Lesson 3 - Intro to Tidy Data: Go Long!

Contents

A quick intro to the Introduction to R for Data Science

This 'Introduction to R for Data Science' is brought to you by the Centre for the Analysis of Genome Evolution & Function's (CAGEF) bioinformatics training initiative. This CSB1020 was developed based on feedback on the needs and interests of the Department of Cell & Systems Biology and the Department of Ecology and Evolutionary Biology.

This lesson is the third in a 6-part series. The idea is that at the end of the series, you will be able to import and manipulate your data, make exploratory plots, perform some basic statistical tests, test a regression model, and make some even prettier plots and documents to share your results.

How do we get there? Today we are going to be learning about tidy data and how it makes data analysis less of a pain. We will perform some basic statistics on our newly transformed dataset. The next lesson will be data cleaning and string manipulation; this is really the battleground of coding - getting your data into the format where you can analyse it. After that, we will make all sorts of plots - from simple data exploration to interactive plots - this is always a fun lesson.

The structure of the class is a code-along style. It is hands on. The lecture AND code we are going through are available on GitHub for download at https://github.com/eacton/CAGEF, so you can spend the time coding and not taking notes. As we go along, there will be some challenge questions and multiple choice questions on Socrative. At the end of the class if you could please fill out a post-lesson survey (https://www.surveymonkey.com/r/VNQZ3KS), it will help me further develop this course and would be greatly appreciated.

Highlighting

`grey background` - a package, function, code, command or directory
italics - an important term or concept or an individual file or folder
bold - heading or a term that is being defined
blue text - named or unnamed hyperlink

Data Wrangling Is The
HARDEST
Part
Photo by Pascal under Creative Commons (CC0 1.0) - Adaptation by LutzFinger.com

Figure 1:

Packages Used in This Lesson

The following packages are used in this lesson:

`tidyverse` (`tidyr`, `dplyr`, `tibble`)

Please install and load these packages for the lesson. In this document I will load each package separately, but I will not be reminding you to install the package. Remember: these packages may be from CRAN OR Bioconductor.

Data Files Used in This Lesson

-ENV_pitlatrine.csv
-gapminder_wide.csv
-SPE_pitlatrine.csv

These files can be downloaded at https://github.com/eacton/CAGEF/tree/master/Lesson_3/data. Right-click on the filename and select 'Save Link As…' to save the file locally. The files should be saved in the same folder you plan on using for your R script for this lesson.

Or click on the blue hyperlink at the start of the README.md at https://github.com/eacton/CAGEF/tree/master/Lesson_3 to download the entire folder at DownGit.

Objective: At the end of this session you will know the principles of tidy data, and be able to subset and transform your data to perform simple calculations. You will be able to add new rows and columns to your data frame.

Dataset: Sequencing of the V3-V5 hypervariable regions of the 16S rRNA gene

16S rRNA sequencing of 30 latrines from Tanzania and Vietnam at different depths (multiples of 20cm). Microbial abundance is represented in Operational Taxonomic Units (OTUs). Operational Taxonomic Units (OTUs) are groups of organisms defined by a specified level of DNA sequence similarity at a marker gene (e.g. 97% similarity at the V4 hypervariable region of the 16S rRNA gene). Intrinsic environmental factors such as pH, temperature, organic matter composition were also recorded.

We have 2 csv files:

1. A metadata file (Naming conventions: [Country_LatrineNo_Depth]) with sample names and environmental variables.

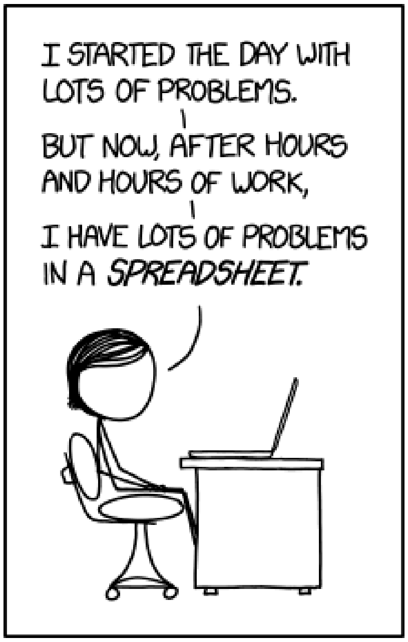
2. OTU abundance table.

B Torondel, JHJ Ensink, O Gunvirusdu, UZ Ijaz, J Parkhill, F Abdelahi, V-A Nguyen, S Sudgen, W Gibson, AW Walker, and C Quince. Assessment of the influence of intrinsic environmental and geographical factors on the bacterial ecology of pit latrines Microbial Biotechnology, 9(2):209-223, 2016. DOI:10.1111/1751-7915.12334

In this lesson we want to answer 3 simple questions:

```
- Which latrine depth has the greatest mean number of OTUs?
  (remember the Country_LatrineNo_Depth site encoding)
- Is there more Clostridia in Tanzania or Vietnam?
- Which site had the greatest number of Taxa represented?
```

Last lesson, we learned how to filter and select data subsets we were interested in. However, we can make data manipulation more efficient by controlling the overall structure or format of our data.


I STARTED THE DAY WITH LOTS OF PROBLEMS.
BUT NOW, AFTER HOURS AND HOURS OF WORK,
I HAVE LOTS OF PROBLEMS IN A SPREADSHEET.

