00:00:00         10
##  4      4 God      2016-11-17 00:00:00          7
##  5      5 God      2016-11-17 00:00:00          7
##  6      6 God      2016-11-17 00:00:00          3
```

Filter and Mutate

We may be interested in investigating whether being a member of an irrigation association had any effect on the ratio of household members to rooms. To look at this relationship, we will first remove data from our dataset where the respondent didn't answer the question of whether they were a member of an irrigation association. These cases are recorded as "NULL" in the dataset.

To remove these cases, we could insert a `filter()` in the chain:

```
interviews %>%
    filter(!is.na(memb_assoc)) %>%
    mutate(people_per_room = no_membrs / rooms)
```

```
## # A tibble: 131 x 15
##    key_ID village  interview_date       no_membrs years_l
##     <dbl> <chr>    <dttm>                   <dbl>     <db
##  1      1 God      2016-11-17 00:00:00          3
##  2      2 God      2016-11-17 00:00:00          7
##  3      3 God      2016-11-17 00:00:00         10
##  4      4 God      2016-11-17 00:00:00          7
##  5      5 God      2016-11-17 00:00:00          7
##  6      6 God      2016-11-17 00:00:00          3
##  7      7 God      2016-11-17 00:00:00          6
##  8      8 Chirodzo 2016-11-16 00:00:00         12
##  9      9 Chirodzo 2016-11-16 00:00:00          8
## 10     10 Chirodzo 2016-12-16 00:00:00         12
## # i 121 more rows
## # i 9 more variables: rooms <dbl>, memb_assoc <chr>, aft
```

Split-apply-combine data analysis and the summarize() function

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results. **`dplyr`** makes this very easy through the use of the `group_by()` function.

The `summarize()` function

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to compute the average household size by village:

```
interviews %>%
    group_by(village) %>%
    summarize(mean_no_membrs = mean(no_membrs))
```

```
## # A tibble: 3 x 2
##   village  mean_no_membrs
##   <chr>             <dbl>
## 1 Chirodzo           7.08
## 2 God                6.86
## 3 Ruaca              7.57
```

Counting

When working with data, we often want to know the number of observations found for each factor or combination of factors. For this task, **dplyr** provides `count()`. For example, if we wanted to count the number of rows of data for each village, we would do:

```
interviews %>%
    count(village)
```

```
## # A tibble: 3 x 2
##   village      n
##   <chr>    <int>
## 1 Chirodzo    39
## 2 God         43
## 3 Ruaca       49
```

Reshaping with pivot_wider() and pivot_longer()

There are essentially three rules that define a "tidy" dataset:

1. Each variable has its own column
2. Each observation has its own row
3. Each value must have its own cell

Long and wide data formats

In the `interviews` data, each row contains the values of variables associated with each record collected (each interview in the villages), where it is stated that the `key_ID` was "added to provide a unique Id for each observation" and the `instance_ID` "does this as well but it is not as convenient to use."

However, with some inspection, we notice that there are more than one row in the dataset with the same `key_ID` (as seen below). However, the `instanceIDs` associated with these duplicate `key_IDs` are not the same. Thus, we should think of `instanceID` as the unique identifier for observations!

```
interviews %>%
  select(key_ID, village, interview_date, instanceID)
```

```
## # A tibble: 131 x 4
##    key_ID village  interview_date      instanceID
##     <dbl> <chr>    <dttm>              <chr>
##  1      1 God      2016-11-17 00:00:00 uuid:ec241f2c-060
##  2      2 God      2016-11-17 00:00:00 uuid:099de9c9-3e5
##  3      3 God      2016-11-17 00:00:00 uuid:193d7daf-958
##  4      4 God      2016-11-17 00:00:00 uuid:148d1105-778
##  5      5 God      2016-11-17 00:00:00 uuid:2c867811-969
##  6      6 God      2016-11-17 00:00:00 uuid:daa56c91-c8e
##  7      7 God      2016-11-17 00:00:00 uuid:ae20a58d-56f
##  8      8 Chirodzo 2016-11-16 00:00:00 uuid:d6cee930-7be
##  9      9 Chirodzo 2016-11-16 00:00:00 uuid:846103d2-b1c
## 10     10 Chirodzo 2016-12-16 00:00:00 uuid:8f4e49bc-da8
## # i 121 more rows
```

As seen in the code below, for each interview date in each village no `instanceIDs` are the same. Thus, this format is what is called a "long" data format, where each observation occupies only one row in the dataframe.

```
interviews %>%
  filter(village == "Chirodzo") %>%
  select(key_ID, village, interview_date, instanceID) %>%
  sample_n(size = 10)
```

```
## # A tibble: 10 x 4
##    key_ID village  interview_date      instanceID
##     <dbl> <chr>    <dttm>              <chr>
##  1     35 Chirodzo 2016-11-17 00:00:00 uuid:ff7496e7-984
##  2     48 Chirodzo 2016-11-16 00:00:00 uuid:e180899c-761
##  3     46 Chirodzo 2016-11-17 00:00:00 uuid:35f297e0-aa5
##  4     10 Chirodzo 2016-12-16 00:00:00 uuid:8f4e49bc-da8
##  5     56 Chirodzo 2016-11-16 00:00:00 uuid:973c4ac6-f88
##  6      8 Chirodzo 2016-11-16 00:00:00 uuid:d6cee930-7be
##  7     59 Chirodzo 2016-11-16 00:00:00 uuid:1936db62-573
##  8     62 Chirodzo 2016-11-16 00:00:00 uuid:c6597ecc-cc3
```

We notice that the layout or format of the `interviews` data is in a format that adheres to rules 1-3, where

- each column is a variable
- each row is an observation
- each value has its own cell

This is called a "long" data format. But, we notice that each column represents a different variable. In the "longest" data format there would only be three columns, one for the id variable, one for the observed variable, and one for the observed value (of that variable). This data format is quite unsightly and difficult to work with, so you will rarely see it in use.

Questions which warrant different data formats

In interviews, each row contains the values of variables associated with each record (the unit), values such as the village of the respondent, the number of household members, or the type of wall their house had. This format allows for us to make comparisons across individual surveys, but what if we wanted to look at differences in households grouped by different types of housing construction materials?

To facilitate this comparison we would need to create a new table where each row (the unit) was comprised of values of variables associated with housing material (e.g. the `respondent_wall_type`). In practical terms this means the values of the wall construction materials in `respondent_wall_type` (e.g. muddaub, burntbricks, cement, sunbricks) would become the names of column variables and the cells would contain values of `TRUE` or `FALSE`, for whether that house had a wall made of that material.

Exporting data

Now that you have learned how to use **`dplyr`** and **`tidyr`** to wrangle your raw data, you may want to export these new data sets to share them with your collaborators or for archival purposes.

Similar to the `read_csv()` function used for reading CSV files into R, there is a `write_csv()` function that generates CSV files from dataframes.

Before using `write_csv()`, we are going to create a new folder, `data_output`, in our working directory that will store this generated dataset. We don't want to write generated datasets in the same directory as our raw data. It's good practice to keep them separate. The `data` folder should only contain the raw, unaltered data, and should be left alone to make sure we don't delete or modify it. In contrast, our script will generate the contents of the `data_output` directory, so even if the files it contains are deleted, we can always re-generate them.