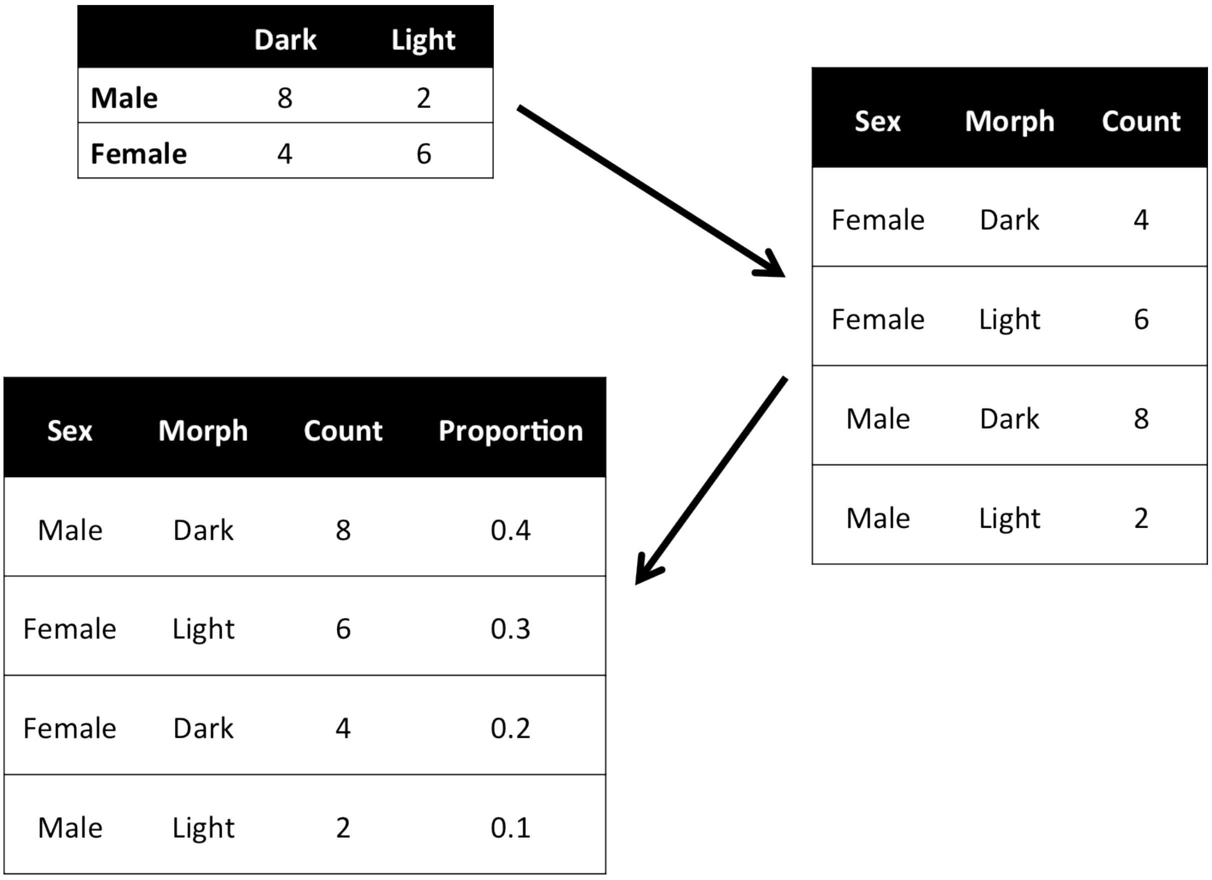
PR~statistics
Delivering ecology based courses & workshops
http://prstatistics.co.uk/

Advancing in R



	Dark	Light
Male	8	2
Female	4	6

Sex	Morph	Count
Female	Dark	4
Female	Light	6
Male	Dark	8
Male	Light	2

Sex	Morph	Count	Proportion
Male	Dark	8	0.4
Female	Light	6	0.3
Female	Dark	4	0.2
Male	Light	2	0.1

Module 1: Data Manipulation

Introduction

In this practical, we will cover the basics of manipulating data, making particular use of the {tidyr}, and {dplyr} packages. We will assume that you are familiar with the general syntax of coding in R, loading data (e.g. from CSV files), and with different variable types (e.g. numeric, string, factor,…).