Let's read in our dataset, store it in a variable, and remind ourselves about the original structure.

```
library(readr)

dat <- read_csv("data/SPE_pitlatrine.csv")
head(dat)
```

```
## # A tibble: 6 x 82
##   Taxa T_2_1 T_2_10 T_2_12 T_2_2 T_2_3 T_2_6 T_2_7 T_2_9 T_3_2 T_3_3 T_3_5
##   <ch> <int>  <int>  <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 2 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 3 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 4 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 5 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 6 Aci~     0      0      0     2     2     5     0    15     2     1     0
## # ... with 70 more variables: T_4_3 <int>, T_4_4 <int>, T_4_5 <int>,
## #   T_4_6 <int>, T_4_7 <int>, T_5_2 <int>, T_5_3 <int>, T_5_4 <int>,
## #   T_5_5 <int>, T_6_2 <int>, T_6_5 <int>, T_6_7 <int>, T_6_8 <int>,
## #   T_9_1 <int>, T_9_2 <int>, T_9_3 <int>, T_9_4 <int>, T_9_5 <int>,
## #   V_1_2 <int>, V_10_1 <int>, V_11_1 <int>, V_11_2 <int>, V_11_3 <int>,
## #   V_12_1 <int>, V_12_2 <int>, V_13_1 <int>, V_13_2 <int>, V_14_1 <int>,
## #   V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>, V_15_3 <int>,
## #   V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>, V_18_1 <int>,
## #   V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>, V_19_2 <int>,
## #   V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>, V_20_1 <int>,
## #   V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>, V_22_4 <int>,
## #   V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>, V_5_1 <int>,
## #   V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>, V_7_1 <int>,
## #   V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>, V_9_2 <int>,
## #   V_9_3 <int>, V_9_4 <int>
```

Intro to tidy data

Why tidy data?

Data cleaning (or dealing with ‘messy’ data) accounts for a huge chunk of data scientist’s time. Ultimately, we want to get our data into a ‘tidy’ format where it is easy to manipulate, model and visualize. Having a consistent data structure and tools that work with that data structure can help this process along.

Tidy data has:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

This seems pretty straight forward, and it is. It is the datasets you get that will not be straight forward. Having a map of where to take your data is helpful to unraveling its structure and getting it into a usable format.

The 5 most common problems with messy datasets are:

- common headers are values, not variable names
- multiple variables in one column
- variables stored in both rows and columns
- a single variable stored in multiple tables
- multiple types of observational units stored in the same table

Fortunately, Hadley has also given us the tools to solve these problems.

Intro to the tidyverse

The tidyverse is the universe of packages created by Hadley Wickham for data analysis. There are packages to help import, tidy, transform, model and visualize data. His packages are pretty popular, so he made a package to load all of his packages at once. This wrapper package is `tidyverse`. In this lesson series we have used `readr` and `readxl`, and we will be using `dplyr` and `tidyr` today, and `stringr` and `ggplot2` in future lessons. Install the package now.

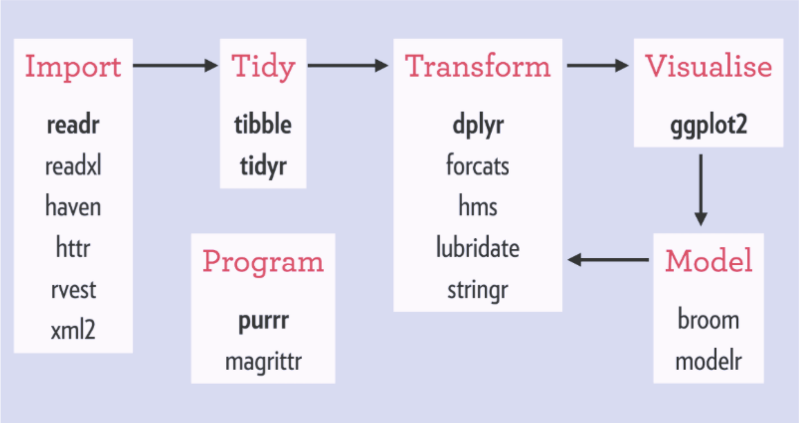


Figure 2:

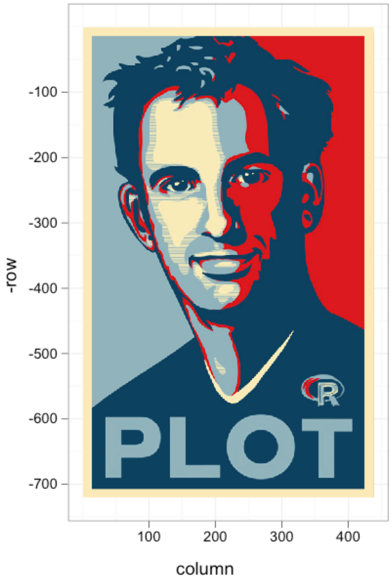


Figure 3:

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

Hadley has a large fan-base. Someone even made a plot of Hadley using his own package, `ggplot2`.

Back to the normalverse…

Assessing our data frame

Which tidy data rules might our data frame break?

At first glance we can see that the column names are actually 3 different variables: 'Country', 'LatrineNumber', and 'Depth'. This information will likely be useful in our study, as we expect different bacteria at different depths, sites, and geographical locations. Each of these is a variable and should have its own separate column.

We could keep the column names as the sample names (as they are meaningful to the researcher) and add the extra variable columns, or we could make up sample names (ie. Sample_1) knowing that the information is not being lost, but rather stored in a more useful format.

Some of the Taxa also appear to have an additional variable of information (ie. _Gp1), but not all taxa have this information. We could also make a separate column for this information.

Each result is the same observational unit (ie. relative abundances of bacteria), so having one table is fine.

Intro to tidyr

`tidyr` is a package with functions that help us turn our ‘messy’ data into ‘tidy’ data. It has 2 major workhorse functions and 2 other tidying functions:

1. `gather()` - convert a data frame from wide to long format
2. `spread()` - convert a data frame from long to wide format
3. `separate()` - split a column into 2 or more columns based on a string separator
4. `unite()` - merge 2 or more columns into 1 column using a string separator

`gather()` and `spread()` rely on key-value pairs to collapse or expand columns.

