Chapter_5

Henry Bourne

2022-11-03

Tidyverse

The Tidyverse is a set of inter-compatible R packages sharing the same programming philosophy. The Tidyverse provides tools for building and transforming dataframes so that they are in the correct ("tidy") format for us to carry out modelling and visualization. The tidyverse is extensive and so in this chapter we will focus on some core important aspects of the Tidyverse, in particular:

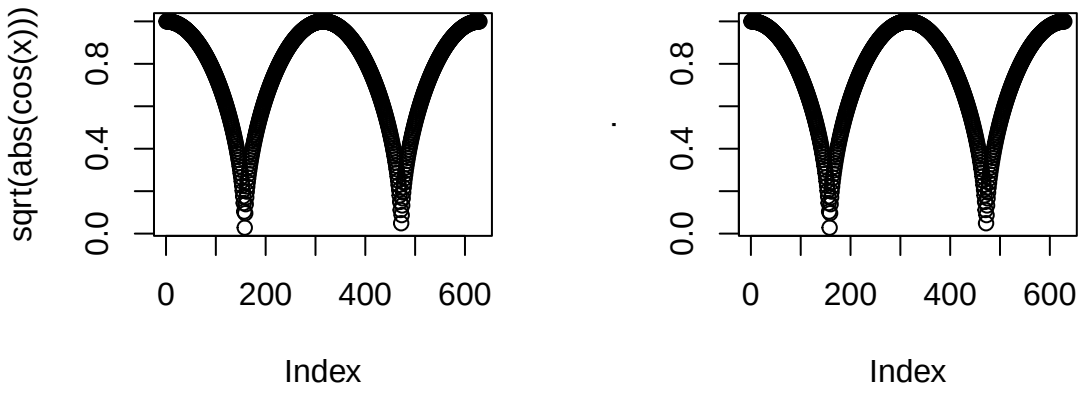
- Pipes, %>%
- Visualizations with the *ggplot2* package: we will go through the basics and also show why its layer-based framework is very useful when building a library of tools.
- The *dplyr* and *tidyr* packages and how to manipulate and transform data using them to streamline visualization and modelling.

Pipes

To use the pipe operator we must have:

```
library(magrittr)
```

The **pipe operator** is written: %>%, or can be produced (in Rstudio) using the shortcut *Ctrl + Shift + M*. What the pipe operator does is "feed" whats to the left of it into whats to the right of it. For example, after setting x, the following to code chunks are equivalent:

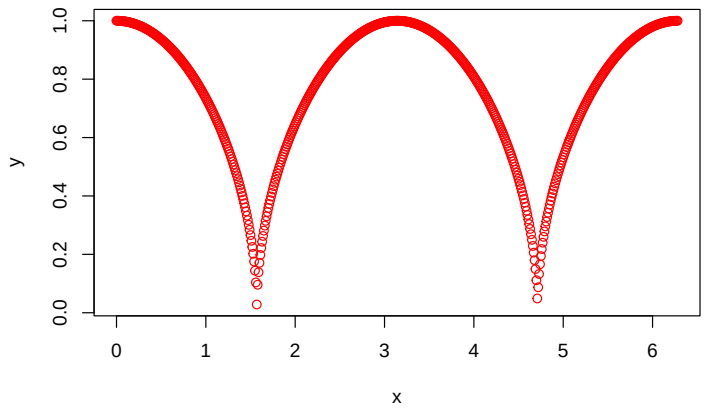


More concretely what the pipe operator is doing is the following:

```
 f <- function(.){

 . <- cos( . )
 . <- abs( . )
 . <- sqrt( . )
 plot( . )

}

f(x)
```

ie. it applies a sequence of functions to whats on the left of the left-most pipe and stores after the execution of each function the results as ".", which is then fed to the next function in the sequence. This also explains why the second graph has a "." on the y-axis as opposed to "sqrt(abs(cos(x)))". What if along the way we want to have more arguments for our functions than simply whats being "ran through the pipe"? Well we can use the "." as a placeholder for the argument that is being piped in this scenario, for example, let's say we now want our plot to be red and to have the label "y" on the y-axis we can do this as follows:

```
x %>% cos %>% abs %>% sqrt %>% plot(x = x, y = ., ylab = "y", col="red")
```