Learning outcomes

Know what constitutes 'tidy data'.

Learn how to convert 'messy data' into tidy format.

Know what 'wide' and 'long' data are, and how to reshape data between these two formats.

Know how to inspect large data sets using commands such as select, filter, and group_by.

Know how to pipe multiple commands together.

Know how to aggregate and summarise data.

Know how to create new variables from existing data.

Learn the 'split-apply-combine' method of data manipulation.

Some useful commands

rename()

gather()

arrange()

%>%

filter()

select()

distinct()

summarise()

group_by()

mutate()

separate()

Tidying data

1: Tidying moth count data

Begin by loading the RStudio package. Start a new project and a new R script, and save each of these using meaningful and descriptive filenames. Recall the preamble to a well-annotated script (from the tutorial): what information and commands should you include at the beginning of your script?

For this exercise, we shall use the 'tidyr' and 'dplyr' packages. If you have not installed these previously, you can install them by going to the 'Tools' menu and selecting 'Install packages…'. In the dialog box, type in the names of the packages (they should autofill). Make sure that you have selected 'Repository (CRAN)' under the 'Install from' section, and that the box indicating "Install dependencies" is checked. You will notice that clicking "Install": causes a command to be automatically entered in the console, which is another way to install named packages:

```
install.packages(c("tidyr", "dplyr"))
```

Now that you have installed the required packages, load these into your project with the following commands:

```
library(tidyr)

library(dplyr)
```

Find the data file called "MothCount.csv" in the folder for this practical, and read it into a well-named data frame. You can ignore the warning message about an 'incomplete final line'. Verify that it has loaded properly, and examine the data object.

```
# import the data; note that read.csv works for comma delimited
files

MOTHS <- read.csv("MothCount.csv")

# examine data

MOTHS
```

Consider what variables and observations we have: what are the **key/value pairs**? (A key/value pair is the association between a variable name and a value within it: e.g., for a key 'CITY' then a value could be 'Glasgow'). Should we rename any of our columns before tidying the data? Think about (a) whether any columns are already key/value pairs that need to be named properly, and (b) what the values will be when we reorganise data into key/value pairs…

```
##

# First, sex is already in a column, but needs to be named

#  (it was left blank in the original spreadsheet).

# Second, when we rearrange our morph types into key/value pairs,
#  we don't want each value to be 'x.morph'; we want 'x'.
#
# Let's use dplyr's 'rename' function to specify new column names.
##
MOTHS <- rename(MOTHS,
               Sex = X,
               Dark = Dark.morph,
               Light = Light.morph)


# Check these changes:
MOTHS
```

Now, let's use the 'gather' function of tidyr to convert our data into key/value pairs. We pass the following arguments to this function: the data frame to use, the names of key/value columns to be created, and the columns that we will gather together.

```
##
# Convert our data into key/value pairs using 'gather'
# - 1. data [MOTHS]
# - 2. key / value column names to create [Morph, Count]
# - 3. columns to gather [Dark:Light]
##
MOTHS_T <- gather(MOTHS,
                      Morph, Count,
                      Dark:Light)
```

Inspect your new data frame: each row represents an observation – the `count` for one `morph` type for one `sex` – and each column is a variable. You now have a tidy data frame!

The other main function of {tidyr} is 'spread()'. Can you guess what this does? Have a look at the documentation for it (pass '`?spread`' to the console), and then try using the spread() command with the MOTHS_T data frame.

2: Tidying time series data

A common use for these tidying functions is to convert time series data from a 'wide' to 'long' format, making it easier to explore, plot, and model. In the data file 'TOM_CITIES.csv', I have recorded 10 days worth of temperature readings for each of the cities I have visited this month.

Read this data in, and take a look. It is easy for us to understand in this format, but again we need to think about making it easier for manipulating, plotting and modelling the data: i.e., using key/value pairs. Try restructuring this data set into long format using the 'gather' function.

