Tidy data management in R

Antonio Gasparrini, Arturo de la Cruz Libardi

01 November 2023

Contents

This session introduces advanced data management tools based on *tidyverse*, a collection of R packages built following a consistent design philosophy and *tidy* data formats. The tidyverse system offers a set of dedicated functions to perform a wide range of data management tasks using a simple and intuitive syntax. and its consistent data structure facilitates the use of pipes to program sequential operations in an efficient and straightforward way. The first section discusses the concept of tidy data and presents the key tidyverse features. The second section introduces methods to manage variables, introducing the basic usage of tidyverse functions. The next three sections illustrate tools to perform common data management tasks on datasets, from subsetting and reordering to merging, reshaping, and aggregating them. The last section briefly mentions other tidyverse packages used in more specialised tasks.

Introducing tidyverse

The design philosophy of the R software is heavily influenced by its nature as a language based on object-oriented and functional programming. As such, the emphasis is on the provision of a limited set of generic functions that can flexibly connect objects to perform a wide range of tasks. However, this requires a high-level grammar and the existence of complex data structures. In practical terms, R offers a flexible environment to directly program non-standard operations, but its core tools are less suitable for performing standardised procedures that require consistent processes.

This issue is pretty clear in data management, which involves routine but specialised and coordinated operations that require consistent data formats. In R, data management tasks often necessitate tools such as indexing and operators that imply high-level programming skills. This is one of the reasons why, historically,

R has not been considered a good option for data management, especially for users with limited expertise in programming.

This limitation has been addressed with the development of the *tidyverse*, a collection of R packages that are built using a coherent design and the same underlying data format. The philosophy of tidyverse is, in some sense, the opposite of what is described above: the series of packages provides a long list of highly-specialised functions to perform specific operations, but based on a simple and unified syntax and the same underlying data structure. This makes it simple to perform such tasks without having an in-depth knowledge of the language and programming aspects of R.

In this session, we will introduce the basic tools of tidyverse, focusing on functions available within two packages, DPLYR and TIDYR. We start by loading them, assuming they have been already installed, possibly together with the whole tidyverse set:

```
library(dplyr) ; library(tidyr)
```

In the next sections, we will discuss the key features of tidyverse and illustrate some of its applications for data management, showing how these complement and can be used together with standard R functions.

Tidy data and tidyverse features

Tidyverse is based on the idea of *tidy data*, a format in which each variable forms a column, each observation forms a row, and each value forms a cell. This consistent structure makes it easier to develop functions with common usage, each of them representing a *verb* of a simple grammar to perform different operations.

An example will clarify the issue. We can compare how to perform a basic operation, namely ordering a dataset, using both standard R tools and tidyverse functions (details will be provided in later sections). Let's start with the standard R method:

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

Now, using tidyverse functions:

```
arrange(mtcars, mpg)
```

The results of the two expressions (not shown) are identical, with the data frame `mtcars` ordered by the variable `mpg`. However, the two approaches are completely different. The first method relies on general tools, such as the generic function `order` (returning a vector with the sequence of order permutation indices) and basic indexing and extract operators such as '`[]`' and '`$`'. However, this simple operation already requires high-level programming skills, and involves cumbersome coding such as calling the `mtcars` object twice. In contrast, the tidyverse version needs the dedicated function `arrange()`, but it uses a simpler and more straightforward syntax.

The simple code above exemplifies key aspects of tidyverse. First, the first argument of tidyverse functions is always a dataset, and the following arguments define expressions or parameters to apply operations on it. This feature allows and encourages to building series of sequential operations through the *pipe* operator '`|>`' (or the alternative '`%>%`' available within tidyverse). In addition, the tidyverse functions make use of non-standard evaluation, whereby variables can be named directly in the expressions without the need to use operators or quotes, thus simplifying the syntax. Finally, all the tidyverse function works with *tibbles*, an extension of the standard `data.frame` class that offers some advantages for printing and internal functionalities.

