2 A tidyverse primer

The tidyverse is a collection of R packages for data analysis that are developed with common ideas and norms. From Wickham et al. (2019):

> "At a high level, the tidyverse is a language for solving data science challenges with R code. Its primary goal is to facilitate a conversation between a human and a computer about data. Less abstractly, the tidyverse is a collection of R packages that share a high-level design philosophy and low-level grammar and data structures, so that learning one package makes it easier to learn the next."

In this chapter, we briefly discuss these principles and how they apply in the context of modeling, before diving into specific examples of tidyverse syntax.

2.1 PRINCIPLES

The full set of strategies and tactics for writing R code in the tidyverse style can be found at the website `https://design.tidyverse.org` . Here we can briefly describe several of the design principles, their motivation, and how we think about modeling as an application of these principles.

2.1.1 DESIGN FOR HUMANS

The tidyverse focuses on designing R packages and functions that can be easily understood and used by a broad range of people. Both historically and today, a substantial percentage of R users are not people who create software or tools but instead people who create analyses or models. As such, R users do not typically have (or need) computer science backgrounds, and many are not interested in writing their own R packages.

For this reason, it is critical that R code be easy to work with to accomplish your goals. Documentation, training, accessibility, and other factors play an important part in achieving this. However, if the syntax itself is difficult for people to easily comprehend, documentation is a poor solution. The software itself must be intuitive.

To contrast the tidyverse approach with more traditional R semantics, consider sorting a data frame. Using only the core language, we can sort a data frame using one or more columns by reordering the rows via R's subscripting rules in conjunction with `order()`; you *cannot* successfully use a function you might be tempted to try in such a situation because of its name, `sort()`. To sort the `mtcars` data by two of its columns, the call might look like:

```
mtcars[order(mtcars$gear, mtcars$mpg), ]
```

While very computationally efficient, it would be difficult to argue that this is an intuitive user interface. In **dplyr** by contrast, the tidyverse function `arrange()` takes a set of variable names as input arguments directly:

```
library(dplyr)
arrange(.data = mtcars, gear, mpg)
```

The variable names used here are "unquoted"; many traditional R functions require a character string to specify variables, but tidyverse functions take unquoted names or *selector functions*. The selectors allow for one or more readable rules that are applied to the column names. For example, `ends_with("t")` would select the `drat` and `wt` columns of the `mtcars` data frame.

Additionally, naming is crucial. If you were new to R and were writing data analysis or modeling code involving linear algebra, you might be stymied when searching for a function that computes the matrix inverse. Using `apropos("inv")` yields no candidates. It turns out that the base R function for this task is `solve()`, for solving systems of linear equations. For a matrix `X`, you would use `solve(X)` to invert `X` (with no vector for the right-hand side of the equation). This is only documented in the description of one of the *arguments* in the help file. In essence, you need to know the name of the solution to be able to find the solution.

The tidyverse approach is to use function names that are descriptive and explicit over those that are short and implicit. There is a focus on verbs (e.g. `fit` , `arrange` , etc.) for general methods. Verb-noun pairs are particularly effective; consider `invert_matrix()` as a hypothetical function name. In the context of modeling, it is also important to avoid highly technical jargon in names such as Greek letters or obscure terms. Names should be as self-documenting as possible.

When there are similar functions in a package, function names are designed to be optimized for tab-completion. For example, the **glue** package has a collection of functions starting with a common prefix (`glue_`) that enables users to quickly find the function they are looking for.

2.1.2 REUSE EXISTING DATA STRUCTURES

Whenever possible, functions should avoid returning a novel data structure. If the results are conducive to an existing data structure, it should be used. This reduces the cognitive load when using software; no additional syntax or methods are required.

One data structure that is used as much as possible in tidyverse and tidymodels packages is the data frame. Data frames can represent different types of data in each column, and multiple values in each row. Tibbles, a type of data frame described below, are preferred since they have additional properties that are helpful for data analysis.

