Wrangling Basics

Emily Malcolm-White

Installing and Using Packages

Sometimes everything we need (data, functions, etc) are not available in base R. In R, expert users will package up useful things like data and functions into packages that be download and used.

First, you need to download the package from the right hand menu –> You only need to do this once.

In each new .qmd document, you need to call any packages you want to use but adding the code `library(packagename)` inside an R chunk.

The `tidyverse` package

In this class we will use the `tidyverse` package a lot.

```
library(tidyverse) #<1>
```

① Loads the `tidyverse` package

There are actually many commonly used packages wrapped up inside one `tidyverse` package.

Today we are specifically going to be talking about the package `dplyr` which is useful to manipulating data sets.

Figure 1: Credit: https://uopsych-r-bootcamp-2020.netlify.app/

`can_lang` dataset

In this class, we are going to be working with a dataset relating to the languages spoken at home by Canadian residents. Many Indigenous peoples exist in Canada with their own languages and cultures. Sadly, colonization has led to the loss of many of these languages. This data is a subset of data collected during the 2016 census.

Importing Data

What is a .csv file?

- It's plain text file that stores data
- Each value is seperated by a comma (,) - hence the name "*c*omma *s*eperated *v*alues"
- It's readable with tools like Excel, Good Sheets, R, and more.

How do we import it into R? Use `read.csv()`! Note that your data file (`.csv`) needs to be saved in the same folder as your notes template document (`.qmd`).

```
can_lang <- read.csv("data/can_lang.csv") #<1>
```

① Takes the `can_lang.csv` file (located in the same folder as your .qmd file), reads it into R, and saves it as the dataset `can_lang`

Alternatively, you can download it directly from the internet. Github user `ttimbers` hosts this file to share with the public at the link: https://raw.githubusercontent.com/ttimbers/canlang/master/inst/extdata/can_lang.csv

```
can_lang <- read.csv("https://raw.githubusercontent.com/ttimbers/canlang/master/inst/extda
```

① Takes the dataset located at the given url, reads it into R, and saves it as the dataset `can_lang`

Let's take a look at this data for a minute to see what information has been recorded. In the environment in the top left, if you click on the word `can_lang` (not the blue play button, the word itself) it will open the object so you can see what is saved inside. Alternatively you can use the `head()` function to display just the first few rows of the dataset.

```
head(can_lang)
```

```
              category                          language mother_tongue most_at_home
1 Aboriginal languages Aboriginal languages, n.o.s.           590          235
  most_at_work lang_known
1           30        665
 [ reached 'max' / getOption("max.print") -- omitted 5 rows ]
```

`filter()`

We can use the `filter` function to extract ***rows*** from the data that have a particular characteristic.

For example, we may be interested in only looking at only the languages in this dataset that are Aboriginal languages.

Start with the `can_lang` dataset, the pipe `%>%` means apply the action on the following line to the previous line.

```
can_lang  %>%                                 #<1>
  filter(category == "Aboriginal languages")   #<2>
```

① begin with the `can_lang` dataset
② only include the rows were the category variable is "Aboriginal languages"

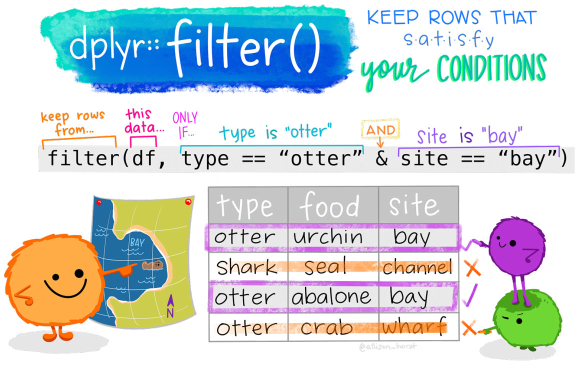

Figure 2: Artwork by @allisonhorst

```
              category                          language mother_tongue most_at_home
1 Aboriginal languages Aboriginal languages, n.o.s.           590          235
  most_at_work lang_known
1           30        665
 [ reached 'max' / getOption("max.print") -- omitted 66 rows ]
```

Some notes:

- the aboriginal languages is text/categorical and so quotation marks are needed.
- R doesn't care about whether they are double quotation marks (") or single ('). They work the same.
- If we don't assign it to an object, then it just prints out for us to see!

