

Manipulating, analyzing and exporting data with tidyverse

Data Carpentry contributors

Contents

Data Manipulation using <code>dplyr</code> and <code>tidyr</code>	1
What are <code>dplyr</code> and <code>tidyr</code> ?	2
Selecting columns and filtering rows	3
Pipes	3
Mutate	4
Split-apply-combine data analysis and the <code>summarize()</code> function	5
Counting	7
Reshaping by pivoting data wider or longer	9
Exporting data	12
Summary of functions in this lesson	13
Manipulating and analyzing data with <code>dplyr</code>	

Learning Objectives

- Describe the purpose of the `dplyr` and `tidyr` packages.
- Select certain columns in a data frame with the `dplyr` function `select`.
- Select certain rows in a data frame 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 data frame 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 data frame into groups of observations, apply 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 what key-value pairs are.
- Reshape a data frame from long to wide format and back with the `pivot_wider` and `pivot_longer` functions from the `tidyr` package.
- Export a data frame to a .csv file.

Data Manipulation using `dplyr` and `tidyr`

Bracket subsetting is handy, but it can be cumbersome and difficult to read, especially for complicated operations. Enter `dplyr`. `dplyr` is a package for making tabular data manipulation easier. It pairs nicely with `tidyr` which enables you to swiftly convert between different data formats for plotting and analysis.

Packages in R are sets of additional functions that let you do more stuff. The functions we've been using so far, like `str()` or `data.frame()`, come built into R. Adding packages gives you access to more functions. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent R session when you need it. You should already have installed the **tidyverse** package. This is an "umbrella-package" that installs several packages useful for data analysis which work together well such as **tidyr**, **dplyr**, **ggplot2**, **tibble**, etc.

Advanced note: The **tidyverse** package tries to address 3 common issues that arise when doing data analysis with some of the functions that come with R: (1) The results from a base R function sometimes depend on the type of data. (2) Using R expressions in a non standard way, which can be confusing for new learners. (3) Hidden arguments, having default operations that new learners are not aware of.

We have seen in our previous lesson that when building or importing a data frame, the columns that contain characters (i.e., text) are coerced (=converted) into the **factor** data type. We had to set **stringsAsFactors** to **FALSE** to avoid this hidden argument to convert our data type.

This time we will use the **tidyverse** package to read the data and avoid having to set **stringsAsFactors** to **FALSE**

If we haven't already done so, we can type `install.packages("tidyverse")` straight into the console. In fact, it's better to write this in the console than in our script for any package, as there's no need to re-install packages every time we run the script.

Then, to load the package type:

```
## load the tidyverse packages, incl. dplyr
library(tidyverse)
```

What are dplyr and tidyr?

The package **dplyr** provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with R in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can connect to a database of many hundreds of GB, conduct queries on it directly, and pull back into R only what you need for analysis.

The package **tidyr** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups - like plots or aquaria. Moving back and forth between these formats is non-trivial, and **tidyr** gives you tools for this and more sophisticated data manipulation.

To learn more about **dplyr** and **tidyr** after the workshop, you may want to check out this handy data transformation with **dplyr** cheatsheet and this one about **tidyr**.

We'll read in our data using the `read_csv()` function, from the tidyverse package **readr**, instead of `read.csv()`. (Note the underscore instead of dot.)

```
surveys <- read_csv("data_raw/portal_data_joined.csv")
#> Rows: 34786 Columns: 13
#> -- Column specification -----
#> Delimiter: ","
#> chr (6): species_id, sex, genus, species, taxa, plot_type
#> dbl (7): record_id, month, day, year, plot_id, hindfoot_length, weight
```

```
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

You will see the message `Parsed with column specification`, followed by each column name and its data type. When you execute `read_csv` on a data file, it looks through the first 1000 rows of each column and guesses the data type for each column as it reads it into R. For example, in this dataset, `read_csv` reads `weight` as `col_double` (a numeric data type), and `species` as `col_character`. You have the option to specify the data type for a column manually by using the `col_types` argument in `read_csv`.

```
## inspect the data
str(surveys)
```

```
## preview the data
View(surveys)
```

Notice that the class of the data is now `tbl_df`

This is referred to as a “tibble”. Tibbles tweak some of the behaviors of the data frame objects we introduced in the previous episode. The data structure is very similar to a data frame. For our purposes the only differences are that:

1. In addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen.
2. Columns of class `character` are never converted into factors.

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 data frame, use `select()`. The first argument to this function is the data frame (`surveys`), and the subsequent arguments are the columns to keep.