Managing variables

In this section, we start our illustration by demonstrating basic data management operations involving the creation or modification of variables in a dataset. We will use the dataset BIRTHS from the package EPI. This

package is neither a standard nor a recommended package, and therefore it must be installed (the first time) and then loaded. In addition, the data frame BIRTHS should be loaded as well into the session:

```
library(Epi)
data(births)
```

The first example features the modification of an existing variable, transforming a numeric vector into a categorical variable. Specifically, we want to transform the variable `hyp` in `births`, which represents an indicator of maternal hypertension, and it is provided as a simple numeric code 0/1 in the original dataset (see `help(births)` for details). This operation, which can be carried out using basic R tools such as indexing and replacement expressions, is performed here using the function `mutate()` from the package DPLYR, offering the opportunity to illustrate the standard usage of tidyverse functions. Let's see:

```
mutate(births, preterm = factor(preterm, labels=c("no","yes"))) |> head(3)
```

```
##   id bweight lowbw gestwks preterm matage hyp sex
## 1  1    2974     0   38.52      no     34   0   2
## 2  2    3270     0      NA    <NA>     30   0   1
## 3  3    2620     0   38.15      no     35   0   2
```

The function includes the data frame `births` as its first argument, as customary in tidyverse, followed by an expression where the variable `hyp` is (re)assigned as categorical using a call to `factor()` with specific labels for the levels (see `help(factor)`). The expression is followed in a pipe with a call to `head()` to show the first rows of the modified dataset `births`. The process can be used both for modifying existing variables and for creating new ones.

The function `mutate()` is flexible, and it can accept multiple expressions as separate arguments to create/modify several variables. For example:

```
births <- mutate(births,
  preterm = factor(preterm, labels=c("no","yes")),
  hyp = factor(hyp, labels=c("no","yes")),
  sex = factor(sex, labels=c("male","female")),
  matagegr = cut(matage, breaks=c(0,30,35,100), right=F)
)
head(births, 3)
```

```
##   id bweight lowbw gestwks preterm matage hyp    sex matagegr
## 1  1    2974     0   38.52      no     34  no female  [30,35)
## 2  2    3270     0      NA    <NA>     30  no   male  [30,35)
## 3  3    2620     0   38.15      no     35  no female [35,100)
```

Here, for clarity, the different expressions related to the different variables are written in new lines. The function `transform()` represents the counterpart of `mutate()` available in standard BASE packages of R.

The function `mutate()` also offers more specialised features to perform more complex tasks, for example through other functions such as `across()`. See `help(mutate)` for details and examples.

Labelling, subsetting, and reordering datasets

Other common data management operations involve changes to the whole dataset, for instance adding or modifying labels, selecting rows or columns, or reordering them. Again, such tasks can be completed using basic R functionalities such as indexing and replacement expressions, in addition to existing functions in standard R packages. However, tools made available in tidyverse packages can simplify these operations.

In this section, we will show some examples using the dataset ESOPH available in the DATASET package, storing the results of a case-control study of oesophageal cancer in France (see `help(esoph)`). Let's have a look:

```
head(esoph, 3)
```

```
##   agegp     alcgp    tobgp ncases ncontrols
## 1 25-34 0-39g/day 0-9g/day      0        40
## 2 25-34 0-39g/day    10-19      0        10
## 3 25-34 0-39g/day    20-29      0         6
```

The first example is about changing the name of the variables of the dataset. This can be simply accomplished with basic *labelling* methods, for example using a replacement expressions involving the function `names()`. However, this requires replacing all the variable names or otherwise writing more complex indexing expression to identify the specific variables. A simpler way is offered by the function `rename()` from the package DPLYR. Let's see:

```
esoph <- rename(esoph, age=agegp, alcohol=alcgp, tobacco=tobgp, cases=ncases,
  controls=ncontrols)
head(esoph, 3)
```

```
##     age   alcohol  tobacco cases controls
## 1 25-34 0-39g/day 0-9g/day     0       40
## 2 25-34 0-39g/day    10-19     0       10
## 3 25-34 0-39g/day    20-29     0        6
```

You can check that the names of the variables are now replaced. The function `rename()`, as usual in tidyverse, includes the dataset as its first argument, followed by one or multiple expressions with the form `old = new` to replace existing names with new ones. Compared to the standard operation involving replacement expressions with the functions `name()` or `colnames()`, this easily allows changing only the names of selected variables and being explicit about the selection.

