Data Visualization with R

OSDC MiniSeries: Reproducible Research

These courses were created from curriculum originally developed by The Carpentries.

The aim of the OSDC Mini-series workshops is to teach researchers basic concepts, skills, and tools for working with data so that they can get more done in less time, and with less pain. The lessons below were designed for those interested in working with social sciences data in R.

This 2-hour course teaches students to create customized data visualizations in R. Using the ggplot2 package, students will produce plots, histograms, density distribution, and other useful visualizations to bring their data to life.

Getting Started

The teaching style of these workshops is hands-on, so participants are encouraged to use their own computers to ensure the proper setup of tools for an efficient workflow.

To get started, follow the directions in the "Setup" tab to download data to your computer and follow any installation instructions.

Prerequisites

This lesson requires a working copy of **R** and **RStudio**. To most effectively use these materials, please make sure to install everything before working through this lesson.

This lesson also assumes basic familiarity with **R** and the **RStudio** environment. Participants are expected to have completed the Introduction to R mini prior to enrolling in this course, or to have an equivalent level of understanding of the material contained in that lesson.

For Instructors

If you are teaching this lesson in a workshop, please see the Instructor notes.

Schedule

	Setup	Install R and RStudio Download files required for the lesson
00:00	Before we Start	How do I set up my working directory? How do I import data? How to install packages?
00:20	Data Visualization with ggplot2	What are the components of a ggplot? How do I create scatterplots, boxplots, and barplots? How can I change the aesthetics (color, transparency, etc.) of my plot? How can I create multiple plots at once?
02:00	Finish	

(The actual schedule may vary slightly depending on the topics and exercises chosen by the instructor.)

Setup instructions

Overview

Questions

- How to install R and RStudio?

Objectives

- Install latest version of R.
- Install latest version of RStudio.

R and **RStudio** are separate downloads and installations. R is the underlying statistical computing environment, but using R alone is no fun. RStudio is a graphical integrated development environment (IDE) that makes using R much easier and more interactive. You need to install R before you install RStudio. After installing both programs, you will need to install the **tidyverse** package from within RStudio. Follow the instructions below for your operating system, and then follow the instructions to install **tidyverse**.

Windows

If you already have R and RStudio installed

- Open RStudio, and click on 'Help' > 'Check for updates'. If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check which version of R you are using, start RStudio and the first thing that appears in the console indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go to the CRAN website and check whether a more recent version is available. If so, please download and install it. You can check here for more information on how to remove old versions from your system if you wish to do so.

If you don't have R and RStudio installed

- Download R from the CRAN website.
- Run the .exe file that was just downloaded
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Windows Vista/7/8/10** (where x, y, and z represent version numbers)
- Double click the file to install it
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

MacOS

If you already have R and RStudio installed

- Open RStudio, and click on 'Help' > 'Check for updates'. If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check the version of R you are using, start RStudio and the first thing that appears on the terminal indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it.

If you don't have R and RStudio installed

- Download R from the CRAN website.
- Select the .pkg file for the latest R version
- Double click on the downloaded file to install R
- It is also a good idea to install XQuartz (needed by some packages)
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Mac OS X 10.6+ (64-bit)** (where x, y, and z represent version numbers)

- Double click the file to install RStudio
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

Linux

- Follow the instructions for your distribution from CRAN, they provide information to get the most recent version of R for common distributions. For most distributions, you could use your package manager (e.g., for Debian/Ubuntu run sudo apt-get install r-base, and for Fedora sudo yum install R), but we don't recommend this approach as the versions provided by this are usually out of date. In any case, make sure you have at least R 3.3.1.
- Go to the RStudio download page
- Under *Installers* select the version that matches your distribution, and install it with your preferred method (e.g., with Debian/Ubuntu sudo dpkg -i rstudio-x.yy.zzz-amd64.deb at the terminal).
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

Before We Start

Overview

Teaching: 20 min
Questions

- How do I set up my working directory?
- How do I import data?
- How do I install packages?

Objectives

- Learn the benefits of using R for data management.
- Learn to install R packages.
- Learn to import data.

Getting set up

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder called the **working directory**. All the scripts within this folder can then use relative paths to files. Relative paths indicate where inside the project a file is located (as opposed to absolute paths, which point to where a file is on a specific computer). Working this way makes it

a lot easier to move your project around on your computer and share it with others without having to directly modify file paths in the individual scripts.

RStudio provides a helpful set of tools to do this through its “Projects” interface, which not only creates a working directory for you but also remembers its location (allowing you to quickly navigate to it). The interface also (optionally) preserves custom settings and open files to make it easier to resume work after a break.

Create a new project

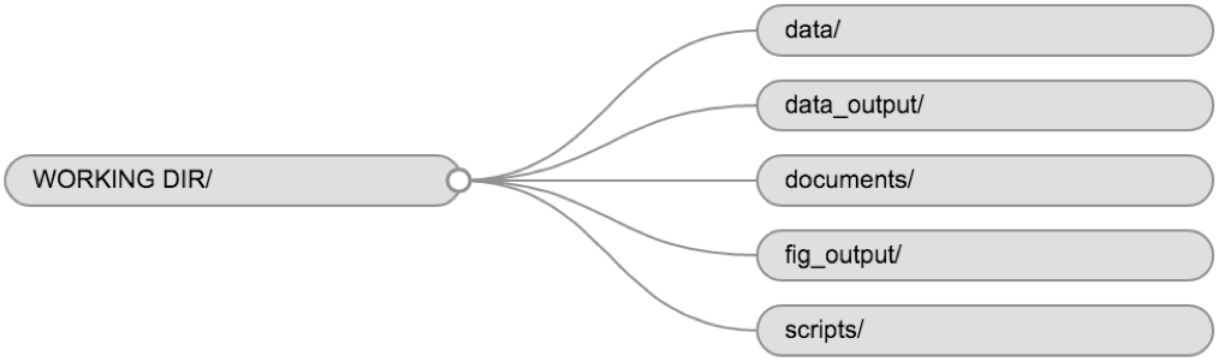
- Under the File menu, click on New project, choose New directory, then New project
- Enter a name for this new folder (or “directory”) and choose a convenient location for it. This will be your working directory for the rest of the day (e.g., ~/data-viz-with-r)
- Click on Create project
- Create a new file where we will type our scripts. Go to File > New File > R script. Click the save icon on your toolbar and save your script as “script.R”.

Organizing your working directory

Using a consistent folder structure across your projects will help keep things organized and make it easy to find/file things in the future. This can be especially helpful when you have multiple projects. In general, you might create directories (folders) for **scripts**, **data**, and **documents**. Here are some examples of suggested directories:

- `data/` Use this folder to store your raw data and intermediate datasets. You should always keep a copy of your raw data accessible and do as much of your data cleanup and preprocessing programmatically (i.e., with scripts, rather than manually) as possible.
- `data_output/` When you need to modify your raw data, it might be useful to store the modified versions of the datasets in a different folder.
- `documents/` Used for outlines, drafts, and other text.
- `fig_output/` This folder can store the graphics that are generated by your scripts.
- `scripts/` A place to keep your R scripts for different analyses or plotting.

You may want additional directories or subdirectories depending on your project needs, but these should form the backbone of your working directory.



The working directory

The working directory is an important concept to understand. It is the place where R will look for and save files.

Using RStudio projects makes this easy and ensures that your working directory is set up properly. If you need to check it, you can use `getwd()`. If for some reason your working directory is not what it should be, you can change it in the RStudio interface by navigating in the file browser to where your working directory should be, clicking on the blue gear icon "More", and selecting "Set As Working Directory".

Alternatively, you can use `setwd("/path/to/working/directory")` to reset your working directory. However, your scripts should not include this line, because it will fail on someone else's computer.

Downloading the data and getting set up

For this lesson we will use the following folders in our working directory: `data/`, `data_output/`, and `fig_output/`. Let's write them all in lowercase to be consistent. We can create them using the RStudio interface by clicking on the "New Folder" button in the file pane (bottom right), or directly from R by typing at console:

```
dir.create("data")
dir.create("data_output")
dir.create("fig_output")
```

Go to the Figshare page for this curriculum and download the dataset called `"SAFI_clean.csv"`. The direct download link is: https://ndownloader.figshare.com/files/11492171. Place this downloaded file in the `data/` you just created. You can do this directly from R by copying and pasting this in your terminal (your instructor can place this chunk of code in the Etherpad):

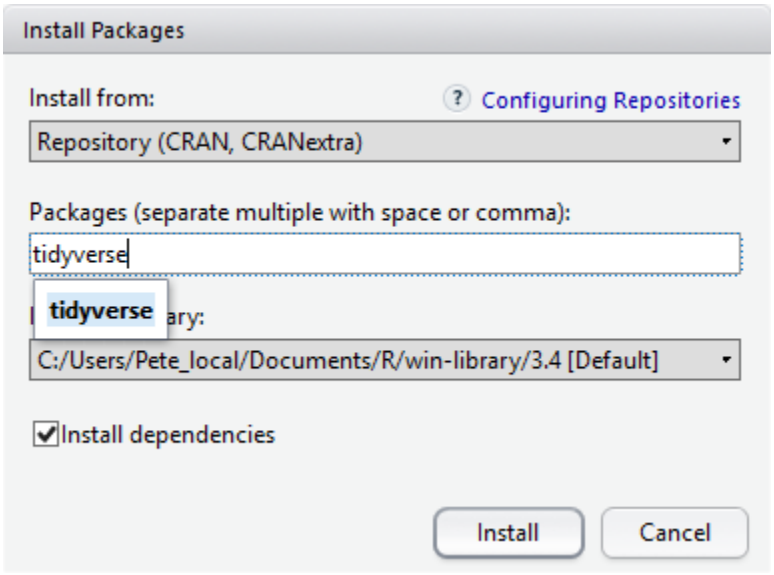
```
download.file("https://ndownloader.figshare.com/files/11492171",
              "data/SAFI_clean.csv", mode = "wb")
```

Installing additional packages using the packages tab

In addition to the core R installation, there are more than 10,000 additional packages that can be used to extend the functionality of R. Many of these have been written by R users and have been made available in central repositories for anyone to download and install into their own R environment.

You can see if you have a package installed by looking in the packages tab (on the lower-right by default). You can also type the command `installed.packages()` into the console and examine the output.

Additional packages can be installed from the 'packages' tab. On the packages tab, click the 'Install' icon and start typing the name of the package you want in the text box. As you type, packages matching your starting characters will be displayed in a drop-down list so that you can select them.



At the bottom of the Install Packages window is a check box to 'Install' dependencies. This is ticked by default, which is usually what you want. Packages can (and do) make use of functionality built into other packages, so for the functionality contained in the package you are installing to work properly, there may be other packages which have to be installed with them. The 'Install dependencies' option makes sure that this happens.

Note: Because the install process accesses the CRAN repository, you will need an Internet connection to install packages.

Installing additional packages using R code

If you were watching the console window when you started the install of 'tidyverse', you may have noticed that the line

```
install.packages("tidyverse")
```

was written to the console before the start of the installation messages. You could also have installed the **tidyverse** packages by running this command directly at the R terminal.

To easily access the documentation for a package within R or RStudio, use `help(package = "package_name")`.

Key Points

- Use RStudio to write and run R programs.
- Use `install.packages()` to install packages (libraries).

Load the SAFI dataset

SAFI (Studying African Farmer-Led Irrigation) is a study looking at farming and irrigation methods in Tanzania and Mozambique. The survey data was collected through interviews conducted between November 2016 and June 2017. For this lesson, we will be using a subset of the available data. (For information about the full teaching dataset used in other lessons in this workshop, see the dataset description.)

We are using a subset of the cleaned version of the dataset that was produced through cleaning in OpenRefine. Each row holds information for a single interview respondent, and the columns represent:

column_name	description
key_id	Added to provide a unique Id for each observation. (The InstanceID field does this as well but it is not as convenient to use)
village	Village name
interview_date	Date of interview
no_membrs	How many members in the household?
years_liv	How many years have you been living in this village or neighboring village?

respondent_wall_type	What type of walls does their house have (from list)
rooms	How many rooms in the main house are used for sleeping?
memb_assoc	Are you a member of an irrigation association?
affect_conflicts	Have you been affected by conflicts with other irrigators in the area?
liv_count	Number of livestock owned.
items_owned	Which of the following items are owned by the household? (list)
no_meals	How many meals do people in your household normally eat in a day?
months_lack_food	Indicate which months, In the last 12 months have you faced a situation when you did not have enough food to feed the household?
instanceID	Unique identifier for the form data submission

Just as we did in the previous session, we are going to load the data in R's memory using the function `read_csv()` from the **readr** package, which is part of the **tidyverse**.

Before we can use the `read_csv()` function, we need to load the package. Also, if you recall, the missing data is encoded as "NULL" in the dataset. We'll tell R in the function to automatically convert all the "NULL" entries in the dataset into `NA`.

```
library(tidyverse)
interviews <- read_csv("data/SAFI_clean.csv", na = "NULL")
```

Exporting data

In the previous session (Introduction to R), we also created a new data file from a subset of the SAFI data. For this exercise, we're going to recreate that data file, export it to the `data_output` folder we just created, and then import it again to use in our visualization exercises.

Note: Technically, this last step is unnecessary since the object containing our new data set, `interviews_plotting`, is already stored in our global environment. The reason we're loading it again is to simulate the steps that we would ordinarily take if we had created the object in a previous session, and then started a new project from scratch.

```
## Recreate interviews_plotting dataset from previous exercise
```

```
interviews_plotting <- interviews %>%
  ## pivot wider by items_owned
  separate_rows(items_owned, sep = ";") %>%
  ## if there were no items listed, changing NA to no_listed_items
  replace_na(list(items_owned = "no_listed_items")) %>%
  mutate(items_owned_logical = TRUE) %>%
  pivot_wider(names_from = items_owned,
              values_from = items_owned_logical,
              values_fill = list(items_owned_logical = FALSE)) %>%
  ## pivot wider by months_lack_food
  separate_rows(months_lack_food, sep = ";") %>%
  mutate(months_lack_food_logical = TRUE) %>%
  pivot_wider(names_from = months_lack_food,
              values_from = months_lack_food_logical,
              values_fill = list(months_lack_food_logical =
FALSE))%>%
  ## add some summary columns
  mutate(number_months_lack_food = rowSums(select(., Jan:May))) %>%
  mutate(number_items = rowSums(select(., bicycle:car)))
```

Now we can save this data frame to our `data_output` directory.

```
write_csv(interviews_plotting, file =
"data_output/interviews_plotting.csv")
```

Shortcut

For those who didn't participate in the previous session (Introduction to R), feel free to download the revised dataset directly using the code below.

```
## download the revised dataset
download.file("https://cmu-lib.github.io/os-workshops/reproducible-re
search/data/interviews_plotting.csv",
     "data_output/interviews_plotting.csv", mode = "wb")

## store the revised dataset in a new variable
interviews_plotting <-
read_csv("data_output/interviews_plotting.csv")
```

Data Visualization with ggplot2

Overview

Teaching: 70 min
Exercises: 30 min
Questions

- What are the components of a ggplot?
- How do I create scatterplots, boxplots, and barplots?
- How can I change the aesthetics (e.g., color, transparency) of my plot?
- How can I create multiple plots at once?