Note that the placeholder by default is given to the first argument of the function its being piped to. Note that we must also be careful when we write the placeholder explicitly, as if we write it in a nested expression then this will be evaluated then piped to the rest of the expression, eg.

```
x %>% plot(x = x, y = . + 1, ylab = "y")
```

returns an error as it is equivelant to writing:

```
x %>%  plot(., x = x, y = . + 1, ylab = "y")
```

ie. (.+1) is evaluated then is passed as the first argument to the *plot* function. To override this behaviour we can make explicit the expression that we are piping to which we do with curly brackets as follows:

```
x %>% { plot(x = x, y = . + 1, ylab = "y") }
```

Once we've made explicit the expression we are piping to we can actually pipe to multiple placeholder values, for example we can compute the sum of x summed together 3 times using the following code:

```
x %>% {sum( . , . , .)}
```

```
## [1] 5925.18
```

Or we could sum together the first three elements of x by writing:

```
x %>% {sum(.[1], .[2], .[3])}
```

```
## [1] 0.03
```

Pipes are useful as they can make our code more readable in many instances. If we want to compute a big sequence of functions normally we would have to write the inner most bracket put that inside the brackets of another function and so on... this means we have to read from the inside out and it can get confusing (especially with all the brackets!). Whereas with piping we just "lay down the pipes" through which the information will flow, this can make it easier to write (easier not to mismatch brackets etc.) and also easier to read as you can read from easily from left to right where the information is flowing in the sequence of computations that is being performed.

There are also some more exotic pipes such as the **assignment pipe operator**, *%<>%*. What it does is pipe whats on its left to what on its right and then assign the resulting value to whats on its left, ie. the following are equivelant

```
x <- c(5, 1, 7, 9, 3)
x %<>% sort ; x
```

```
## [1] 1 3 5 7 9
```

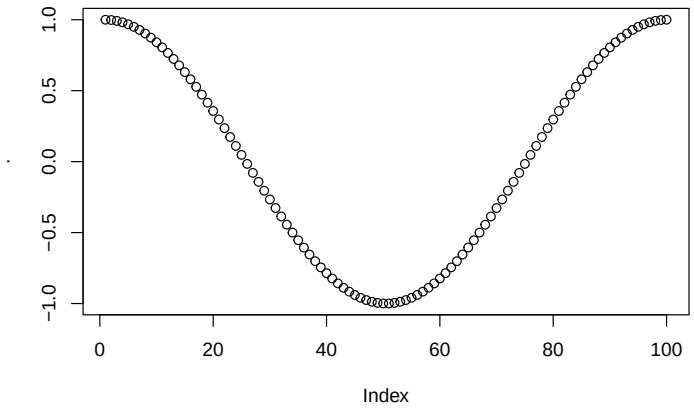
```
x <- x %>% sort ; x
```

```
## [1] 1 3 5 7 9
```

Note that *%<>%* can only be the first pipe in a pipeline.

Another pipe is the **tee pipe operator**, *%T>%*. What it does is store partial results along the pipeline when called. For example:

```
x <- 100 %>%
  seq(0, 2*pi, length.out = .) %>%
  cos %T>%
  plot
```



```
head(x)
```

```
## [1] 1.0000000 0.9979867 0.9919548 0.9819287 0.9679487 0.9500711
```

Here the output at the left of the tee pipe is saved as x, it then pipes whats on the left of it not to what is on its right, but to the right of the next pipe operator, essentially skipping what is to its right (in regards to the pipeline, it still computes what is to its right).

Finally we have the **exposition pipe operator**, *%$%*. This becomes especially useful when working with names lists and data frames, what it does is make the names of the object on the left available in the function call on the right, for example:

```
iris %>%
  subset(Species == "setosa") %$%
  cor(Sepal.Length, Sepal.Width)
```

```
## [1] 0.7425467
```

ggplot2

To plot using ggplot2 we must first load the package and create a ggplot object,

```
library(ggplot2)
pl <- ggplot(data = iris)
class(pl)
```

```
## [1] "gg"     "ggplot"
```

We assign to the ggplot object the data we want to produce plots for and then print its class. To then produce visualizations we need to add graphical **layers** to the plot, currently if we call pl it will just produce an empty screen. Now we will add a graphical layer:

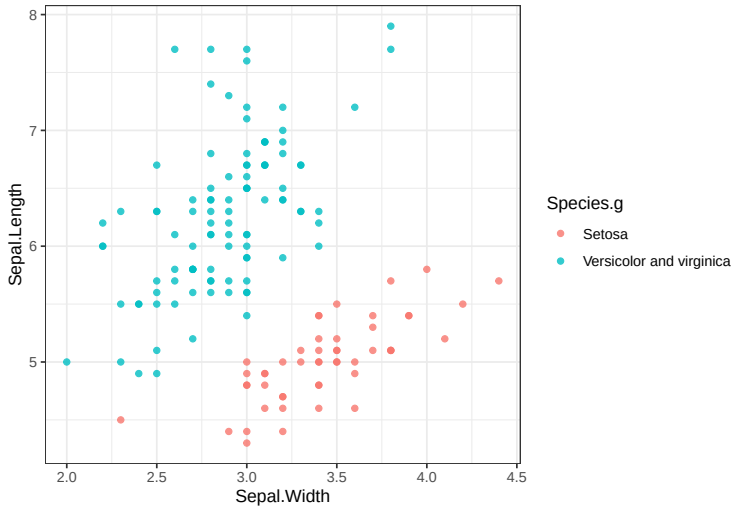
```
pl <- pl + geom_point(mapping = aes(x = Sepal.Width, y = Sepal.Length)) ; pl
```

Note that compared to base-r plot functions nothing prints automatically, we have to call the print function for the object to generate a plot (we do this by simply calling the object which in turn calls the print statement), also note that we only give the variable names not the actual variables like we would do in base-r. What we do in the above is first call + which is an overloaded operator that will dispatch to *+.gg* (when to the left of the + is a object of class gg). The right hand-side of the operator can be a graphical layer or a function that modifies the plot, what happens is the *+.gg* operator gets the plot to the left of it and modifies it using what

is to the right of it. A **graphical layer** produces some visualization and each layer needs a **mapping** to know which variables (from the data frame) should be plotted. In the above code the *aes* function is our mapping, it maps the variable names we give to the correspoinding data in the data frame. Then *geom_point* builds a scatter plot visualization using the mapping.

We will now show a brief example of it in action along with some pipes, to give some intuition for how it works and provide some further information on what we can do with ggplot2. Observe the following:

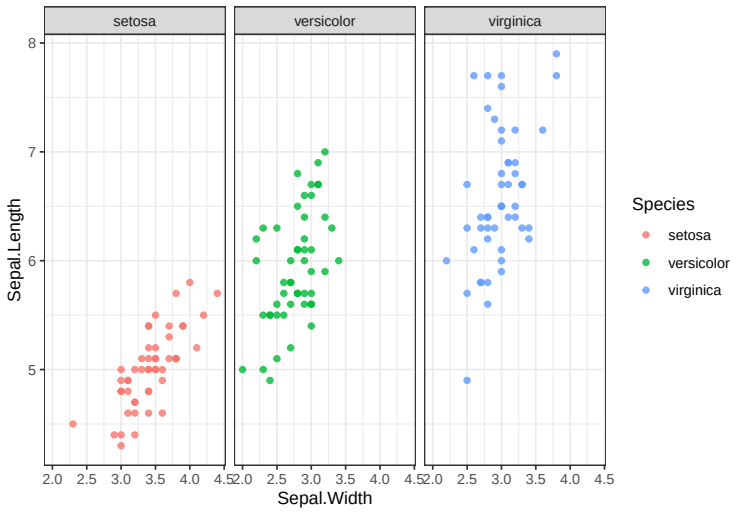
```
library(magrittr)
iris$Species.g <- iris %$% factor(Species == "versicolor" | Species=="virginica", labels
↪  = c("Setosa","Versicolor and virginica"))

iris %>% ggplot(mapping = aes(x = Sepal.Width, y = Sepal.Length, col = Species.g)) +
           geom_point(alpha = 0.8) +
           theme_bw()
```



We first load *magrittr* so that we can pipe, we then create a new column in the data frame that tells us whether it belongs to the Setosa species or one of the other two species. We then create a ggplot object where we specify the mapping we want and then we add a scatter plot layer (specifying the opacity of the points) and finally add a theme which tweaks the graphical scatter plot layer visually.

Another thing we can do is create a multi-plot, we can do this using faceting, for example lets say we wanted to plot side by side the sepal length and widths of each of the three species separately:

```
iris %>% ggplot(mapping = aes(x = Sepal.Width, y = Sepal.Length, col = Species)) +
           geom_point(alpha = 0.8) +
           theme_bw() +
           facet_wrap(~Species)
```



There is much much more you can do with ggplot2, however, we will now move on to other aspects of the Tidyverse.

Data transformation with dplyr

To use ggplot2 effectively we must first have everything we want to plot stored in the data frame. The dplyr package provides some convenient tools for manipulating data stored in a tabular format (ie. data.frames). One of the simplest functions supplied is **select** which allows us to select one or more columns of a *data.frame*, we can use it as follows:

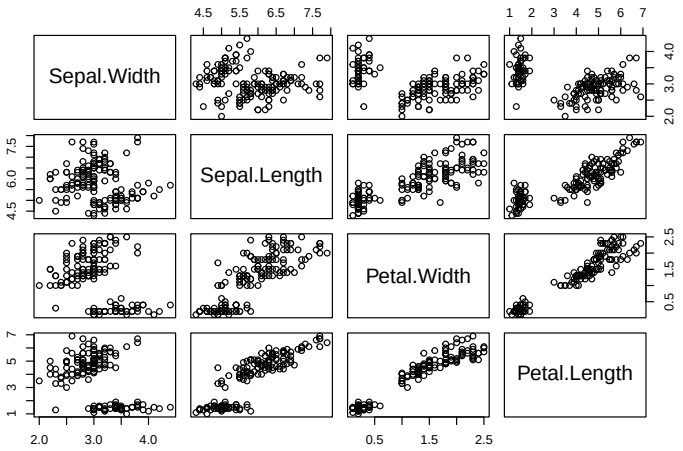
```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(magrittr)
#library(GGally)
iris %>% select(Sepal.Width, Sepal.Length, Petal.Width, Petal.Length) %>% pairs
```



Here what we do is select certain columns of the data frame and then plot them against each other using ggplot2's pairs function so we can view if any of the variables appear correlated. By including a "-" in front of a variable name we can also remove that variable from the dataframe. Another usefull function is **filter** which allows us to filter out some of our data using logical conditions, eg.

```
small_sepals <- iris %>% filter(Sepal.Length < 6 & Sepal.Width < 3.5) %T>%
↪  {print(head(.), digits = 2)}
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Species.g
## 1          4.9         3.0          1.4         0.2  setosa    Setosa
## 2          4.7         3.2          1.3         0.2  setosa    Setosa
## 3          4.6         3.1          1.5         0.2  setosa    Setosa
## 4          4.6         3.4          1.4         0.3  setosa    Setosa
## 5          5.0         3.4          1.5         0.2  setosa    Setosa
## 6          4.4         2.9          1.4         0.2  setosa    Setosa
```

(note that we use & not && as we want to perform element wise comparisons, the former is for programming control-flow doesn't evaluate each element)

We could also use the **arrange** function which allows us to sort the rows of the data using one or more variables, like in the following:

```
iris %<>% arrange(desc(Sepal.Length), desc(Sepal.Width), desc(Petal.Length),
↪  desc(Petal.Width))
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          7.9         3.8          6.4         2.0 virginica
## 2          7.7         3.8          6.7         2.2 virginica
## 3          7.7         3.0          6.1         2.3 virginica
## 4          7.7         2.8          6.7         2.0 virginica
## 5          7.7         2.6          6.9         2.3 virginica
## 6          7.6         3.0          6.6         2.1 virginica
##                  Species.g
## 1 Versicolor and virginica
## 2 Versicolor and virginica
## 3 Versicolor and virginica
## 4 Versicolor and virginica
## 5 Versicolor and virginica
```

```
## 6 Versicolor and virginica
```

```
tail(iris)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width Species Species.g
## 145          4.6         3.1          1.5         0.2  setosa    Setosa
## 146          4.5         2.3          1.3         0.3  setosa    Setosa
## 147          4.4         3.2          1.3         0.2  setosa    Setosa
## 148          4.4         3.0          1.3         0.2  setosa    Setosa
## 149          4.4         2.9          1.4         0.2  setosa    Setosa
## 150          4.3         3.0          1.1         0.1  setosa    Setosa
```

Here what we are doing is first arranging the rows of the dataframe in order of descending Sepal.Length, then where there are ties we sort by descending Sepal.Width and then by descending Petal.Length and finally by descending Petal.Width. We can also use **mutate** which allows us to modify one or more variables of the data frame, for example we could use it to modify the levels of the species names (which tells us their order) if we wanted to arrange the rows by species in a certain order.

There are also some functions that allow us to group or summarise data in data frames. For example **summarise** which lets us turn a dataframe into a summary, for example we can create a summary including the maximal measures of sepal width, height, petal width and height by writing:

```
data(iris)
iris %>% summarise(max_SepalLength = max(Sepal.Length),
                   max_SepalWidth = max(Sepal.Width),
                   max_PetalLength = max(Petal.Length),
                   max_PetalWidth = max(Petal.Width)
                  )
```

```
##   max_SepalLength max_SepalWidth max_PetalLength max_PetalWidth
## 1             7.9            4.4             6.9           2.5
```

There is also a function called **group_by** which takes a data frame as input and groups its rows using one or more variables. Where it differs from arrange is that as opposed to just changing the ordering of the rows it also creates groups which when we call dplyr based operations to it will be applied by group, to stop this we call **ungroup**.

```
iris_species <- iris %>% group_by(Species)
```

Here we have grouped iris by species, if we now run summarise like before on it we can see that it will perform it by group (which in this case is species):

```
iris_species %>% summarise(max_SepalLength = max(Sepal.Length),
                   max_SepalWidth = max(Sepal.Width),
                   max_PetalLength = max(Petal.Length),
                   max_PetalWidth = max(Petal.Width)
                  )
```

```
## # A tibble: 3 x 5
##   Species    max_SepalLength max_SepalWidth max_PetalLength max_PetalWidth
##   <fct>                <dbl>          <dbl>           <dbl>          <dbl>
## 1 setosa                 5.8            4.4             1.9            0.6
## 2 versicolor             7              3.4             5.1            1.8
## 3 virginica              7.9            3.8             6.9            2.5
```

Note that the output of group_by is of type *tibble*, a tibble is simply a "tidy" version of a data.frame.

Tidyr and some more usefull functions

The goal of tidyr is to help create "tidy" data, where:

- Every column is a variable
- Every row is an obervation
- Every cell is a single value

There are many functions in the tidyr package and we will just briefly go through a few of them here. The first is **pivot_longer** which "lengthens" data, what we mean by this is that it increases the number of rows and decreases the number of columns. You give it the data frame and then the columns which you want to pivot into a longer format (stack), you can use the argument *values_to* to give the name of the new value column you are creating and you can use the argument **names_to** to name the new character column you create (names_to can be used to reduce memory by making it a factor variable column). **slice** let's you select, remove or duplicate rows based on the data frames rows' indices. **pivot_wider** works like *pivot_longer* except it widens instead of lengthens ie. is used to increase the number of columns and decrease the number of rows.

Some other useful tidyr functions are: **separate** which allows you to break a variable into its components, **unite** which does the opposite of *separate* and **complete** which is useful to find out whether your data set has implicit missing values.

There are also a whole host of functions (in dplyr) which help us merge data frames. **left_join** takes two data frames x and y, it then adds columns from y to x and matches rows based on all the rows in x. **right_join** does the same but matches rows according to all the rows in y. **inner_join** matches by all the rows in x **and** y. **full_join** matches by all the rows in x **or** y.