

Lab 2B

Advanced Data Management

In this lab, we will work on data manipulation with the tidyverse package. Since a data frame is a key data structure in statistics and in R, it's important that we have good tools for dealing with them. In previous section of Lab2 we have already discussed some tools like the use of `[]`, `$` operators and `subset()` to extract subsets of data frames. However, other operations, like filtering, re-ordering, and collapsing, can often be tedious operations with the base-R functions whose syntax is not very intuitive.

To increase the readability of the code, we will use the `tidyr` and `dplyr` package of tidyverse collection of packages (<http://tidyverse.org>). One important contribution of these packages is providing a consistent and intuitive function names and syntax for data manipulation and for operating on data frames.

To start, install and load the tidyverse package either with the commands below or the R-Studio menu interface as we covered it in the first lab:

```
install.packages("tidyverse")
library(tidyverse) #all packages of tidyverse will be loaded
```

2.1. Data preparation

Go to the course website and download three files that represent the measurements of an experiment. This is, of course, a simplified example, but you often have situations where you are given data in strange formats and separate chunks. If you have many and/or very large files like this, the following functions will come in handy to quickly create a master data table that you can use for a statistical analysis.

In this experiment three lentil varieties (A, B, C) were tested with two levels of fertilizer (F1, F2) within three blocks. What you see below are sample results of the yield in kg/ha and the experimenter also entered the quality of the lentil harvest in the same cell for each experimental plot. There are three files that look like this if you open them in Excel:

BLOCK	VARIETY	F1	F2
1	A	510/fine	559/ok
1	B	632/ok	649/ok
1	C	dead	588/poor

- Let's start by automating the import of files. What works for three files in this exercise would also work for dozens or hundreds of separate files that you may receive for analysis. Sometimes, you only want to import a particular set of files with a certain character string in the file name. Here is how you can do this:
- First, we identify all csv files that end with `'_lab2b.csv'`. Check the result by calling the object it should show all the files that it found in the working directory that matched the pattern above:

```
filenames <- list.files(pattern = '_lab2b.csv')
filenames
```
- The next command reads all the files listed in the object `filenames`, with the `read.csv()` function. The function is wrapped with the list of filenames through the `lapply()` command which is used to do something with lists, hence `_lapply()`.
- All the files are then combined with the `bind_rows()` command. Note that the `bind_rows()` command only works if your variable names are the same in every file.

- In order to execute the `bind_rows()` function with a number of data tables, we need another wrapper `do.call()`, which can be applied with any function plus multiple objects. There may be some warning messages. If you get any, carefully check your result by calling the output object:

```
dat <- do.call(bind_rows, lapply(filenames, read.csv))
dat
```

- Next, let's check the structure of the data by `str(dat)`. Does it make sense to you? Is this data table well-organized? There clearly is some problem with the variables, which we'll address in a minute. First, let's check whether the data set contain any missing values by `is.na()` function: `is.na(dat)` For larger data tables you can use this command: `sum(is.na(dat))`

- Even though the missing data check is negative, we actually do have some missing data in the data frame. It's just not coded in a way that R can understand: 'dead' Therefore, we need to replace the 'dead' notes in the data table with NA:

```
dat <- na_if(dat, 'dead') # 'dead' represents NA
is.na(dat) #check again
```

- Again, for larger datasets there is a handy function to identify the rows with missing values. `complete.cases()` returns rows without missing values but in many cases you can reverse the meaning of functions by adding an exclamation mark in front. So, `!complete.cases()` will return the rows with missing values:

```
!complete.cases(dat)
which(!complete.cases(dat)) # returns the row number
```

2.2. Tidy the data table by tidyr package

One of the things that we covered in the lecture is that for analysis to work, each column of the data table must be a variable, and each row should represent characterize an experimental or sampling unit. Tidyr is a collection of functions that helps you efficiently convert messy data tables that you may receive from others into a tidy master data table fit for analysis!

The four most popular functions to tidy a mess are: `gather()`, `separate()`, `spread()`, and `unite()`.

2.2.1. `gather()` : from wide to narrow

You may have noticed that measurements of different experimental units are in the same row. For example, in the data table above, the cells 510/fine and 559/ok are completely different measurements under different fertilized treatments (F1 and F2) and they represent independent measurements. They have no business being in the same row, and this needs to be fixed for any subsequent analysis.

The function `gather()` will take multiple columns and move their content into different rows, while replicating all the relevant variable names and creating new variable names as needed. It produces a "long" data format from a "wide" one. The syntax is as follows

```
gather(data, key, value, columns to be gathered)
```

- data: a data frame we work with
- key: a new variable name for the columns that will be gathered
- value: a new variable name for the measurements



```
long_dat <- gather(dat, key = 'FERTILIZER', value = 'YIELD', c(F1, F2))
long_dat
```

2.2.2. `spread()`: from narrow to wide

The function `spread()` does the reverse of `gather()`. It formats a wide dataset by taking two columns (key and value) and spreading into multiple columns. This is not a common problem you are likely to encounter, but there are some instances where this function can come in handy. Automated data loggers, for example, tend to produce long lists of measurements, and sometimes it is handy to summarize repeat observations by first converting certain time intervals into separate columns. Here, just for the purpose of the exercise, we use our correctly formatted `long_dat` dataset, and convert it back to the incorrect format that we started with. The syntax of `spread()` function is:

```
spread(data, key, value)
```

- `key`: the (unquoted) name of the variable whose values will be used as column headings.
- `value`: the (unquoted) names of the column whose values will populate the cells.