As an example, the **rsample** package can be used to create *resamples* of a data set, such as cross-validation or the bootstrap (described in Chapter 10). The resampling functions return a tibble with a column called `splits` of objects that define the resampled data sets. Three bootstrap samples of a data set might look like:

```
boot_samp <- rsample::bootstraps(mtcars, times = 3)
boot_samp
#> # Bootstrap sampling
#> # A tibble: 3 × 2
#>   splits          id
#>   <list>          <chr>
#> 1 <split [32/13]> Bootstrap1
#> 2 <split [32/11]> Bootstrap2
#> 3 <split [32/11]> Bootstrap3
class(boot_samp)
#> [1] "bootstraps" "rset"       "tbl_df"     "tbl"        "data.frame"
```

With this approach, vector-based functions can be used with these columns, such as `vapply()` or `purrr::map()` [2]. This `boot_samp` object has multiple classes but inherits methods for data frames (`"data.frame"`) and tibbles (`"tbl_df"`). Additionally, new columns can be added to the results without affecting the class of the data. This is much easier and more versatile for users to work with than a completely new object type that does not make its data structure obvious.

One downside to relying on common data structures is the potential loss of computational performance. In some situations, data can be encoded in specialized formats that are more efficient representations of the data. For example:

- In computational chemistry, the structure-data file format (SDF) is a tool to take chemical structures and encode them in a format that is computationally efficient to work with.
- Data that have a large number of values that are the same (such as zeros for binary data) can be stored in a *sparse matrix format*. This format can reduce the size of the data as well as enable more efficient computational techniques.

These formats are advantageous when the problem is well scoped and the potential data processing methods are both well defined and suited to such a format[3]. However, once such constraints are violated, specialized data formats are less useful. For example, if we perform a transformation of the data that converts the data into fractional numbers, the output is no longer sparse; the sparse matrix representation is helpful for one specific algorithmic step in modeling but this is often not true before or after that specific step.

A specialized data structure is not flexible enough for an entire modeling workflow in the way that a common data structure is.

One important feature in the tibble produced by **rsample** is that the `splits` column is a list. In this instance, each element of the list has the same type of object: an `rsplit` object that contains the information about which rows of `mtcars` belong in the bootstrap sample. *List columns* can be very useful in data analysis and, as will be seen throughout this book, are important to tidymodels.

2.1.3 DESIGN FOR THE PIPE AND FUNCTIONAL PROGRAMMING

The **magrittr** pipe operator (`%>%`) is a tool for chaining together a sequence of R functions. To demonstrate, consider the following commands which sort a data frame and then retain the first 10 rows:

```r
small_mtcars <- arrange(mtcars, gear)
small_mtcars <- slice(small_mtcars, 1:10)

# or more compactly:
small_mtcars <- slice(arrange(mtcars, gear), 1:10)
```

The pipe operator substitutes the value of the left-hand side of the operator as the first argument to the right-hand side, so we can implement the same result as above with:

```r
small_mtcars <- 
  mtcars %>% 
  arrange(gear) %>% 
  slice(1:10)
```

The piped version of this sequence is more readable; this readability increases as more operations are added to a sequence. This approach to programming works in this example because all of the functions we used return the same data structure (a data frame) that is then the first argument to the next function. This is by design. When possible, create functions that can be incorporated into a pipeline of operations.

If you have used **ggplot2**, this is not unlike the layering of plot components into a `ggplot` object with the `+` operator. To make a scatter plot with a regression line, the initial `ggplot()` call is augmented with two additional operations:

```
library(ggplot2)
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  geom_smooth(method = lm)
```

While similar to the **dplyr** pipeline, note that the first argument to this pipeline is a data set (`mtcars`) and that each function call returns a `ggplot` object. Not all pipelines need to keep the returned values (plot objects) the same as the initial value (a data frame). Using the pipe operator with **dplyr** operations has acclimated many R users to expect to return a data frame when pipelines are used; as shown with **ggplot2**, this does not need to be the case. Pipelines are incredibly useful in modeling workflows but modeling pipelines can return, instead of a data frame, objects such as model components.

R has excellent tools for creating, changing, and operating on functions, making it a great language for *functional programming*. This approach can replace iterative loops in many situations, such as when a function returns a value without other side effects.[4]

Let’s look at an example. Suppose you are interested in the logarithm of the ratio of the fuel efficiency to the car weight. To those new to R and/or coming from other programming languages, a loop might seem like a good option:

```
n <- nrow(mtcars)
ratios <- rep(NA_real_, n)
for (car in 1:n) {
  ratios[car] <- log(mtcars$mpg[car]/mtcars$wt[car])
}
head(ratios)
#> [1] 2.081 1.988 2.285 1.896 1.693 1.655
```

Those with more experience in R may know that there is a much simpler and faster *vectorized version* that can be computed by:

```
ratios <- log(mtcars$mpg/mtcars$wt)
```

However, in many real-world cases, the element-wise operation of interest is too complex for a vectorized solution. In such a case, a good approach is to write a *function* to do the computations. When we design for functional programming, it is important that the output only depends on the inputs and that the function has no side effects. Violations of these ideas in the following function are shown with comments:

```
compute_log_ratio <- function(mpg, wt) {
  log_base <- getOption("log_base", default = exp(1)) # gets external data
  results <- log(mpg/wt, base = log_base)
  print(mean(results))                                # prints to the console
  done <<- TRUE                                       # sets external data
  results
}
```

A better version would be:

```
compute_log_ratio <- function(mpg, wt, log_base = exp(1)) {
  log(mpg/wt, base = log_base)
}
```

The **purrr** package contains tools for functional programming. Let's focus on the `map()` family of functions, which operates on vectors and always returns the same type of output. The most basic function, `map()`, always returns a list and uses the basic syntax of `map(vector, function)`. For example, to take the square-root of our data, we could:

```
map(head(mtcars$mpg, 3), sqrt)
#> [[1]]
#> [1] 4.583
#>
#> [[2]]
#> [1] 4.583
#>
#> [[3]]
#> [1] 4.775
```

There are specialized variants of `map()` that return values when we know or expect that the function will generate one of the basic vector types. For example, since the square-root returns a double-precision number:

```
map_dbl(head(mtcars$mpg, 3), sqrt)
#> [1] 4.583 4.583 4.775
```

There are also mapping functions that operate across multiple vectors:

```
ratios <- map2_dbl(mtcars$mpg, mtcars$wt, compute_log_ratio)
head(ratios)
#> [1] 2.081 1.988 2.285 1.896 1.693 1.655
```

The `map()` functions also allow for temporary, anonymous functions defined using the tilde character. The argument values are `.x` and `.y` for `map2()`:

```
map2_dbl(mtcars$mpg, mtcars$wt, ~ log(.x/.y)) %>%
  head()
#> [1] 2.081 1.988 2.285 1.896 1.693 1.655
```

These examples have been trivial but, in later sections, are applied to more complex problems.

For functional programming in tidy modeling, functions should be defined so that functions like `map()` can be used for iterative computations.

2.2 EXAMPLES OF TIDYVERSE SYNTAX

Before diving into examples, let's discuss how the tidyverse relies on a type of data frame called a "tibble". Tibbles have slightly different rules than basic data frames in R. For example, tibbles naturally work with column names that are not syntactically valid variable names:

```
# Wants valid names:
data.frame(`variable 1` = 1:2, two = 3:4)
#>   variable.1 two
#> 1          1   3
#> 2          2   4
# But can be coerced to use them with an extra option:
df <- data.frame(`variable 1` = 1:2, two = 3:4, check.names = FALSE)
df
#>   variable 1 two
#> 1          1   3
#> 2          2   4

# But tibbles just work:
tbbl <- tibble(`variable 1` = 1:2, two = 3:4)
tbbl
#> # A tibble: 2 × 2
#>   `variable 1`   two
#>          <int> <int>
#> 1            1     3
#> 2            2     4
```

Standard data frames enable *partial matching* of arguments so that code using only a portion of the column names still work. Tibbles prevent this from happening since it can lead to accidental errors.

```
df$tw
#> [1] 3 4

tbbl$tw
#> Warning: Unknown or uninitialised column: `tw`.
#> NULL
```

Tibbles also prevent one of the most common R errors: dropping dimensions. If a standard data frame subsets the columns down to a single column, the object is converted to a vector. Tibbles *never* do this:

```
df[, "two"]
#> [1] 3 4

tbbl[, "two"]
#> # A tibble: 2 × 1
#>     two
#>   <int>
#> 1     3
#> 2     4
```

There are various other advantages to using tibbles instead of data frames, such as better printing and more. Chapter 10 of Wickham and Grolemund (2016) has more details on tibbles.

To demonstrate some syntax, let's use tidyverse functions to read in data that could be used in modeling. The data set comes from the city of Chicago's data portal and contains daily ridership data for the city's elevated train stations. The data set has columns for:

- the station identifier (numeric),
- the station name (character),
- the date (character in `mm/dd/yyyy` format),
- the day of the week (character), and
- the number of riders (numeric).

Our tidyverse pipeline will conduct the following tasks, in order:

1. We will use the tidyverse package **readr** to read the data from the source website and convert them into a tibble. To do this, the `read_csv()` function can determine the type of data by reading an initial number of rows. Alternatively, if the column names and types are already known, a column specification can be created in R and passed to `read_csv()` .
2. We filter the data to eliminate a few columns that are not needed (such as the station ID) and change the column `stationname to station` . The function `select()` is used for this. When filtering, use either the names of the column names or a **dplyr** selector function. When selecting names, a new variable name can be declared using the argument format `new_name = old_name` .

3. The date field is converted to the R date format using the `mdy()` function from the **lubridate** package. We also convert the ridership numbers to thousands. Both of these computations are executed using the `dplyr::mutate()` function.

4. There are a small number of days that have replicate ridership numbers at certain stations. To mitigate this issue, we use the maximum number of rides for each station and day combination. We *group* the ridership data by station and day, and then *summarize* within each of the 1999 unique combinations with the maximum statistic.

The tidyverse code for these steps is:

```r
library(tidyverse)
library(lubridate)

url <- "http://bit.ly/raw-train-data-csv"

all_stations <- 
  # Step 1: Read in the data.
  read_csv(url) %>% 
  # Step 2: filter columns and rename stationname
  dplyr::select(station = stationname, date, rides) %>% 
  # Step 3: Convert the character date field to a date encoding.
  # Also, put the data in units of 1K rides
  mutate(date = mdy(date), rides = rides / 1000) %>% 
  # Step 4: Summarize the multiple records using the maximum.
  group_by(date, station) %>% 
  summarize(rides = max(rides), .groups = "drop")
```

This pipeline of operations illustrates why the tidyverse is popular. A series of data manipulations is used that have simple and easy to understand user interfaces; the series is bundled together in a streamlined and readable way. The focus is on how the user interacts with the software. This approach enables more people to learn R and achieve their analysis goals, and adopting these same principles for modeling in R has the same benefits.

2.3 CHAPTER SUMMARY

This chapter introduced the tidyverse, with a focus on applications for modeling. We described differences in conventions between the tidyverse and base R, and introduced two important components of the tidyverse system, tibbles and the pipe operator `%>%`. Data cleaning and processing can feel mundane at times, but these tasks are important for modeling in the real world; we illustrated how to use tibbles, the pipe, and tidyverse functions in an example data import and processing exercise.

REFERENCES

Wickham, H, M Averick, J Bryan, W Chang, L McGowan, R François, G Grolemund, et al. 2019. "Welcome to the Tidyverse." *Journal of Open Source Software* 4 (43).

Wickham, H, and G Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc.

2. If you've never seen `::` in R code before, it is an explicit method for calling a function. The value of the left-hand side is the *namespace* where the function lives (usually a package name). The right-hand side is the function name. In cases where two packages use the same function name, this syntax ensures that the correct function is called.↩
3. Not all algorithms can take advantage of sparse representations of data. In such cases, a sparse matrix must be converted to a more conventional format before proceeding.↩
4. Examples of function side effects could include changing global data or printing a value.↩