Oftentimes, we want to take our subset and give it a new name. This takes our subset and assigns it to a new dataset called `aboriginal_lang`.

```
aboriginal_lang <- can_lang  %>%             #<1>
  filter(category == "Aboriginal languages")
```

① The code `aboriginal_lang <-` takes the given data (the Aboriginal languages in the `can_lang` dataset) and saves it as a new object called `aboriginal_lang`.

Notes:

- Notice if you assign it to an object that it doesn't print out the contents.
- You'll see the new object in your environment on the top right —>

It can also be used with numeric criteria.

Suppose we want a list of all the languages in Canada that are spoken by less than 100 people as their mother tongue.

```
rare_lang <- can_lang  %>%         #<1>
  filter(mother_tongue < 100)      #<2>
                                   #<3>
```

① begin with the `can_lang` dataset
② only include the rows were the number of people who speak the language as their mother tongue is more than 100 people
③ data saved to the object `rare_lang`

The logical operators are given below:

Operator	Description
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to
`!x`	Not x
`x | y`	x OR y
`x & y`	x AND y

`select()`

`select` is used to extract only certain ***columns***. For example, perhaps we only want to print out a list names of the aboriginal languages (language column).

```
aboriginal_lang %>%  #<1>
  select(language)  #<2>
```

① Begin with the `aboriginal_lang` dataset
② only include the language column

```
                      language
1  Aboriginal languages, n.o.s.
2  Algonquian languages, n.i.e.
3                     Algonquin
```

```
4  Athabaskan languages, n.i.e.
5                     Atikamekw
6           Babine (Wetsuwet'en)
7                        Beaver
8                     Blackfoot
9                       Carrier
10                       Cayuga
 [ reached 'max' / getOption("max.print") -- omitted 57 rows ]
```

We can combine criteria together as well in one command with multiple pipes:

```
can_lang %>%
  filter(category == "Aboriginal languages") %>%
  select(language)
```

```
                       language
1  Aboriginal languages, n.o.s.
2  Algonquian languages, n.i.e.
3                     Algonquin
4  Athabaskan languages, n.i.e.
5                     Atikamekw
6           Babine (Wetsuwet'en)
7                        Beaver
8                     Blackfoot
9                       Carrier
10                       Cayuga
 [ reached 'max' / getOption("max.print") -- omitted 57 rows ]
```

arrange()

The `arrange` function allows us to order the rows of the data frame by the values of a particular column.

For example, arrange all the aboriginal languages in canada by from most to least spoken as mother tongue.

```
aboriginal_lang %>%
  arrange(desc(mother_tongue))  #<1>
```

① arranges the languages from the language with the most to the least people who speak the language as their mother tongue

```
              category     language mother_tongue most_at_home most_at_work
1 Aboriginal languages Cree, n.o.s.         64050        37950         7800
  lang_known
1      86115
 [ reached 'max' / getOption("max.print") -- omitted 66 rows ]
```

Note:

- use arrange(variable) to go from least to most
- use arrange(desc(variable)) to go from most to least, arrange(-variable) also works

`slice()`

The slice function will allow us to pick only a subset of the rows based on their numeric order (1st through last).

For example, if I want a list of the 10 most commonly spoken aboriginal languages.

```
aboriginal_lang %>%
  arrange(desc(mother_tongue)) %>%
  slice(1:10)   #<1>
```

① Only include the first 10 rows of the dataset

```
              category     language mother_tongue most_at_home most_at_work
1 Aboriginal languages Cree, n.o.s.         64050        37950         7800
  lang_known
1      86115
 [ reached 'max' / getOption("max.print") -- omitted 9 rows ]
```

`mutate()`

`mutate()` creates new columns that are functions of existing variables.

For example, if I want to create a new column called `mother_tongue_K` which represents the number of people who speak the language their mother tongue in thousands. You may want to save this new dataset over top of the original dataset so you could use this new column in the future.

Figure 3: Artwork by @allisonhorst

```
aboriginal_lang <- aboriginal_lang %>%
  mutate(mother_tongue_K = mother_tongue/1000) #<1>
```

① Creates a new column called `mother_tongue_K` calculated by taking the `mother_tongue` column and dividing it by 1000.

This can be useful for unit conversions. It also be useful for making new calculations based on existing data (for example, price and number of square feet could be used to calculate price per square foot).