Objectives

- Produce scatter plots, boxplots, and barplots using ggplot.
- Set universal plot settings.
- Describe what faceting is and apply faceting in ggplot.
- Modify the aesthetics of an existing ggplot plot (including axis labels and color).
- Build complex and customized plots from data in a data frame.

We start by loading the required package. **ggplot2** is also included in the **tidyverse** package.

```
library(tidyverse)
```

Next, we'll load the data set we just created in the previous section. (Remember, this step is not strictly necessary since the `interviews_plotting` object is already stored in our global environment, but it's good practice.)

```
interviews_plotting <-
read_csv("data_output/interviews_plotting.csv")
```

Plotting with ggplot2

ggplot2 is a plotting package that makes it simple to create complex plots from data stored in a data frame. It provides a programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

ggplot2 functions work best with data in the 'long' format, i.e., a column for every dimension, and a row for every observation. Well-structured data will save you lots of time when making figures with **ggplot2**.

ggplot graphics are built step by step by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

To build a **ggplot**, we will use the following basic template that can be used for different types of plots:

```
<DATA> %>%
    ggplot(aes(<MAPPINGS>)) +
    <GEOM_FUNCTION>()
```

Remember from the last lesson that the pipe operator `%>%` places the result of the previous line(s) into the first argument of the function. **ggplot** is a function that expects a data frame to be the first argument. This allows for us to change from specifying the `data =` argument within the **ggplot** function and instead pipe the data into the function.

- use the `ggplot()` function and bind the plot to a specific data frame.

```
interviews_plotting %>%
    ggplot()
```

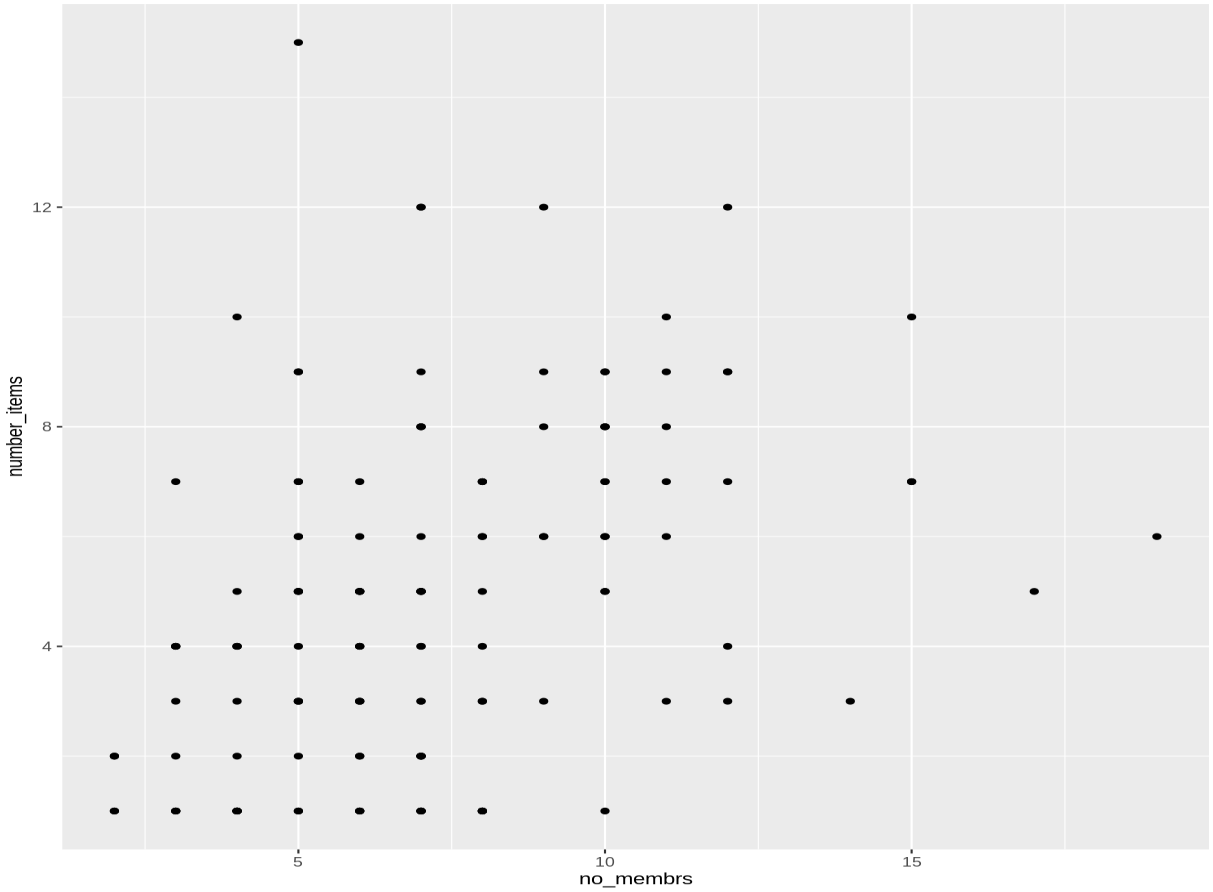
- define a mapping (using the aesthetic (`aes`) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g. as x/y positions or characteristics such as size, shape, color, etc.

```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items))
```

- add ‘geoms’ – graphical representations of the data in the plot (points, lines, bars). **ggplot2** offers many different geoms; we will use some common ones today, including:
    - `geom_point()` for scatter plots, dot plots, etc.
    - `geom_boxplot()` for, well, boxplots!
    - `geom_line()` for trend lines, time series, etc.

To add a geom to the plot use the `+` operator. Because we have two continuous variables, let’s use `geom_point()` first:

```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items)) +
    geom_point()
```



The `+` in the **ggplot2** package is particularly useful because it allows you to modify existing **ggplot** objects. This means you can easily set up plot templates and conveniently explore different types of plots, so the above plot can also be generated with code like this, similar to the “intermediate steps” approach in the previous lesson:

```
# Assign plot to a variable
interviews_plot <- interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items))

# Draw the plot as a dot plot
interviews_plot +
    geom_point()
```

Notes

- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x- and y-axis mapping you set up in `aes()`.

- You can also specify mappings for a given geom independently of the mapping defined globally in the `ggplot()` function.
- The `+` sign used to add new layers must be placed at the end of the line containing the *previous* layer. If, instead, the `+` sign is added at the beginning of the line containing the new layer, **ggplot2** will not add the new layer and will return an error message.

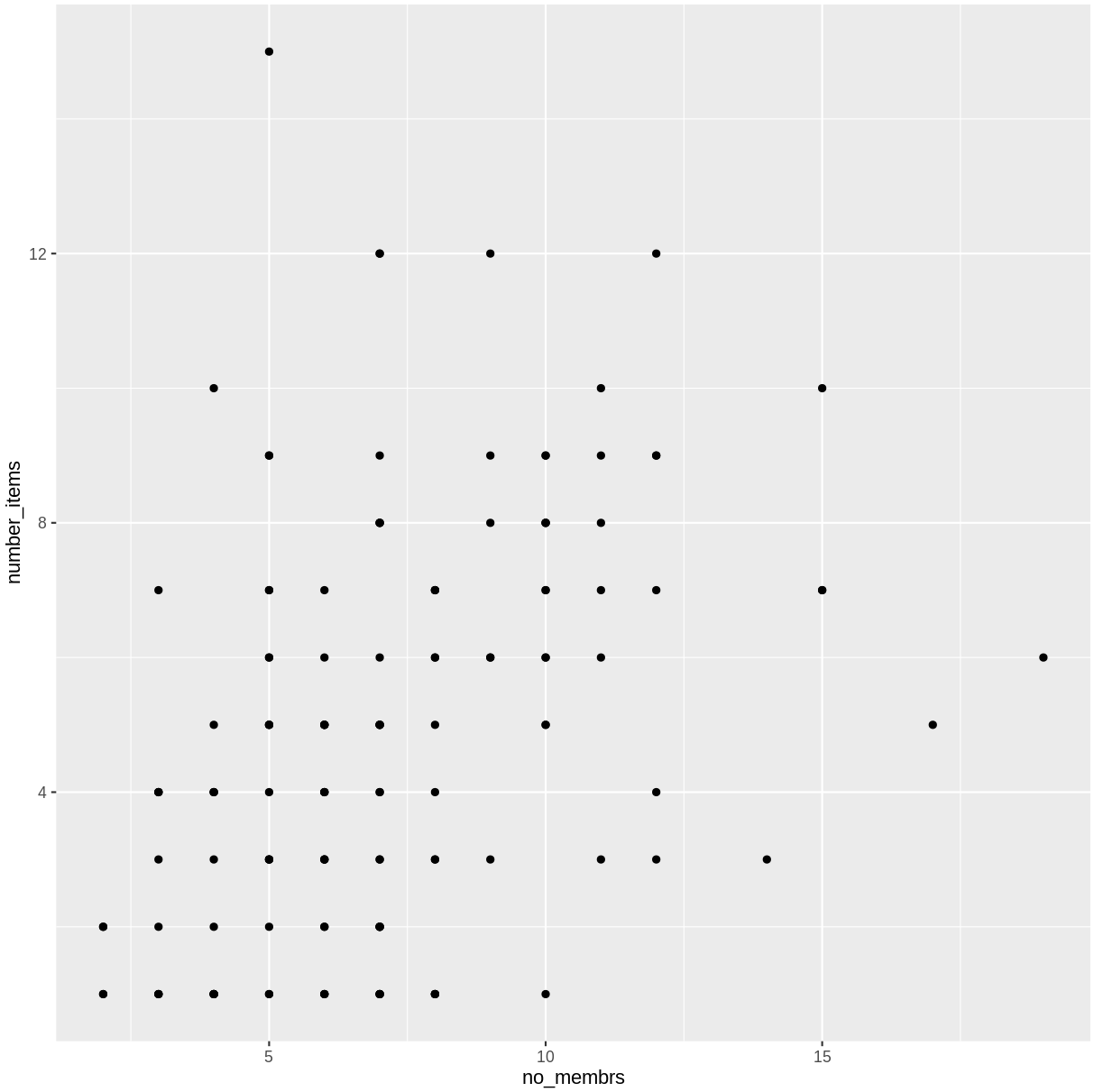
```
## This is the correct syntax for adding layers
interviews_plot +
    geom_point()

## This will not add the new layer and will return an error message
interviews_plot
+ geom_point()
```

Building your plots iteratively

Building plots with **ggplot2** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items)) +
    geom_point()
```



Then, we start modifying this plot to extract more information from it. For instance, when inspecting the plot we notice that points only appear at the intersection of whole numbers of `no_membrs` and `number_items`. Also, from a rough estimate, it looks like there are far fewer dots on the plot than there rows in our dataframe. This should lead us to believe that there may be multiple observations plotted on top of each other (e.g. three observations where `no_membrs` is 3 and `number_items` is 1).

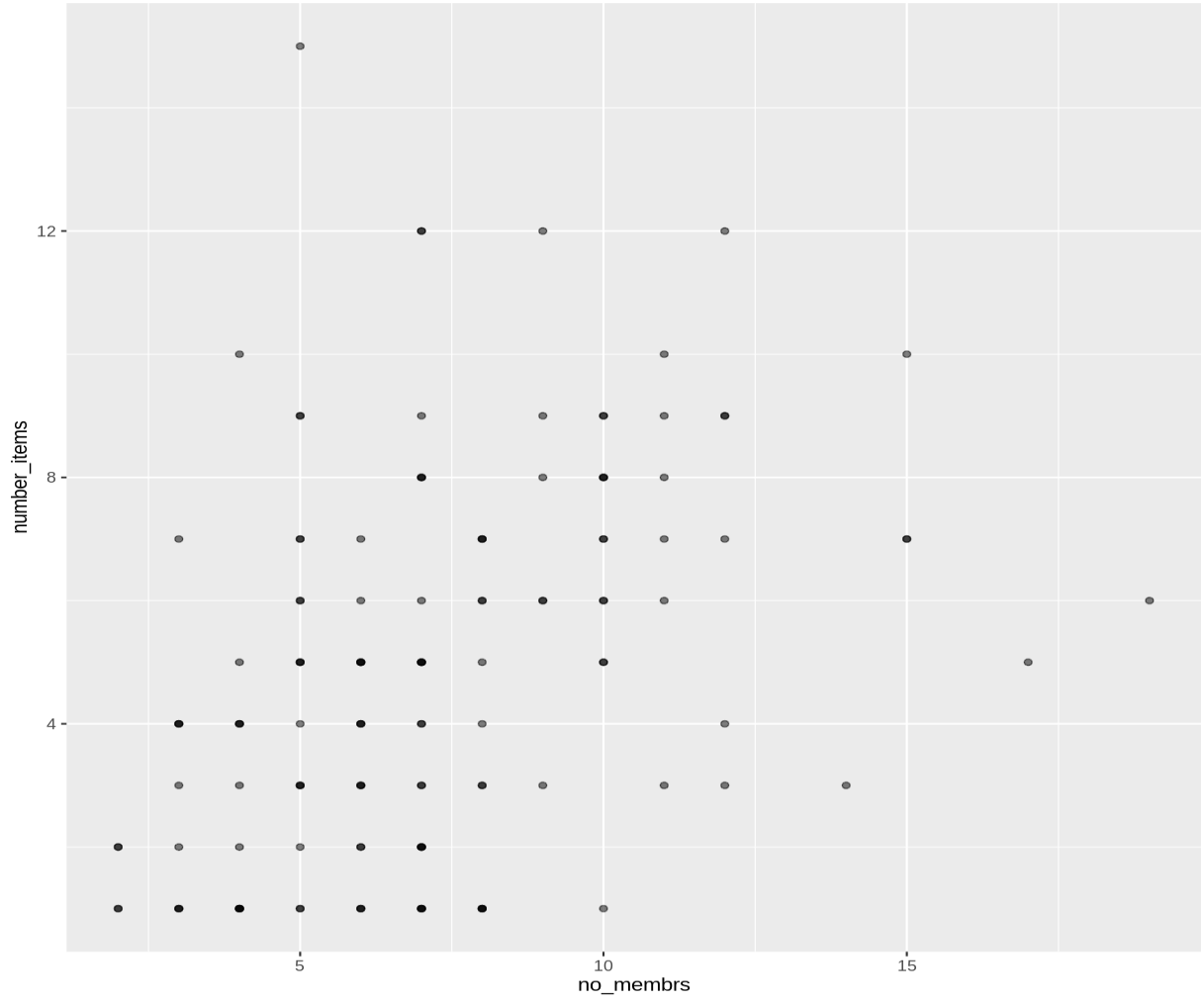
There are two main ways to alleviate overplotting issues:

1. changing the transparency of the points
2. jittering the location of the points

Let's first explore option 1, changing the transparency of the points. What we mean when we say "transparency" we mean the opacity of point, or your ability to see through the point. We can control the transparency of the points with the `alpha` argument to `geom_point`. Values of `alpha` range from 0 to 1, with lower values corresponding to more transparent colors (an `alpha` of 1 is the default value).

Here, we change the `alpha` to 0.5, in an attempt to help fix the overplotting. While the overplotting isn't solved, adding transparency begins to address this problem, as the points where there are overlapping observations are darker (as opposed to lighter gray):

