

Principles and Practice of Data Analysis
for Reproducible Research in R

Data Handling

Heather Turner

Department of Statistics, University of Warwick

2017-09-12

RStudio Projects

An Rstudio *project* is a context for work on a specific project, with its own working directory, workspace and command history.

A new project can be created from the Project tab

- ▶ in a brand new directory
- ▶ from an existing directory

It is possible to save the workspace on exit, and restore when the project is re-opened.

Project Structure

By opening the project, the working directory is automatically set to the project root folder.

Sub-directories can be created to organise work, e.g.

- ▶ raw data
- ▶ processed data
- ▶ R scripts
- ▶ outputs (figures/documents)

Relative paths should then be used to specify files, e.g.

"../data/survey.csv". If it is not practical to store the data under the project tree, assign a name to the data directory, so it is easy to change

```
dir <- "//network/directory" # put at top of script
files <- list.files(file.path(dir, "experiment1.csv"))
```

Data Input via Import Dataset

The *Import Dataset* dialog in RStudio can import files stored locally or online in the following formats:

- ▶ `.txt/.csv` via `read_delim/read_csv` from `readr`.
- ▶ `.xlsx` via `read_excel` from `readxl`.
- ▶ `.sav/.por` , `.sas7bdat` and `.dta` via `read_spss`, `read_sas` and `read_stata` respectively from `haven`.

Most of these functions also allow files to be compressed, e.g. as `.zip`.

Tibbles

The functions used by *Import Dataset* return enhanced data frames of class `"tbl_df"`, aka **tibbles**. These have certain advantages over the standard R data frame:

- ▶ no partial name matching with `$` (warns if name does not exist)
- ▶ safe printing: default 10-20 rows and columns to fit visible width
- ▶ single column indexing, e.g. `tbl[, 1]` returns a tibble not a vector
- ▶ rows are numbered not named
- ▶ no copy created when column names changed
- ▶ only recycle values of length 1

In addition the `read_` functions do not convert character vectors to factors as the standard `read.` functions do.

Data Input via Code

For writing code it is convenient to use the **rio** package, which provides a common interface to the functions used by *Import Dataset* as well as many others.

The data format is automatically recognised from the file extension; to read the data in as a tibble, we use the **setclass** argument.

```
library(rio)
compsci <- import("compsci.csv", setclass = "tibble")
cyclist <- import("cyclist.xlsx", setclass = "tibble")
```

See **?rio** for the underlying functions used for each format and the corresponding optional arguments, e.g. the **skip** argument to **read_excel** to skip a certain number of rows.

Basic Checks

Using `View` to view the data in RStudio is a good way to check the data has been read in as expected. Other useful summaries include

- ▶ `head/tail` to look at the first/last few rows of data
- ▶ `dim` to find out the dimensions (number of rows and columns)
- ▶ `summary` to summarise each variable in the data

The print method for tibbles has arguments `n` and `width` to set the number of rows and the output width, in number of characters

```
print(compsci, n = 2, width = 80)

## # A tibble: 44 x 7
##           Year `Bachelor's - Men` `Bachelor's - Women`
##           <chr>           <int>           <int>
## 1 1970-71 .....                2064                324
## 2 1971-72 .....                2941                461
## # ... with 42 more rows, and 4 more variables: `Master's -
## #   Men` <int>, `Master's - Women` <int>, `Doctor's - Men` <int>,
## #   `Doctor's - Women` <int>
```

Tidy Data

To make it easier to work with, i.e. to operate on, visualise, or model, data should be *tidy*, i.e.

- ▶ each column is a variable
- ▶ each row is an observation

Then you have a consistent way to refer to variables (by name) and to observations (by number).

