Converting hand-smoothed norms tables into print-format output

Overview

The norms development process (i.e., the process of creating raw-to-norm-score[1] lookup tables) has three stages:

1. Model the raw-to-norm-score relationship using `cNORM` (or custom R code) and create basic lookup tables as output.
2. Apply manual smoothing to the basic lookup tables, as needed, to prepare the norms for clinical use.
3. Convert the modified basic lookup tables into print-format lookup tables (or, alternatively, into digital-format lookup tables).

Because Stage 2 is a manual modification of `cNORM` output, Stage 3 cannot be reintegrated into the `cNORM` workflow. Stage 3 must be handled separately, and that is the purpose of this R script. The script takes as input the modified basic lookup tables and transforms them into print-format tables. The script is a template that can handle multiple tests or subtests in a batch process. The script is set up for age-stratified norms, but can be modified for grade-based norms.

The documentation refers frequently to the lookup relationship between raw and norm scores. It's important to keep in mind that this relationship is is *many-to-one*. That is, each raw score value maps onto one and only one norm score value, but each norm score value *may* map on to more than one raw score value. This many-to-one relationship is preserved through the Stage 3 transformations imposed by this script, as the commented snippets below explain.

Essentially, the Stage 3 process transforms two structures in the input tables:

1. The hierarchical organization of the data;
2. The directional flow of the tables from lookup input to lookup output (i.e., the "lookup direction").

The input tables embody a data hierarchy in which age groups are a subordinate category of tests. Thus, there is one table per test, and each of these tables holds the lookup columns for all age groups. In this documentation, we use the shorthand expressions `test>>age` or `ta` to label the input data hierarchy.

The input tables have a left-to-right lookup direction. That is, one looks up raw scores in a column at the left margin of the table, and reads to the right to find the column with the associated standard scores for a particular age group.

Below is the head of a typical input table, showing the `test>>age` data hierarchy and the left-to-right lookup direction.

raw	5.0-5.3	5.4-5.7	5.8-5.11	6.0-6.5	6.6-6.11	7.0-7.5	7.6-7.11	8.0-8.5
0	66	62	58	55	50	47	44	40
1	70	66	62	58	53	50	47	44
2	74	70	66	61	57	53	49	46
3	77	74	70	65	60	56	53	49

[1] In this general overview we refer to "norm scores", rather than specific types of norm scores, such as T scores or IQ-type standard scores. Once the discussion turns to the specific input files used by this script, we will refer to standard scores.

4	80	77	73	68	63	58	55	51
5	83	80	75	70	65	60	56	53
6	86	82	77	72	67	62	58	55
7	89	84	79	74	68	64	60	56
8	91	86	81	76	70	65	61	57
9	93	88	83	77	72	67	62	59

Note that the column names are the age range (in `years.months` format) of each age group. Also, reading left-to-right along the rows, each raw score maps onto only one standard score per age group, reflecting the many-to-one relationship between raw and standard scores.

Creating the print-format lookup tables requires an inversion of both the data hierarchy and and the lookup direction. In the print-format tables, tests are a subordinate category of age groups. That is, there is one table per age group, and each of these tables holds the lookup columns for all tests. We use the shorthand expressions `age>>test` or `at` to label the print-format data hierarchy.

The print-format tables have a right-to-left lookup direction. That is, one looks up raw scores for a particular test in one of the the rightward columns, and reads to the left to find the associated standard score in a column on the left margin.

Below is the head of a typical print-format table, showing the `age>>test` data hierarchy and the right-to-left lookup direction.

perc	ss	LSK-E	LSW-E	RHY-E	RLN-E	SEG-E	SPW-E
98	130	27-33	25-38	21-30	73-120	21-25	20-32
97	129	-	24	-	72	20	19
97	128	26	-	20	71	-	-
96	127	-	23	-	70	-	-
96	126	-	-	19	68-69	19	18
95	125	25	22	-	67	-	-
95	124	-	-	18	66	-	-
94	123	24	21	17	65	18	17
93	122	-	-	-	64	-	-
92	121	23	20	16	63	-	-

Note that the column names are now acronyms, one for each test. Some cells within these columns hold a range of raw scores, rather than a single score. This embodies the many-to-one relationship between raw and standard scores, in which each standard score *may* map onto more than one raw score.

