**Formatting datasets for the core-transient project**

In this project, we evaluate the proportion of core and transient species at a given site using data collected across a wide range of taxa, regions of the world, and environmental systems. A challenge that we face is that ecological data are collected at highly variable spatial, temporal, and biological scales and it is necessary to consistently format the data in preparation for analysis. Here we provide instructions associated with formatting datasets. The goal is to format a dataset as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| datasetID | site | species | year | count |
| 01 | d01\_Treatment1PlotA | A.\_schoenobaenus | 1928 | 1 |
| 01 | d01\_Treatment1PlotA | Buteo\_buteo | 1928 | 3 |
| 01 | d01\_Treatment1PlotB | Corvus\_corax | 1929 | 7 |
| 01 | d01\_Treatment2PlotA | Cuculus\_canorus | 1929 | 1 |
| 01 | d01\_Treatment2PlotB | Gallinula\_chloropus | 1928 | 2 |

This document is written in two parts. Section One describes typical workflow one should follow when formatting a dataset. Section Two describes the typical steps required to modify a given field in the dataset. Section Three provides a cheatsheet for the R functions used in data exploration and formatting.

**SECTION ONE: WORKFLOW**

Below are the steps that you should take when exploring and formatting datasets. These steps should be followed in the order that they’re presented. I also suggest using the example\_cleaning\_script in the scripts/R-scripts/data\_cleaning\_scripts folder as a guide. Following and modifying this script will also ensure that all of our scripts are in similar format and are thus easier to follow. ***IMPORTANT****: throughout this process, if there are any problems that keep you from successfully formatting a dataset, add an asterisk to the “flag” column of the data\_source\_table, git-add-commit-pull the data\_source\_table, and* ***add an issue to GitHub*** *(assigning the issue to me).*