The **tidyr** package provides functions to help get data into this tidy form. The key functions are **gather** and **separate**.

gather

The `gather` function gathers a variable that is spread over multiple columns into *two* columns: one for the key (original column name) and one for the value.

```
library(tidyr)
compsci2 <- gather(compsci, key = "Student Group",
                  value = "Number of students", -Year)
print(compsci2, n = 3)
```

A tibble: 264 x 3

	Year	`Student Group`	`Number of students`
	<chr>	<chr>	<int>
## 1	1970-71	Bachelor's - Men	2064
## 2	1971-72	Bachelor's - Men	2941
## 3	1972-73	Bachelor's - Men	3664

... with 261 more rows

The last value specifies the columns to gather: `-Year` means everything minus `Year`. Alternatively name each column as a separate argument, or specify a sequence: ``Bachelor's - Men`:`Doctor's - Women``.

separate

The `separate` function separates values that have been concatenated into a single variable.

```
compsci2 <- separate(compsci2, col = `Student Group`,
                      into = c("Degree", "Gender"),
                      sep = " - ")
print(compsci2, n = 3)
```

```
## # A tibble: 264 x 4
##           Year      Degree Gender `Number of students`
## *      <chr>      <chr> <chr>           <int>
## 1 1970-71 Bachelor's Men             2064
## 2 1971-72 Bachelor's Men             2941
## 3 1972-73 Bachelor's Men             3664
## # ... with 261 more rows
```

By default (if no `sep` is specified), the column will be split on any sequence of non-alphanumeric characters. Since the degree names have apostrophes, we specify a custom `sep` to avoid two splits.

Further Tidying

Other functions in **tidyr** are focused on the following tasks

More separating `separate_rows` separates concatenated values into multiple rows

Expanding e.g. `complete` to include missing combinations of values

Handling missing values replacing/filling in/dropping missing values

Reverse operations create messy data! Can be useful for table output.

Data Wrangling

Often we need to go beyond tidying the data to create *derived* data sets.

The `dplyr` package provides the following key functions to operate on data frames

- ▶ `filter()`
- ▶ `arrange()`
- ▶ `select()` (and `rename()`)
- ▶ `distinct()`
- ▶ `mutate()` (and `transmute()`)
- ▶ `summarise()`

filter()

`filter()` selects rows of data by criteria

```
library(dplyr)
filter(compsci2, Gender == "Men" & `Number of students` > 40000)

## # A tibble: 5 x 4
##       Year      Degree Gender `Number of students`
##   <chr>    <chr>  <chr>         <int>
## 1 2002-03 Bachelor's Men           41950
## 2 2003-04 Bachelor's Men           44585
## 3 2004-05 Bachelor's Men           42125
## 4 2012-13 Bachelor's Men           41874
## 5 2013-14 Bachelor's Men           45393
```

The second argument can be anything that returns a logical vector.

Logical Filters

The following components are useful for defining filters

Binary comparisons `>`, `<`, `==`, `<=`, `>=` and `!=`

Logical operators `or` `|`; and `&`, not `!`

Value matching e.g. `Degree %in% c("Master's", "Doctor's");`
`grepl`

Missing value indicator `is.na`

arrange()

`arrange()` orders the rows of data by one or more variables.

By default, ordering is in ascending order; use `desc()` for descending order

```
print(arrange(compsci2, desc(Year), Gender), n = 4)
```

```
## # A tibble: 264 x 4
```

```
##           Year      Degree Gender `Number of students`
##           <chr>      <chr>  <chr>                <int>
## 1 2013-14 ..... Bachelor's      Men                45393
## 2 2013-14 .....   Master's      Men                17484
## 3 2013-14 .....   Doctor's      Men                 1566
## 4 2013-14 ..... Bachelor's    Women                9974
## # ... with 260 more rows
```

select()

`select()` selects variables from the data frame. Columns are selected in the same way as for `gather()`

```
select(compsci2, Year, Gender, `Number of students`)  
select(compsci2, Year:Degree, `Number of students`)  
select(compsci2, -Degree, -Gender)  
select(compsci2, -(Year:Gender))
```

Blocks of variables can be selected using `starts_with()`, `ends_with()`, `contains()` and `num_range()`

```
print(select(compsci, starts_with("Bachelor's")), n = 2)  
  
## # A tibble: 44 x 2  
##   `Bachelor's - Men` `Bachelor's - Women`  
##           <int>           <int>  
## 1             2064             324  
## 2             2941             461  
## # ... with 42 more rows
```


Renaming Variables

Variables can be renamed when they are selected using named arguments, e.g.

```
select(compsci2, `Academic Year` = Year, Gender, `Number of students`)
```

However this drops any variables not specified in the selection. To rename without selection, use `rename()`

```
rename(compsci2, `Academic Year` = Year)
```

N.B. the new name is given on the left!