To track the transformation of the data hierarchy explicitly in the code, we name objects with the `ta` suffix when they express the input (`test>>age`) hierarchy. Similarly, we use the suffix `at` to name objects that embody the print-format (`age>>test`) hierarchy. The suffix `flat` is used to name objects that lack hierarchical structure.

Executable Code

```
suppressMessages(library(here))
suppressMessages(library(tidyverse))
suppressMessages(library(writexl))

input_test_names <- c("lske", "lswe", "rhme", "rlne", "sege", "snwe")
output_test_names <- c("LSK-E", "LSW-E", "RHY-E", "RLN-E", "SEG-E", "SPW-E")
tod_form <- "TOD-E"
norm_type <- "age"
input_file_path <- "INPUT-FILES/PRINT-FORMAT-NORMS-TABLES/"
output_file_path <- "OUTPUT-FILES/PRINT-FORMAT-NORMS-TABLES/"

input_files_ta <- map(
  input_test_names,
  ~
  suppressMessages(read_csv(here(str_c(
  input_file_path, .x, "-", norm_type, ".csv"
))))
) %>%
  set_names(input_test_names)

perc_ss_cols <- suppressMessages(read_csv(here(str_c(
  input_file_path, "perc-ss-cols.csv"
))))

age_strat <- input_files_ta[[1]] %>%
  select(-raw) %>%
  names()

print_lookups_ta <- input_files_ta %>%
  map(~
        .x %>%
  pivot_longer(contains("-"), names_to = "age_strat", values_to = "ss") %>%
  arrange(age_strat) %>%
  group_by(age_strat) %>%
  complete(ss = 40:130) %>%
  group_by(age_strat, ss) %>%
  filter(n() == 1 | n() > 1 & row_number()  %in% c(1, n())) %>%
  summarize(raw = str_c(raw, collapse = '--')) %>%
  mutate(across(raw, ~ case_when(is.na(.x) ~ '-', TRUE ~ .x))) %>%
  arrange(age_strat, desc(ss)) %>%
  pivot_wider(names_from = age_strat,
              values_from = raw) %>%
  filter(!is.na(ss)) %>%
  right_join(perc_ss_cols, by = "ss") %>%
  relocate(perc, .before = "ss")
) %>%
  set_names(input_test_names)

age_strat_cols_ta <-  print_lookups_ta %>%
  map( ~
```

```
        map(age_strat,
            ~
              .y %>%
              select(perc, ss,!!sym(.x)), .y = .x) %>%
        set_names(age_strat))

age_test_names_flat <- cross2(age_strat, input_test_names) %>%
  map_chr(str_c, collapse = "_")

age_test_cols_flat <- flatten(age_strat_cols_ta) %>%
  set_names(age_test_names_flat)

age_test_cols_at <- map(
  age_strat,
  ~
  keep(age_test_cols_flat, str_detect(names(age_test_cols_flat), .x))
)

print_lookups_at <- age_test_cols_at %>%
  map(
    ~
      .x %>%
      reduce(left_join, by = c("perc", "ss")) %>%
      rename_with(~ output_test_names, contains("-"))
  ) %>%
  set_names(age_strat)

write_xlsx(print_lookups_at,
           here(
             str_c(
               output_file_path, tod_form, "-print-lookup-tables-", norm_type, ".xlsx"
             ))
           )
```

Commented Snippets

I. Load packages, set up tokens, read input files Load packages for file path specification (`here`), data wrangling (`tidyverse`), and writing files in .xlsx format (`writexl`). Initialize tokens for test names, file paths, and other input parameters. Tokenization of these elements facilitates adaptation of the template to different projects with different input parameters. Project-specific values are entered only once, at the beginning of the script, thus reducing the chance of human input error.

```
suppressMessages(library(here))
suppressMessages(library(tidyverse))
suppressMessages(library(writexl))

input_test_names <- c("lske", "lswe", "rhme", "rlne", "sege", "snwe")
output_test_names <- c("LSK-E", "LSW-E", "RHY-E", "RLN-E", "SEG-E", "SPW-E")
tod_form <- "TOD-E"
norm_type <- "age"
input_file_path <- "INPUT-FILES/PRINT-FORMAT-NORMS-TABLES/"
output_file_path <- "OUTPUT-FILES/PRINT-FORMAT-NORMS-TABLES/"
```

Read input files into a list named `input_files_ta`. In this example, the script processes six tests from the

Tests of Dyslexia, Early (TOD-E) project[2]. There are six input files, one for each test, each embodying the `test>>age` data hierarchy. To read the files, we use `map()` to call `read_csv()` iteratively over the set of input files. The first argument to `map()` is the vector `input_test_names`, one of the tokens initialized upstream in the script. The second argument to `map()` is the function to be applied to iteratively to the input files, namely `read_csv()`. Within `map()`, the function call is set off with the formula notation `~`.