1. Git pull! Before you begin to work on a dataset, make sure to do a **git pull** to ensure that you’re working on the most up-to-date version of the core-transient folder. Taking a few seconds to do this may end up saving you minutes in trying to figure out how to deal with git conflicts if they arise.
2. Scratch paper! I suggest always working with a piece of scrap paper to keep track of various issues with the dataset you’re working with.
3. Open a new script in RStudio. Save your script with the naming convention dwork[datasetID].R and git add-commit-push.
4. Explore! When you first load a dataset into R, take some time to explore the data. Common R commands that should be used whenever you start formatting a new dataset include:
   1. names(example\_df): Used to observer the field names of the data frame. This is great first look at how you can modify the fields of a data frame to fit the core-transient format.
   2. dim(example\_df): Used to observe the number of rows and columns of the data frame. To observe just the number of rows, you could always look at the length of the first column of data using: length(example\_df[,1])
   3. str(example\_df): Used to observe the structure of the data frame, including how each of the fields are formatted and the fields. Some fields may require to be changed from one format to another. This can be done using: as.character(example\_field), as.numeric(example\_field), and factor(example\_field). *Warning! If you are changing a factor to numeric and want to maintain the field values, you need to use: as.numeric(as.character(example\_field)).*
   4. head(example\_df): Used to observe the first few rows of the data frame. Note that it may sometimes be necessary to observe more rows than the default. You can specify this; for example, if you’d like to view the first 10 rows, type head(example\_df, 10). You can also look at the last few rows of data using tail(example\_df) and modify the number of rows shown as above.
   5. summary(example\_df): Provide summary data of the data frame. This can be especially useful, for example, to find out if there are zeros in the count data that need to be removed.
5. Once you are done with the exploration of the larger dataset, save your script and git-add-commit-push.
6. Explore and format **site** data. Prior to formatting sites (see Section Two) take a moment to explore how sites are coded. Of importance are:
   1. **How many sites are there?** You need to ensure that there are a reasonable number of sites. To determine the number of sites, use: length(unique(example\_df$site)). There have been instances in which the number of sites comes close to the number of records in the data frame. This sort of situation is most likely due to miscoding of the site field. Try to find out how the sites are miscoded. If you can find the problem, see the site section below in how the site data can be modified. Make sure to provide a comment in your data cleaning script that tells exactly what you’ve changed and why. Also, after the modification make sure to git add-commit-push and provide a message that details the modification. If the problem is not clear to you, add an issue to the core-transient git hub repository, describe the problem in detail and assign the issue to me.
   2. **How many records are there per site?** Sites that are mis-defined can also be determined by observing the number of records across sites. If sites are mis-defined, this can be identified if a large proportion of sites have very few records. There are many ways to determine this. To observe the number of records per site using the table function in base R, use either table(example\_df$site) to observe the records in wide format or data.frame(table(example\_df$site)) to observe the records in long format. The latter can also be done in Hadley Wickham’s plyr package using: ddply(example\_df,.(site),'nrow'). If there are a large number of sites, it can be cumbersome to search through them all. You can avoid this by ordering from the smallest to largest number of records per site. First, assign a name to your site table: xy <- data.frame(table(example\_df$site)). Next, order by frequency: xy2[order(xy2$Freq),]. Modify as necessary (see Section Two), providing descriptive comments in your script for your modification, and add-commit-push to GitHub. Again, if the problem is not clear to you, add an issue to the core-transient git hub repository, describe the problem in detail and assign the issue to me.
7. Once you are done with site exploration and formatting, save your script and git-add-commit-push.
8. Explore and format **species** data. Your primary goal in exploring species data is to remove problem species. For example, there may be NA’s in the dataset, as well as “species” such as “bare ground”. Use unique(example\_df$species) to explore the species in the dataset (*Note: prior to this species columns may need to be concatenated (see Section Two) if genus and species are listed in separate fields*). If there is a problem, and it can be easily remedied, make sure to add comments to your code that describe the steps you have taken and why they were necessary. Make no assumptions when removing species! If a species designation is not ENTIRELY clear, add an issue to the core-transient git hub repository, describe the problem in detail and assign the issue to me.
9. Once you are done with species exploration and formatting, save your script and git-add-commit-push.
10. Explore and format **time** data. Open data\_source\_table.csv. Here, you will see a column called analysis\_grain\_temporal This refers to the smallest time units of the study. If the temporal grain is less than one year, you will have to adjust the year column to represent partial years (we are coding these as decimals of a year). It is important that decimal years are rounded to the appropriate scale. If, for example, data are collected on a monthly basis, date objects should reflect months rather than days. See Section Two for how to adjust the year column.
11. Once you are done with time sample exploration and formatting, save your script and git-add-commit-push.
12. Explore and format count data. Of primary importance is ensuring that the count data are all non-zero (remove any zero counts, see Section Two) and that you note the type of count conducted (for example abundance, density, and cover). If the data\_source\_data currently contains the entry “FILL” and you are able to interpret the type of count data represented, please modify the table (and git add-commit-push). Once you have removed zeros and ensured the type of count data present, you will create a new data frame that summarizes the counts by site, species, and time unit (see Section Two).
13. Once you are done with count data exploration and formatting, save your script and git-add-commit-push.
14. You now have a reduced dataset and are ready to add the datasetID field. Follow the steps in Section Two for doing so and git-add-commit-push.
15. As a final step, you will compare a summary of your dataset with the data\_source\_table. Adjust the data\_source\_table as necessary. With every adjustment, git-add-commit-push and explain what you have done and why.
    1. Nrecs: dim(example\_df)[1]
    2. nSites: length(unique(example\_df$site))
    3. nSpecies: length(unique(example\_df$species))
    4. start\_year: min(example\_df$year)
    5. end\_year: min(example\_df$year)

**SECTION TWO: MODIFYING FIELDS**

**datasetID:**

Goal: Add a column that repeats the name of the dataset.

DatasetID’s are available from the dataset\_summary\_table. This must be the first column of your formatted dataset. To do so, you simply use the rep command in the base package and tell it to repeat the value for the number of rows of the data frame. *Note: example\_df is the unformatted dataset being prepared for analysis.*

datasetID = rep(01, dim(example\_df)[1])

**site:**

Goal: Add a column that provides a unique site ID for each site. Determining sites can sometimes be challenging the steps required to create unique site ID’s depends on how researchers have coded their site data. The examples below provide the most common solutions to create unique site ID’s

***Concatenating multiple site columns into a single site:***

In the example above, the sites were broken down into different treatments and plots. To construct the sites field, we paste the datasetID as well as any site information provided. If the above example were constructed from a data frame with the fields “Treatment” and “Plot”, the site column would be made using the following code:

site = paste(‘d’, datasetID, example\_df$Treatment, example\_df$Plot, sep = ‘’)

***Removing site information:***

Some data sources include information in the site field that are problematic for analysis. For example, some sites include the time that a sample was collected as a part of the site field. Consider a column named “site” where year is included in the field, such that the first entry might be “Treatment1PlotA1928”. You can extract the plot information easily in one of two ways:

1) Substring the plot information by from the first to the last plot character. The following code extracts all characters from the first to the 15th within the site field. This is ONLY to be used is the plot information contains the same number of characters!

site1 = substr(example\_df$site, 1, 15)

*Note: If the field is not a character field, you can convert it on the fly using:*

site1 = substr(as.character(example\_df$site) , 1, 15)

2) Substring the plot information by removing the last characters in a field. This method is valid if the number characters that make up the true site field are not the same across sites but there is an equal number of characters that need to be removed. To do so, the easiest way is to use the str\_sub function in Hadley Wickham’s **stringr** package (though this can be easily accomplished by writing your own function in base).

require(stringr)

x = “hello world”

str\_sub(x, 1, -7)

[1] "hello"

***Separating a field to extract site information:***

It is also often necessary to separate the site field by some common character (such as, in the example below “\_”). This is done using the transform and colsplit functions. Colsplit is located in Hadley Wickham’s package **reshape2**. The output of this function is a multiple field dataset containing the original data (field 1) and a column for each split. In this case, the second column contains the site information, so using “[,2]” returns a vector with just the relevant site information.

require(reshape2)

x = 'Treatment1PlotB\_1927'

site1a = transform(x, site = colsplit(x, pattern = '\\\_', names = c('site','year')))[,2]

***Using latitude and longitude to define sites:***

It is sometimes necessary to define sites using latitude and longitude if this is the only site information provided. While making decisions for the appropriate scale to analyze the data requires an understanding of the taxa and method of collection, the process of creating the site information is relatively straitforward. Here we will use the “round\_any” function in Hadley Wickham’s **plyr** package to turn decimal latitude and longitude data sites composed of 2 degree lat-lon blocks.

x = 13.35679

y = 46.87

site1 = paste(round\_any(x, 2), round\_any(y, 2))

**species:**

Goal: Subset dataset to unique species records. It is occasionally necessary to modify species records, such as if genus and species are provided in separate fields (in which case you would concatenate the two fields as above) and remove records that are not valid species.

***Subsetting a dataset to valid species observations:***

There are several methods for removing problem records; here are a few examples.

Removing NA’s:

example\_df1 = na.omit(example\_df)

example\_df1 = example\_df[!is.na(example\_df$species),]

Removing a given species record (example is records called “Bare Ground”):

example\_df1 = example\_df[example\_df!='Bare Ground',]

example\_df1 = subset(example\_df1, species!= 'Bare Ground')

**year:**

Goal: Create a time column. The two challenges that may be associated with this are if the date is provided as a date formatted object (example 01/01/2015) or if there are multiple samples per year. If the latter is the issue, the appropriate temporal scale must be determined prior to formatting the dataset (see Allen, Ethan, or myself). Once the time scale is determined, data are reported as a decimal year and simply requires a bit of math (for example, if sampling was done monthly and a sample was taken on 1 Mar 2015, the time of the sample would be 2015 + 3/12 as March is the third month of the year).

***Extracting year from a date object:***

Convert the date column to an R formatted date (in this case pretending that our unformatted dataset contains a column called record\_date):

date = strptime(example\_df $record\_date, '%m/ %d/ %y')

Add a sampling year line (summarize by year):

example\_df $year = as.numeric(format(date, '%Y'))

**count:**

Goal: Summarize the dataset to the count of individuals per species, site, and year for a given dataset. To do so, we will use Hadley Wickham’s “ddply” function in the **plyr** package. Below is an example in which there is a count column that must be summarized.

example\_df2 = ddply(example\_df, .(site, year, species), summarize, count = sum(count))

**SECTION THREE: DATA FORMATTING AND EXPLORATION CHEAT SHEET**

ddply(example\_df, .(site, year, species), summarize, count = sum(count))

**Create data object:**

strptime(example\_df $record\_date, '%m/ %d/ %y')

**Extract year from date object:**

as.numeric(format(date, '%Y'))

**Remove NA's:**

na.omit(example\_df)

example\_df[!is.na(example\_df$species),]

**Subset records:**

example\_df[example\_df!='bad\_record',]

subset(example\_df1, species!= 'bad\_record')

**Round numeric vector (x) to a given accuracy:**

round\_any(x, accuracy = 2)

**Split a column:**

colsplit(string, pattern, names)