Another basic operation is *subsetting*, applied to select specific records or variables in a dataset. This operation can be accomplished using the function `subset()` from the BASE package. Alternatively, the two functions `filter()` and `select()` from the package DPLYR can be applied to extract specific records (rows) or variables (columns), respectively. Let's see an example where we select records in `esoph` with age corresponding to the group 55-64 and tobacco consumption (number of cigarettes a day) equal to 10-19 or 20-29. The code:

```
esoph |> filter(age=="55-64", tobacco %in% c("10-19","20-29")) |>
  select(alcohol, tobacco, cases, controls)
```

```
##     alcohol tobacco cases controls
## 1 0-39g/day   10-19     3       19
## 2 0-39g/day   20-29     3        9
## 3     40-79   10-19     6       15
## 4     40-79   20-29     4       13
## 5    80-119   10-19     8        7
## 6    80-119   20-29     3        3
## 7      120+   10-19     6        1
## 8      120+   20-29     2        1
```

The function `filter()` defines the selection of the rows through logical statements added as separate arguments, while `select()` includes the names of the variables to be selected, in the requested order. Note that the latter involves expressions with unquoted variables, as typical in tidyverse. in addition, operators such as ':', '!', '-', as well as functions such as `starts_with()` or `contains()`, can be used to select a range of variables or exclude some, respectively. See `help(select)` for details.

In the expression above, we see the standard syntax commonly used in tidyverse, where a call to the data frame is followed by a series of instructions through pipes, each of them involving a call to specific functions to perform sequential operations. The pipe is naturally used in tidyverse functions, whereby the dataset

is passed as the first argument of sequential function calls without the need to directly specify it in the expressions.

There are other useful tools in the package DPLYR for subsetting and extracting information from a dataset. For instance, `slice()` and similar functions can be used to extract rows based on their indices, while `pull()` extracts single columns (see the related help pages). Some examples:

```
slice(esoph, 4:5)
```

```
##     age   alcohol  tobacco cases controls
## 1 25-34 0-39g/day      30+     0        5
## 2 25-34     40-79 0-9g/day     0       27
```

```
pull(esoph, tobacco) |> head()
```

```
## [1] 0-9g/day 10-19    20-29    30+      0-9g/day 10-19
## Levels: 0-9g/day < 10-19 < 20-29 < 30+
```

These functions perform similar operations to indexing and extraction operators '`[]`', '`[[]]`', and '`$`', but they are more easily applied within piped calls (however, see `help('|>')` for the use of these operators together with the placeholder '_' in pipes).

Another usual task is *reordering* a dataset. This usually involves changing the order of the rows, or (less often) the variables. The reordering of the rows is commonly performed by sorting by one or more variables. As before, this task can be performed simply using indexing and the function `order()` (as shown earlier), but it can be easier using dedicated tidyverse functions. As an example, we reorder `esoph` by tobacco and then alcohol consumption, and then age. In addition, we change the order of the variables accordingly. here is the code:

```
esoph <- arrange(esoph, tobacco, alcohol, age) |> select(c(3:1,4:5))
```

You can check that the dataset has been reordered (not shown). In the piped expressions above, the first call is to the function `arrange()`, which includes as additional arguments the variable(s) used for ordering the rows (with the optional second and following ones used to break optional ties). You can use the function `desc()` applied to one of the variables to arrange in descending order. See `help(arrange)` for more information. The second function call is for `select()`, used here to reorder columns, adopting an alternative syntax that defines their index positions rather than variable names.

The specific definition of the tidyverse function makes it straightforward to perform all the different operations above using piped expressions, without the need to create intermediate objects or re-assign existing ones.

Appending, merging, and reshaping datasets

Very often, common data management tasks require more complex operations, for example when information must be gathered from multiple data sources, or when the format of the dataset needs to be changed. To illustrate some examples, we make use of the dataset MTCARS from the package DATASET, which includes technical aspects of 32 cars featured in the US magazine *Motor Trend*. Let's have a look at it:

```
head(mtcars, 3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

We start by describing a relatively simple operation that involves *appending* datasets, that is stacking them by columns. The DPLYR package provides the function `bind_rows()` (and similarly `bind_cols()`), which

extends the standard function `rbind()` (and `cbind()`) by allowing the binding of more than two data frames. See the related help pages for details.

More complex computations are required to combine information from datasets that store information on different variables, possibly referring to different levels, for example individual and group-level measures. The process requires a *merging* operation, that is combining two datasets in a way that the independent variables provided by each one are linked through the values of the common variables. As an example, let's create a small hypothetical dataset with information on a certain score determined by the number of forward gears in a car:

```
gearscore <- data.frame(gear=3:6, score=c(2.1, 7.5, 8.1, 8.3))
```

This information can be merged with the main data frame `mtcars`, using the common information on the forward gears. This can be done through the standard function `merge()` from the BASE package. Instead, DPLYR provides a set of alternative functions, for example `inner_join()`. Let's see how to use it:

```
inner_join(mtcars, gearscore, by="gear")
```

The resulting data frame (not shown here) now includes the variable `score`. The argument `by` defines the merging variable (or more than one), and can be omitted when this corresponds to common variable names (as in this case). You can notice that `inner_join()` only keeps the rows that have a matching key between the two datasets. Alternatively, `left_join()`, `right_join()`, and `full_join()` can be used to keep all observations in the first, second, or both datasets, respectively. See the related help pages for additional features and details.

Often, datasets contain multiple values of a variable for the same observational unit. In this case, the information can be stored in different formats, which are usually categorised as *wide* or *long*. The process of transforming a dataset from wide to long, or in the opposite direction, is called *reshaping*. This is particularly important here, as the idea of tidy data and the usage of tidyverse functions always imply that the data are stored in long format.

To show some example of reshaping, we use the dataset SLEEP, also available in the DATASET package. This contains the results of a study on the effect of two soporific drugs in increasing the sleep time in ten patients (see `help(sleep)`). Let's have a look:

```
head(sleep, 3)
```

```
##   extra group ID
## 1   0.7     1  1
## 2  -1.6     1  2
## 3  -0.2     1  3
```

The data frame `sleep` has a long format, as each row refers to a unique measurement of the variable `extra`. The ten subjects (defined by `ID`) have two different measurements taken in relation to each drug, defined by `group`. Here we show how to reshape the dataset in a wide format using the function `pivot_wider()` from the package TIDYR. The code:

```
sleepwide <- pivot_wider(sleep, id_cols=ID, values_from=extra, names_from=group,
  names_prefix="extra")
head(sleepwide, 3)
```

```
## # A tibble: 3 x 3
##   ID    extra1 extra2
##   <fct>  <dbl>  <dbl>
## 1 1        0.7    1.9
## 2 2       -1.6    0.8
## 3 3       -0.2    1.1
```

The new dataset `sleepwide` stores the same information but in a wide format: each row includes the in-

formation for each subject and the two measurements are now included in separate variables `extra1` and `extra2`. The arguments of `pivot_wider()` include `id_cols`, which set the column(s) identifying multiple records from the same group in the long format, as well as `values_from` and `names_from` that refer to the variable(s) storing the information on the measurements and group identifier, respectively. The additional argument `names_prefix` is used to add a common stub for the names of the new variables in the wide format. You can also notice that the resulting object has a tibble structure.

The opposite transformation, from wide to long, is performed using the function `pivot_longer()`. Let's use it to back-transform `sleepwide`:

```
pivot_longer(sleepwide, cols=2:3, values_to="extra", names_to="group") |>
  head(3)
```

```
## # A tibble: 3 x 3
##   ID    group  extra
##   <fct> <chr>  <dbl>
## 1 1     extra1   0.7
## 2 1     extra2   1.9
## 3 2     extra1  -1.6
```

The function has a similar usage and arguments. Specifically, `cols` identifies the column(s) to be pivoted into a longer format, while `values_to` and `names_to` set the names of the new variables to store the information on multiple measurements and related groups, respectively.

Note that `pivot_wider()` and `pivot_longer()` provide a way to reshape datasets with more complex data structures, involving multiple matching, measurement, and grouping variables. in these cases, their usage and sets of arguments can be relatively more sophisticated. Specific information can be found in the related help pages. Finally, it is important to mention that these functions have replaced the old functions `spread()` and `gather()`, respectively, also available in the package TIDYR.

Aggregating datasets

A final data management operation illustrated in this session is about *aggregating* a dataset, which is a procedure to create a more compact version with information collapsed to a grouped level. In practical terms, the aggregation is done by recomputing the value of some variables with summary statistics repeated for groups defined by one or more factors.

The standard R function to perform such an operation is `aggregate()`. This can be used to apply a specific function to obtain an aggregated statistic for a variable, or optionally multiple variables. However, it has the big limitation that only one aggregation function can be applied, and for instance it is not straightforward to obtain different summaries for one or more variables.

In tidyverse, operations involving aggregation can be more flexibly and efficiently performed using functions in the DPLYR package. The main function is `summarise()`, which is its basic usage returns a data frame with a single row that includes one or more summary statistics. Let's see an example using again the dataset MTCARS introduced above:

```
summarise(mtcars, avgmpg=mean(mpg), medhp=median(hp))
```

```
##     avgmpg medhp
## 1 20.09062   123
```

Here, `summarise()` includes the data frame as the first argument (as usual), followed by expressions involving existing functions to obtain summary statistics assigned to new variables. This syntax provides a flexible way to obtain different summaries for the same or different variables.

However, this usage is rather limited, as it produces only a one-row dataset with related statistics. A more common and useful aggregation procedure involves the definition of cross-classifying factors that define

groups from which the statistics are repeatedly computed. This can be achieved using the argument `.by` of `summarise()`:

```
summarise(mtcars, avgmpg=mean(mpg), medhp=median(hp), .by=c(vs,am))
```

```
##   vs am   avgmpg medhp
## 1  0  1 19.75000 142.5
## 2  1  1 28.37143  66.0
## 3  1  0 20.74286 105.0
## 4  0  0 15.05000 180.0
```

Here, the original dataset is collapsed by computing the average consumption (miles per gallon in the variable `mpg`) and the median gross horsepower (variable `hp`) by engine and transmission types (variables `vs` and `am`, respectively). The output is therefore a data frame with multiple rows corresponding to each group, and with columns representing the summary statistics in addition to the cross-classifying factors.

A more flexible method involves the use of an additional function named `group_by()`. This function specified a grouped structure for the data frame, so that the following calls within `summarise()` are applied to each group separately. The function `ungroup()` removes the grouped structure. Let's repeat the previous example using this alternative method (results not shown):

```
mtcars |> group_by(vs, am) |> summarise(avgmpg=mean(mpg), medhp=median(hp))
```

Here, we apply a composite piped expression that includes a call to `group_by()` before computing the summary statistics. The arguments of the function simply represent the cross-classifying factors (or some computations involving them) to define the groups.

It is important to note that the results of the functions applied within `summarise()` are expected to produce a single summary statistic, so that the final dataset will have a single value (and row) for each combination of the cross-classifying factors. However, in some cases, more flexible aggregating operations are needed, where complex functions return multiple statistics. In this case, you can rely on the function `reframe()`, which works in a similar way as `summarise()` but can return an arbitrary number of rows per group. Let's see an example using the function `quantile()` to obtain multiple percentiles of `mpg`, using a single cross-classifying factor to reduce the size of the output:

```
mtcars |>
  group_by(vs) |>
  reframe(probs=paste0(1:3*25, "%"), mpg=quantile(mpg, 1:3*25/100))
```

```
## # A tibble: 6 x 3
##      vs probs   mpg
##   <dbl> <chr> <dbl>
## 1     0 25%    14.8
## 2     0 50%    15.6
## 3     0 75%    19.1
## 4     1 25%    21.4
## 5     1 50%    22.8
## 6     1 75%    29.6
```

Here the function `reframe()` returns three rows for each group, corresponding to the quartiles of `mpg`, as well as another variable `probs` identifying them. Its use allows more complex aggregating operations.

Other tidyverse packages

In this session, we have focused on basic tidyverse tools provided by the packages DPLYR and TIDYR. However, the tidyverse system features many other packages dedicated to specific tasks. In particular, the core

collection also includes the package GGPLOT2 for creating graphics, STRINGR for working with strings, FORCATS to deal with factors and categorical variables, READR as a fast way to read rectangular data, PURRR for functional programming tools, and TIBBLE for extensions of the data frame. Worth mentioning is also the LUBRIDATE package for dealing with date/time data. In addition, other contributed packages follow the tidyverse design philosophy and extend the core collection. You can refer to the related documentation for a detailed overview.