Pairwise zipping in R

Roy Ratcliffe

Some languages and systems (Erlang, Redis) treat implied list pairs as a special data structure. Named R lists can transform to flat lists where the first element in each pairing retains the name, and where the value takes the second pairing position. Development iterations pursue three alternative approaches to the forward and reverse transformation operations, all using R's base function library only.

Introduction

Various computing applications apply the concept of a flattened and interlaced *pairwise list* or list of pairs. Erlang utilises a list of implied two-element tuple pairings for configuration terms, called property lists (Erlang 2022). Redis applies the identical concept within its streaming interface where an event corresponds to a flat list of interlaced keys and values. An example of such a list appears below, in R.

```
unnamed_pairs <- list("key1", "value1", "key2", "value2")
```

Operations on such lists exist:

- pairing up keys and values from lists of lists;
- reversing that process by flattening nested lists but by only one level; and
- finding the value (or values because uniqueness **not** guaranteed) of some given key.

Extending a pair list proves easy: just append the new key and value. Structuring and destructuring is another story yet an important one for situations where software components require a translation to and from implied pairs. Two complementary operations flatten and de-interlace pairs would be quite handy for such scenarios in R.

Iteratative development

The experimental steps below attempt to simplify structuring operations with list pairs. Keep it simple but make it reliable, correctly failing when pairing proves impossible.

First notion

Assume that *x* is a named list. The function below applies its names to its scalar elements by multivariate application of the `list` function. It does not simplify the result; the outcome is a nested list of lists. Nor does it apply names to the resulting super-list.

```
named_pairs <- function(x)
  mapply(list, names(x), x, SIMPLIFY = FALSE, USE.NAMES = FALSE)
```

Worth noting that R list names are character vectors of names. The list's name vector in R is `NULL` (an empty vector) if the list has no names. List names default to an empty string when lists mix named and unnamed elements. List elements may share the same name, not unique as if the list were a map. R automatically encapsulates that notable feature of pair lists.

Applying the function `unlist` to the result will flatten the list. No recursion needs to apply (calling `unlist(x, recursive = FALSE)` where `x` binds some pair list) because only two levels of nesting concern pairwise decomposition. Flattening to a vector using `unlist` also unifies their type; numbers become strings, logical `TRUE` becomes string `"TRUE"`. Sub-lists become `NULL` however.

Second notion, sequence generation

Generate indices and apply them to the list by sub-setting. Assumes that list positions translate to one-based contiguous integer indices—a safe assumption.

```
unzip <- function(x) {
  if (length(x) == 0L) return(list())
  i <- 2L * 1:length(x)
  named <- list()
  named[i - 1L] <- names(x)
  named[i] <- x
  named
}
```

To zip or unzip? That is the question! The previous function calls it "zip," the procedure for taking a flat list and building a named list. Not necessarily the best name. Imagination conjures other possibilities: flatten, unflatten, interlace, de-interlace. If zip, the reverse naturally becomes "unzip."

Sequences of indices

"Pairs to named list" using `seq` to generate the index sequences:

```
zip <- function(x) {
  if (length(x) == 0L) return(list())
```

```
  stopifnot("length must be even" = length(x) %% 2L == 0L)
  named <- x[seq(2L, length(x), 2L)]
  names(named) <- x[seq(1L, length(x), 2L)]
  named
}
```

Disadvantageously, the implementation presumes sequence $1 \ldots n$ gives the correct list indices, or more mathematically, the inclusive interval $[1, n]$ where $\{i | 1 \leq i \leq n, n \in \mathbb{Z}\}$. R makes this true in all if not almost all imaginable cases. Yet better if the implementation does not care about indexing details and therefore makes no assumptions about where they start and finish.

Still, a step in the right direction.

Final notion, matrix sub-setting, pure base

The final approach attempts a pure matrix sub-setting strategy. It only uses `base::seq_along` for indexing because it collapses predictably for empty lists, and fails for non-even lists, without requiring additional tests and conditional execution. Nor does it make assumptions about one-based indexing, an additional bonus.

```
flatten_named_list <- function(x) {
  i <- 2L * seq_along(x)
  unnamed <- list()
  unnamed[i] <- x
  unnamed[i - 1L] <- names(x)
  unnamed
}
```

Its complement:

```
unflatten_named_list <- function(x) {
  i <- 2L * seq_along(x)
  dim(i) <- c(length(x) / 2L, 2L)
  i <- i[, 1L]
  named <- x[i]
  names(named) <- x[i - 1L]
  named
}
```

Some redundancy exists. Expression `2L * seq_along(x)` unnecessarily multiplies the second half of the indices by two which the function subsequently ignores and then throws into the trash bin. Only the first column has interesting indices: those of the even-indexed values. Subtracting one from these gives the corresponding keys.

Wrong dimensions

Applying the dimensions by halving the length catches odd-length errors; the product of the dimension must match the length.

```
unflatten_named_list(list("a", 1, "b"))
```

```
Error in dim(i) <- c(length(x)/2L, 2L): dims [product 2] do not match the length of object [3]
```

Failure is the correct response.

Conclusions

- Basic matrix-oriented R sub-setting adds useful advantages: simple, collapses predicable, requires no conditionals. Matrix means a vector with added dimensions.
- The function names have developed during development: unzip and zip; flatten and unflatten. The subject of the operation: a named list or an unnamed list.
- Note that keys are **not** unique. The pairwise list is *not* a map or dictionary.

References

Erlang. 2022. *STDLIB User's Guide*. Version 4.0.1. Stockholm, Sweden: Ericsson AB.