```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items)) +
    geom_point(alpha = 0.5)
```

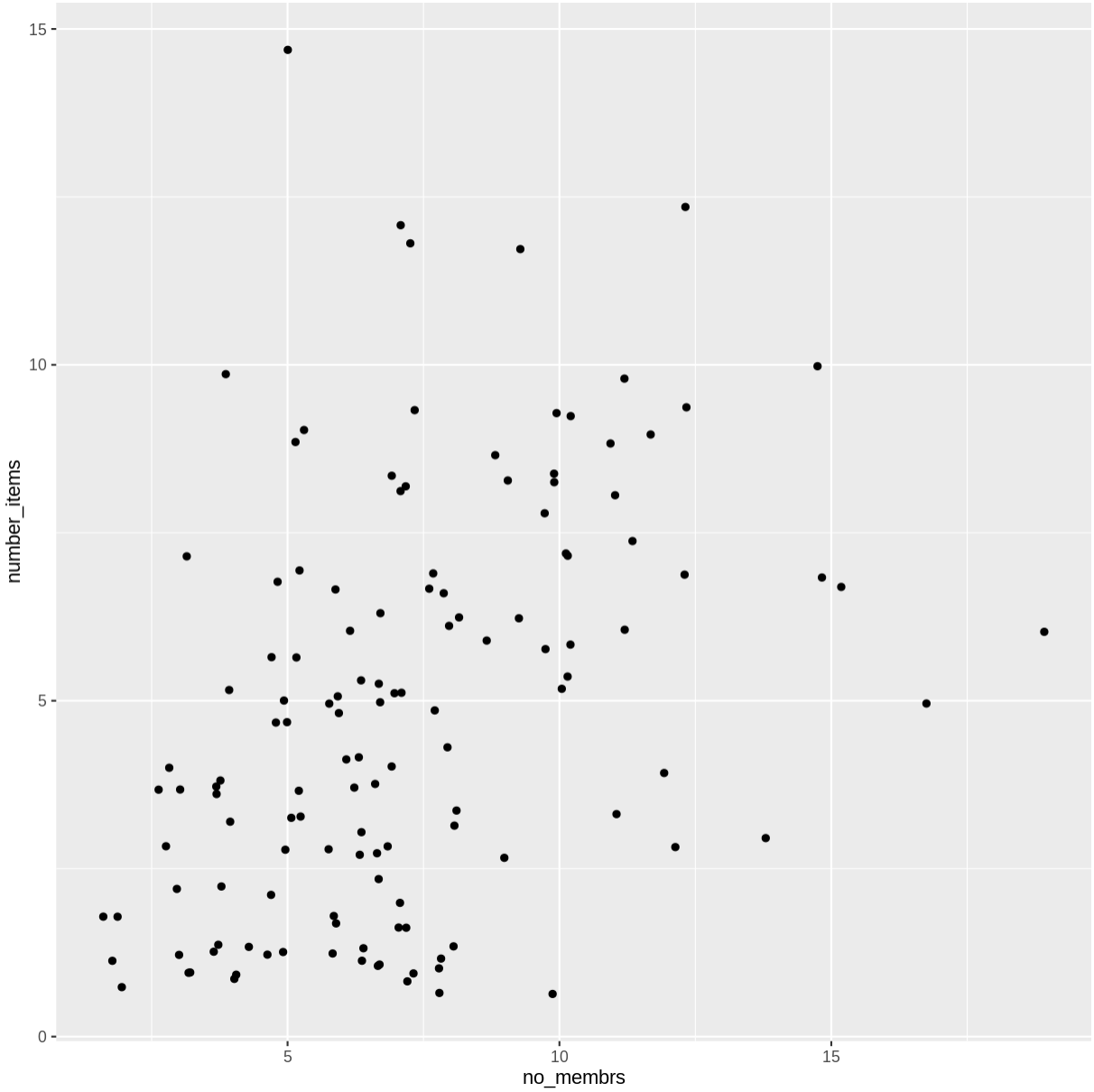


That only helped a little bit with the overplotting problem, so let's try option two. We can jitter the points on the plot, so that we can see each point in the locations where there are overlapping points. Jittering introduces a little bit of randomness into the position of our points. You can think

of this process as taking the overplotted graph and giving it a tiny shake. The points will move a little bit side-to-side and up-and-down, but their position from the original plot won't dramatically change.

We can jitter our points using the `geom_jitter()` function instead of the `geom_point()` function, as seen below:

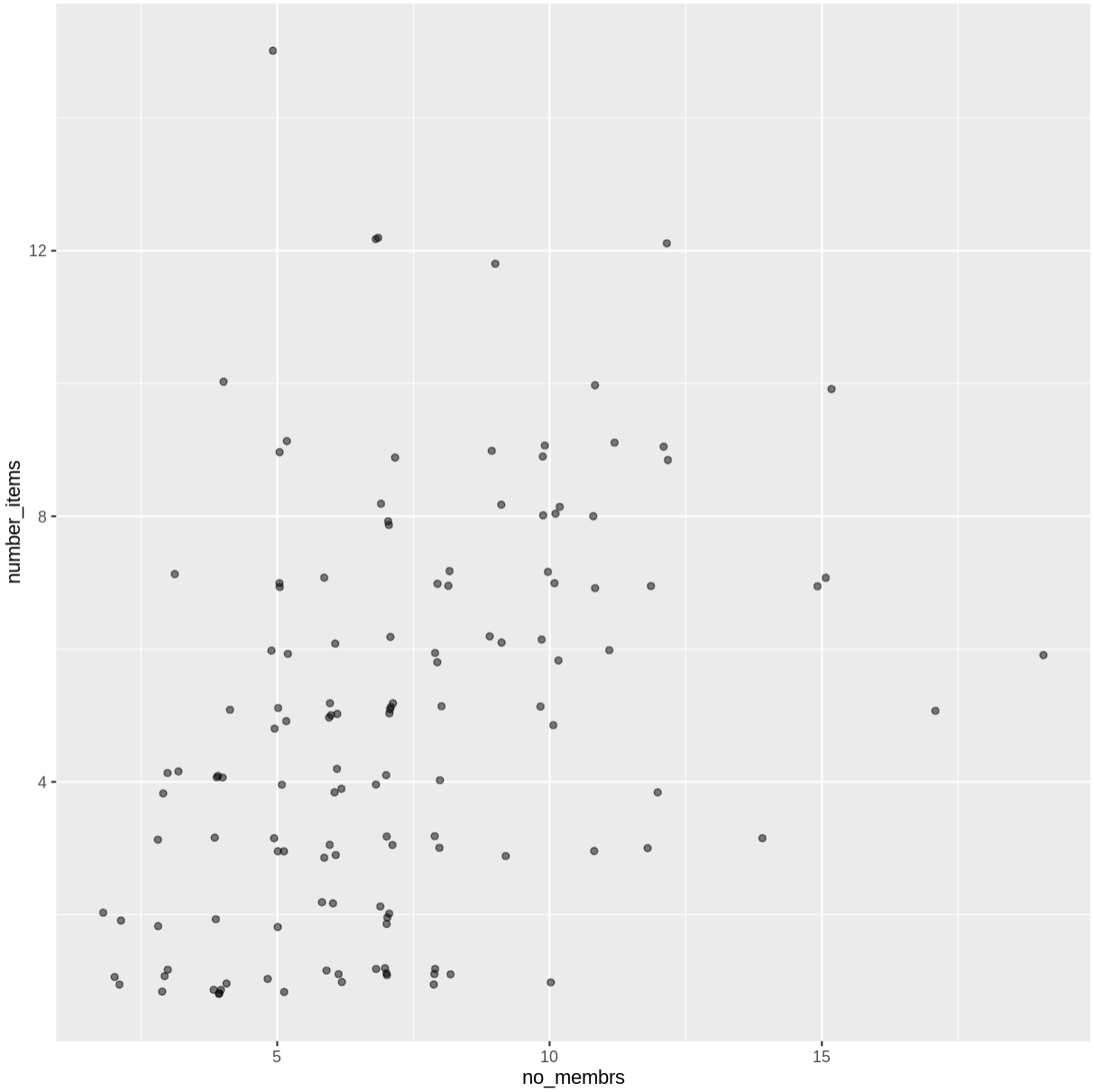
```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items)) +
    geom_jitter()
```



The `geom_jitter()` function allows for us to specify the amount of random motion in the jitter, using the `width` and `height` arguments. When we don't specify values

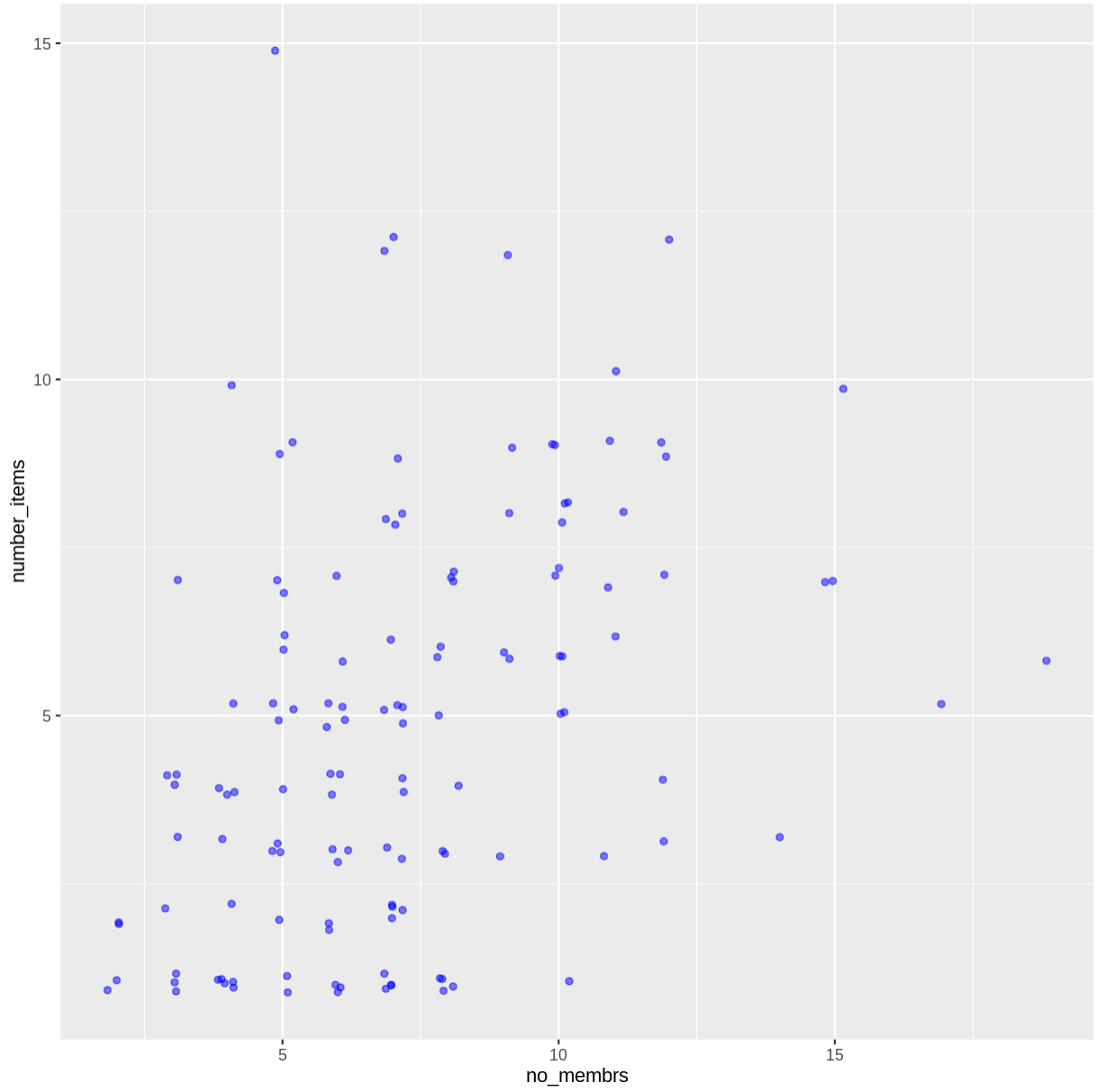
for `width` and `height`, `geom_jitter()` defaults to 40% of the resolution of the data (the smallest change that can be measured). Hence, if we would like *less* spread in our jitter than was default, we should pick values between 0.1 and 0.4. Experiment with the values to see how your plot changes.