```
select(surveys, plot_id, species_id, weight)
```

To select all columns *except* certain ones, put a “-” in front of the variable to exclude it.

```
select(surveys, -record_id, -species_id)
```

This will select all the variables in `surveys` except `record_id` and `species_id`.

To choose rows based on a specific criterion, use `filter()`:

```
filter(surveys, year == 1995)
```

Pipes

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 data frame and use that as input to the next function, like this:

```
surveys2 <- filter(surveys, weight < 5)
surveys_sml <- select(surveys2, species_id, sex, weight)
```

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.

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

```
surveys_sml <- select(filter(surveys, weight < 5), species_id, sex, weight)
```

This is handy, but can be difficult to read if too many functions are nested, as R evaluates the expression from the inside out (in this case, filtering, then selecting).

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. Pipes in R look like `%>%` and are made available via the `magrittr` package, installed automatically with `dplyr`. If you use RStudio, you can type the pipe with `Ctrl + Shift + M` if you have a PC or `Cmd + Shift + M` if you have a Mac.

```
surveys %>%  
  filter(weight < 5) %>%  
  select(species_id, sex, weight)
```

In the above code, we use the pipe to send the `surveys` dataset first through `filter()` to keep rows where `weight` is less than 5, then through `select()` to keep only the `species_id`, `sex`, and `weight` columns. Since `%>%` takes the object on its left (either an object or the result of a function call) and passes it as the **first argument to the function on its right**, we don't need to explicitly include the data frame as an argument to the `filter()` and `select()` functions any more.

Some may find it helpful to read the pipe like the word “then”. For instance, in the above example, we took the data frame `surveys`, *then* we **filtered** for rows with `weight < 5`, *then* we **selected** columns `species_id`, `sex`, and `weight`. The `dplyr` functions by themselves are somewhat simple, but by combining them into linear workflows with the pipe, we can accomplish more complex manipulations of data frames.

If we want to create a new object with this smaller version of the data, we can assign as the value of a new object using the assignment operator:

```
surveys_sml <- surveys %>%  
  filter(weight < 5) %>%  
  select(species_id, sex, weight)
```

```
surveys_sml
```

Challenge

3.1 Using pipes, subset the `surveys` data to include animals collected before 1995 and retain only the columns `year`, `sex`, and `weight`.

Answer

```
surveys %>%  
  filter(year < 1995) %>%  
  select(year, sex, weight)
```

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()`.

To create a new column of weight in kg:

```
surveys %>%  
  mutate(weight_kg = weight / 1000)
```

Note that we use the single = symbol here to store the value on the right in the new column named on the left.

You can also create a second new column based on the first new column within the same call of `mutate()`:

```
surveys %>%
  mutate(weight_kg = weight / 1000,
         weight_lb = weight_kg * 2.2)
```

If this runs off your screen and you just want to see the first few rows, you can use a pipe to view the `head()` of the data. (Pipes work with non-`dplyr` functions, too, as long as the `dplyr` or `magrittr` package is loaded).

```
surveys %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

The first few rows of the output are full of NAs, so if we wanted to remove those we could insert a `filter()` in the chain:

```
surveys %>%
  filter(!is.na(weight)) %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

`is.na()` is a function that determines whether something is an NA. The `!` symbol negates the result, so we're asking for every row where weight *is not* an NA.

Challenge

3.2 Create a new data frame from the `surveys` data that meets the following criteria:

- contains only the `species_id` column and a new column called `hindfoot_cm`, which contains the `hindfoot_length` values converted to centimeters.
- In this `hindfoot_cm` column, there are no NAs and all values are less than 3.

Hint: think about how the commands should be ordered to produce this data frame

Answer

```
surveys_hindfoot_cm <- surveys %>%
  filter(!is.na(hindfoot_length)) %>%
  mutate(hindfoot_cm = hindfoot_length / 10) %>%
  filter(hindfoot_cm < 3) %>%
  select(species_id, hindfoot_cm)
```

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 and the `summarize()` function

`group_by()` creates groups within your data by taking as arguments the column name(s) that contain the **categorical** variable(s) for which you want to make the groups (e.g. by sex and by species).

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row by calculating some summary statistic(s).

So to compute the mean weight by sex, you would group by the sex column and then summarise the weight column:

```
surveys %>%
  group_by(sex) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

You may also have noticed that the output from these calls doesn't run off the screen anymore. It's one of the advantages of `tbl_df` over data frame.