First let’s load our library. Loading `tidyverse` will include the `tidyr` package that the `gather()` function is from.

```
library(tidyverse)
```

```
## -- Attaching packages -------------------------------------------------------------------- tidyverse
```

```
## v ggplot2 3.0.0     v purrr   0.2.5
## v tibble  1.4.2     v dplyr   0.7.6
## v tidyr   0.8.1     v stringr 1.3.1
## v ggplot2 3.0.0     v forcats 0.3.0
```

```
## -- Conflicts ----------------------------------------------------------------------- tidyverse_confl
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Note that 8 different packages are loaded, and that 2 functions from the `stats` package have been replaced by functions of the same name by `dplyr`. Note that you can still access the `stats` version of the function by calling it directly as `stats::filter()`.

We can use the `gather()` function to collect our columns. This will make our dataset ‘long’ instead of ‘wide’.

We need to provide `gather()` with information on our new columns. The first argument is our data frame, the second argument is the **key**, which is the name for the variable we want to gather (a set of column names). In this case our columns represent latrine sites, so we can assign the name ‘Sites’ to our key. The next argument is the **value**, which is the name for the measurements (usually numeric or integer) we have. In this case our values are Operational Taxonomic Units, so we can assign the name ‘OTUs’ to our value. The third argument is all of the columns that we want to gather. You can specify the columns by listing their names or positions.

```
dat %>% gather(key = Site, value = OTUs, T_2_1:V_9_4) %>% head()
```

```
## # A tibble: 6 x 3
##   Taxa                Site   OTUs
##   <chr>               <chr> <int>
## 1 Acidobacteria_Gp1   T_2_1     0
## 2 Acidobacteria_Gp10 T_2_1     0
## 3 Acidobacteria_Gp14 T_2_1     0
## 4 Acidobacteria_Gp16 T_2_1     0
## 5 Acidobacteria_Gp17 T_2_1     0
## 6 Acidobacteria_Gp18 T_2_1     0
```

```
#equivalent to
dat %>% gather(key = Site, value = OTUs, 2:82) %>% head()
#equivalent to
dat %>% gather(Site, OTUs, -1) %>% head()
#equivalent to
dat %>% gather(Site, OTUs, -Taxa) %>% head()
```

In the above examples "-" means gather every column except the 1st, or gather every column except Taxa. Taxa would still be retained as a column but its elements are not grouped in with 'Sites' (ie. we do not want 'V_9_4', 'T_2_9', and 'Clostridia' gathered into the same column).

Let's save the last variation into a data frame called spread_dat.

```
spread_dat <- dat %>% gather(Site, OTUs, -Taxa)
```

Note how the dimensions of your dataframe have changed relative to dat. Instead of 52 rows and 82 columns, we now have a data frame with 4212 rows and 3 columns (which is the 81 columns we gathered x 52 rows). spread_dat is now in a long format instead of wide.

Next, we can use the `separate()` function to get the Country, Latrine_Number, and Depth information from our Site column. `separate()` takes in your dataframe, the name of the column to be split, the names of your new columns, and the character that you want to split the columns by (in this case an underscore). Note that the default is to remove your original column - if you want to keep it, you can add the additional argument `remove = FALSE`, keeping in mind that you now have redundant data.

```
split_dat <- spread_dat %>%
  separate(Site, c("Country", "Latrine_Number", "Depth"), sep = "_")
## #equivalent to
## split_dat <- spread_dat %>%
##   separate(Site, c("Country", "Latrine_Number", "Depth"), sep = "_", remove = TRUE)
```

We may also want to do this for the 'Group' of Acidobacteria. Try the code, but do not save the answer in a variable.

```
split_dat %>% separate(Taxa, c("Taxa", "Group"), sep = "_Gp") %>% head(20)
```

```
## Warning: Expected 2 pieces. Missing pieces filled with `NA` in 3078 rows
## [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
## 33, 34, ...].

## # A tibble: 20 x 6
##    Taxa                Group Country Latrine_Number Depth  OTUs
##    <chr>               <chr> <chr>   <chr>          <chr> <int>
##  1 Acidobacteria       1     T       2              1         0
##  2 Acidobacteria       10    T       2              1         0
##  3 Acidobacteria       14    T       2              1         0
##  4 Acidobacteria       16    T       2              1         0
##  5 Acidobacteria       17    T       2              1         0
##  6 Acidobacteria       18    T       2              1         0
##  7 Acidobacteria       21    T       2              1         0
##  8 Acidobacteria       22    T       2              1         0
##  9 Acidobacteria       3     T       2              1         0
## 10 Acidobacteria       4     T       2              1         0
## 11 Acidobacteria       5     T       2              1         0
## 12 Acidobacteria       6     T       2              1         0
## 13 Acidobacteria       7     T       2              1         0
## 14 Acidobacteria       9     T       2              1         0
## 15 Actinobacteria      <NA>  T       2              1       110
## 16 Alphaproteobacteria <NA>  T       2              1        11
## 17 Anaerolineae        <NA>  T       2              1         2
## 18 Bacilli             <NA>  T       2              1        19
## 19 Bacteroidia         <NA>  T       2              1      1547
## 20 Betaproteobacteria  <NA>  T       2              1         2
```

We get a warning from R that it has filled in 'NA' for the bacteria that did not have groups. Note that I chose to split Taxa using '_Gp' since I did not need 'Gp'.

Challenge

Use the `glimpse()` function to look at the type of each variable in our new data frame, split_dat. Are those the types you expected? Why or why not? How is `glimpse()` different from the `str()` function?

There is a useful function `group_by()` that you can use to group variables or sets of variables together. This is useful for calculations and plotting on subsets of your data without having to turn your variables into factors. Say I wanted to look at a combination of Country and Well Depth. While visually, you wouldn't notice any changes to your data frame, if you look at the structure it will now be a 'grouped_df'. There are 15 groupings resulting from Country and Depth. After I have performed my desired operation, I can return my data frame to its original structure by calling `ungroup()`. First we will examine the structure of grouped and ungrouped output without any additional operations.