Obtaining Distinct Records

`distinct()` extracts records with unique combinations of the specified variables

```
distinct(compsci2, Degree, Gender)
```

```
## # A tibble: 6 x 2
##   Degree Gender
##   <chr> <chr>
## 1 Bachelor's Men
## 2 Bachelor's Women
## 3 Master's Men
## 4 Master's Women
## 5 Doctor's Men
## 6 Doctor's Women
```

Computing New Columns

`mutate()` computes new columns based on existing columns. Re-using an existing name replaces the old variable

```
dat <- mutate(compsci2,
              Postgrad = Degree != "Bachelor's",
              Year = gsub(".", "", Year, fixed = TRUE),
              Year = sub(" ", "", Year, fixed = TRUE))
print(dat, n = 2)
```

```
## # A tibble: 264 x 5
##   Year      Degree Gender `Number of students` Postgrad
##   <chr>    <chr> <chr>             <int>    <lgl>
## 1 1970-71 Bachelor's   Men             2064    FALSE
## 2 1971-72 Bachelor's   Men             2941    FALSE
## # ... with 262 more rows
```

Note computations are in the order given, so mutated columns can be used in subsequent computations

Discarding Original Variables

To only keep the computed columns, use `transmute()`

```
dat <- transmute(compsci2,  
                 Postgrad = Degree != "Bachelor's",  
                 Year = gsub(".", "", Year, fixed = TRUE),  
                 Year = sub(" ", "", Year, fixed = TRUE))  
print(dat, n = 2)
```

```
## # A tibble: 264 x 2  
##   Postgrad   Year  
##   <lgl>    <chr>  
## 1     FALSE 1970-71  
## 2     FALSE 1971-72  
## # ... with 262 more rows
```

To keep some original and some computed columns, we could use `mutate` followed by `select`, or set a transmuted variable equal to the original, e.g. `Gender = Gender`.

Summarise Columns

`summarise()` is for computing single number summaries of variables

```
summarise(compsci2,  
           Average = mean(`Number of students`),  
           Total = sum(`Number of students`))  
  
## # A tibble: 1 x 2  
##   Average   Total  
##   <dbl>   <int>  
## 1     6862 1811522
```

Selected variables can be summarised using `summarise_all()`, `summarise_at()` and `summarise_if()`

```
summarise_if(compsci2, is.numeric, mean)  
  
## # A tibble: 1 x 1  
##   `Number of students`  
##                   <dbl>  
## 1                   6862
```

Multiple Steps

Typically data pre-processing will involve multiple steps

```
dat <- mutate(compsci2,  
              Year = gsub(".", "", Year, fixed = TRUE),  
              Year = sub(" ", "", Year, fixed = TRUE))  
dat <- filter(dat, Year == "2013-14" & Degree != "Bachelor's")  
select(dat, -Year)
```

```
## # A tibble: 4 x 3  
##   Degree Gender `Number of students`  
##   <chr> <chr> <int>  
## 1 Master's Men 17484  
## 2 Master's Women 7048  
## 3 Doctor's Men 1566  
## 4 Doctor's Women 416
```

Chaining

Since the first argument to all **dplyr** functions is the data frame to operate on, we can use "%>%" to pipe the data from one step to the next

```
compsci2 %>%
  mutate(Year = gsub(".", "", Year, fixed = TRUE),
         Year = sub(" ", "", Year, fixed = TRUE)) %>%
  filter(Year == "2013-14" & Degree != "Bachelor's") %>%
  select(-Year)

## # A tibble: 4 x 3
##   Degree Gender `Number of students`
##   <chr> <chr> <int>
## 1 Master's Men 17484
## 2 Master's Women 7048
## 3 Doctor's Men 1566
## 4 Doctor's Women 416
```

Pipe-aware Functions

Any function with data as the first argument can be added to the data pipeline, e.g. `tidyr` functions.

```
compsci %>%
  gather(key = "Student Group", value = "Number of students",
         -Year) %>%
  separate(col = `Student Group`, into = c("Degree", "Gender"),
          sep = " - ") %>%
  mutate(Year = gsub(".", "", Year, fixed = TRUE),
         Year = sub(" ", "", Year, fixed = TRUE)) %>%
  filter(Year == "2013-14" & Degree != "Bachelor's") %>%
  select(-Year)

## # A tibble: 4 x 3
##   Degree Gender `Number of students`
##   <chr> <chr> <int>
## 1 Master's Men 17484
## 2 Master's Women 7048
## 3 Doctor's Men 1566
## 4 Doctor's Women 416
```