```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items)) +
    geom_jitter(alpha = 0.5,
                width = 0.2,
                height = 0.2)
```



For our final change, we can also add colors for all the points by specifying a `color` argument inside the `geom_jitter()` function:

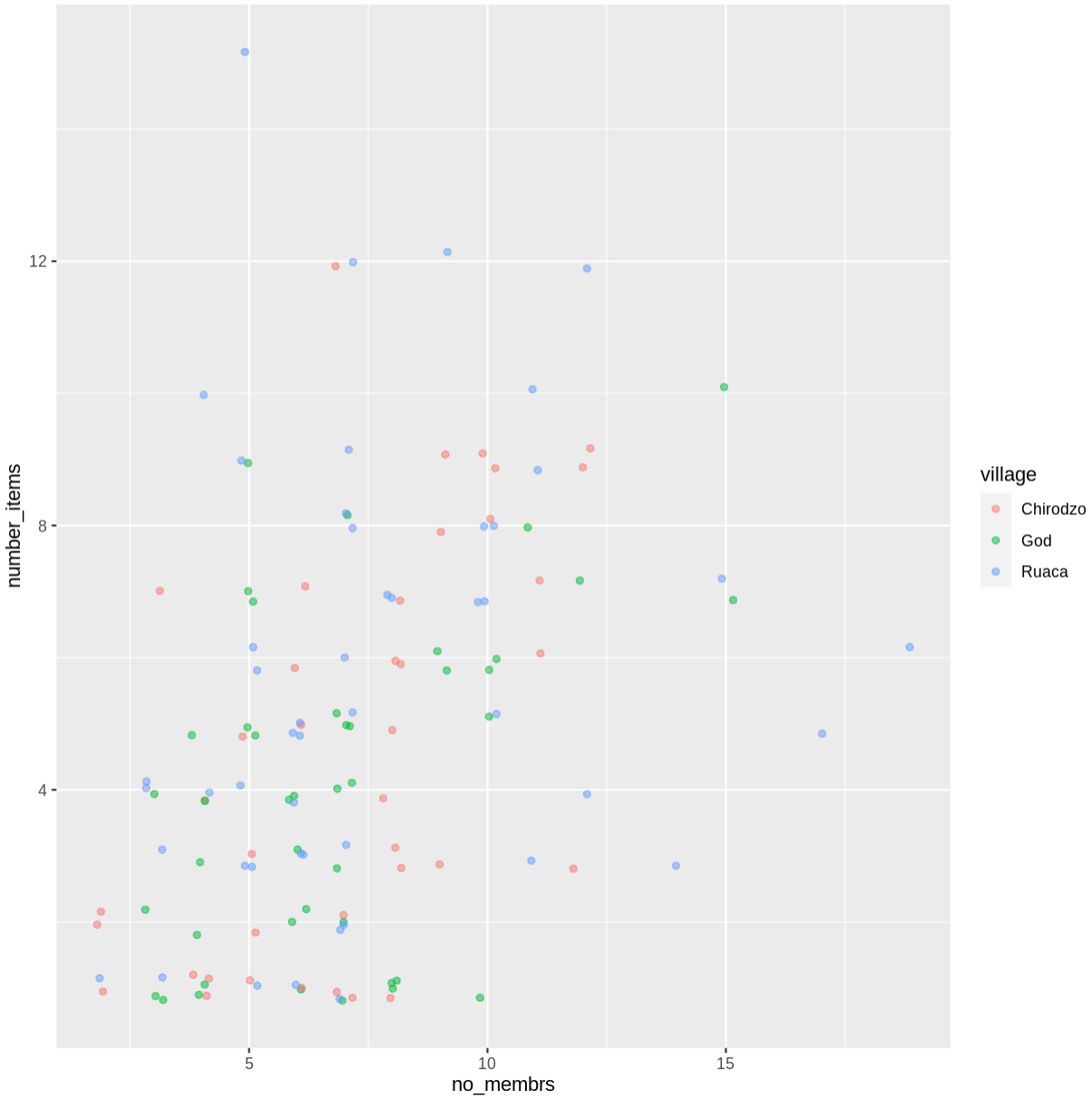
```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items)) +
    geom_jitter(alpha = 0.5,
                color = "blue",
                width = 0.2,
                height = 0.2)
```



To color each species in the plot differently, you could use a vector as an input to the argument `color`. However, because we are now mapping features of the data to a color, instead of setting one color for all points, the color of the points now needs to be set inside a call to the `aes` function. When we map a variable in our data to the color of the points, **ggplot2** will provide a different color corresponding to the different values of the variable. We will continue to

specify the value of `alpha`, `width`, and `height` outside of the `aes` function because we are using the same value for every point. Here is an example where we color points by the `village` of the observation:

```
interviews_plotting %>%
    ggplot(aes(x = no_membrs, y = number_items)) +
    geom_jitter(aes(color = village), alpha = 0.5, width = 0.2,
height = 0.2)
```

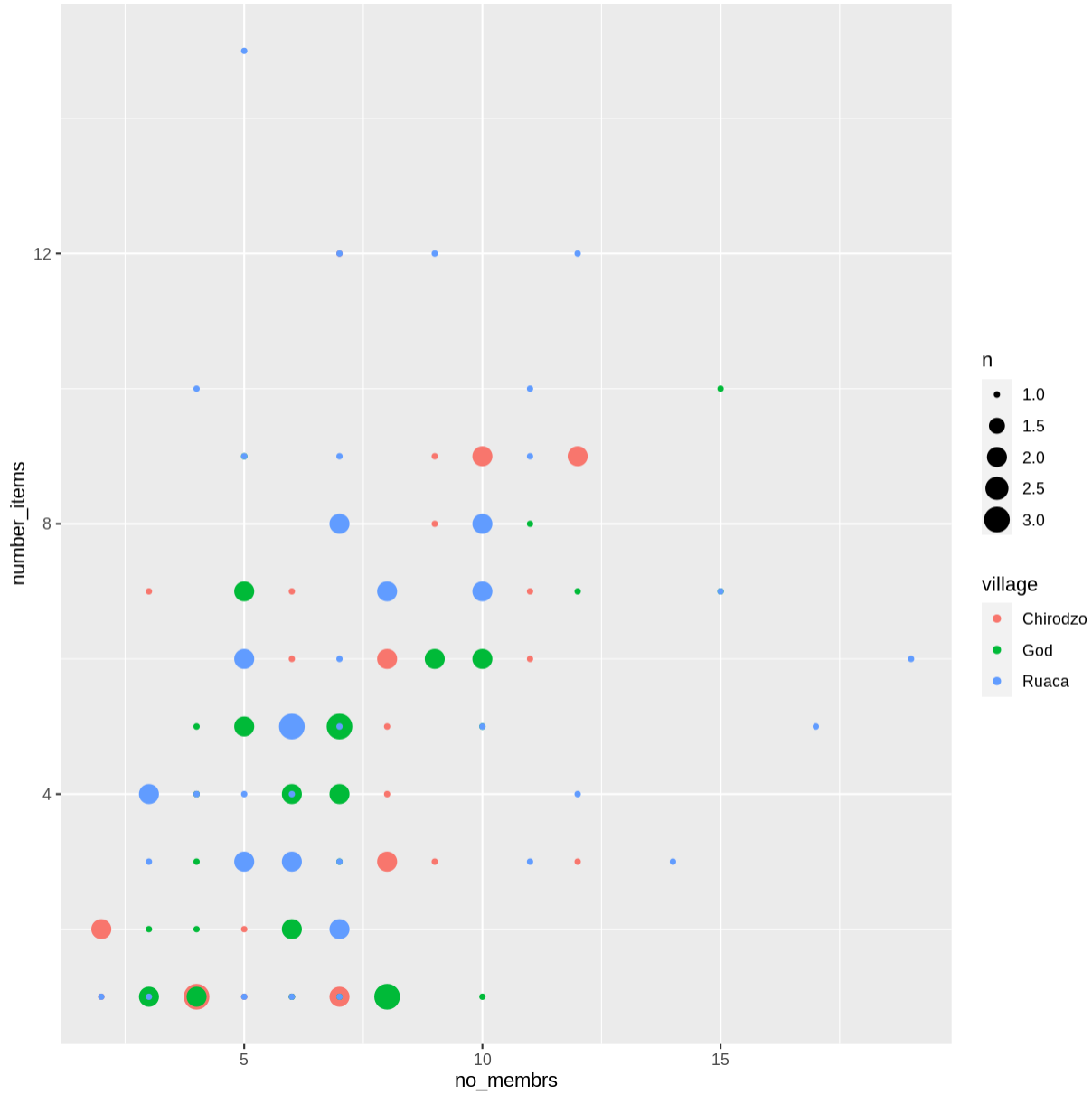


There appears to be a positive trend between number of household members and number of items owned (from the list provided). Additionally, this trend does not appear to be different by village.

Notes

As you will learn, there are multiple ways to plot the relationship between variables. Another way to plot data with overlapping points is to use the `geom_count` plotting function.
The `geom_count()` function makes the size of each point representative of the number of data items of that type and the legend gives point sizes associated to particular numbers of items.

```
interviews_plotting %>%
   ggplot(aes(x = no_membrs, y = number_items, color = village)) +
   geom_count()
```



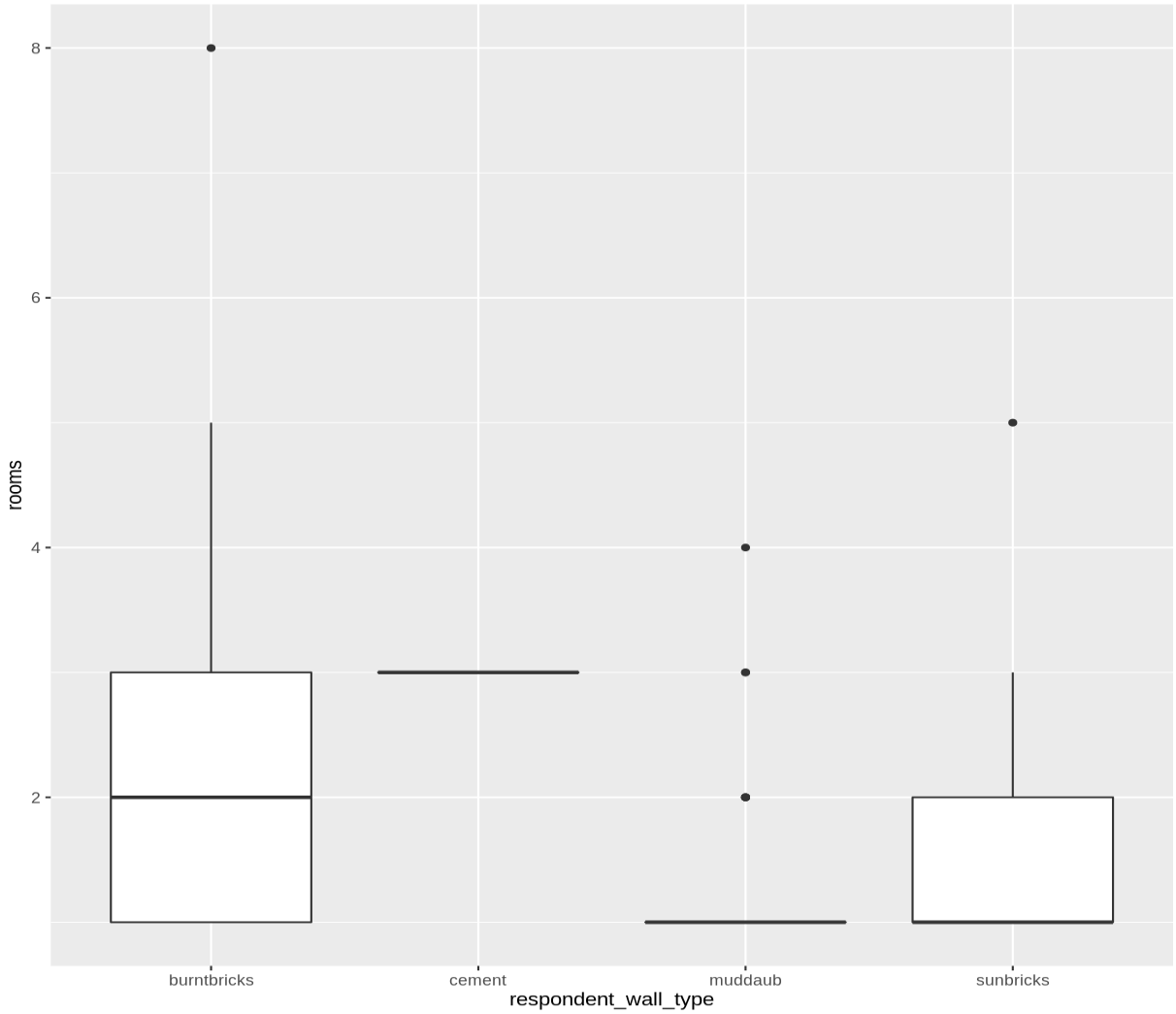
Exercise

Use what you just learned to create a scatter plot of `rooms` by `village` with the `respondent_wall_type` showing in different colors. Does this seem like a good way to display the relationship between these variables? What other kinds of plots might you use to show this type of data?

Boxplot

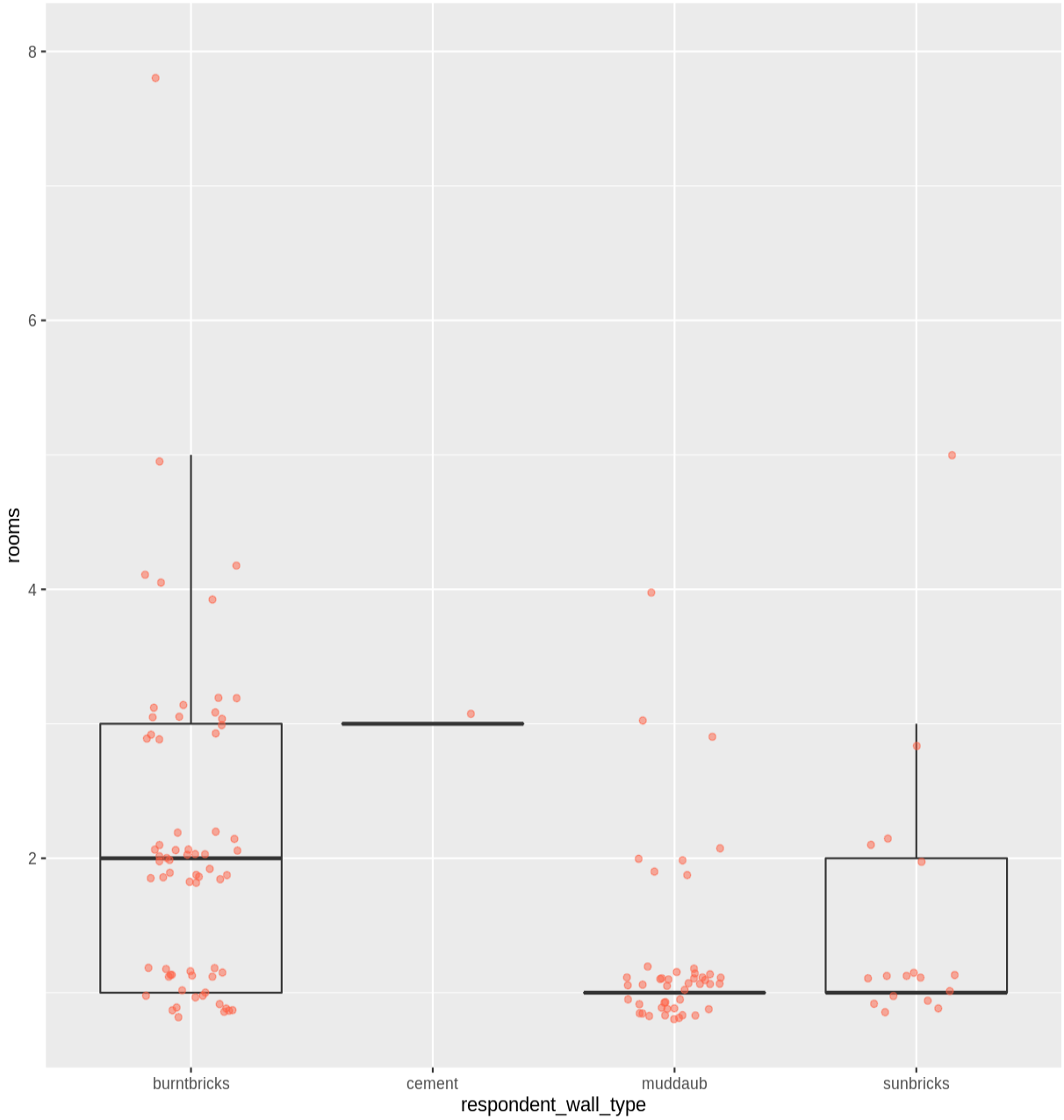
We can use boxplots to visualize the distribution of rooms for each wall type:

```
interviews_plotting %>%
    ggplot(aes(x = respondent_wall_type, y = rooms)) +
    geom_boxplot()
```



By adding points to a boxplot, we can have a better idea of the number of measurements and of their distribution:

```
interviews_plotting %>%
    ggplot(aes(x = respondent_wall_type, y = rooms)) +
    geom_boxplot(alpha = 0) +
    geom_jitter(alpha = 0.5,
                color = "tomato",
                width = 0.2,
                height = 0.2)
```



We can see that muddaub houses and sunbrick houses tend to be smaller than burntbrick houses.

Notice how the boxplot layer is behind the jitter layer? What do you need to change in the code to put the boxplot in behind the points such that it's not hidden?

Exercise 1

Boxplots are useful summaries, but hide the *shape* of the distribution. For example, if the distribution is bimodal, we would not see it in a boxplot. An alternative to the boxplot is the violin plot, where the shape (of the density of points) is drawn.

- Replace the box plot with a violin plot; see `geom_violin()`.

Exercise 2

So far, we've looked at the distribution of room number within wall type. Try making a new plot to explore the distribution of another variable within wall type.

- Create a boxplot for `liv_count` for each wall type. Overlay the boxplot layer on a jitter layer to show actual measurements.

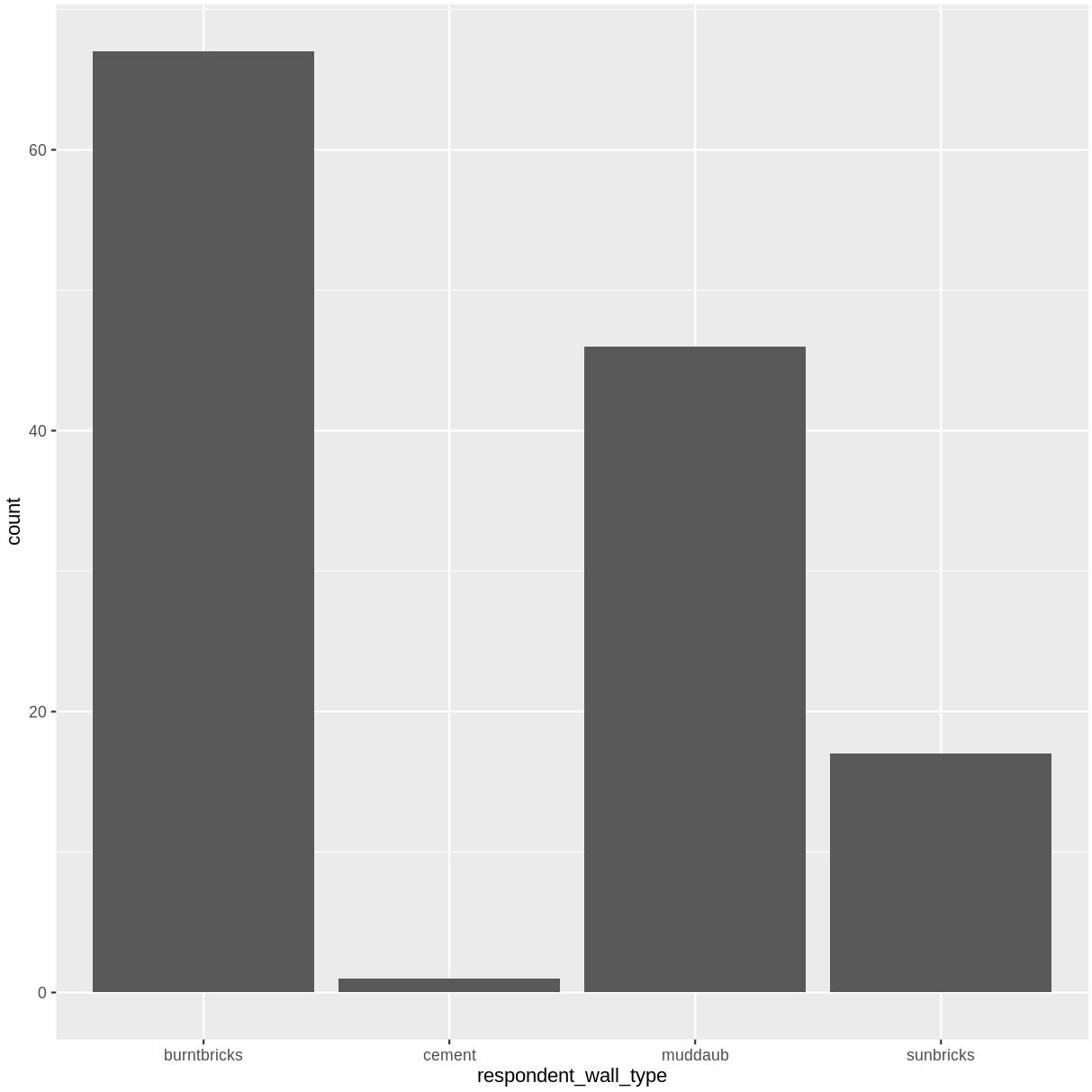
Exercise 3

- Add color to the data points on your boxplot according to whether the respondent is a member of an irrigation association (`memb_assoc`).

Barplots

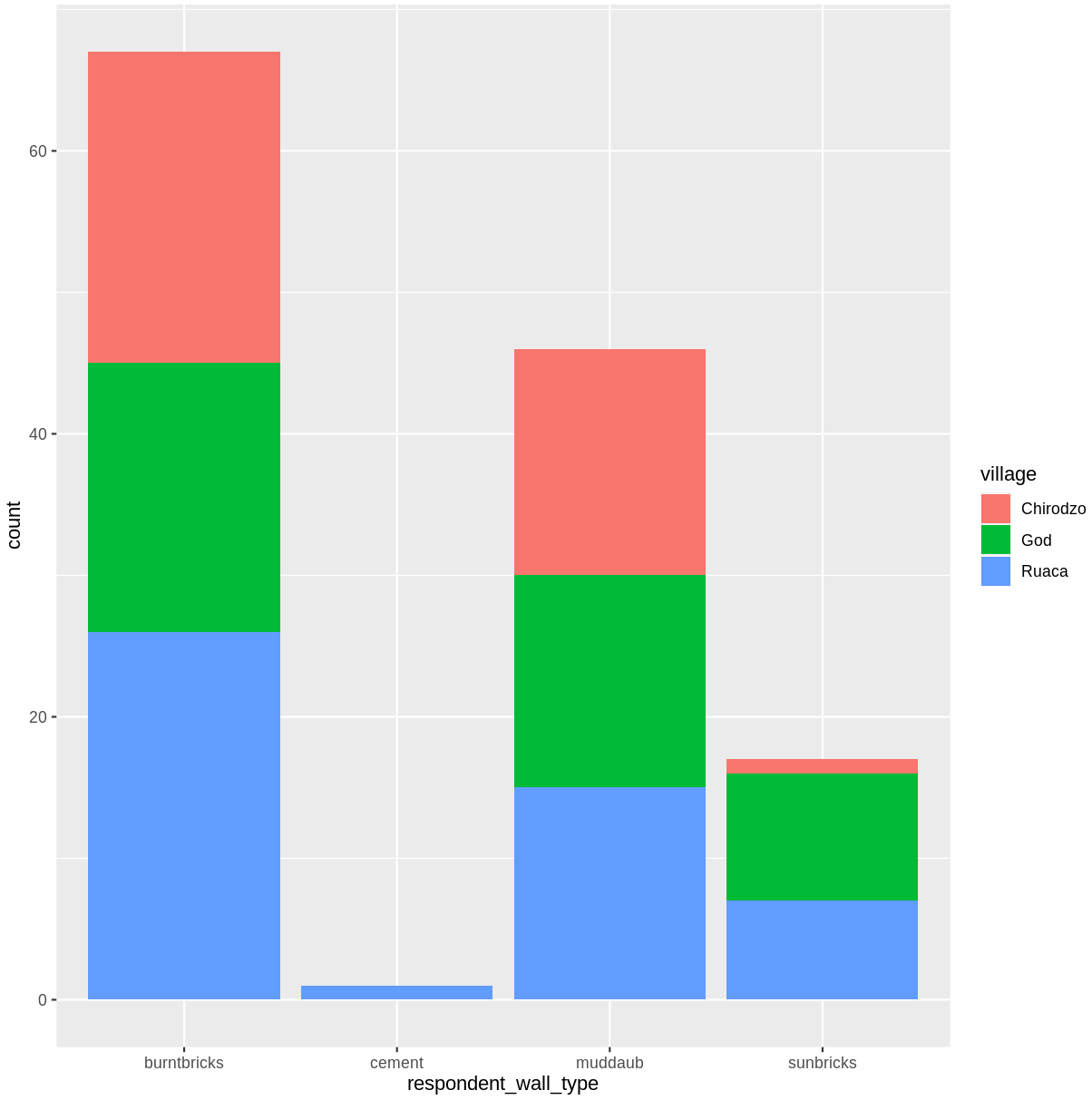
Barplots are also useful for visualizing categorical data. By default, geom_bar accepts a variable for x, and plots the number of instances each value of x (in this case, wall type) appears in the dataset.

```
interviews_plotting %>%
    ggplot(aes(x = respondent_wall_type)) +
    geom_bar()
```



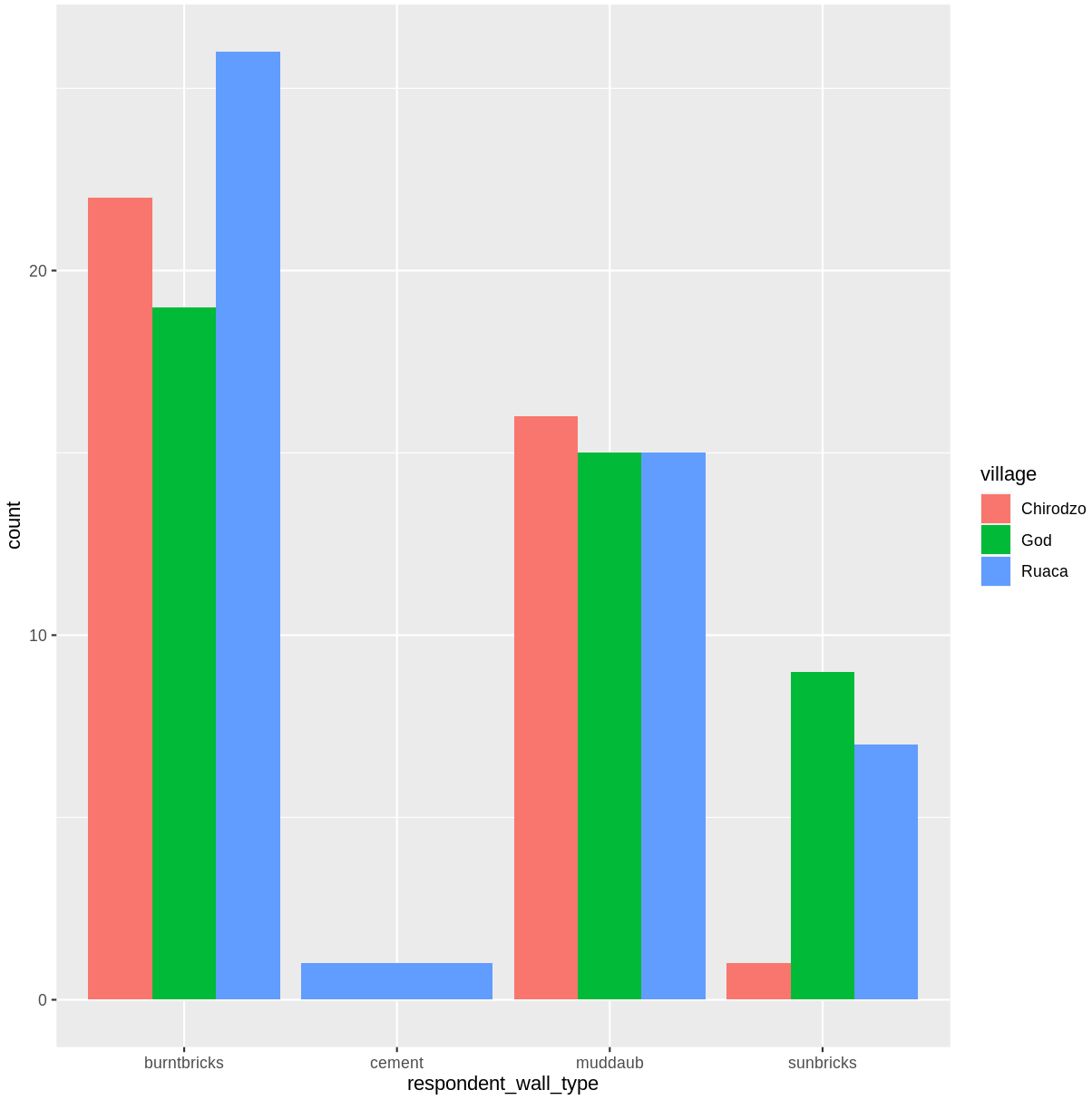
We can use the `fill` aesthetic for the `geom_bar()` geom to color bars by the portion of each count that is from each village.