```r
str(split_dat)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    4212 obs. of  5 variables:
##  $ Taxa          : chr  "Acidobacteria_Gp1" "Acidobacteria_Gp10" "Acidobacteria_Gp14" "Acidobacteria
##  $ Country       : chr  "T" "T" "T" "T" ...
##  $ Latrine_Number: chr  "2" "2" "2" "2" ...
##  $ Depth         : chr  "1" "1" "1" "1" ...
##  $ OTUs          : int  0 0 0 0 0 0 0 0 0 0 ...
```

```r
str(split_dat %>% group_by(Country, Depth))
```

```
## Classes 'grouped_df', 'tbl_df', 'tbl' and 'data.frame':  4212 obs. of  5 variables:
##  $ Taxa          : chr  "Acidobacteria_Gp1" "Acidobacteria_Gp10" "Acidobacteria_Gp14" "Acidobacteria
##  $ Country       : chr  "T" "T" "T" "T" ...
##  $ Latrine_Number: chr  "2" "2" "2" "2" ...
##  $ Depth         : chr  "1" "1" "1" "1" ...
##  $ OTUs          : int  0 0 0 0 0 0 0 0 0 0 ...
##  - attr(*, "vars")= chr  "Country" "Depth"
##  - attr(*, "drop")= logi TRUE
##  - attr(*, "indices")=List of 15
##   ..$ : int  0 1 2 3 4 5 6 7 8 9 ...
##   ..$ : int  52 53 54 55 56 57 58 59 60 61 ...
##   ..$ : int  104 105 106 107 108 109 110 111 112 113 ...
##   ..$ : int  156 157 158 159 160 161 162 163 164 165 ...
##   ..$ : int  208 209 210 211 212 213 214 215 216 217 ...
##   ..$ : int  624 625 626 627 628 629 630 631 632 633 ...
##   ..$ : int  520 521 522 523 524 525 526 527 528 529 ...
##   ..$ : int  260 261 262 263 264 265 266 267 268 269 ...
##   ..$ : int  312 313 314 315 316 317 318 319 320 321 ...
##   ..$ : int  1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 ...
##   ..$ : int  364 365 366 367 368 369 370 371 372 373 ...
##   ..$ : int  1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 ...
##   ..$ : int  1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 ...
##   ..$ : int  1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 ...
##   ..$ : int  2652 2653 2654 2655 2656 2657 2658 2659 2660 2661 ...
##  - attr(*, "group_sizes")= int  104 52 52 260 260 156 260 104 156 52 ...
##  - attr(*, "biggest_group_size")= int 1040
##  - attr(*, "labels")='data.frame':   15 obs. of  2 variables:
##   ..$ Country: chr  "T" "T" "T" "T" ...
##   ..$ Depth  : chr  "1" "10" "12" "2" ...
##   ..- attr(*, "vars")= chr  "Country" "Depth"
##   ..- attr(*, "drop")= logi TRUE
```

```
str(split_dat %>% group_by(Country, Depth) %>% ungroup())
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    4212 obs. of  5 variables:
##  $ Taxa          : chr  "Acidobacteria_Gp1" "Acidobacteria_Gp10" "Acidobacteria_Gp14" "Acidobacteria_
##  $ Country       : chr  "T" "T" "T" "T" ...
##  $ Latrine_Number: chr  "2" "2" "2" "2" ...
##  $ Depth         : chr  "1" "1" "1" "1" ...
##  $ OTUs          : int  0 0 0 0 0 0 0 0 0 0 ...
```

Now we can see an example of how `grouped_by` in action with `summarize()`, can easily calculate summary statistics for groups of data. Whereas in our messy data frame it was difficult to do calculations based on Country, Well Number or Latrine Depth, this is now an easy task. Let's get the mean, median, standard deviation and maximum for the number of OTUs collected in Tanzania vs Vietnam.

```
split_dat %>%
  group_by(Country) %>%
  summarize(mean = mean(OTUs), median = median(OTUs), sd = sd(OTUs), maximum = max(OTUs))
```

```
## # A tibble: 2 x 5
##   Country  mean median    sd maximum
##   <chr>   <dbl>  <dbl> <dbl>   <dbl>
## 1 T        122.      0  896.   17471
## 2 V        186.      0  863.   17572
```

In dealing with grouped data, we no longer have to grab a Country by subsetting or using helper functions to grab letters from their names. Group by recognizes that we have 2 countries and will perform calculations for both of them.

Now that we have tidy data, let's answer our questions:

```
- Which latrine depth has the greatest mean number of OTUs?
  (remember the Country_LatrineNo_Depth site encoding)
- Is there more Clostridia in Tanzania or Vietnam?
- Which site had the greatest number of Taxa represented?
```

Okay, Go!

Which latrine depth has the greatest mean number of OTUs?

```
split_dat %>%
  group_by(Latrine_Number) %>%
  summarize(mean = mean(OTUs)) %>%
  arrange(desc(mean))
```

```
## # A tibble: 22 x 2
##    Latrine_Number  mean
##    <chr>          <dbl>
##  1 17              415.
##  2 16              403.
##  3 11              338.
##  4 20              335.
##  5 10              328.
##  6 8               236.
##  7 15              207.
##  8 3               206.
##  9 18              201.
## 10 12              188.
## # ... with 12 more rows
```

Without data being in ‘tidy’ format - with all variables that were mashed together into a Site (Country_LatrineNo_Depth) having their own columns - this is a difficult question to answer. Once Latrine_Number is a variable, we can simply group our data to perform our mean calculation and get an answer.

Is there more Clostridia in Tanzania or Vietnam?