You might find that 'gather' doesn't format the data exactly the way you want it to, with the values in your 'Day' variable being 'Day.1', 'Day.2', etc, rather than just 1, 2,… Thankfully, there is an easy way to **separate** out the actual value you want. Take a look at the documentation for this by typing '?separate' into the console. The important parts here are 'col' and 'into' – these are where we specify the column containing the values we want to separate out, and the names of the new columns that our separated values will be placed into. The 'sep' argument (see the documentation) enables us to specify how the values should be split, but it has a clever default regular expression that will look for patterns in our values. It's worth having a try with the default before worrying too much about specifying your own regular expressions!

Let's try using the pipe (more on that in the next section) to chain a 'separate' command to our 'gather':

```
gather(CITYTEMPS,

       Day, Temperature,

       Day.1:Day.10) %>%

  separate(Day, into=c("V1","V2"))
```

You will see that you now have two new variables, V1 and V2; the default regular expression has split our values from the 'Day' variable around the '.'. We don't have any need for V1 as it doesn't give us any new information, and we should rename V2 to something more meaningful. In the next section, you will learn more about data manipulation – come back to this data frame later so that you can select only those variables you want to keep, rename as necessary, and perhaps mutate them into different data types (e.g. the Day variable might be best converted into numeric format, using the function 'as.numeric()').

Manipulating data: chaining dplyr commands to interrogate large data sets

We will use a large data set to find answers to specific questions. When working with data, we often have common questions or tasks; the {dplyr} package makes it easy to carry these out.

The dplyr and tidyr packages also enable us to string a sequence of commands together using the 'pipe' operator, `%>%`. The output from each step is 'piped' into the next step.

You can use this technique to chain together many commands, performing complex tasks without having to assign new data frames.

NOTE: RStudio has a shortcut for entering the 'pipe' operator. Hit CTRL+SHIFT+'M' (Windows) or Command+SHIFT+'M' (Mac).

First, start a new script and annotate as usual, then load the file of historical weather data from UK weather stations (provided by the Met Office).

```
# import the data; note that read.csv works for comma delimited
files
```

```
WEATHER <- read.csv("UKWeatherData.csv")
```

Use the 'str()' function to find out more about the data structure.

```
str(WEATHER)
```

Each row gives several weather measurements (TempMax, TempMin, AF, Rainfall, Sun), recorded monthly for each weather station going back to its inception. As you can see, the data set has a lot of observations! This practical should give you an appreciation of the power of data manipulation packages to help tackle the questions you want to answer, especially with large data sets.

TempMax: Mean daily maximum temperature recorded that month (degrees Celsius).

TempMin: Mean daily minimum temperature recorded that month (degrees Celsius).

AF: Days of air frost for the month.

Rainfall: Total rainfall for the month (mm).

Sun: Total hours of sunshine for the month (hours).

1: List the locations with the highest temperature in each year

For this task, we are only interested in a small subset of the variables: Location, Year, Month, and TempMax. Use the 'select()' function and the pipe command to return only these columns.

```
WEATHER %>%
  select(Location, Year, Month, TempMax)
```

Remember that piping commands like this just prints the results to the console; you will see that RStudio doesn't print them all, which is probably for the best! Append 'head' to your command to just get the first few rows. The default for the 'head' function is to return the first 6 rows; you can change the 'n' argument (see the documentation for 'head') to make it 10 rows.

Take a look at the documentation for 'select()' by typing '?select' into the console: you'll see that it gives you a list of 'special functions' for choosing which variables (columns) to keep. Some of these may come in useful later, but a good one to note now is using '-' in front of a variable name to denote that you want to drop it. This can be quicker when you have a large number of variables that you wish to keep lots of variables. While that isn't the case here, you can check how it works by specifying a group of variables to drop:

```
WEATHER %>%
  select(-c(TempMin, AF, Rainfall, Sun))
```

As you saw in the tutorial, the 'c()' function **combines** values into a list or vector; here, we use it to say that we have a group of variables that we want to deselect. It applies the '-' to each of these variables, and is therefore directly equivalent to:

```
WEATHER %>%
  select(-TempMin, -AF, -Rainfall, -Sun)
```