To return the `long-dat` data frame to the original `dat` data table.

```
wide_dat <- spread(long_dat, key = FERTILIZER, value = YIELD)
wide_dat
```

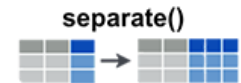


2.2.3. `separate()`: one column into multiple

The function `separate()` takes values inside a single character column and separates them into multiple columns. Simplified format:

```
separate(data, col, into, sep)
```

- `col`: unquoted column names
- `into`: character vector specifying the names of new variables to be created.
- `sep`: separator between columns. If character, is interpreted as a regular expression; if numeric, interpreted as positions to split at. Positive values start at 1 at the far-left



We can use `separate()` function to separate the column `YIELD` into two by using `'/'` as separator:

```
sep_dat <- separate(long_dat, col=YIELD, into=c('YIELD','QUALITY'), sep='/')
sep_dat
```

The `separate()` function does work with character strings, so while the result looks right, it is not yet suitable for analysis. The yield variable should be numeric.

```
str(sep_dat) # correcting columns types
sep_dat$YIELD <- as.numeric(sep_dat$YIELD)
str(sep_dat) # check again
```

2.2.4. `unite()`: multiple columns into one

The `unite()` function is the reverse of the `separate()`, which combine multiple columns into one. This functionality can be handy to create unique IDs that are generally useful in research to consistently tag an experimental unit or sampling unit. For example, for this dataset it would be sensible to create a unique ID from the block, variety and fertilizer information:

```
id <- unite(sep_dat[,1:3], col = 'ID', BLOCK, VARIETY, FERTILIZER, sep='-')
dat <- bind_cols(id, sep_dat)
dat
```



This data table now looks good and is ready to go for analysis! Note that we have overwritten the original dataset `dat` that we imported above by assigning the same name.

2.3. Data manipulation by dplyr package

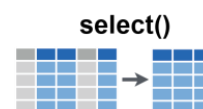
This next section repeats a fair amount of what we covered in the first section of the data management lab, but the dplyr package has more intuitive function names, a syntax that is easier to remember, and additional functionality that may come in handy. You don't critically need this section, but if you plan to do a lot of data management and analysis as part of your research, this package will make your life easier. If you have time, work your way through some of the popular functions covered below, plus the pipe operator covered at the end of the lab, which allows you to string multiple commands together into a data management pipeline without creating intermediate datasets.

2.3.1. select() columns by name

When you have a large master dataset, it is very common to subset this dataset for analyzing or graphing only certain portions of the total data. For selection of specific columns, we can use the `select()` function. Two simple arguments are required for `select()` function, one is the name of data frame, the other one is to identify the columns that you want to subset. The select function is more flexible than the `[,]` commands covered in the previous lab to specify what columns you would like to extract.

- Normally, you would write the output into a new object, but for the purpose of this exercise, we just look at the results of several versions of the command in the log window:

```
select(dat, c(1, 5, 6)) # select by numbered position
select(dat, ID, YIELD) # select by column names
select(dat, ID:YIELD) # range between two variable names
```



- Other useful way to select columns by names

```
select(dat, starts_with('F')) #select columns starting with 'F'
select(dat, contains('A')) # select columns containing letter 'A'
select(dat, ends_with('Y')) # select columns ending with letter 'Y'
```

- `select()` can also be used to change the order of columns. The following code switch the order of 'BLOCK' and 'VARIETY' but keep the rest:

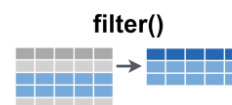
```
select(dat, VARIETY, BLOCK, everything())
```

- This example does not work here, but if you had numbered variables, you could select a subset as follows, e.g. to represent temperature for the June to Aug (columns: 'Temp6', 'Temp7', 'Temp8')

```
select(dat, num_range(prefix = 'Temp', range = 6:8))
```

2.3.2. filter() rows by conditions

We can do similar operations applicable to rows. For example, we may want remove a particular lentil variety from the analysis. In the dplyr package, this is covered by the `filter()` function, which has two arguments: data frame and conditions to filter: `filter(data table, condition)`:



- For example, `filter(dat, VARIETY != 'C')` will remove variety C from the dataset. Note that you need to put the "C" in quotation marks. That is necessary for character/factor variables. For numerical variables you would leave the quotation marks out. For instance, subset data frame with yield greater than 500: `filter(dat, YIELD > 500)`.
- **challenge:** You can create complex queries with multiple conditions. For example, to extract data for variety A from BLOCK1 plus data for variety B from BLOCK2 this would be the syntax:

```
filter(dat, VARIETY=='A' & BLOCK==1 | VARIETY == 'B' & BLOCK==2)
```

Let's investigate the elements of the command: the first part is to choose variety A from Block 1, the second one is for variety B from Block 2, then combine these two together by adding '|' in

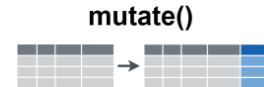
between, which stands for “or”.