```
split_dat %>%
  filter(Taxa == "Clostridia") %>%
  group_by(Country) %>%
  summarise(sum = sum(OTUs), mean = mean(OTUs)) %>%
  arrange(desc(sum))
```

```
## # A tibble: 2 x 3
##   Country    sum  mean
##   <chr>    <int> <dbl>
## 1 V       176079 3386.
## 2 T       101217 3490.
```

Again, being able to filter by Taxa and group by Country (as an isolated variable) helps a lot. With `dplyr` syntax we can perform all data manipulations and calculations in a code block that is readable.

Which site had the greatest number of Taxa represented?

```
split_dat %>%
  group_by(Country, Latrine_Number, Depth) %>%
  filter(OTUs != 0) %>%
  summarize(count = n()) %>%
  arrange(desc(count))
```

```
## # A tibble: 81 x 4
## # Groups:   Country, Latrine_Number [28]
##    Country Latrine_Number Depth count
##    <chr>   <chr>          <chr> <int>
##  1 V       7              3        36
##  2 V       22             1        31
##  3 V       7              2        29
##  4 T       2              6        27
##  5 V       11             1        27
##  6 V       11             3        27
##  7 T       2              9        26
##  8 V       15             2        26
##  9 V       15             3        26
## 10 V       18             4        26
## # ... with 71 more rows
```

Figure 4:

Since we can group by the 3 variables that were in the Site name, there is no disadvantage to having our data in tidy format compared to our original wide data frame. However now we are able to filter for non-zero OTUs, which was impossible in the wide format. Since we know from earlier in the lesson that each Taxa is only represented once for each site, we only have to count and order the number of observations to get our answer.

To get data back into its original format, there are reciprocal functions in the tidyr package, making it possible to switch between wide and long formats.

Fair question: But you've just been telling me how great the 'long' format is?!?! Why would I want the wide format again???

Honest answer: Note that our original data frame was 52 rows and expanded to 4212 rows in the long format. When you have, say, a genomics dataset you might end up with 6,000 rows expanding to 600,000 rows. You probably want to do your calculations and switch back to the more 'human readable' format. Sure, I can save a data frame with 600,000 rows, but I can't really send it to anyone because LibreOffice or Excel might crash opening it.

Challenge

Collapse Country, Latrine_Number and Depth back into one variable, 'Site', using the `unite()` function. Store the output in a data frame called unite_dat.

Challenge

Use the `spread()` function to turn unite_dat into the wide shape of our original dataset. Save the output into a data frame called 'spread_dat'.

Challenge

Read in the gapminder_wide.csv. What rules of tidy data does it break? Transform the dataset to the format below. How many rows do you have?

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.801	8425333	779.4453
Afghanistan	Asia	1957	30.332	9240934	820.8530

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1962	31.997	10267083	853.1007
Afghanistan	Asia	1967	34.020	11537966	836.1971
Afghanistan	Asia	1972	36.088	13079460	739.9811
Afghanistan	Asia	1977	38.438	14880372	786.1134

Adding rows and columns to data frames

We are going to use a subset of the collected environmental data from the study to practice adding rows and columns to a data frame. We can use our pipe while reading in our data to do this subsetting.

```
jdat <- read_csv("data/ENV_pitlatrine.csv") %>% select(1:4)
```

```
## Parsed with column specification:
## cols(
##   Samples = col_character(),
##   pH = col_double(),
##   Temp = col_double(),
##   TS = col_double(),
##   VS = col_double(),
##   VFA = col_double(),
##   CODt = col_integer(),
##   CODs = col_integer(),
##   perCODsbyt = col_integer(),
##   NH4 = col_double(),
##   Prot = col_double(),
##   Carbo = col_double()
## )
```

```
glimpse(jdat)
```

```
## Observations: 81
## Variables: 4
## $ Samples <chr> "T_2_1", "T_2_10", "T_2_12", "T_2_2", "T_2_3", "T_2_6"...
## $ pH      <dbl> 7.82, 9.08, 8.84, 6.49, 6.46, 7.69, 7.48, 7.60, 7.55, ...
## $ Temp    <dbl> 25.1, 24.2, 25.1, 29.6, 27.9, 28.7, 29.8, 25.0, 28.8, ...
## $ TS      <dbl> 14.53, 37.76, 71.11, 13.91, 29.45, 65.52, 36.03, 46.87...
```

There is a function from `dplyr`, `bind_rows()`, which tries to save you from human error when joining a row of observations to a data frame by ensuring that your column names match (so data is not entered in the wrong column). If you do not give column names, an error will be thrown.

What data structure should you be using to add a row to a data frame?

If you give `bind_rows()` the correct column information in vector format, it will only add a vector as a row IF THE CHARACTER TYPE MATCHES ALL DATA FRAME COLUMN TYPES (ie. if our data frame was all character data). Otherwise, it actually gives you the useful error that your character types are not matching.

```
bind_rows(jdat, c(Samples = "V_2_10", pH = 6, Temp = 5, TS = 5)) %>% tail()
```

```
## Error in bind_rows_(x, .id): Column `pH` can't be converted from numeric to character
```

A list allows us to add the data we can't add in vector format, by retaining the data types that match the data frame columns.

```
bind_rows(jdat, list(Samples = "V_2_10", pH = 6, Temp = 5, TS = 5)) %>% tail()
```

```
## # A tibble: 6 x 4
##   Samples    pH  Temp    TS
##   <chr>   <dbl> <dbl> <dbl>
## 1 V_8_2    9.3   19.5  36.8
## 2 V_9_1    7.3   18.8  31.3
## 3 V_9_2    4.04  20.5  32
## 4 V_9_3    4.36  19.9  92.6
## 5 V_9_4    4.3   19.8  36.5
## 6 V_2_10   6      5     5
```

If you give `bind_rows()` the correct column information in list format, it will add your row.

A *sanity check* is making sure things are turning out as you expect them to - it is a way of checking your (or others') assumptions. It is particularly useful in the data exploration stage when you are getting familiar with your data set before you try to run any type of model. However, it is also good to make sure a function is behaving as you expect it to, or trouble-shooting odd behaviours

Here is a sanity check on switching the order of the columns.