We could also take advantage of the fact that we want 4 consecutively arranged columns, using ':' to indicate that we want to select all columns between the ones specified:

```
WEATHER %>%
  select(Location:TempMax)
```

Whichever method you choose to use, we then want to arrange our observations in descending order for Year (first) and TempMax (second); use the 'arrange' function with both of these variables:

```
WEATHER %>%
  select(Location, Year, Month, TempMax) %>%
  arrange(desc(Year), desc(TempMax)) %>%
  head(n = 10)
```

Notice that some of these temperatures seem quite low; this is because the data is current and gives readings up until March 2015.

```
WEATHER %>%
  select(Month, Year) %>%
  distinct(Month, Year) %>%
  arrange(desc(Year), desc(Month)) %>%
  head(n = 20)
```

Let's use 'filter()' to only get those rows up until the end of 2014:

```
WEATHER %>%
  select(Location, Year, Month, TempMax) %>%
  filter(Year <= 2014) %>%
  arrange(desc(Year), desc(TempMax)) %>%
  head(n = 10)
```

Note that we put the filter before the 'arrange' command, as this gives the computer fewer rows to sort.

We don't want to use the 2015 observations at all, so we can safely remove these from our WEATHER data frame. It is up to you whether you create a new data frame (with a different name), or simply overwrite WEATHER by assigning the filtered version to an object of the same name:

```
WEATHER <- WEATHER %>%
  filter(Year <= 2014)
```

Now we don't have to worry about filtering out 2015 each time we run a command (but this doesn't affect the data in your original CSV file). Let's run the select and arrange commands on Year and TempMax again to check:

```
WEATHER %>%
  select(Location, Year, Month, TempMax) %>%
  arrange(desc(Year), desc(TempMax)) %>%
  head(n = 10)
```

Our piped command (up until 'head') returns all the maximum monthly temperatures, arranged by year and maximum monthly temperature, both in descending order. We only want the **highest temperature observation per year**, no matter the location or month, so we can use the 'distinct' function to request only one row for each year. The 'distinct()' function retains only the first of multiple rows with the same values, and we can specify which variables it uses for 'uniqueness'. In this case, we want only one row for any particular year, but we also want that row to contain the highest maximum monthly temperature from all the records for that year. As we have already arranged the data in descending order of temperatures, we can simply specify Year in our 'distinct' command. Make sure you pipe it **after** you have sorted the data!

The 'head' command allows us to check the first few rows of our results, just to see that things are going as we expect. To get the full list of results, simply remove the last part of the command so that we get all rows, not just the first 10.

Notice how our chain of commands also makes readable sense: from our data set, we **select** variable, **filter** rows, then **arrange** these rows before taking a **distinct** value for each year.

Chaining commands together is a great way to investigate different views of data frames without creating multiple sub-data frames. Of course, if we wanted to save a separate data frame of the locations with the highest monthly max temperature in every year, we could simply assign our created table to a new data frame as follows:

```
TopTemps <- WEATHER %>%
  select(Location, Year, Month, TempMax) %>%
  arrange(desc(Year), desc(TempMax)) %>%
  distinct(Year)
```

Bonus question:

- Find the location with the lowest minimum monthly temperature for each month of 1975.
    - Think about which variables to **select** and how you should **filter** the data, as well as how to **arrange** the rows before selecting **distinct** ones.

2: Find the top 5 locations with the highest total rainfall from 2000-2014

This example will show you one of the most powerful strategies for data manipulation: **split**, **apply**, **combine**. After filtering the data for years 2000-2014, we will split the data on values of a particular variable, apply a function on each of these sections, and then combine the results.

Take a minute to think about the question above and how you would approach it with the 'split-apply-combine' technique.

Now, look at the documentation for the functions 'group_by()' and 'summarise()'.