```
interviews_plotting %>%
    ggplot(aes(x = respondent_wall_type)) +
    geom_bar(aes(fill = village))
```



This creates a stacked bar chart. These are generally more difficult to read than side-by-side bars. We can separate the portions of the stacked bar that correspond to each village and put them side-by-side by using the `position` argument for `geom_bar()` and setting it to “dodge”.

```
interviews_plotting %>%
    ggplot(aes(x = respondent_wall_type)) +
    geom_bar(aes(fill = village), position = "dodge")
```



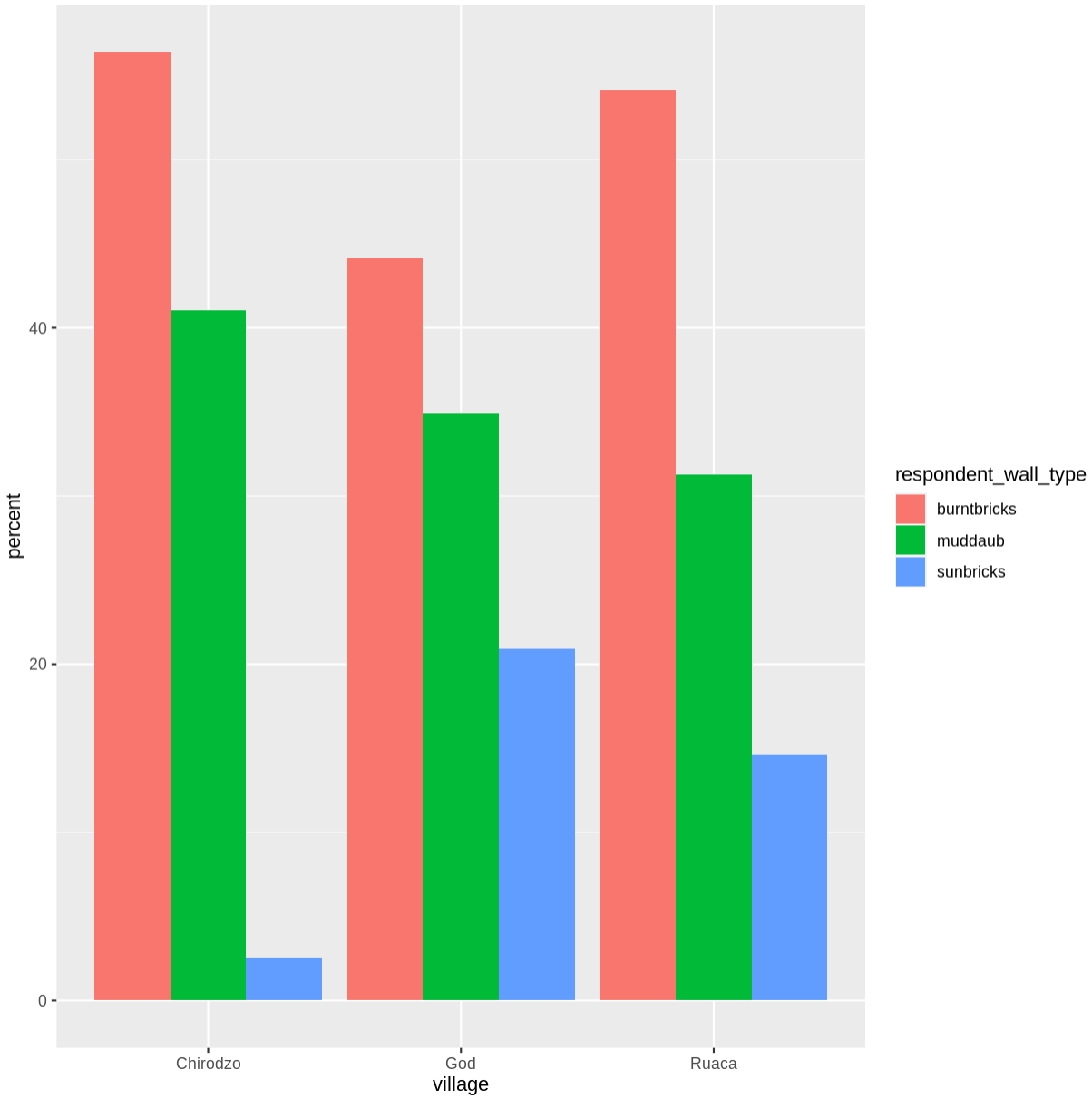
This is a nicer graphic, but we're more likely to be interested in the proportion of each housing type in each village than in the actual count of number of houses of each type (because we might have sampled different numbers of households in each village). To compare proportions, we will first create a new data frame (`percent_wall_type`) with a new column named "percent" representing the percent of each house type in each village. We will remove houses with cement walls, as there was only one in the dataset.

```
percent_wall_type <- interviews_plotting %>%
    filter(respondent_wall_type != "cement") %>%
    count(village, respondent_wall_type) %>%
    group_by(village) %>%
    mutate(percent = (n / sum(n)) * 100) %>%
```

```
    ungroup()
```

Now we can use this new data frame to create our plot showing the percentage of each house type in each village.

```
percent_wall_type %>%
    ggplot(aes(x = village, y = percent, fill =
respondent_wall_type)) +
    geom_bar(stat = "identity", position = "dodge")
```



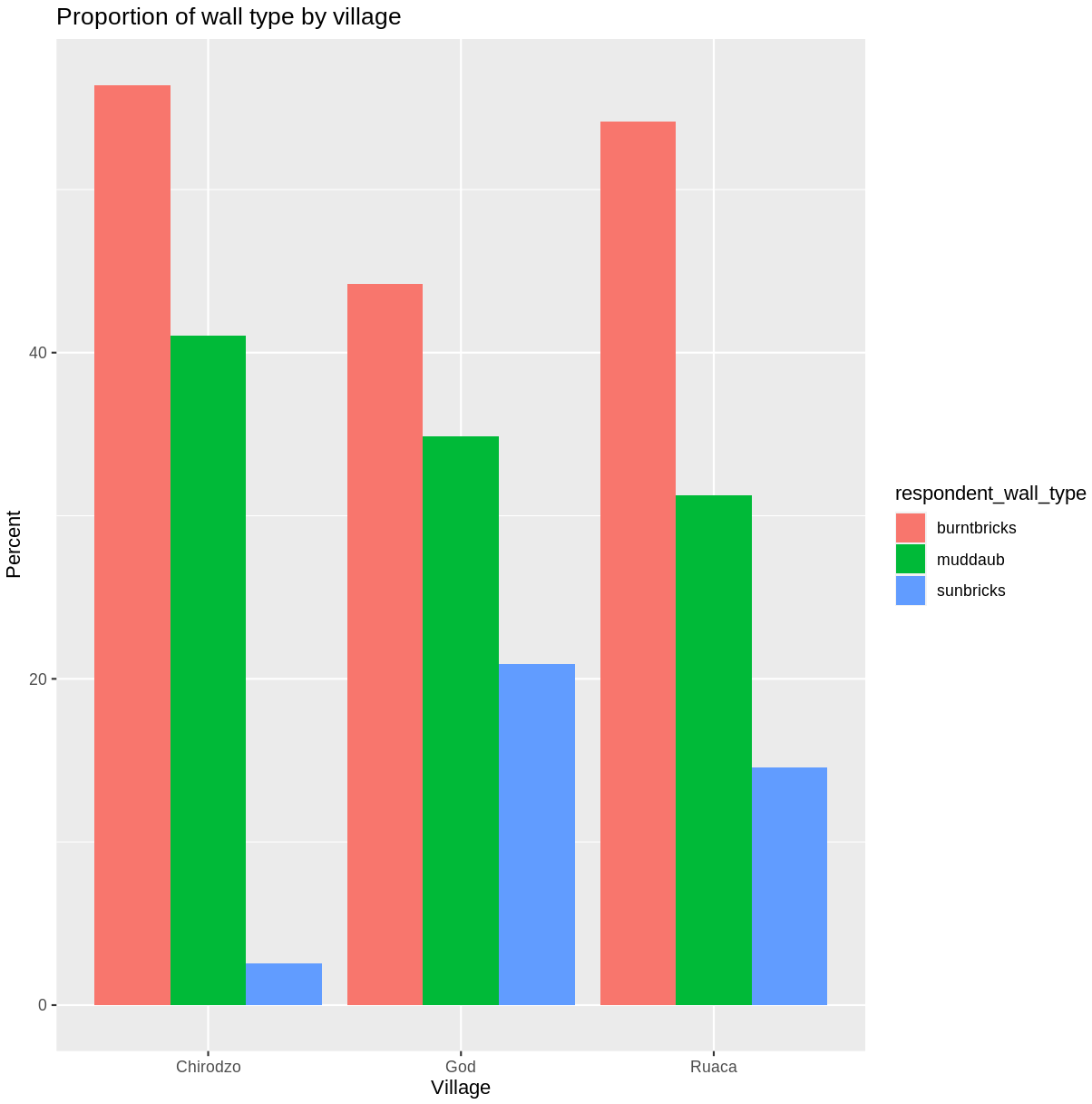
Exercise

Create a bar plot showing the proportion of respondents in each village who are or are not part of an irrigation association (`memb_assoc`). Include only respondents who answered that question in the calculations and plot. Which village had the lowest proportion of respondents in an irrigation association?

Adding Labels and Titles

By default, the axes labels on a plot are determined by the name of the variable being plotted. However, **ggplot2** offers lots of customization options, like specifying the axes labels, and adding a title to the plot with relatively few lines of code. We will add more informative x and y axis labels to our plot of proportion of house type by village and also add a title.

```
percent_wall_type %>%
    ggplot(aes(x = village, y = percent, fill =
respondent_wall_type)) +
    geom_bar(stat = "identity", position = "dodge") +
    labs(title = "Proportion of wall type by village",
         x = "Village",
         y = "Percent")
```

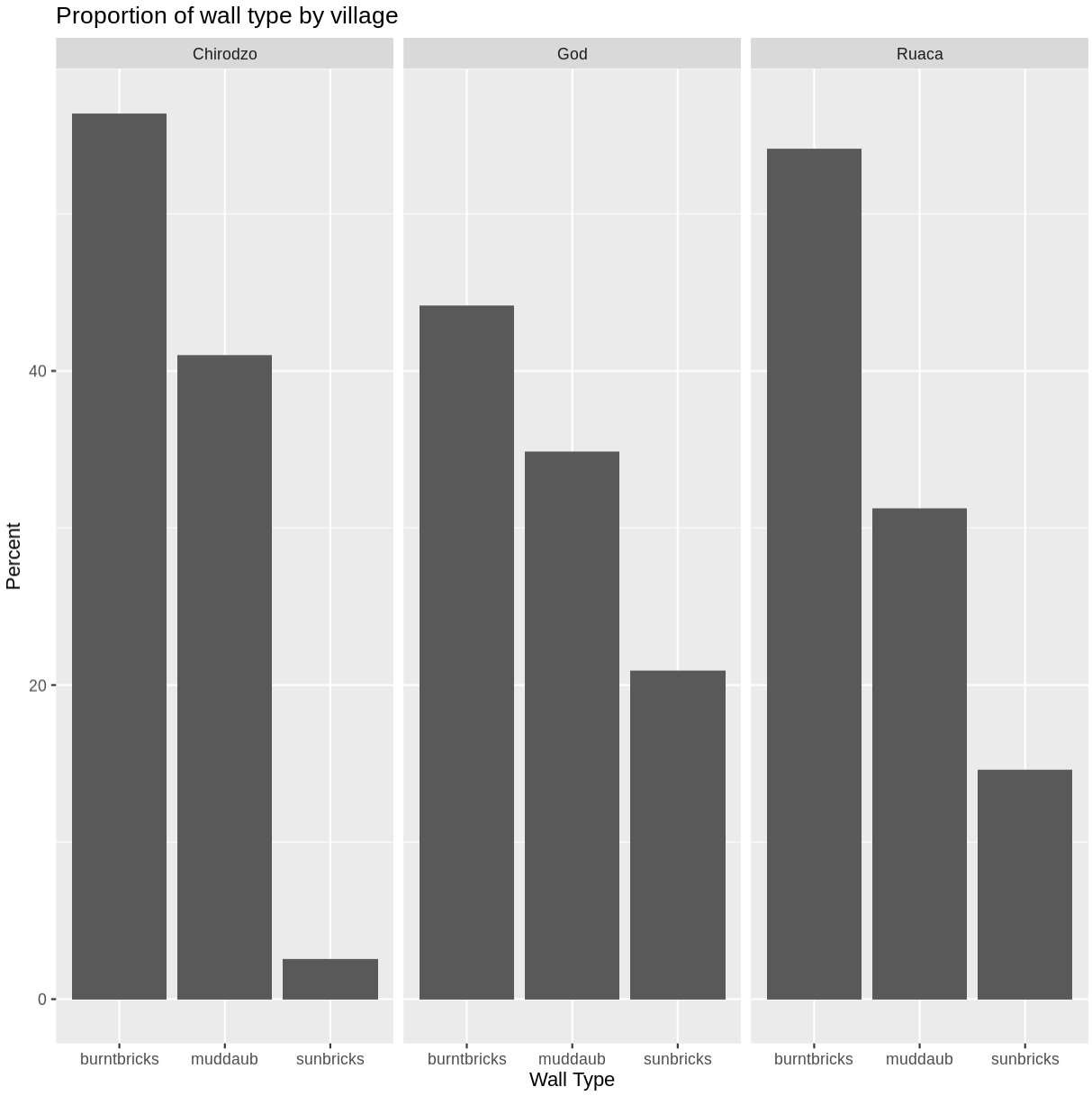


Faceting

Rather than creating a single plot with side-by-side bars for each village, we may want to create multiple plot, where each plot shows the data for a single village. This would be especially useful if we had a large number of villages that we had sampled, as a large number of side-by-side bars will become more difficult to read.

ggplot2 has a special technique called *faceting* that allows the user to split one plot into multiple plots based on a factor included in the dataset. We will use it to split our barplot of housing type proportion by village so that each village has its own panel in a multi-panel plot:

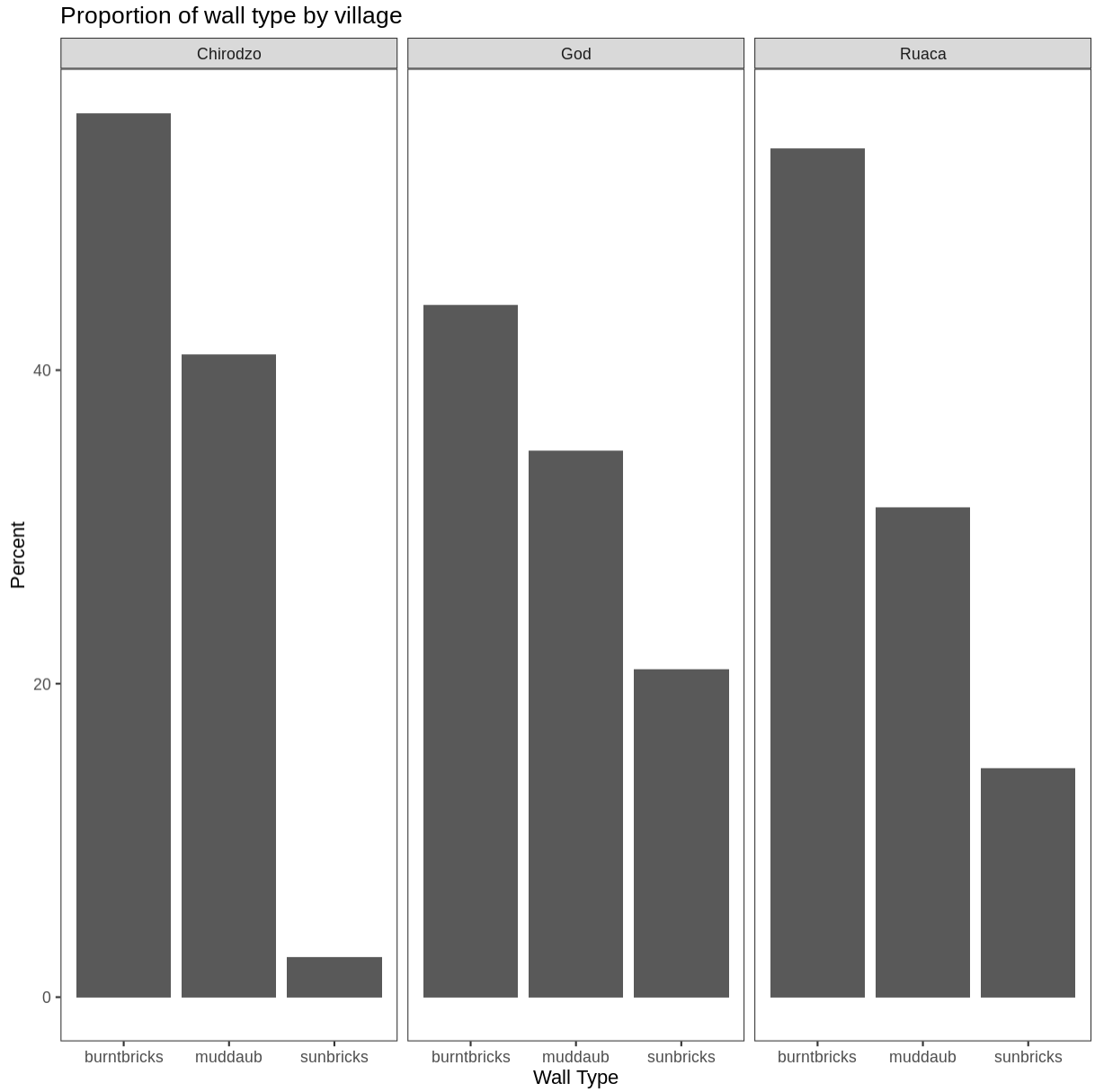
```
percent_wall_type %>%
    ggplot(aes(x = respondent_wall_type, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    labs(title="Proportion of wall type by village",
         x="Wall Type",
         y="Percent") +
    facet_wrap(~ village)
```



Click the “Zoom” button in your RStudio plots pane to view a larger version of this plot.

Usually plots with white background look more readable when printed. We can set the background to white using the function `theme_bw()`. Additionally, you can remove the grid:

```
percent_wall_type %>%
    ggplot(aes(x = respondent_wall_type, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    labs(title="Proportion of wall type by village",
         x="Wall Type",
         y="Percent") +
    facet_wrap(~ village) +
    theme_bw() +
    theme(panel.grid = element_blank())
```



What if we wanted to see the proportion of respondents in each village who owned a particular item? We can calculate the percent of people in each village who own each item and then

create a faceted series of bar plots where each plot is a particular item. First we need to calculate the percentage of people in each village who own each item:

```
percent_items <- interviews_plotting %>%
    group_by(village) %>%
    summarize(across(bicycle:no_listed_items, ~ sum(.x) / n() * 100)) %>%
    pivot_longer(bicycle:no_listed_items, names_to = "items", values_to = "percent")
```

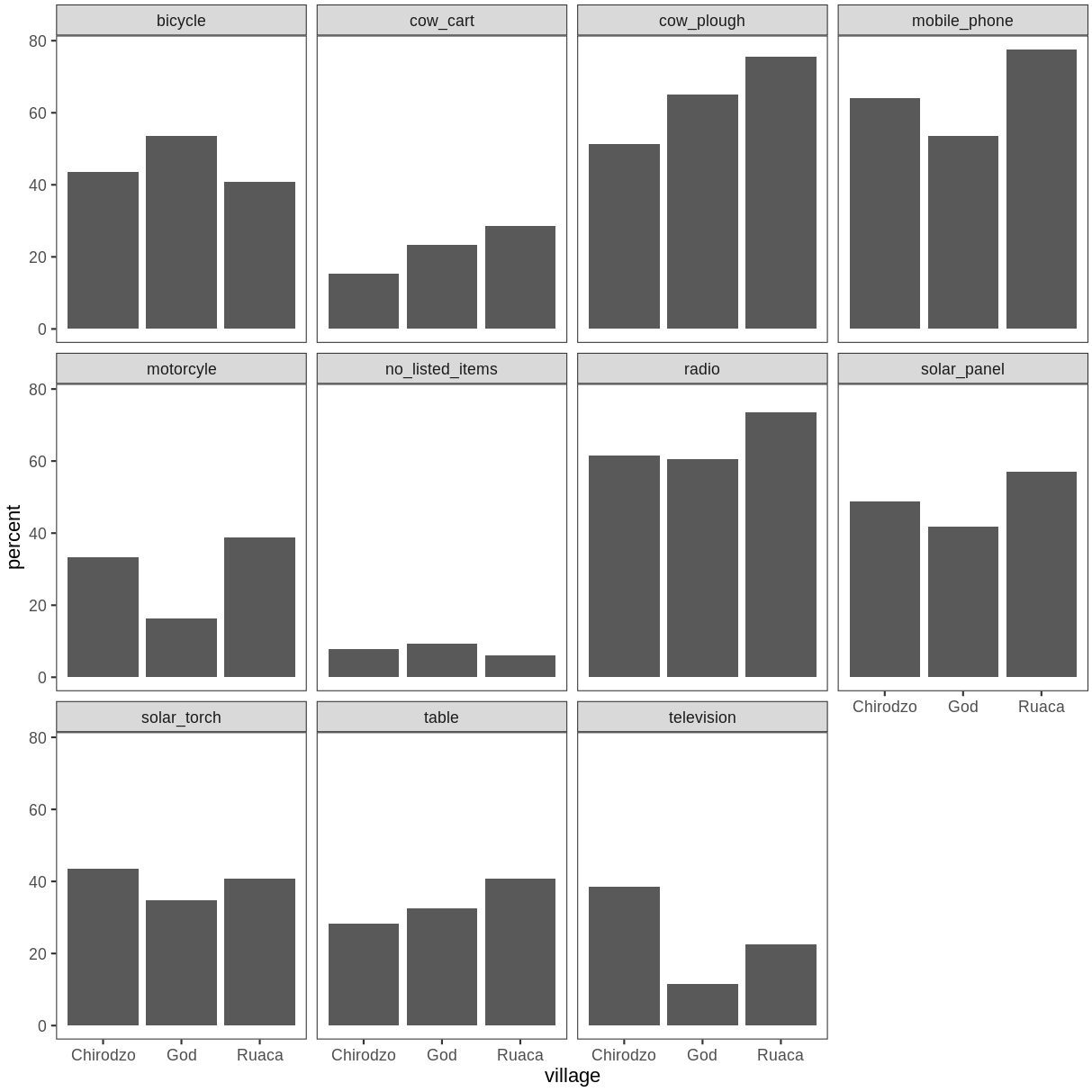
To calculate this percentage data frame, we needed to use the `across()` function within a `summarize()` operation. Unlike the previous example with a single wall type variable, where each response was exactly one of the types specified, people can (and do) own more than one item. So there are multiple columns of data (one for each item), and the percentage calculation needs to be repeated for each column.

Combining `summarize()` with `across()` allows us to specify first, the columns to be summarized (`bicycle:no_listed_items`) and then the calculation. Because our calculation is a bit more complex than is available in a built-in function, we define a new formula:

- `~` indicates that we are defining a formula,
- `sum(.x)` gives the number of people owning that item by counting the number of `TRUE` values (`.x` is shorthand for the column being operated on),
- and `n()` gives the current group size.

After the `summarize()` operation, we have a table of percentages with each item in its own column, so a `pivot_longer()` is required to transform the table into an easier format for plotting. Using this data frame, we can now create a multi-paneled bar plot.

```
percent_items %>%
    ggplot(aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    theme_bw() +
    theme(panel.grid = element_blank())
```



ggplot2 themes

In addition to `theme_bw()`, which changes the plot background to white, **ggplot2** comes with several other themes which can be useful to quickly change the look of your visualization. The complete list of themes is available at https://ggplot2.tidyverse.org/reference/ggtheme.html. `theme_minimal()` and `theme_light()` are popular, and `theme_void()` can be useful as a starting point to create a new hand-crafted theme.

The ggthemes package provides a wide variety of options (including an Excel 2003 theme). The ggplot2 extensions website provides a list of packages that extend the capabilities of **ggplot2**, including additional themes.

Exercise

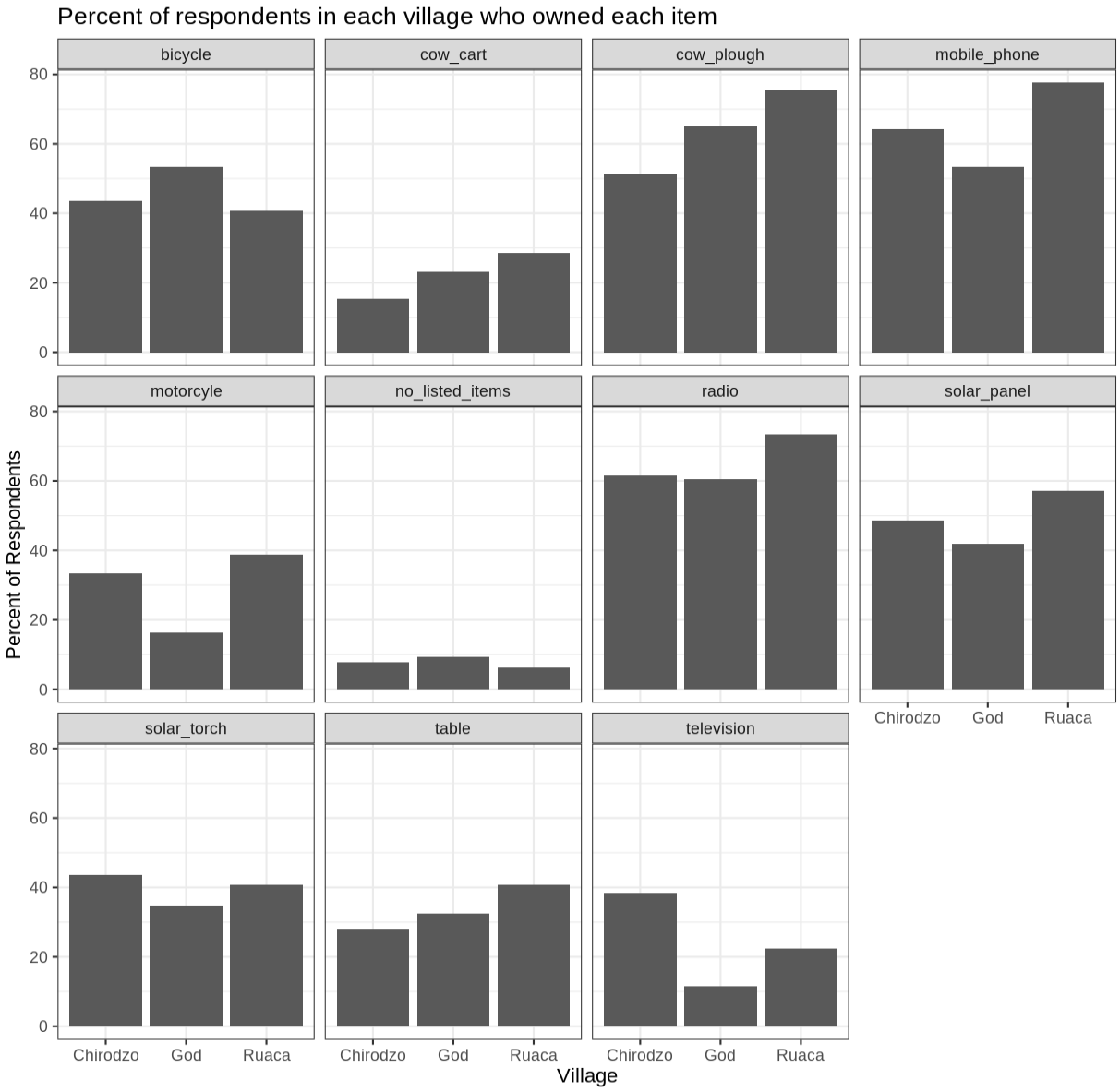
Experiment with at least two different themes. Build the previous plot using each of those themes. Which do you like best?

Customization

Take a look at the ggplot2 cheat sheet, and think of ways you could improve the plot.

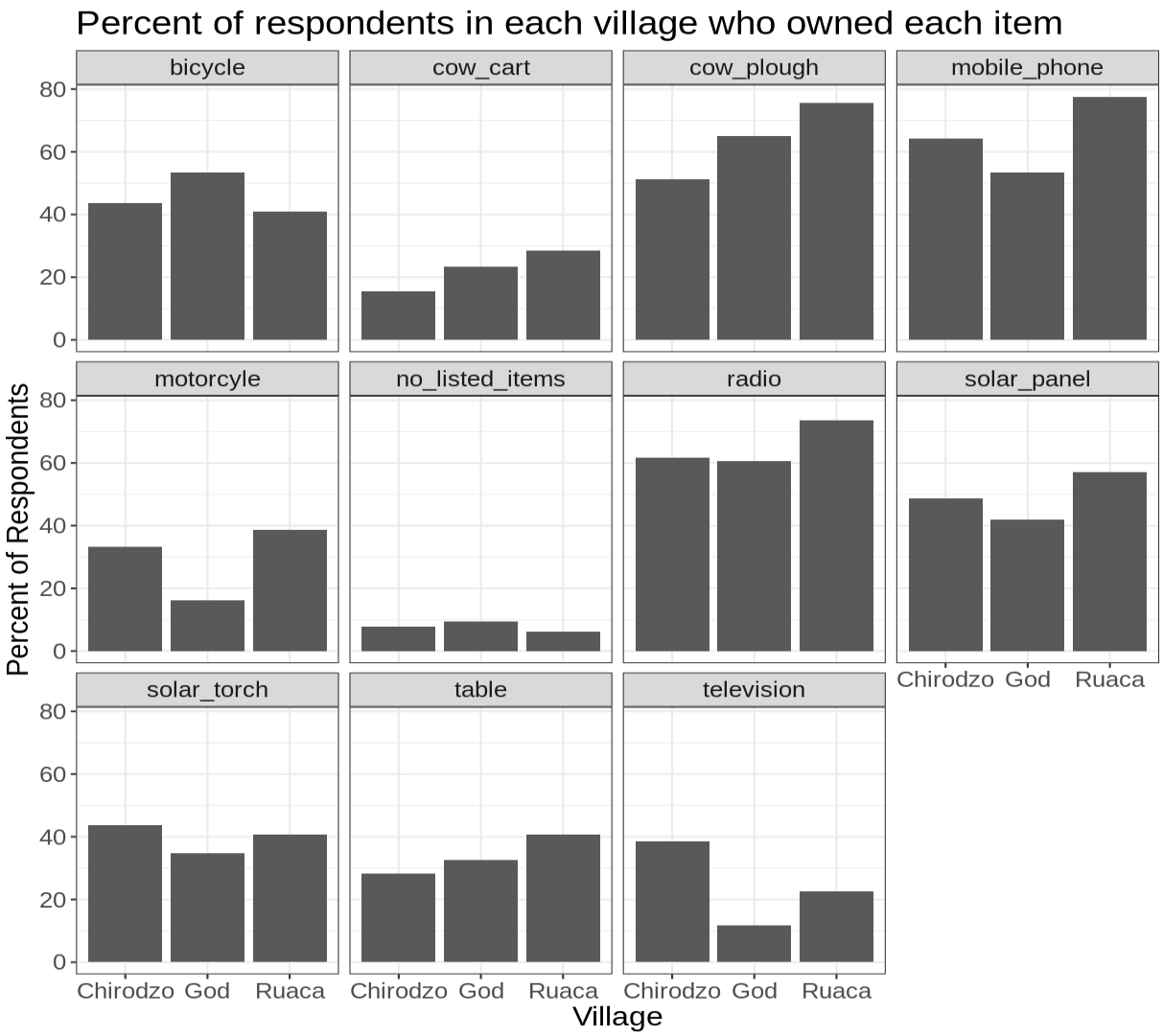
Now, let's change names of axes to something more informative than 'village' and 'percent' and add a title to the figure:

```
percent_items %>%
    ggplot(aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village who owned
each item",
         x = "Village",
         y = "Percent of Respondents") +
    theme_bw()
```



The axes have more informative names, but their readability can be improved by increasing the font size:

```
percent_items %>%
    ggplot(aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village who owned
each item",
         x = "Village",
         y = "Percent of Respondents") +
    theme_bw() +
    theme(text = element_text(size = 16))
```

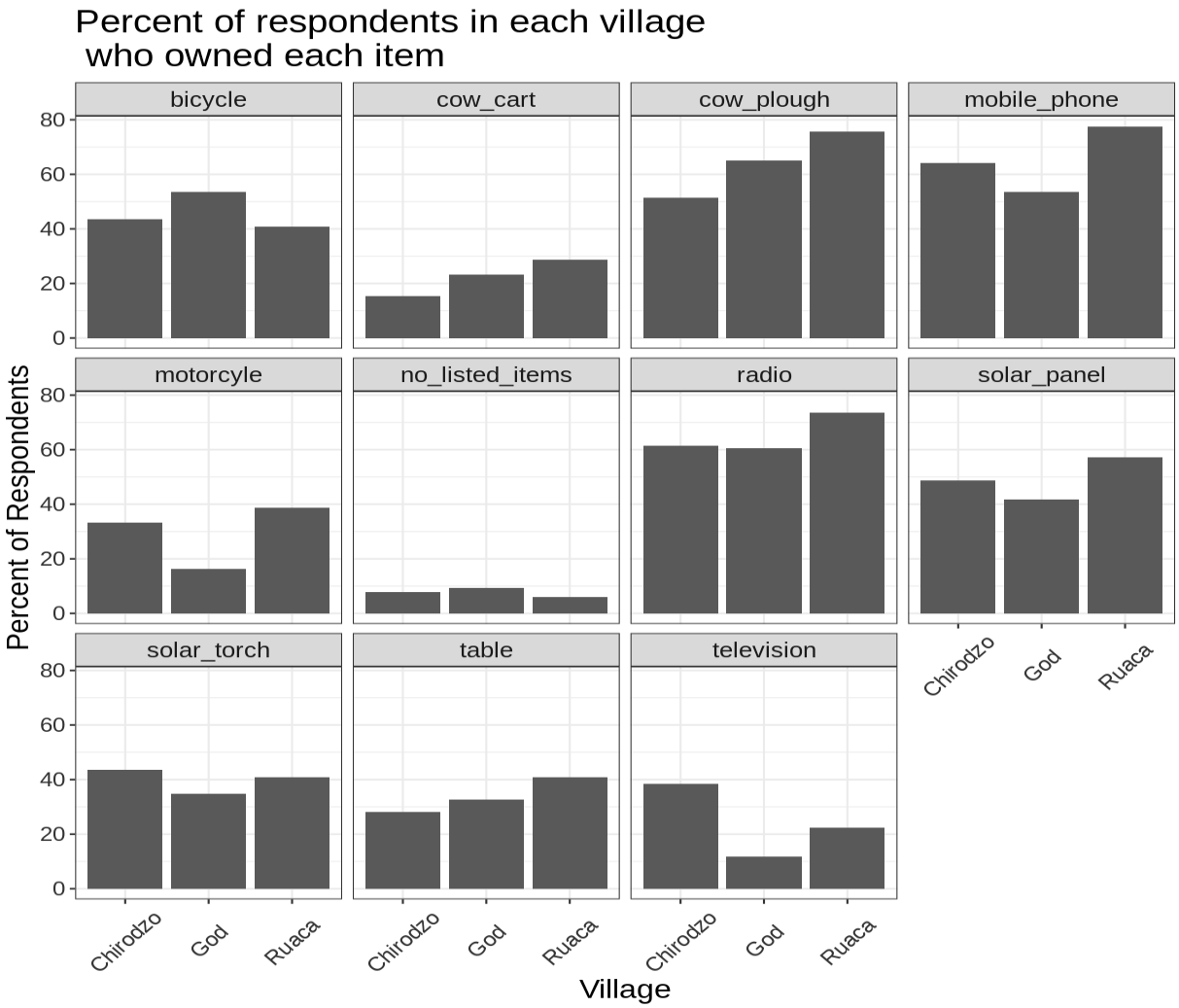


Note that it is also possible to change the fonts of your plots. If you are on Windows, you may have to install the extrafont package, and follow the instructions included in the README for this package.

After our manipulations, you may notice that the values on the x-axis are still not properly readable. Let's change the orientation of the labels and adjust them vertically and horizontally so they don't overlap. You can use a 90-degree angle, or experiment to find the appropriate angle for diagonally oriented labels. With a larger font, the title also runs off. We can add "\n" in the string for the title to insert a new line:

```
percent_items %>%
    ggplot(aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village \n who owned
each item",
```

```
          x = "Village",
          y = "Percent of Respondents") +
     theme_bw() +
     theme(axis.text.x = element_text(color = "grey20", size = 12,
angle = 45,
                                      hjust = 0.5, vjust = 0.5),
           axis.text.y = element_text(color = "grey20", size = 12),
           text = element_text(size = 16))
```

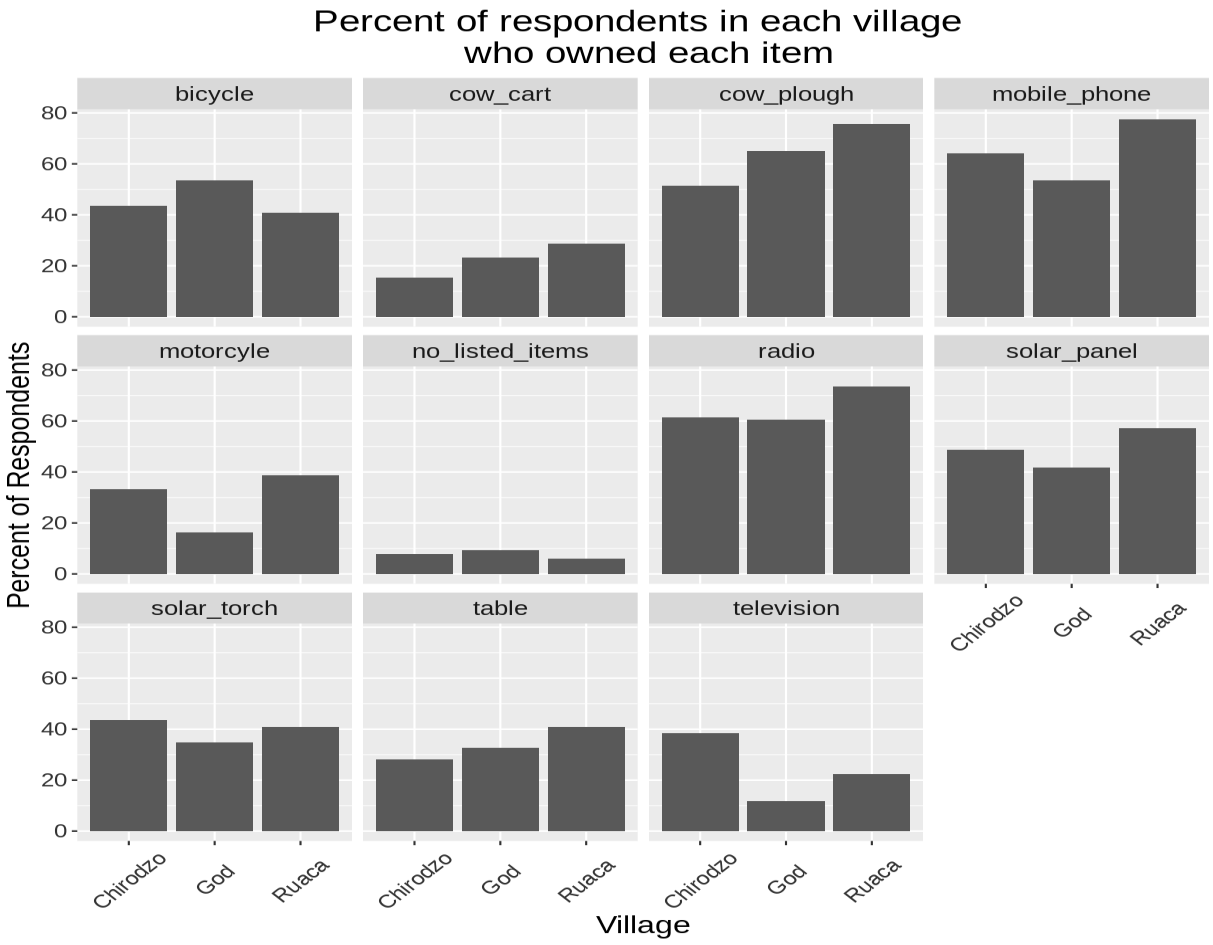


If you like the changes you created better than the default theme, you can save them as an object to be able to easily apply them to other plots you may create. We can also add `plot.title = element_text(hjust = 0.5)` to centre the title:

```
grey_theme <- theme(axis.text.x = element_text(color = "grey20", size = 12,
                                               angle = 45, hjust = 0.5,
                                               vjust = 0.5),
                    axis.text.y = element_text(color = "grey20", size = 12),
```

```
                    text = element_text(size = 16),
                    plot.title = element_text(hjust = 0.5))


percent_items %>%
    ggplot(aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village \n who owned each
item",
         x = "Village",
         y = "Percent of Respondents") +
    grey_theme
```



Exercise

With all of this information in hand, please take another five minutes to either improve one of the plots generated in this exercise or create a beautiful graph of your own. Use the RStudio ggplot2 cheat sheet for inspiration. Here are some ideas:

- See if you can make the bars white with black outline.
- Try using a different color palette
  (see http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/).

After creating your plot, you can save it to a file in your favorite format. The Export tab in the **Plot** pane in RStudio will save your plots at low resolution, which will not be accepted by many journals and will not scale well for posters.

Instead, use the `ggsave()` function, which allows you easily change the dimension and resolution of your plot by adjusting the appropriate arguments (`width`, `height` and `dpi`).

Make sure you have the `fig_output/` folder in your working directory.

```
my_plot <- percent_items %>%
    ggplot(aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village \n who owned each
item",
         x = "Village",
         y = "Percent of Respondents") +
    theme_bw() +
    theme(axis.text.x = element_text(color = "grey20", size = 12, angle = 45,
                                     hjust = 0.5, vjust = 0.5),
          axis.text.y = element_text(color = "grey20", size = 12),
          text = element_text(size = 16),
          plot.title = element_text(hjust = 0.5))

ggsave("fig_output/name_of_file.png", my_plot, width = 15, height = 10)
```

Note: The parameters `width` and `height` also determine the font size in the saved plot.

Key Points

- **ggplot2** is a flexible and useful tool for creating plots in R.
- The data set and coordinate system can be defined using the **ggplot** function.
- Additional layers, including geoms, are added using the `+` operator.
- Boxplots are useful for visualizing the distribution of a continuous variable.
- Barplots are useful for visualizing categorical data.
- Faceting allows you to generate multiple plots based on a categorical variable.