```
bind_rows(jdat, list(Samples = "V_2_10", Temp = 5, pH = 6, TS = 5)) %>% tail()
```

```
## # A tibble: 6 x 4
##   Samples    pH  Temp    TS
##   <chr>   <dbl> <dbl> <dbl>
## 1 V_8_2    9.3   19.5  36.8
## 2 V_9_1    7.3   18.8  31.3
## 3 V_9_2    4.04  20.5  32
## 4 V_9_3    4.36  19.9  92.6
## 5 V_9_4    4.3   19.8  36.5
## 6 V_2_10   6      5     5
```

Since we have named our values, `bind_rows()` will add the value to the appropriate column.

What happens if column name do not match? `bind_rows()` will match your columns as much as possible, and then create new columns for the data that does not fit at the end of your data frame. Note that 'NA's will be created for all missing data.

```
bind_rows(jdat, list(Turtles = "V_2_10", pH = 5, Volatility = 5, TS = 5)) %>% tail()
```

```
## # A tibble: 6 x 6
##   Samples    pH  Temp    TS Turtles Volatility
##   <chr>   <dbl> <dbl> <dbl> <chr>        <dbl>
## 1 V_8_2    9.3   19.5  36.8 <NA>            NA
## 2 V_9_1    7.3   18.8  31.3 <NA>            NA
## 3 V_9_2    4.04  20.5  32   <NA>            NA
## 4 V_9_3    4.36  19.9  92.6 <NA>            NA
## 5 V_9_4    4.3   19.8  36.5 <NA>            NA
## 6 <NA>     5     NA     5   V_2_10           5
```

Why doesn't jdat change after all these operations?

Whenever you are adding rows or columns, I strongly advise checking to see that the dimensions of the resulting data frame are what you expect.

```
jdat <- bind_rows(jdat, list(Samples = "V_2_10", pH = 6, Temp = 5, TS = 5))
dim(jdat)
```

```
## [1] 82  4
```

Challenge

Make a column and add it to jdat using the `dplyr` function `bind_cols()`. Do you forsee any problems in using this function?

All of the joins

Often we have more than one data table that shares a common attribute. For example, with our current dataset, we have other variables (such as pH) for a sample, as well as our OTU table, both of which have site IDs (ie. T_2_9). We want to merge these into one table.

Joins can be tricky, so we are going to use a subset of our tidy data such that we can easily observe the output of our join.

Joins use a 'key' by which we can match up our data frames. When we look at our data frames we can see that they have matching information in 'Samples'. We are going to reduce our dataset by only keeping non-zero values so we can see how the join functions work a bit more easily.

Challenge

Filter unite_dat to remove all non-zero values, and store it in an object ndat. How many rows in unite_dat had the value of zero? Sort ndat to keep the top 20 rows with the highest OTUs.

Answer:

```
ndat <- unite_dat %>% filter(OTUs!=0) %>% arrange(desc(OTUs)) %>% .[1:20, ]
```

We will remove some observations from jdat, just so our key columns don't match perfectly.

```
jdat <- jdat[-c(4,7,9,12,20,22) , 1:4]
```

There are 2 types of joins:

1. *mutating joins* - uses a key to match observations and combines variables from 2 tables (adding columns and potentially rows)
2. *filtering joins* - uses a key to match observations, to subset observations (rows)

In these examples we are joining 2 dataframes: x and y.

Inner join is a mutating join. Inner join returns all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

A set of graphics from *'R for Data Science'* makes the description clearer:

```
inner_join(ndat, jdat, by = c("Site" = "Samples"))
```

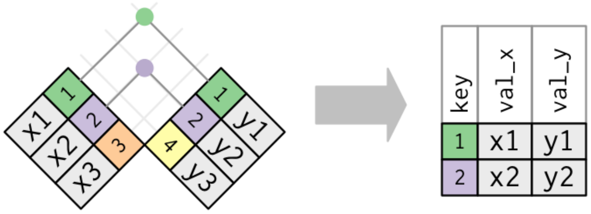
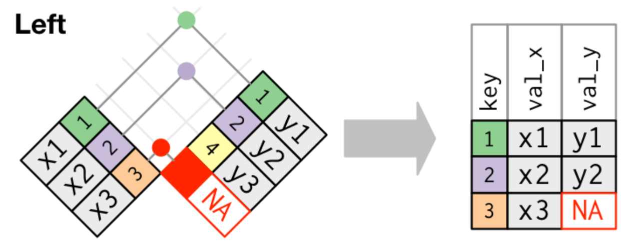


Figure 5: img borrowed from r4ds.had.co.nz

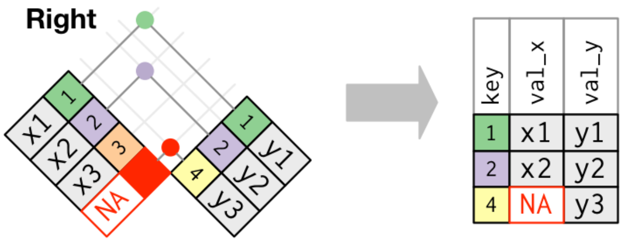
```
## # A tibble: 18 x 6
##    Taxa                Site    OTUs    pH  Temp    TS
##    <chr>               <chr>  <int> <dbl> <dbl> <dbl>
##  1 Clostridia          V_16_2 17572  6.26  16.3  24
##  2 Clostridia          T_2_9  17471  7.6   25    46.9
##  3 Clostridia          V_17_2 13875  8.14  15.4  25.6
##  4 Clostridia          T_2_6  12169  7.69  28.7  65.5
##  5 Bacteroidia         V_17_1 11392  7.39  15.7  25.3
##  6 Clostridia          V_10_1 11180  9.12  18.6  46.3
##  7 Clostridia          T_2_3  10944  6.46  27.9  29.4
##  8 Unknown             T_9_2  10624  7.86  28.5  13.6
##  9 Clostridia          T_4_4  10545  7.84  26.3  28.8
## 10 Clostridia          V_16_1 10043  7.83  17.7  25.1
## 11 Clostridia          T_5_3   8981  7.53  27.5  12.6
## 12 Alphaproteobacteria V_3_2   8660  8.08  20.7  52
## 13 Clostridia          V_15_1  8244  7.44  18.9  25.4
## 14 Clostridia          V_17_1  7722  7.39  15.7  25.3
## 15 Clostridia          V_18_1  7481  7.09  22.5  35.1
## 16 Alphaproteobacteria V_3_1   7322  8.9   22.4  56.2
## 17 Clostridia          V_18_4  6246  8.25  19.4  29.9
## 18 Clostridia          T_2_1   6213  7.82  25.1  14.5
```

We can see that there are 18 rows in the resulting data frame, and columns from jdat have been added. Rows from ndat that did not have a matching site in jdat were removed.

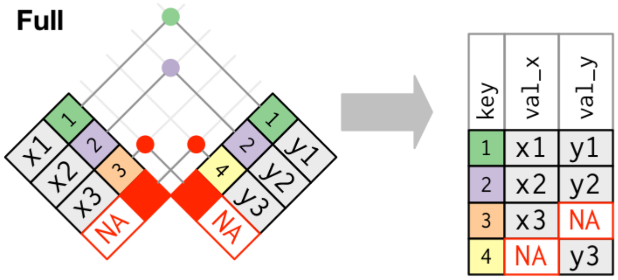
Outer joins are a set of mutating joins. There are 3 outer joins: left, right, and full.



key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA



key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3



key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

Left join returns all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