Grouped Operations

Grouping can be set on a data frame using `group_by`. This affects the **dplyr** functions as follows

- ▶ `select()` adds the grouping variables to the selection if you don't
- ▶ `arrange()` acts as on an unordered data frame
- ▶ `mutate()` and `filter()` operate per group - only differ when involve a summary statistic
- ▶ `summarise()` operate per group

Grouping

```
compsci2 %>%  
  filter(grepl("2013-14", Year)) %>%  
  group_by(Gender) %>%  
  select(Degree, `Number of students`) %>%  
  arrange(`Number of students`)
```

```
## # A tibble: 6 x 3  
## # Groups:   Gender [2]  
##   Gender      Degree `Number of students`  
##   <chr>      <chr>          <int>  
## 1 Women      Doctor's          416  
## 2 Men        Doctor's         1566  
## 3 Women      Master's          7048  
## 4 Women      Bachelor's       9974  
## 5 Men        Master's         17484  
## 6 Men        Bachelor's      45393
```

Grouped Mutate

```
compsci2 %>%
  filter(grepl("2013-14", Year)) %>%
  group_by(Gender) %>%
  mutate(`Relative number` =
          100 * `Number of students`/max(`Number of students`))

## # A tibble: 6 x 5
## # Groups:   Gender [2]
##           Year      Degree Gender `Number of students`
##           <chr>      <chr> <chr>             <int>
## 1 2013-14 Bachelor's Men             45393
## 2 2013-14 Bachelor's Women              9974
## 3 2013-14 Master's  Men             17484
## 4 2013-14 Master's  Women              7048
## 5 2013-14 Doctor's  Men              1566
## 6 2013-14 Doctor's  Women              416
## # ... with 1 more variables: `Relative number` <dbl>
```

Grouped Summarise

```
compsci2 %>%  
  filter(grepl("2013-14", Year)) %>%  
  group_by(Gender) %>%  
  summarise(Total = sum(`Number of students`))  
  
## # A tibble: 2 x 2  
##   Gender Total  
##   <chr> <int>  
## 1   Men 64443  
## 2  Women 17438
```

Factors

For the purpose of data manipulation categorical variables may be stored as character or numeric.

For analysis, particularly modelling, categorical variables must be defined as factors. By default factor levels are the ordered unique values

```
dat <- compsci2 %>%  
  mutate(Year = factor(Year),  
         Degree = factor(Degree),  
         Gender = factor(Gender))  
summary(select(dat, -`Number of students`))
```

##	Year	Degree	Gender
##	1970-71	Bachelor's:88	Men :132
##	1971-72	Doctor's :88	Women:132
##	1972-73	Master's :88	
##	1973-74		
##	1974-75		
##	1975-76		
##	(Other) :		228

Factors

The order of factor levels matters

visualisation geometric objects (bars/lines) displayed in order of levels

modelling first level taken as reference level

Levels and their labels can be specified as arguments to `factor`.

```
mutate(compsci2, Gender = factor(Gender, levels = c("Men", "Women"),
                                labels = c("M", "F")))
```

The **forcats** package provides useful functions to reorder levels, e.g.

```
library(forcats)
dat %>%
  mutate(Degree = fct_inorder(Degree),
         Gender = fct_relevel(Gender, "Men"))
summary(dat)
```

Other functions in **forcats** help to change levels, combine factors, etc.

Saving/Exporting (Processed) Data

The **rio** package also has an **export** function to export data, e.g to share with collaborators

```
export(compsci2, "compsci2.csv")
```

However, if the processed data is only saved as an intermediate step, it is better to save in the binary **.rds** format. This requires less memory, is quicker to load and will retain the tibble class

```
saveRDS(compsci2, "compsci_tidy.rds")  
genderbalance <- readRDS("compsci_tidy.rds")  
print(genderbalance, n = 2)
```