

# TRES Tidyverse Tutorial

Raphael and Pratik

2020-05-20



1

# Contents

2	<b>Outline</b>	<b>5</b>
3	<b>1 Reading files and string manipulation</b>	<b>7</b>
4	1.1 Section on <code>readr</code> . . . . .	7
5	1.2 String manipulation with <code>stringr</code> . . . . .	7
6	1.3 String interpolation with <code>glue</code> . . . . .	16
7	<b>2 Working with lists and iteration</b>	<b>19</b>
8	2.1 Basic iteration with <code>map</code> . . . . .	20
9	2.2 More <code>map</code> variants . . . . .	23
10	2.3 Modification in place . . . . .	23
11	2.4 Working with lists . . . . .	24



## Outline

This is the readable version of the TRES tidyverse tutorial, with these sections:

1. Reading data and string manipulation with readr, stringr, and glue
2. The new data frames with tibble and wrangling them into shape with tidyr
3. Manipulating data with dplyr
4. Iteration and functional programming with purrr
5. Plotting with ggplot2



## 23 Chapter 1

# 24 Reading files and string 25 manipulation



Every use case is ridiculous  
until it happens to you.

26

```
library(readr)
library(stringr)
library(glue)
```

## 27 1.1 Section on readr

## 28 1.2 String manipulation with stringr

29 **stringr** is the tidyverse package for string manipulation, and exists in an interesting  
30 symbiosis with the **stringi** package. For the most part, **stringr** is a wrapper around  
31 **stringi**, and is almost always more than sufficient for day-to-day needs.

32 `stringr` functions begin with `str_`.

### 33 1.2.1 Putting strings together

34 Concatenate two strings with `str_c`, and duplicate strings with `str_dup`. Flatten a list  
35 or vector of strings using `str_flatten`.

```

36 # str_c works like paste(), choose a separator
  str_c("this string", "this other string", sep = "_")

## [1] "this string_this other string"

# str_dup works like rep
  str_dup("this string", times = 3)

37 ## [1] "this stringthis stringthis string"

# str_flatten works on lists and vectors
  str_flatten(string = as.list(letters), collapse = "_")

38 ## [1] "a_b_c_d_e_f_g_h_i_j_k_l_m_n_o_p_q_r_s_t_u_v_w_x_y_z"

  str_flatten(string = letters, collapse = "-")

39 ## [1] "a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"

  str_flatten is especially useful when displaying the type of an object that returns a list
40 when class is called on it.

# get the class of a tibble and display it as a single string
  class_tibble = class(tibble::tibble(a = 1))
  str_flatten(string = class_tibble, collapse = ", ")

42 ## [1] "tbl_df, tbl, data.frame"

```

### 43 1.2.2 Detecting strings

44 Count the frequency of a pattern in a string with `str_count`. Returns an integer. Detect  
45 whether a pattern exists in a string with `str_detect`. Returns a logical and can be used  
46 as a predicate.

47 Both are vectorised, i.e, automatically applied to a vector of arguments.

```

# there should be 5 a-s here
  str_count(string = "ababababa", pattern = "a")

48 ## [1] 5

# vectorise over the input string
# should return a vector of length 2, with integers 5 and 3
  str_count(string = c("ababbababa", "banana"), pattern = "a")

49 ## [1] 5 3

```



```

# vectorise over the pattern to count both a-s and b-s
str_count(string = "ababababa", pattern = c("a", "b"))
## [1] 5 4

```

Vectorising over both string and pattern works as expected.

```

# vectorise over both string and pattern
# counts a-s in first input, and b-s in the second
str_count(string = c("ababababa", "banana"),
          pattern = c("a", "b"))
## [1] 5 1

```

provide a longer pattern vector to search for both a-s and b-s in both inputs

```

str_count(string = c("ababababa", "banana"),
          pattern = c("a", "b",
                     "b", "a"))
## [1] 5 1 4 3

```

`str_locate` locates the search pattern in a string, and returns the start and end as a two column matrix.

```

# the behaviour of both str_locate and str_locate_all is
# to find the first match by default
str_locate(string = "banana", pattern = "ana")
##           start end
## [1,]         2   4

```

`str_detect` detects a sequence in a string

```

str_detect(string = "Bananageddon is coming!",
          pattern = "na")
## [1] TRUE

```

`str_detect` is also vectorised and returns a two-element logical vector

```

str_detect(string = "Bananageddon is coming!",
          pattern = c("na", "don"))
## [1] TRUE TRUE

```

use any or all to convert a multi-element logical to a single logical here we ask if either of the patterns is detected

```

any(str_detect(string = "Bananageddon is coming!",
               pattern = c("na", "don")))
## [1] TRUE

```

Detect whether a string starts or ends with a pattern. Also vectorised. Both have a `negate` argument, which returns the negative, i.e., returns `FALSE` if the search pattern is detected.

```

# taken straight from the examples, because they suffice
fruit <- c("apple", "banana", "pear", "pineapple")
# str_detect looks at the first character
str_starts(fruit, "p")
64 ## [1] FALSE FALSE TRUE TRUE

# str_ends looks at the last character
str_ends(fruit, "e")
65 ## [1] TRUE FALSE FALSE TRUE

# an example of negate = TRUE
str_ends(fruit, "e", negate = TRUE)
66 ## [1] FALSE TRUE TRUE FALSE

str_subset [WHICH IS NOT RELATED TO str_sub] helps with subsetting a character
67 vector based on a str_detect predicate. In the example, all elements containing "ba-
68 nana" are subset.
69
70 str_which has the same logic except that it returns the vector position and not the ele-
71 ments.

# should return a subset vector containing the first two elements
str_subset(c("banana",
             "bananageddon is coming",
             "appleageddon is not real"),
           pattern = "banana")
72 ## [1] "banana" "bananageddon is coming"

# returns an integer vector
str_which(c("banana",
            "bananageddon is coming",
            "appleageddon is not real"),
          pattern = "banana")
73 ## [1] 1 2

```

### 74 1.2.3 Matching strings

75 `str_match` returns all positive matches of the pattern in the string. The return type is a  
 76 list, with one element per search pattern.

77 A simple case is shown below where the search pattern is the phrase “banana”.

```

str_match(string = c("banana",
                     "bananageddon",
                     "bananas are bad"),
          pattern = "banana")

```

```

78 ##      [,1]
79 ## [1,] "banana"
80 ## [2,] "banana"
81 ## [3,] "banana"

```

82 The search pattern can be extended to look for multiple subsets of the search pattern.  
 83 Consider searching for dates and times.

84 Here, the search pattern is a `regex` pattern that looks for a set of four digits (`\\d{4}`) and  
 85 a month name (`\\w+`) separated by a hyphen. There's much more to be explored in dealing  
 86 with dates and times in `[lubridate]` (<https://lubridate.tidyverse.org/>),  
 87 another `tidyverse` package.

88 The return type is a list, each element is a character matrix where the first column is  
 89 the string subset matching the full search pattern, and then as many columns as there  
 90 are parts to the search pattern. The parts of interest in the search pattern are indicated  
 91 by wrapping them in parentheses. For example, in the case below, wrapping `[-.]` in  
 92 parentheses will turn it into a distinct part of the search pattern.

```

# first with [-.] treated simply as a separator
str_match(string = c("1970-somemonth-01",
                     "1990-anothermonth-01",
                     "2010-thismonth-01"),
          pattern = "(\\d{4})[-.](\\w+)")

93 ##      [,1]      [,2]  [,3]
94 ## [1,] "1970-somemonth" "1970" "somemonth"
95 ## [2,] "1990-anothermonth" "1990" "anothermonth"
96 ## [3,] "2010-thismonth" "2010" "thismonth"

```

```

# then with [-.] actively searched for
str_match(string = c("1970-somemonth-01",
                     "1990-anothermonth-01",
                     "2010-thismonth-01"),
          pattern = "(\\d{4})([-.])(\\w+)")

97 ##      [,1]      [,2]  [,3] [,4]
98 ## [1,] "1970-somemonth" "1970" "-" "somemonth"
99 ## [2,] "1990-anothermonth" "1990" "-" "anothermonth"
100 ## [3,] "2010-thismonth" "2010" "-" "thismonth"

```

101 Multiple possible matches are dealt with using `str_match_all`. An example  
 102 case is uncertainty in date-time in raw data, where the date has been entered as  
 103 1970-somemonth-01 or 1970/anothermonth/01.

104 The return type is a list, with one element per input string. Each element is a character  
 105 matrix, where each row is one possible match, and each column after the first (the full  
 106 match) corresponds to the parts of the search pattern.

```

# first with a single date entry
str_match_all(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01"),

```

```

pattern = "(\\d{4})[\\-\\\\/](\\[a-z\\]+)"

107 ## [[1]]
108 ##      [,1]      [,2]      [,3]
109 ## [1,] "1970-somemonth" "1970" "somemonth"
110 ## [2,] "1990/anothermonth" "1990" "anothermonth"

# then with multiple date entries
str_match_all(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01",
                        "1990-somemonth-01 or maybe 2001/anothermonth/01"),
              pattern = "(\\d{4})[\\-\\\\/](\\[a-z\\]+)")

111 ## [[1]]
112 ##      [,1]      [,2]      [,3]
113 ## [1,] "1970-somemonth" "1970" "somemonth"
114 ## [2,] "1990/anothermonth" "1990" "anothermonth"
115 ##
116 ## [[2]]
117 ##      [,1]      [,2]      [,3]
118 ## [1,] "1990-somemonth" "1990" "somemonth"
119 ## [2,] "2001/anothermonth" "2001" "anothermonth"

```

#### 1.2.4 Simpler pattern extraction

The full functionality of `str_match_*` can be boiled down to the most common use case, extracting one or more full matches of the search pattern using `str_extract` and `str_extract_all` respectively.

`str_extract` returns a character vector with the same length as the input string vector, while `str_extract_all` returns a list, with a character vector whose elements are the matches.

```

# extracting the first full match using str_extract
str_extract(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01",
                      "1990-somemonth-01 or maybe 2001/anothermonth/01"),
            pattern = "(\\d{4})[\\-\\\\/](\\[a-z\\]+)")

127 ## [1] "1970-somemonth" "1990-somemonth"

# extracting all full matches using str_extract_all
str_extract_all(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01",
                          "1990-somemonth-01 or maybe 2001/anothermonth/01"),
                pattern = "(\\d{4})[\\-\\\\/](\\[a-z\\]+)")

128 ## [[1]]
129 ## [1] "1970-somemonth" "1990/anothermonth"
130 ##
131 ## [[2]]
132 ## [1] "1990-somemonth" "2001/anothermonth"

```

### 1.2.5 Breaking strings apart

`str_split`, `str_sub`, In the above date-time example, when reading filenames from a path, or when working sequences separated by a known pattern generally, `str_split` can help separate elements of interest.

The return type is a list similar to `str_match`.

```
# split on either a hyphen or a forward slash
str_split(string = c("1970-somemonth-01",
                     "1990/anothermonth/01"),
          pattern = "[\\-\\/]" )

## [[1]]
## [1] "1970"      "somemonth" "01"
##
## [[2]]
## [1] "1990"      "anothermonth" "01"

This can be useful in recovering simulation parameters from a filename, but may require
some knowledge of regex.

# assume a simulation output file
filename = "sim_param1_0.01_param2_0.05_param3_0.01.ext"

# not quite there
str_split(filename, pattern = "_")

## [[1]]
## [1] "sim"      "param1"   "0.01"     "param2"   "0.05"     "param3"   "0.01.ext"

# not really
str_split(filename,
          pattern = "sim_")

## [[1]]
## [1] ""
## [2] "param1_0.01_param2_0.05_param3_0.01.ext"

# getting there but still needs work
str_split(filename,
          pattern = "(sim_)|_*param\\d{1}_|(.ext)")

## [[1]]
## [1] ""      ""      "0.01" "0.05" "0.01" ""

str_split_fixed split the string into as many pieces as specified, and can be especially
useful dealing with filepaths.

# split on either a hyphen or a forward slash
str_split_fixed(string = "dir_level_1/dir_level_2/file.ext",
```

```

        pattern = "/",
        n = 2)

154 ##          [,1]          [,2]
155 ## [1,] "dir_level_1" "dir_level_2/file.ext"

```

### 1.2.6 Replacing string elements

`str_replace` is intended to replace the search pattern, and can be co-opted into the task of recovering simulation parameters or other data from regularly named files. `str_replace_all` works the same way but replaces all matches of the search pattern.

```

# replace all unwanted characters from this hypothetical filename with spaces
filename = "sim_param1_0.01_param2_0.05_param3_0.01.ext"
str_replace_all(filename,
    pattern = "(sim_)|_*param\\d{1}|(.ext)",
    replacement = " ")

160 ## [1] " 0.01 0.05 0.01 "

161 str_remove is a wrapper around str_replace where the replacement is set to "". This
162 is not covered here.

163 Having replaced unwanted characters in the filename with spaces, str_trim offers a
164 way to remove leading and trailing whitespaces.

# trim whitespaces from this filename after replacing unwanted text
filename = "sim_param1_0.01_param2_0.05_param3_0.01.ext"
filename_with_spaces = str_replace_all(filename,
    pattern = "(sim_)|_*param\\d{1}|(.ext)",
    replacement = " ")

filename_without_spaces = str_trim(filename_with_spaces)
filename_without_spaces

165 ## [1] "0.01 0.05 0.01"

# the result can be split on whitespaces to return useful data
str_split(filename_without_spaces, " ")

166 ## [[1]]
167 ## [1] "0.01" "0.05" "0.01"

```

### 1.2.7 Subsetting within strings

When strings are highly regular, useful data can be extracted from a string using `str_sub`. In the date-time example, the year is always represented by the first four characters.

```

# get the year as characters 1 - 4
str_sub(string = c("1970-somemonth-01",
    "1990-anothermonth-01",

```

177 **1.2.8 Padding and truncating strings**

```
# pad so all values have three digits
str_pad(string = c("1", "10", "100"),
        width = 3,
        side = "left",
        pad = "0")
```

182 Strings can also be truncated if they are too long.

```
str_trunc(string = c("bananas are great and wonderful  
and more stuff about bananas and  
it really goes on about bananas"),
```

```

width = 27,
side = "right", ellipsis = "etc. etc.")
183 ## [1] "bananas are great etc. etc."

```

### 184 1.2.9 Stringr aspects not covered here

185 Some stringr functions are not covered here. These include:

```

186 - str_wrap (of dubious use),
187 - str_interp, str_glue* (better to use glue; see below),
188 - str_sort, str_order (used in sorting a character vector),
189 - str_to_case* (case conversion), and
190 - str_view* (a graphical view of search pattern matches).

```

191 stringi, of which stringr is a wrapper, offers a lot more flexibility and control.

## 192 1.3 String interpolation with glue

193 The idea behind string interpolation is to procedurally generate new complex strings  
194 from pre-existing data.

195 glue is as simple as the example shown.

```

# print that each car name is a car model
cars = rownames(head(mtcars))
glue('The {cars} is a car model')

196 ## The Mazda RX4 is a car model
197 ## The Mazda RX4 Wag is a car model
198 ## The Datsun 710 is a car model
199 ## The Hornet 4 Drive is a car model
200 ## The Hornet Sportabout is a car model
201 ## The Valiant is a car model

```

202 This creates and prints a vector of car names stating each is a car model.

203 The related glue\_data is even more useful in printing from a dataframe. In this example,  
204 it can quickly generate command line arguments or filenames.

```

# use dataframes for now
parameter_combinations = data.frame(param1 = letters[1:5],
                                     param2 = 1:5)

# for command line arguments or to start multiple job scripts on the cluster
glue_data(parameter_combinations,
           'simulation-name {param1} {param2}')

205 ## simulation-name a 1
206 ## simulation-name b 2
207 ## simulation-name c 3

```



```
208 ## simulation-name d 4
209 ## simulation-name e 5

    # for filenames
    glue_data(parameter_combinations,
              'sim_data_param1_{param1}_param2_{param2}.ext')

210 ## sim_data_param1_a_param2_1.ext
211 ## sim_data_param1_b_param2_2.ext
212 ## sim_data_param1_c_param2_3.ext
213 ## sim_data_param1_d_param2_4.ext
214 ## sim_data_param1_e_param2_5.ext

215 Finally, the convenient glue_sql and glue_data_sql are used to safely write SQL
216 queries where variables from data are appropriately quoted. This is not covered here,
217 but it is good to know it exists.

218 glue has some more functions — glue_safe, glue_collapse, and glue_col, but
219 these are infrequently used. Their functionality can be found on the glue github page.
```



## 220 Chapter 2

## 221 Working with lists and iteration

Every use case is ridiculous  
until it happens to you.

```
222 # load the tidyverse  
library(tidyverse)  
  
223 ## -- Attaching packages ----- tidyverse 1.3.0 --  
  
224 ## v ggplot2 3.3.0      v purrr  0.3.4  
225 ## v tibble  3.0.1      v dplyr  0.8.5  
226 ## v tidyr   1.0.2      v forcats 0.5.0  
  
227 ## -- Conflicts ----- tidyverse_conflicts() --  
228 ## x dplyr::collapse() masks glue::collapse()  
229 ## x dplyr::filter()   masks stats::filter()  
230 ## x dplyr::lag()      masks stats::lag()
```

## 231 2.1 Basic iteration with map

232 Iteration in base R is commonly done with `for` and `while` loops. There is no readymade  
 233 alternative to `while` loops in the tidyverse. However, the functionality of `for` loops is  
 234 spread over the `map` family of functions.

235 `purrr` functions are *functionals*, i.e., functions that take another function as an argument.  
 236 The closest equivalent in R is the `*apply` family of functions: `apply`, `lapply`, `vapply`  
 237 and so on.

238 A good reason to use `purrr` functions instead of base R functions is their consistent and  
 239 clear naming, which always indicates how they should be used. This is explained in the  
 240 examples below.

241 These reasons, as well as how `map` is different from `for` and `lapply` are best explained  
 242 in the Advanced R book.

### 243 2.1.1 map basic use

244 `map` works on any list-like object, which includes vectors, and always returns a list. `map`  
 245 takes two arguments, the object on which to operate, and the function to apply to each  
 246 element.

```

247 # get the square root of each integer 1 - 10
248 some_numbers = 1:10
249 map(some_numbers, sqrt)
250 ## [[1]]
251 ## [1] 1
252 ##
253 ## [[2]]
254 ## [1] 1.414214
255 ##
256 ## [[3]]
257 ## [1] 1.732051
258 ##
259 ## [[4]]
260 ## [1] 2
261 ##
262 ## [[5]]
263 ## [1] 2.236068
264 ##
265 ## [[6]]
266 ## [1] 2.44949
267 ##
268 ## [[7]]
269 ## [1] 2.645751
270 ##
  
```

```

268 ## [[8]]
269 ## [1] 2.828427
270 ##
271 ## [[9]]
272 ## [1] 3
273 ##
274 ## [[10]]
275 ## [1] 3.162278

```

### 2.1.2 map variants returning vectors

Though map always returns a list, it has variants named `map_*` where the suffix indicates the return type. `map_chr`, `map_dbl`, `map_int`, and `map_lgl` return character, double (numeric), integer, and logical vectors.

```

# use map_dbl to get a vector of square roots
some_numbers = 1:10
map_dbl(some_numbers, sqrt)

280 ## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
281 ## [9] 3.000000 3.162278

# map_chr will convert the output to a character
map_chr(some_numbers, sqrt)

282 ## [1] "1.000000" "1.414214" "1.732051" "2.000000" "2.236068" "2.449490"
283 ## [7] "2.645751" "2.828427" "3.000000" "3.162278"

# map_int will NOT round the output to an integer

# map_lgl returns TRUE/FALSE values
some_numbers = c(NA, 1:3, NA, NaN, Inf, -Inf)
map_lgl(some_numbers, is.na)

284 ## [1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE

```

### Integrating map and tidyr::nest

The example show how each map variant can be used. This integrates `tidyr::nest` with `map`, and the two are especially complementary.

```

# nest mtcars into a list of dataframes based on number of cylinders
some_data = as_tibble(mtcars, rownames = "car_name") %>%
  group_by(cyl) %>%
  nest()

# get the number of rows per dataframe
# the mean mileage
# and the first car

```

```
some_data = some_data %>%
  mutate(n_rows = map_int(data, nrow),
         mean_mpg = map_dbl(data, ~mean(.$mpg)),
         first_car = map_chr(data, ~first(.$car_name)))
```

```
some_data
```

```
288 ## # A tibble: 3 x 5
289 ## # Groups:   cyl [3]
290 ##   cyl data                n_rows mean_mpg first_car
291 ##   <dbl> <list>              <int>   <dbl> <chr>
292 ## 1     6 <tibble [7 x 11]>         7     19.7 Mazda RX4
293 ## 2     4 <tibble [11 x 11]>      11     26.7 Datsun 710
294 ## 3     8 <tibble [14 x 11]>     14     15.1 Hornet Sportabout
```

map accepts multiple functions that are applied in sequence to the input list-like object, but this is confusing to the reader and ill advised.

### 2.1.3 map variants returning dataframes

map\_df returns data frames, and by default binds dataframes by rows, while map\_dfr does this explicitly, and map\_dfc does returns a dataframe bound by column.

```
# split mtcars into 3 dataframes, one per cylinder number
some_list = split(mtcars, mtcars$cyl)
```

```
# get the first two rows of each dataframe
map_df(some_list, head, n = 2)
```

```
300 ##   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
301 ## 1 22.8  4 108.0  93 3.85 2.320 18.61  1  1   4    1
302 ## 2 24.4  4 146.7  62 3.69 3.190 20.00  1  0   4    2
303 ## 3 21.0  6 160.0 110 3.90 2.620 16.46  0  1   4    4
304 ## 4 21.0  6 160.0 110 3.90 2.875 17.02  0  1   4    4
305 ## 5 18.7  8 360.0 175 3.15 3.440 17.02  0  0   3    2
306 ## 6 14.3  8 360.0 245 3.21 3.570 15.84  0  0   3    4
```

map accepts arguments to the function being mapped, such as in the example above, where head() accepts the argument n = 2.

map\_dfr behaves the same as map\_df.

```
# the same as above but with a pipe
some_list %>%
  map_dfr(head, n = 2)
```

```
310 ##   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
311 ## 1 22.8  4 108.0  93 3.85 2.320 18.61  1  1   4    1
312 ## 2 24.4  4 146.7  62 3.69 3.190 20.00  1  0   4    2
313 ## 3 21.0  6 160.0 110 3.90 2.620 16.46  0  1   4    4
```

```

314 ## 4 21.0    6 160.0 110 3.90 2.875 17.02  0  1    4    4
315 ## 5 18.7    8 360.0 175 3.15 3.440 17.02  0  0    3    2
316 ## 6 14.3    8 360.0 245 3.21 3.570 15.84  0  0    3    4

```

317 `map_dfc` binds the resulting 3 data frames of two rows each by column, and automatically  
 318 repairs the column names, adding a suffix to each duplicate.

```

some_list %>%
  map_dfc(head, n = 2)

319 ##      mpg cyl  disp hp drat   wt  qsec vs am gear carb mpg1 cyl1 disp1 hp1 drat1
320 ## 1 22.8   4 108.0 93 3.85 2.32 18.61 1  1   4   1  21   6  160 110   3.9
321 ## 2 24.4   4 146.7 62 3.69 3.19 20.00 1  0   4   2  21   6  160 110   3.9
322 ##      wt1 qsec1 vs1 am1 gear1 carb1 mpg2 cyl2 disp2 hp2 drat2 wt2 qsec2 vs2 am2
323 ## 1 2.620 16.46  0  1     4     4 18.7   8  360 175  3.15 3.44 17.02  0  0
324 ## 2 2.875 17.02  0  1     4     4 14.3   8  360 245  3.21 3.57 15.84  0  0
325 ##      gear2 carb2
326 ## 1         3     2
327 ## 2         3     4

```

## 328 2.1.4 Selective mapping

329 • `map_at` and `map_if`

## 330 2.2 More map variants

### 331 2.2.1 `map2`

332 `imap` here

### 333 2.2.2 `pmap`

### 334 2.2.3 `walk`

335 `walk2` and `pwalk`

## 336 2.3 Modification in place

337 `modify`

## 338 **2.4 Working with lists**

### 339 **2.4.1 Filtering lists**

### 340 **2.4.2 Summarising lists**

### 341 **2.4.3 Reduction and accumulation**

### 342 **2.4.4 Miscellaneous operation**