Our approach will be to **split** the data by grouping rows by location; we will then **apply** a summary function to total the rainfall for each location; finally, we **combine** the results so that we can then arrange them to view just the top 5 locations for total rainfall:

```
WEATHER %>%
  filter(Year >= 2000) %>%
  group_by(Location) %>%
  summarise(totalrain = sum(Rainfall, na.rm = TRUE)) %>%
  arrange(desc(totalrain)) %>%
  head(n = 5)
```

Practise this strategy on your own with some other questions on this data set:

- Find the average (mean) monthly rainfall for each location in each year during the period 2010-2014.
    - *Hint: you can group by more than one variable…*
    - *The R function for finding the mean of a group of numbers is simply 'mean()'; you can use '?mean' to check the documentation for this.*
    - *Note: don't worry if R doesn't print all the rows… but if you do want to see the whole data frame produced, append 'print.data.frame()' to your chain of commands.*
- Find both the mean and the standard deviation of monthly rainfall for each location in each year, 2010-2014
    - *Hint: create more than one summary variable simply by using a comma within 'summarise' to denote a new variable*
    - *Reduce the number of decimal places in the output by nesting your call to 'mean' or 'sd' **within** a call to the 'round' function, e.g.* round(mean(Rainfall, na.rm = TRUE), digits = 2)
- How many hours of sun were recorded altogether each month in 2014?
- For each year, how much rainfall has been recorded in total?
- For **only** those weather stations in Eskdalemuir, Leuchars, Paisley, and Stornoway, find the mean and standard deviations for monthly hours of sun during the year 2014:
    - *You can filter by a group of options by creating a vector of your selected locations, then using R's '%in%' keyword to filter by looking for matches, i.e.* filter(Location %in% c("Eskdalemuir", "Leuchars", "Paisley", "Stornoway"))

3: Which locations have 2014's biggest monthly temperature range?

We have previously used 'summarise' to create summary statistics for grouped data. However, sometimes we want to create a new variable that will have a value for **each row** from existing data.

Say we wanted to define a new variable TempRange, which denoted the range between minimum and maximum temperature (for each month of each year at each location). We could use the '**mutate()**' function to do this.

Let's think again about the steps involved in doing this. First, we would need to **select** the variables of interest (here we need Location, Year, Month, TempMax, TempMin). Next, we should **filter** by year so that we only get results from 2014. Having selected our columns and filtered our rows, we can '**mutate**' a new variable ('TempRange') through subtracting TempMin from TempMax. Finally, we **arrange** our results in descending order, and show the top 10.

You can practise this technique with creating other new variables from the available data.

Bonus question: for each year between 2010 and 2014, what is the average monthly temperature range? (i.e., find the mean TempRange across all locations for 2010, 2011, 2012…)

4. For the whole of the 20th century, which **location** has the highest ratio of total hours of sun to total mm of rainfall?

Put together what you've learned so far in this practical to answer the above question.

HINT: you will need to **group** by location, **filter** years, and find **summary** statistics before '**mutating**' a new variable from those summary variables…

Bonus question: we sometimes want to create categories to partition our data into. Try using a second 'mutate' call to create a new variable 'isNice', which will take the value '1' if your ratio of sun hours to rainfall mm is >=2, and the value '0' if that ratio is <2. Take a look at the documentation for the function '**ifelse**' to see how you might create such a variable…

5. Is there a relationship between location 'northernliness' and temperature?

Load in the 'WeatherStations.csv' file as STATIONS, and take a look at this data set. You will see that each location has coordinates for degrees north (DegN) and east (DegE). For this part of the exercise, you will need to **join** the two data sets together. We do this via a 'left_join' (check the documentation for more details), using 'Location' as the variable on which to join:

```
WEATHERLOC <- left_join(WEATHER, STATIONS,
                        by = "Location")
```

Take a look at the new data frame that you have created – it should have the same number of rows as WEATHER, but 2 additional columns: DegE and DegN. Try filtering for only data from June 2014, and sort by DegN. You may notice something of a trend in terms of how far north a location is with its temperature… and you may not be surprised to learn that Loch Lomond is 56.083 degrees north!

If you wish, you can practice doing more data tidying and manipulation by consulting the supplementary exercises for this module. These exercises will show you how to tidy time series data, and to interrogate large data sets with complex information. Remember to thoroughly annotate your script, and to save your project and R script.

~ End of Practical ~