The big difference between `mutate()` and `summarize()` is that `mutate()` adds columns while `summarize()` only keeps what you have grouped by and the summaries you have calculated.

You can also group by multiple columns:

```
surveys %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE)) %>%
  tail()
```

```
#> `summarise()` has grouped output by 'sex'. You can override using the `.groups`
#> argument.
```

Here, we used `tail()` to look at the last six rows of our summary. Before, we had used `head()` to look at the first six rows. We can see that the `sex` column contains `NA` values because some animals had escaped before their sex and body weights could be determined. The resulting `mean_weight` column does not contain `NA` but `NaN` (which refers to "Not a Number") because `mean()` was called on a vector of `NA` values while at the same time setting `na.rm = TRUE`. To avoid this, we can remove the missing values for weight before we attempt to calculate the summary statistics on weight. Because the missing values are removed first, we can omit `na.rm = TRUE` when computing the mean:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight))
```

```
#> `summarise()` has grouped output by 'sex'. You can override using the `.groups`
#> argument.
```

Here, again, the output from these calls doesn't run off the screen anymore. If you want to display more data, you can use the `print()` function at the end of your chain with the argument `n` specifying the number of rows to display:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight)) %>%
  print(n = 15)
```

```
#> `summarise()` has grouped output by 'sex'. You can override using the `.groups`
#> argument.
```

Once the data are grouped, you can also summarize multiple variables at the same time (and not necessarily on the same variable). For instance, we could add a column indicating the minimum weight for each species for each sex:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight))
```

```
#> `summarise()` has grouped output by 'sex'. You can override using the `.groups`
#> argument.
```

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can sort on `min_weight` to put the lighter species first using the `arrange()` function:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(min_weight)
```

```
#> `summarise()` has grouped output by 'sex'. You can override using the ` .groups `
#> argument.
```

To sort in descending order, we need to add the `desc()` function. If we want to sort the results by decreasing order of mean weight:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(desc(min_weight))
```

```
#> `summarise()` has grouped output by 'sex'. You can override using the ` .groups `
#> argument.
```

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 sex, we would do:

```
surveys %>%
  count(sex)
```

The `count()` function is shorthand for something we've already seen: grouping by a variable, and summarizing it by counting the number of observations in that group. In other words, `surveys %>% count()` is equivalent to:

```
surveys %>%
  group_by(sex) %>%
  summarise(count = n())
```

Where `n()` is a function that gives us the number of rows in a group.

For convenience, `count()` provides the `sort` argument:

```
surveys %>%
  count(sex, sort = TRUE)
```

If we wanted to count a *combination of values*, such as `sex` and `species`, we would specify the both columns as the arguments of `count()`:

```
surveys %>%
  count(sex, species)
```

With the above code, we can proceed with `arrange()` to sort the table according to a number of criteria so that we have a better comparison. For instance, we might want to arrange the table above in (i) alphabetical order of the species and (ii) descending order of the count:

```
surveys %>%
  count(sex, species) %>%
  arrange(species, desc(n))
```

From the table above, we may learn that, for instance, there are 75 observations of the *albigula* species that have missing data for sex (i.e. NA).

Challenge

3.3 How many animals were caught in each `plot_type` surveyed?

Answer

```
surveys %>%
  count(plot_type)
```

3.4 Use `group_by()` and `summarize()` to find the mean, min, and max hindfoot length for each species (using `species_id`). Also add the number of observations (hint: see `?n` and don't forget to handle any missing values).

Answer

```
surveys %>%
  filter(!is.na(hindfoot_length)) %>%
  group_by(species_id) %>%
  summarize(
    mean_hindfoot_length = mean(hindfoot_length),
    min_hindfoot_length = min(hindfoot_length),
    max_hindfoot_length = max(hindfoot_length),
    n = n()
  )
```

3.5 What was the heaviest animal measured in each year? Return the columns `year`, `genus`, `species_id`, and `weight`.

Answer

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(year) %>%
  filter(weight == max(weight)) %>%
  select(year, genus, species, weight) %>%
  arrange(year)
```

3.6 Create a new data frame `surveys_1977` that (i) contains only the columns `year`, `plot_type`, `species`, and `weight` and (ii) contains only the observations from the year 1977 and where `weight` is not NA.

Answer

```
surveys_1977 <- surveys %>%
  select(species, plot_type, weight, year) %>%
  filter(year == 1977 & (!is.na(weight)))
```

3.7 Use `group_by()` to group `surveys_1977` by species and then `arrange()` by weight. What changes when you set the `.by_group` argument in `arrange()` to `TRUE`?

Answer

```
surveys_1977 %>%
  group_by(species) %>%
  arrange(weight)
```

```
surveys_1977 %>%
  group_by(species) %>%
  arrange(weight, .by_group = TRUE)
```

Time for a git commit

Reshaping by pivoting data wider or longer

There are many ways to store data. When working with `tidyverse` functions, it is helpful to follow the four rules that they define as “tidy data”:

1. Each variable has its own column
2. Each observation has its own row
3. Each value must have its own cell
4. Each type of observational unit forms a table

For more on this, after the workshop, see this [Data Carpentry lesson on handling data and spreadsheets](#).

Here we examine the fourth rule: Each type of observational unit forms a table.

In `surveys`, the rows of `surveys` contain the values of variables associated with each record (the unit), such as the weight or sex of each animal. What if instead of comparing records, we wanted to compare the different mean weight of each genus between plots? (Ignoring `plot_type` for simplicity).

Note: here, “plot” means an area of land.

We’d need to create a new table where each row (the unit) is comprised of values of variables associated with each plot (so each row would be a plot). In practical terms this means the values in the `genus` column would become the names of columns and the values within these columns would be the mean weight observed on each plot.

Having created this new table, it would be straightforward to explore the relationship between the weight of different genera within, and between, the plots.

The key point here is that we are still following a tidy data structure, but we have **reshaped** the data according to the observations of interest: average genus weight per plot instead of records per date.

We can do this transformation with the `tidyr` function `pivot_wider()`.

Pivoting wider

Note: in older versions of the `tidyr` package, this function was called `spread()`

`pivot_wider()` takes three principal arguments:

1. the data
2. the column whose values will become new **column names**. (The ‘key’ column.)
3. the column whose values will fill the new **column values**. (The ‘value’ column.)

Let’s use `pivot_wider()` to transform `surveys` to find the mean weight of each genus in each plot over the entire survey period. We use `filter()`, `group_by()` and `summarise()` to filter our observations and variables of interest, and create a new variable for the `mean_weight`.

```
surveys_gw <- surveys %>%
  filter(!is.na(weight)) %>%
  group_by(plot_id, genus) %>%
  summarize(mean_weight = mean(weight))
```

```
#> `summarise()` has grouped output by 'plot_id'. You can override using the
#> `.groups` argument.
```

```
str(surveys_gw)
```

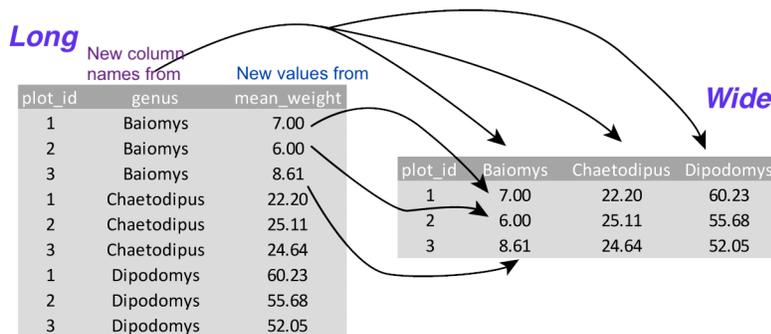
This yields `surveys_gw` where the observations for each plot are spread across multiple rows (196 observations of 3 variables).

Using `pivot_wider()` we can make a new column for each value in the `genus` column with values for those new columns coming from the `mean_weight` column.

```
surveys_wide <- surveys_gw %>%  
  pivot_wider(names_from = genus,  
             values_from = mean_weight)
```

```
head(surveys_wide)
```

Here we now have 24 observations of 11 variables, one row for each plot.



After the workshop, see the `tidyr` documentation for more details. This documentation format is called a ‘vignette’ and is popular in the R community. You may want to use this term when googling.

Pivoting longer

Note: in older versions of the `tidyr` package, this function was called `gather()`

The opposing situation could occur if we had been provided with data in the form of `surveys_wide`, where the genus names are column names, but we wish to treat them as values of a genus variable instead.

In this situation we are gathering the column names and turning them into a pair of new variables. One variable represents the column names as values, and the other variable contains the values previously associated with the column names.

`pivot_longer()` takes four principal arguments:

1. the data
2. which columns we want to gather (pivot on)
3. the name of the column we wish to create from the column names we will gather. (key)
4. the name of the column we wish to create that will contain the values that used to be in the gathered columns (values)

To recreate `surveys_gw` from `surveys_wide` we would create a key called `genus` and value called `mean_weight` and use all columns except `plot_id` for the key variable. Here we exclude `plot_id` from being used in the pivot.

```
surveys_longer <- surveys_wide %>%  
  pivot_longer( cols = -plot_id,  
              # columns to pivot into longer format, here we want  
              # everything *except* the plot_id (note the minus sign)  
  
              names_to = "genus",
```

```

# name of the new column that will store the old column names

values_to = "mean_weight"
# name of the new column that will store the old column values)
)
head(surveys_longer)
str(surveys_longer)

```

Why are some words quoted here (“genus” and “mean_weight”) and others are not (plot_id)? It’s because plot_id refers to an *existing* column in the data frame. The quoted words “genus” and “mean_weight” are the names of new columns. Since the columns *don’t exist yet*, these words need quotation marks because they don’t refer to variables that exist in R yet. If you don’t include the quotation marks then R will look for a column called `genus` and will throw an error because it doesn’t exist.

Note that now the NA genera are included in the re-gathered format. Spreading and then gathering can be a useful way to balance out a dataset so every replicate has the same composition.

Alternatively, we could have specified which columns to include in the `cols` argument (instead of what not to include). This can be useful if you have a large number of identifying columns that you don’t want to pivot longer, because then it’s easier to specify what to pivot (gather) than what to leave alone.

```

surveys_wide %>%
  pivot_longer( cols = c(Baiomys, Chaetodipus, Dipodomys, Neotoma,
                        Onychomys, Perognathus, Peromyscus, Reithrodontomys,
                        Sigmodon, Spermophilus),
               # columns to pivot into longer format, here we specify
               # all the genera with a vector of names
               names_to = "genus",
               values_to = "mean_weight" ) %>%
  head()

```

There are many shortcuts for selecting columns. For more on that after the workshop, see the `tidyr` package documentation on pivoting.

As an example, if the columns are directly adjacent, we don’t even need to list them all out - just use the `:` operator!

```

surveys_wide %>%
  pivot_longer( cols = Baiomys:Spermophilus,
               # columns to pivot into longer format, here we specify
               # all the genera between the first and last
               names_to = "genus",
               values_to = "mean_weight" ) %>%
  head()

```

Challenge

3.8 Widen the `surveys` data frame with `years` as columns, `plot_ids` as rows, and the number of genera per plot as the values. You will need to summarize before reshaping. You can use the function `n_distinct()` to get the number of unique genera within a particular group of data. It’s a powerful function! See `?n_distinct` for more.

Answer

```

surveys_wide_genera <- surveys %>%
  group_by(plot_id, year) %>%
  summarize(n_genera = n_distinct(genus)) %>%
  pivot_wider(names_from = year,
              values_from = n_genera)

```

```
#> `summarise()` has grouped output by 'plot_id'. You can override using the
#> `.groups` argument.
```

```
head(surveys_wide_genera)
```

3.9 Now take that data frame and `pivot_longer()` it again, so each row is a unique `plot_id` by year combination.

Answer

```
surveys_wide_genera %>%
  pivot_longer(cols = -plot_id,
              names_to = "year",
              values_to = "n_genera")
```

3.10 The `surveys` data set has two measurement columns: `hindfoot_length` and `weight`. This makes it difficult to do things like look at the relationship between mean values of each measurement per year in different plot types. Let's walk through a common solution for this type of problem. First, use `pivot_longer()` to create a dataset where we have a key column called `measurement` and a `value` column that takes on the value of either `hindfoot_length` or `weight`. *Hint:* You'll need to specify which columns are being pivoted on (gathered). Which columns have the values that we want to put into one column?

Answer

```
surveys_long <- surveys %>%
  pivot_longer(cols=c(hindfoot_length,weight),
              names_to = "measurement",
              values_to = "value")
```

3.11 With this new data set (`surveys_long`): (a) calculate the mean of each `measurement` in each `year` for each different `plot_type` (Hint: How do you need to group your data to answer this question? What column values do you need to summarise?) (b) Then separate out the mean values into separate columns for `hindfoot_length` and `weight` using `pivot_wider()`

Answer

```
surveys_long %>%
  group_by(year, measurement, plot_type) %>%
  summarize(mean_value = mean(value, na.rm=TRUE)) %>%
  pivot_wider(names_from = measurement, values_from = mean_value)

#> `summarise()` has grouped output by 'year', 'measurement'. You can override
#> using the `.groups` argument.
```

Exporting data

Now that you have learned how to use `dplyr` to extract information from or summarize your raw data, you may want to export these new data sets to share them with your collaborators or for archival.

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

Before using `write_csv()`, we are going to create a new folder, `data`, 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_raw` 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` directory, so even if the files it contains are deleted, we can always re-generate them.

In preparation for our next lesson on plotting, we are going to prepare a cleaned up version of the data set that doesn't include any missing data.

Let's start by removing observations of animals for which `weight` and `hindfoot_length` are missing, or the `sex` has not been determined:

```
surveys_complete <- surveys %>%
  filter(!is.na(weight),           # remove missing weight
         !is.na(hindfoot_length), # remove missing hindfoot_length
         !is.na(sex))             # remove missing sex
```

Because we are interested in plotting how species abundances have changed through time, we are also going to remove observations for rare species (i.e., that have been observed less than 50 times). We will do this in two steps: first we are going to create a data set that counts how often each species has been observed, and filter out the rare species; then, we will extract only the observations for these more common species:

```
## Extract the most common species_id
species_counts_min50 <- surveys_complete %>%
  count(species_id) %>%
  # count() gets the number of times each species_id occurs in the data
  filter(n >= 50)
# count() created a column called "n" for its results. Here we keep
# only those rows where the number of times a species_id was observed is > 50

## Only keep the most common species
surveys_complete <- surveys_complete %>%
  filter(species_id %in% species_counts_min50$species_id)
# remember %in% from the previous lesson?
```

To make sure that everyone has the same data set, check that `surveys_complete` has 30463 rows and 13 columns by typing `dim(surveys_complete)`.

If `dim(surveys_complete)` returns different numbers (e.g. 30521 instead of 30463), make sure the original data was loaded with `read_csv` and not `read.csv`. Some NAs are handled differently by the two functions. See for instance the `sex` column.

Now that our data set is ready, we can save it as a CSV file in our `data` folder.

```
write_csv(surveys_complete, path = "data/surveys_complete.csv")
```

Time for a git commit

Summary of functions in this lesson

- `read_csv()` # load a csv formatted file into R memory
 - `str()` # check structure of the object and information about the class, length and content of each column
 - `View()` # invoke a spreadsheet-style data viewer
 - `select()` # select columns of a data frame
 - `filter()` # allows you to select a subset of rows in a data frame
 - `%>%` # pipes to select and filter at the same time
 - `mutate()` # create new columns based on the values in existing columns
 - `head()` # shows the first 6 rows
 - `group_by()` # split the data into groups (so that you can apply some analysis to each group)
 - `summarize()` # collapses each group into a single-row summary of that group
 - `n()` # get the the number of rows in your group
 - `mean()` # calculate the mean value of a vector
-
- `!is.na()` # test if there are no missing values
 - `print()` # print values to the console

- `min()` # return the minimum value of a vector
- `arrange()` # arrange rows by variables
- `desc()` # transform a vector into a format that will be sorted in descending order (use in `arrange()`)
- `count()` # counts the total number of records for each category
- `pivot_wider()` # reshape a data frame by a key-value pair across multiple columns (new version of `spread()`)
- `pivot_longer()` # reshape a data frame by collapsing into a key-value pair (new version of `gather()`)
- `n_distinct()` # get a count of unique values
- `write_csv()` # save to a csv formatted file

Page built on: 2023-04-18 13:20:33