```r
left_join(ndat, jdat, by = c("Site" = "Samples"))
```

```
## # A tibble: 20 x 6
##    Taxa                Site    OTUs    pH  Temp    TS
##    <chr>               <chr>  <int> <dbl> <dbl> <dbl>
##  1 Clostridia          V_16_2 17572  6.26  16.3  24
##  2 Clostridia          T_2_9  17471  7.6   25    46.9
##  3 Clostridia          V_17_2 13875  8.14  15.4  25.6
##  4 Clostridia          T_2_6  12169  7.69  28.7  65.5
##  5 Bacteroidia         V_17_1 11392  7.39  15.7  25.3
##  6 Clostridia          V_10_1 11180  9.12  18.6  46.3
##  7 Clostridia          T_2_3  10944  6.46  27.9  29.4
##  8 Unknown             T_9_2  10624  7.86  28.5  13.6
##  9 Clostridia          T_4_4  10545  7.84  26.3  28.8
## 10 Clostridia          V_16_1 10043  7.83  17.7  25.1
## 11 Clostridia          T_2_2   8999 NA     NA    NA
## 12 Clostridia          T_5_3   8981  7.53  27.5  12.6
## 13 Alphaproteobacteria V_3_2   8660  8.08  20.7  52
## 14 Clostridia          V_15_1  8244  7.44  18.9  25.4
## 15 Clostridia          V_17_1  7722  7.39  15.7  25.3
## 16 Betaproteobacteria  T_4_3   7710 NA     NA    NA
## 17 Clostridia          V_18_1  7481  7.09  22.5  35.1
## 18 Alphaproteobacteria V_3_1   7322  8.9   22.4  56.2
```

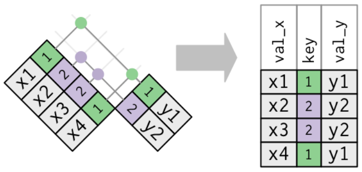


Figure 6: img borrowed from r4ds.had.co.nz

```
## 19 Clostridia          V_18_4  6246  8.25  19.4  29.9
## 20 Clostridia          T_2_1   6213  7.82  25.1  14.5
```

That means that we will have our 20 rows from ndat, and any items that weren't found in jdat, in this case T__2__2 and T__4__3 will be filled with NA.

Right join returns all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

```
right_join(ndat, jdat, by = c("Site" = "Samples"))
```

```
## # A tibble: 77 x 6
##    Taxa       Site    OTUs    pH  Temp    TS
##    <chr>      <chr>  <int> <dbl> <dbl> <dbl>
##  1 Clostridia T_2_1   6213  7.82  25.1  14.5
##  2 <NA>       T_2_10    NA  9.08  24.2  37.8
##  3 <NA>       T_2_12    NA  8.84  25.1  71.1
##  4 Clostridia T_2_3  10944  6.46  27.9  29.4
##  5 Clostridia T_2_6  12169  7.69  28.7  65.5
##  6 Clostridia T_2_9  17471  7.6   25    46.9
##  7 <NA>       T_3_3     NA  7.68  28.9  14.6
##  8 <NA>       T_3_5     NA  7.69  28.7  14.9
##  9 Clostridia T_4_4  10545  7.84  26.3  28.8
## 10 <NA>       T_4_5     NA  7.95  27.9  46.8
## # ... with 67 more rows
```

That means that we will have our 75 rows from jdat, and any items that weren't found in ndat will be filled with NA. Since there are 77 rows in the final data frame, this means there must have been multiple rows matching from ndat. In other words, we had a duplicate key (Site) in ndat.

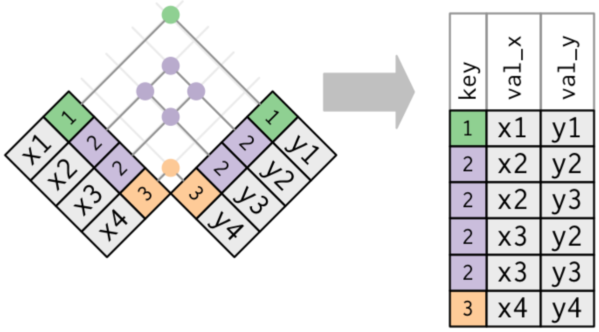
Let's try to find which Site was duplicated by using `n()`.