- When working with large datasets, it is also sometimes useful to just randomly select a subset of rows to try out an analysis. Another application for randomly splits of datasets is bootstrapping and independent validations, which we will cover later in class. A random sample of 10 rows can be generated with the command: `sample_n(dat, 10)`

2.3.3. Calculations and adding columns by `mutate()`

Although there are other options in R, carrying out any type of calculation on your data table should be done with the transform command, which has the syntax: `mutate(data table, operation1, operation2, ...)`

For example we may want to change the units from *kg/acre* to *kg/hectares* or make a log-transformation of our YIELD variable:



- `dat2 <- mutate(dat, YIELD = YIELD * 0.4)` transforms the units of yield from *kg/acre* to *kg/hectares* and over-writes your original variable values. It's good practice to assign the results of transform operation to a new dataset, especially if you over-write variables.

- You can also do two or more operations in one transform statement:

```
dat3 <- mutate(dat, YIELD_HA = YIELD * 0.4, LOG_YIELD = log(YIELD))
dat3
```

- Here is how you would correct this by changing the units to *kg/ha* only in Block 1 data:

```
mutate(dat, YIELD = ifelse(BLOCK == 1, YIELD * 0.4, YIELD))
```

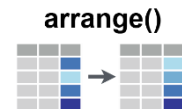
- Just like lab2A, clean the outliers from the database. NA means “missing” or “not applicable”:

```
mutate(dat, YIELD = ifelse(YIELD > 3000, NA, YIELD))
```

2.3.4. `arrange()` the orders of rows

- `arrange(data frame, variable)` can be used to reorder rows of a data frame by the values of certain variables. For example, to reorder the `dat` data table by `LOG_YIELD` (ascending).

```
arrange(dat3, LOG_YIELD)
```



- If you want to reorder the variable(s) in descending order:

```
arrange(dat3, desc(LOG_YIELD))
```

- Same for reorder data frame (`dat3`) by multiple variables (`LOG_YIELD`, `YIELD_HA`):

```
arrange(dat3, LOG_YIELD, YIELD_HA)
```

2.3.5. `rename()` columns

To rename column(s) of a data table, you can utilize `rename(data frame, new_name = old_name)` (No quotation needed!). For instance, in `dat3`, changing the name of column ‘VARIETY’ to ‘variety’:

```
dat4 <- rename(dat3, variety = VARIETY)
names(dat4)
```

Or change multiple variable names: `rename(dat3, block = BLOCK, variety = VARIETY)`





2.3.6. ..._join() multiple data tables

A frequent task in data management is that you have to join two tables that don't quite match up. One table may be longer than the other, for example if additional data was collected for a subset of the samples. This is where the join command comes in, which will match the new data to specific rows through a key variable. Normally this variable would be your ID, but for simplicity, we are using the variable VARIETY here, just to explore this functionality.

- You can create an additional dataset with this command:

```
new <- data.frame(VARIETY = c('A', 'B', 'D', 'G'), NEWVAR = c(42, 20, 17, 46))
```

- We can now play with various join options:

o left_join(data frame1, data frame2, by) prioritizes left data set (data frame1)	left_join()	
o right_join(data frame 1, data frame2, by) prioritizes right data set (data frame2)	right_join()	
o inner_join (data frame 1, data frame 2, by) only retains rows in both data sets (1 & 2)	inner_join()	
o full_join (data frame 1, data frame 2, by) retains all rows of both data sets	full_join()	

```
left_join(dat, new, by = 'VARIETY')  
inner_join(dat3, new, by = 'VARIETY')
```

- What about two data tables with different columns names for the key variable used for merging? Here is the syntax:

```
left_join(dat4, new, by = c('variety' = 'VARIETY'))
```

2.3.7. bind() data frames

Sometimes, you are in a situation where data tables perfectly align (i.e. you have the exact same number of rows, or the exact variable (column) structure. In that case you can use simplified bind commands, which you have already come across earlier in this lab:

```
bind_cols(dat1, dat2)    see bottom of page 3  
bind_rows(dat1, dat2)   see top of page 2
```

2.4. Pipe operator %>%

The pipeline operator %>% is very handy for stringing together multiple dplyr functions in a sequence. You can type the pipe with **Ctrl + Shift + M** if you use PC or **Cmd + Shift + M** for Mac.

```
dat2 <- dat %>%  
  filter(VARIETY != 'C') %>%  
  select(- QUALITY) %>%  
  mutate(YIELD_HA=YIELD*0.4)  
dat2
```

This is a very nice execution of a rather complex operation in one step. If you encounter errors, you can highlight portions of the code from the beginning to the end of a step to trouble-shoot.