As `map()` iterates over the `input_test_names` vector, it passes one complete file path at a time to `read_csv()`. These file paths are strings, concatenated from tokens and quoted sub-strings with `str_c()`. `map()` returns a list, `input_files_ta`, holding the input files as six data frames. `set_names()` applies matching test-specific names to to the six data frames.

```
input_files_ta <- map(
  input_test_names,
  ~
  suppressMessages(read_csv(here(str_c(
  input_file_path, .x, "-", norm_type, ".csv"
))))
) %>%
  set_names(input_test_names)
```

The print-format lookup tables include two static columns, one holding all possible standard scores, and the other holding the percentile rank associated with each standard score. We use `read_csv()` to read these columns into a data frame `perc_ss_cols`.

To finalize the print-format tables, a vector containing the names of the age groups is required. These names exist in each of the input files, as column names. To obtain the required vector `age_strat`, we start with a single input file, which is extracted from the list `input_files_ta` with double-bracket `[[]]` subsetting. We call `select(-raw)` to drop the `raw` column, retaining only the age-group columns. We extract the names of these columns into a vector with `names()`.

```
perc_ss_cols <- suppressMessages(read_csv(here(str_c(
  input_file_path, "perc-ss-cols.csv"
))))

age_strat <- input_files_ta[[1]] %>%
  select(-raw) %>%
  names()
```

II. Impose print-style formatting on the input tables In the next snippet, we again use `map()` for iterative processing of multiple tables held in a list. Here, `map()` is used to apply a long pipeline of functions to the tables held in the list `input_files_ta`. This list is piped into `map()`, where it is represented by the `.x` token, which sits at the beginning of the pipeline of functions set off by `~`. `map()` thus applies the function chain iteratively to each element of `.x`, returning a list of transformed tables, `print_lookups_ta`. The suffix of this output list denotes that the tables at this stage retain the `test>>age` data hierarchy.

We begin by calling `pivot_longer()` to transform each input table to a long, multilevel format, in which a set of standard scores (one for each age group) is nested within each value of raw score. The first argument to `pivot_longer()`, which we specify with the tidyselect helper `contains("-")`, names the columns to be pivoted to long format (i.e., the columns in the input table that hold the standard scores for each age group). The second argument, `names_to = "age_strat"`, designates a new column whose rows, in long format, will hold the names of the pivoted columns. The third argument, `values_to = "ss"`, designates a new column whose rows, in long format, will hold the cell values (that is, the standard scores) contained in the pivoted columns.

[2]The six TOD-E tests are: Letter and Sound Knowledge (LSK-E), Letter and Sight Word Recognition (LSW-E), Ryming (RHY-E), Rapid Letter and Number Naming (RLN-E), Segmenting (SEG-E), Sounds and Pseudowords (SPW-E).

Here we are excluding the `raw` column from the pivot, meaning that in the long table, `raw` remains in the left-most column and becomes a Level 2 variable[3]. The rows of `raw` are expanded such that each raw score value has eight rows, one for each of the eight age groups whose names now appear in the `age_strat` column. The new columns, `age_strat` and `ss`, are now Level 1 variables, because they are nested within the Level 2 variable `raw`. We then use `arrange()` to sort the data by `age_strat`.

To visualize the transformation, here are a few rows from the long-format table:

raw	age_strat	ss
2	6.6-6.11	57
2	7.0-7.5	53
2	7.6-7.11	49
2	8.0-8.5	46
3	5.0-5.3	77
3	5.4-5.7	74
3	5.8-5.11	70
3	6.0-6.5	65

As noted above, each value of `raw` now has eight rows, one for each value of `age_strat`. Each value of `age_strat` is paired with its associated value of `ss`, taken from the matching column of the wide-format input table. The sequence of `age_strat` values repeats itself for each new value of `raw`, going down the table.

```
print_lookups_ta <- input_files_ta %>%
  map(~
        .x %>%
  pivot_longer(contains("-"), names_to = "age_strat", values_to = "ss") %>%
  arrange(age_strat) %>%
```

The current `ss` column is the antecedent of the left-ward standard-score lookup column of the print-format table. As such, the columns needs to include all possible values of `ss`, sorted descending, with no duplicates. To get it to this state, we call `tidyr::complete()`, which fills in missing values of a variable, adding a new row for each filled-in value. As an argument to `complete()`, we pass `ss = 40:130`, which adds rows for any values in the range of `40` to `130` that are missing from the current `ss` column. To ensure that values are filled in consecutively within each age group, we `group_by(age_strat)` before calling `complete()`.

```
  group_by(age_strat) %>%
  complete(ss = 40:130) %>%
```

As noted earlier, the lookup relationship between raw and standard scores is many-to-one, meaning that each value of `ss` may map onto multiple values of `raw`. This mapping is represented explicitly in the current data object, where consecutive rows may share the same `ss` value while being each associated with a different value of `raw`. Below are a set of rows from the current data object, showing the many-to-one mapping of five different values of `raw` to the single `ss` value of 130.

age_strat	ss	raw
5.0-5.3	119	NA
5.0-5.3	120	16

[3] Referring to variables as "Level 1", "Level 2", etc., is part of the nomenclature of multilevel modeling.

```
5.0-5.3     121    NA
5.0-5.3     122    NA
5.0-5.3     123    17
5.0-5.3     124    NA
5.0-5.3     125    NA
5.0-5.3     126    18
5.0-5.3     127    NA
5.0-5.3     128    NA
5.0-5.3     129    19
5.0-5.3     130    20
5.0-5.3     130    21
5.0-5.3     130    22
5.0-5.3     130    23
5.0-5.3     130    24
```

In the finalized print-format lookup tables, there is only a single row for each possible value of standard score. In the example above, where there are five rows mapping `ss = 130` onto five consecutive values of `raw`, we need to collapse multiple values of `raw` into a single range of raw scores. We can then replace the five rows above with a single row in which `ss` is `130` and `raw` is `20-24`.

In addition to the one-to-many mapping for `ss = 130`, the rows above include one-to-one mappings for other values of `ss` (e.g., `ss = 126` maps *only* to `raw = 18`). In the next code snippet, we use `dplyr::filter()` to collapse the table vertically and retain rows according to the following rules:

- **One-to-one** `raw` to `ss` mapping: retain that single row.
- **Many-to-one** `raw` to `ss` mapping: retain the *first* and *last* rows of the series, which contain, respectively, the values of `raw` that are the lower and upper bounds of the required raw-score range.

To execute this transformation, we first re-group the data object hierarchically (`group_by(age_strat, ss)`), so that ranges of `raw` can be assembled within each value of `ss`, and, in turn, within each `age_strat`. After grouping, the one-to-one mappings are single-row groups, and the many-to-one mappings are multi-row groups (in which each row has an identical value of `ss`).

`filter()` operates on these groups according to a complex predicate: `n() == 1 | n() > 1 & row_number() %in% c(1, n())`. The predicate makes use of `dplyr::n()`, which returns the number of rows in the groups defined by `group_by()`. The single-row groups are captured by the expression `n() == 1`.

To get the first and last rows of the multi-row groups, we use the expression `n() > 1 & row_number() %in% c(1, n()))`. The LHS side of this expression (`n() > 1`) captures all rows of the multi-row groups. The RHS of this expression (`row_number()  %in% c(1, n()`) uses `row_number()` to retain only the first and last rows of the multi-row groups. The expression captures only the two rows where `row_number() == 1` (the first row) or `row_number == n()` (i.e, the row number is equal to the total number of rows in the group, which is also the row number of the *last* row of the group). Rather than specify these latter two sub-predicates separately, we can combine them by using `%in%` to specify that `row_number()` is equal to *either* of the two elements of the vector `c(1, n())`.

After applying the `filter()` step, the set of rows depicted above appears as follows:

```
age_strat     ss   raw

5.0-5.3      119    NA
5.0-5.3      120    16
5.0-5.3      121    NA
5.0-5.3      122    NA
5.0-5.3      123    17
5.0-5.3      124    NA
5.0-5.3      125    NA
```

```
5.0-5.3      126     18
5.0-5.3      127     NA
5.0-5.3      128     NA
5.0-5.3      129     19
5.0-5.3      130     20
5.0-5.3      130     24
```

Note that the group of rows for `ss = 130`, which previously included five rows, now consists of only two: those containing the values of `raw` for the lower and upper bounds of the required raw score range (20-24). All other rows in the example are single row groups, representing one-to-one `raw` to `ss` mappings.

Because the data object is grouped by `ss`, we can call `dplyr::summarize()` to further collapse the table so there is a single row for each possible value of `ss`. `summarize()` operates within groups, and is used to aggregate the values of a variable, returning a new column containing a single summary value for each group.[4]

As an argument to `summarize()`, we pass the expression `raw = str_c(raw, collapse = '--')`, which returns the required values for both types of `raw` to `ss` mappings. Note that the RHS is wrapped in `str_c`, meaning that numbers will be formatted as strings in the new `raw` column. One-to-one mappings are represented in the current object as single rows, so the summarizing expression simply returns the "old" value of `raw` in that row. Many-to-one mappings are now represented by two-row groups, with the rows containing the lower and upper bounds of the required raw-score ranges. For these two-row groups, `str_c()` joins the two "old" values of `raw`, collapsing them into a single string with the separator `--`.[5]

Because the mapping of `raw` to `ss` is many-to-one, there may be values of `ss` that are not mapped onto any value of `raw`. In the rows shown above, for example, `ss = 119` is "unmapped" (i.e., `raw = NA` for `ss = 119`). To format the `raw` column properly for final output, we use `mutate(across(case_when()))` to recode `NA` to `-` in the `raw` column. We then use `arrange()` to sort the table ascending by `age_strat`, and descending by `ss`.

```
  group_by(age_strat, ss) %>%
  filter(n() == 1 | n() > 1 & row_number()  %in% c(1, n())) %>%
  summarize(raw = str_c(raw, collapse = '--')) %>%
  mutate(across(raw, ~ case_when(is.na(.x) ~ '-', TRUE ~ .x))) %>%
  arrange(age_strat, desc(ss)) %>%
```

Recall that the original data input to this process was a wide-format data frame (i.e., an element of the list `input_files_ta`). We pivoted this input to long format, so that we could transform it by applying `dplyr` functions on a row-wise basis. As a result, the current data object is a long-format, three-column table whose cell values are formatted as required for the print-format lookup table. The `ss` column thus contains all possible values of standard score in descending order (nested within each value of `age_strat`).

To advance the transformation toward the final output requirements, we call `pivot_wider()` to return the data object to wide format. We include two of the three existing columns in arguments to `pivot_wider()`: `names_from = age_strat`, to specify that names for the new columns in the wide table are to be drawn from the existing `age_strat` column; and `values_from = raw`, to indicate that cell values for the new columns are to be drawn from the existing `raw` column. `pivot_wider()` thus returns a wide table in which the existing `ss` column becomes the left-most column, the existing `age_strat` and `raw` columns are dropped, and the new rightward columns are named for the age groups and contain the raw-score lookup values corresponding to each standard score. Here is the head of the resulting table:

ss	5.0-5.3	5.4-5.7	5.8-5.11	6.0-6.5	6.6-6.11	7.0-7.5	7.6-7.11	8.0-8.5	8.6-9.3
130	20--32	21--32	23--32	26--32	28--32	30--32	32	-	-
129	19	-	-	25	-	-	31	32	-

[4] To avoid confusion, keep in mind that the "new" `raw` column summarizes values in the "old" `raw` column, which is dropped from the data object returned by `summarize()`.

[5] We use two dashes (`--`), instead of a single dash, to prevent the score ranges from being erroneously read as dates when the saved output is opened in MS Excel.

128	-	-	-	-	27	29	-	-	-
127	-	20	22	24	-	-	-	-	32
126	18	-	-	-	-	-	30	-	-
125	-	-	-	-	26	28	-	31	-

To complete this section of the code, `filter(!is.na(ss))` is used to drop extraneous rows that are missing standard scores. `right_join()` brings in a static column `perc` of percentile rank values corresponding to the standard score values, and that column is placed in the required output position with `relocate()`.

At this point, `map()` finishes iterating, and the list of input tables has been transformed into a list (`print_lookups_ta`) of print-format lookup tables. `set_names()` is then used to apply test-specific names to the lookup tables.

```
  pivot_wider(names_from = age_strat,
              values_from = raw) %>%
  filter(!is.na(ss)) %>%
  right_join(perc_ss_cols, by = "ss") %>%
  relocate(perc, .before = "ss")
) %>%
  set_names(input_test_names)
```

III. Flatten tables with `test>>age` hierarchy into list of age-specific columns with no hierarchy
The code in the preceding section accomplished one part of the Stage 3 transformation: changing the format of the raw score columns into that required for the print-format lookup tables. But the tables in `print_lookups_ta` still retain the `test>>age` hierarchy of the input tables.

Completing the transformation into print format involves two additional steps: flattening the `test>>age` hierarchy into a non-hierarchical flat structure, and then restructuring again to yield the `age>>test` data hierarchy required for final output.

Prior to flattening, the tables held in `print_lookups_ta` must undergo a decomposition process. Recall that those tables have the structure shown in the example above, where there are multiple right-ward columns holding the raw scores for each age group. Decomposition of these tables means transforming a single wide table into a list of narrower tables, each containing the left-ward `perc` and `ss` columns, as well as a single right-ward column with the raw scores for a single age group. Essentially, a single table is broken up into a list of three-column sub-tables, one for each age group. The data object is now a "list of lists", retaining the top-level structure of `print_lookups_ta`, which is a list of tables, one for each test. In the decomposed structure, each input table is transformed into a list of three-column tables, one for each age group. Thus, where `print_lookups_ta` was a list of six tables, the current data object is a list of six lists.

The code accomplishes this restructuring via nested `map()` calls. In the outer call, `map()` iterates over the list of tables `print_lookups_ta`. That outer `map()` call feeds one table at a time to the inner `map()` call, which iterates over `age_strat`, a vector holding the names of the age groups.

Isolating the inner `map()` call, the data input is one table from the `print_lookups_ta` list. That input is represented by the `.y` token. The input is piped into `select()`, which keeps three columns: `perc`, `ss`, and one column of raw scores, corresponding to the element of `age_strat` named in the current iteration of the inner `map()` call. That latter element is represented by the `.x` token. Because `.x` in this case is a quoted string, we need to unquote it by wrapping it in `!!sym()` in order it to pass it to `select()`. The `.y = .x` argument to the inner `map()` call ensures that the inputs are processed in the correct sequence. That is, the test-specific elements of `print_lookups_ta` are processed within the iterative structure that applies the elements of `age_strat`, not the other way around.

After the inner `map()` call iterates completely over `age_strat`, it returns a test-specific list of three-column lookup tables, one for each age group. The elements of this list are named for their respective age groups with `set_names(age_strat)`. Control is then passed back to the outer `map()` call, which feeds the next table from `print_lookups_ta` to the inner `map()` call for processing, and so on, until all tables from `print_lookups_ta` have been decomposed. The final output is a "list of lists", `age_strat_cols_ta`, containing a list for each test, each of which holds the age-group-specific three-column lookup tables for that test.

```
age_strat_cols_ta <-  print_lookups_ta %>%
  map( ~
         map(age_strat,
             ~
               .y %>%
               select(perc, ss,!!sym(.x)), .y = .x) %>%
         set_names(age_strat))
```

Before applying the `age>>test` data hierarchy required for the print-format lookup tables, the existing `test>>age` hierarchy must be removed. Essentially this involves collapsing the "list of lists" structure into a single list, a process known as flattening. The resulting flattened list contains 54 age-group-specific, three-column lookup tables[6]. Naming is critical here, to preserve the association of each list element with its correct test and age group. Accordingly, we use `purrr::cross2()` to create all possible combinations of the `input_test_names` and `age_strat` vectors. `cross2()` returns a list, not the required vector of names (strings). To reformat the list as a character vector, we call `map_chr()`, passing `str_c()` to join the test-plus-age-group combinations into single strings. The resulting vector of names is `age_test_names_flat`.

We then apply `purrr::flatten()` to the list-of-lists `age_strat_cols_ta`, which removes one level of list structure, along with the `test>>age` data hierchy. The result is the 54-element list `age_test_cols_flat`, with the `flat` suffix signaling the removal of the `test>>age` structure. We use `set_names(age_test_names_flat)` to preserve identification of the 54 elements.

```
age_test_names_flat <- cross2(age_strat, input_test_names) %>%
  map_chr(str_c, collapse = "_")

age_test_cols_flat <- flatten(age_strat_cols_ta) %>%
  set_names(age_test_names_flat)
```

IV. Impose `age>>test` hierarchy, reduce list elements to age-group specific lookup tables, write output to .xlsx Once all 54 age-group specific lookup tables are held in a single flattened list, it is a straightforward process to rebuild the list-of-lists structure with the `age>>test` data hierarchy. Earlier in the narrative, we characterized this hierarchy as one in which tests are a subordinate category of age groups. Analagously, we can think of these category relationships in terms of subsets. Thus, in the input `test>>age` hierarchy, six subsets are present, one for each test. Each of these subsets holds nine columns that express the raw-to-standard-score lookup relationships for the nine age groups, for that particular test. To build the `age>>test` data hierarchy into a list-of-lists structure, we need to invert the subsetting structure and create subsets for the nine age groups, each of which holds the raw-to-standard-score lookup relationships for six tests.

This new subsetting structure will be imposed on the flattened, 54-element list `age_test_cols_flat`. The resulting list-of-lists is named `age_test_cols_at`, with the `at` suffix signaling that it embodies the `age>>test` data hierarchy.

We use `purrr::keep()` to subset a list. The first argument to `keep()` is the source list for drawing subsets, in this instance the flattened list `age_test_cols_flat`. The second argument is a predicate function which is applied to the elements of `age_test_cols_flat`. `keep()` retains only those elements of `age_test_cols_flat` for which the predicate evaluates to `TRUE`.

In this instance, the predicate function is `str_detect(names(age_test_cols_flat), .x)`, which returns `TRUE` when the name of an element in `age_test_cols_flat` contains `.x`. In the current snippet, we call `map()` to apply `keep()` iteratively over the `age_strat` vector. Thus, in each iteration, we retain only the elements of `age_test_cols_flat` that correspond to a single age group. This creates the subsetting structure

[6] Fifty-four elements result from the crossing of six tests with nine age groups.

required for output, because each age-group specific subset contains six three-column lookup tables, one for each test. In this way, `map()` returns a list-of-lists, `age_test_cols_at`, which has the required `age>>test` data hierarchy.

```
age_test_cols_at <- map(
  age_strat,
  ~
  keep(age_test_cols_flat, str_detect(names(age_test_cols_flat), .x))
)
```

The next step is to transform `age_test_cols_at` from a list-of-lists into a list of data frames suitable for writing out as tabbed .xlsx output. For this operation we `map()` a set of functions over the nine elements of `age_test_cols_at`, which are themselves age-group specific lists, each containing six three-column lookup tables, one for each test.

To each age-group specific list (represented by `.x` in the `map()` call), we apply `purrr::reduce()`, which in turn recursively applies `left_join()` to the three-column lookup tables contained in `.x`. `by = c("perc", "ss")` identifies the index columns for the join. The end result is that a subset of three-column lookup tables is reduced to a single lookup table, with `perc` and `ss` as the left-most columns, and with six right-ward columns, each containing the corresponding raw scores for one test.

We then call `rename_with()` to improve the names for the raw-score columns. We use a shortcut specification for `rename_with()`, in which the first argument is simply a vector of the new names (here `output_test_names`), and the second argument is a vector, of equal length, specifying the existing column names that are to be replaced (here we use the `tidyselect` helper `contains("-")` to identify only the raw-score columns, whose existing names contain a dash character).

`map()` returns `print_lookups_at`, a list of nine lookup tables, one for each age group. We use `set_names(age_strat)` to name the nine list elements appropriately.

```
print_lookups_at <- age_test_cols_at %>%
  map(
    ~
      .x %>%
      reduce(left_join, by = c("perc", "ss")) %>%
      rename_with(~ output_test_names, contains("-"))
  ) %>%
  set_names(age_strat)
```

The required format for the final output is a single .xlsx workbook, with nine named tabs, each containing the lookup table, in print format, for one age group. To create this output, we use `writexl::write_xlsx()`. This function takes the named list `print_lookups_at` and writes each table onto a separate .xlsx tab (within a single workbook), using the name of the list element for the tab name. The output file path for the resulting .xlsx workbook is specified using `here()` (to anchor the path to R project folder) and `str_c` (to concatenate quoted substrings and previously specified tokens into a single string that names the required file path).

```
write_xlsx(print_lookups_at,
           here(
             str_c(
               output_file_path, tod_form, "-print-lookup-tables-", norm_type, ".xlsx"
             ))
           )
```