```
right_join(ndat, jdat, by = c("Site" = "Samples")) %>%
  group_by(Site) %>%
  filter(n()>1)
```

```
## # A tibble: 2 x 6
## # Groups:   Site [1]
##   Taxa        Site    OTUs    pH  Temp    TS
##   <chr>       <chr>  <int> <dbl> <dbl> <dbl>
## 1 Bacteroidia V_17_1 11392  7.39  15.7  25.3
## 2 Clostridia  V_17_1  7722  7.39  15.7  25.3
```

Note that Bacteriodia and Clostridia have different OTUs (from ndat), but the same pH, Temp and TS (from jdat).

Full join returns all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing.



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

Figure 7: img borrowed from r4ds.had.co.nz

```
full_join(ndat, jdat, by = c("Site" = "Samples"))
```

```
## # A tibble: 79 x 6
##    Taxa        Site    OTUs    pH  Temp    TS
##    <chr>       <chr>  <int> <dbl> <dbl> <dbl>
##  1 Clostridia  V_16_2 17572  6.26  16.3  24
##  2 Clostridia  T_2_9  17471  7.6   25    46.9
##  3 Clostridia  V_17_2 13875  8.14  15.4  25.6
##  4 Clostridia  T_2_6  12169  7.69  28.7  65.5
##  5 Bacteroidia V_17_1 11392  7.39  15.7  25.3
##  6 Clostridia  V_10_1 11180  9.12  18.6  46.3
##  7 Clostridia  T_2_3  10944  6.46  27.9  29.4
##  8 Unknown     T_9_2  10624  7.86  28.5  13.6
##  9 Clostridia  T_4_4  10545  7.84  26.3  28.8
## 10 Clostridia  V_16_1 10043  7.83  17.7  25.1
## # ... with 69 more rows
```

The full join returns all of the rows from both ndat and jdat - we have the 75 rows from jdat, plus the second V_17_1 as seen in the right_join and T_2_2 and T_4_3 that we present in ndat but NA in jdat for a total of 79 rows.

Lastly, we have the 2 filtering joins. These will not add columns, but rather filter the rows based on what is present in the second data frame.

Semi join returns all rows from x where there are matching values in y, keeping just columns from x. A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x.

```
semi_join(ndat, jdat, by = c("Site" = "Samples"))
```

```
## # A tibble: 18 x 3
##    Taxa                Site    OTUs
##    <chr>               <chr>  <int>
##  1 Clostridia          V_16_2 17572
##  2 Clostridia          T_2_9  17471
##  3 Clostridia          V_17_2 13875
##  4 Clostridia          T_2_6  12169
##  5 Bacteroidia         V_17_1 11392
##  6 Clostridia          V_10_1 11180
##  7 Clostridia          T_2_3  10944
##  8 Unknown             T_9_2  10624
##  9 Clostridia          T_4_4  10545
## 10 Clostridia          V_16_1 10043
## 11 Clostridia          T_5_3   8981
## 12 Alphaproteobacteria V_3_2   8660
## 13 Clostridia          V_15_1  8244
## 14 Clostridia          V_17_1  7722
## 15 Clostridia          V_18_1  7481
## 16 Alphaproteobacteria V_3_1   7322
## 17 Clostridia          V_18_4  6246
## 18 Clostridia          T_2_1   6213
```

`semi_join()` returns the 18 rows of ndat that have a Site match in jdat. Note that the columns from jdat have not been added.

Anti join returns all rows from x where there are not matching values in y, keeping just columns from x.

```
anti_join(ndat, jdat, by = c("Site" = "Samples"))
```

```
## # A tibble: 2 x 3
##   Taxa               Site   OTUs
##   <chr>              <chr> <int>
## 1 Clostridia         T_2_2  8999
## 2 Betaproteobacteria T_4_3  7710
```

This returns our 2 rows in ndat that did not have a match in jdat. Note that the columns from jdat have not been added.

Challenge

Given the definitions for the inner_, left_, right_, full_, semi_ and anti_joins, what would you expect the resulting data frame to be if jdat and ndat were reversed? Write down the number of rows and columns you would expect the data frame to have, then run the code. Did you find any surprises?

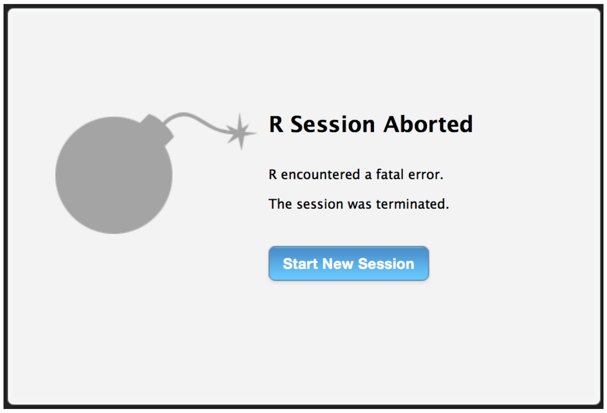
Resources

https://github.com/wmhall/tidyr_lesson/blob/master/tidyr_lesson.md
http://vita.had.co.nz/papers/tidy-data.pdf
https://thinkr.fr/tidyverse-hadleyverse/
http://stat545.com/bit001_dplyr-cheatsheet.html
http://dplyr.tidyverse.org/articles/two-table.html

Post-Lesson Assessment

Your feedback is essential to help the next cohort of trainees. Please take a minute to complete the following short survey: https://www.surveymonkey.com/r/VNQZ3KS

Thanks for coming!!!



R Session Aborted

R encountered a fatal error.

The session was terminated.

Figure 8: