An Introduction to the Tidyverse

Chapter 3 (2)

Guani Wu

Stats 102A: Introduction to Computational Statistics with R

UCLA

Acknowledgements: Miles Chen and Michael Tsiang

(This chapter is largely drawn from Hadley Wickham's Advanced R, 2019)

1. Data Manipulation with `dplyr`
2. Relational Data with dplyr
3. Functional Programming with `purrr`

Section 1

Data Manipulation with dplyr

The Five Functions of dplyr

There are five main functions in the dplyr package that are useful for data manipulation.

The data is assumed to be tidy.

- **filter()** extracts rows (observations) by their values.
- **select()** extracts columns (variables) by their names.
- **arrange()** reorders rows.
- **summarize()** makes summarizing by groups easier.
- **mutate()** creates new variables from existing variables.

Common Syntax

Just like the `tidyr` functions, all of the `dplyr` functions have the same basic syntax:

1. The first argument is a tibble.
2. Subsequent arguments describe what to do with the tibble object (variable names can be used without quotations).
3. The output is a tibble (same class as the input).

This syntax makes the `dplyr` functions particularly well suited to using the pipe %>%, especially when manipulating data in multiple ways.

The filter() Function

The **filter()** function subsets rows of a data frame based on their values. The subsequent arguments are logical expressions that filter the data frame.

```
filter(diamonds, cut == "Ideal")
```

Using the pipe:

```
diamonds %>% filter(cut == "Ideal")
```

The equivalent command in base R:

```
diamonds[diamonds$cut == "Ideal",]
```

The filter() Function

The filter() function is particularly helpful when subsetting using complicated logical expressions.

```
filter(diamonds, price == 326 | price == 327)

filter(diamonds,  price %in% c(326, 327))
```

Adding arguments is similar to using the & relational operator.

```
filter(diamonds, cut == "Ideal", price > 10000, color == "J")
```

Missing Values

filter() only includes rows where the condition is TRUE; it excludes both FALSE and NA values. If you want to preserve missing values, ask for them explicitly:

```
df <- tibble(x = c(1, NA, 3))

filter(df, is.na(x) | x > 1)
```

```
## # A tibble: 2 x 1
##       x
##   <dbl>
## 1    NA
## 2     3
```

The select() Function

The **select()** function selects columns of a data frame based on their names. The subsequent arguments are the column (variable) names (with or without quotations) that we want to select.

```
select(diamonds, carat, price, color)
```

Using the pipe:

```
diamonds %>%
  select(carat, price, color)
```

The equivalent command in base R:

```
diamonds[, c("carat", "price", "color")]
```

The select() Function

The `select()` function also accepts the `:` operator for consecutive columns and negative `-` signs for removing columns, which cannot be done with square brackets `[`.

Select all the columns from carat to clarity

```
select(diamonds, carat:clarity)
```

Select all columns except those from cut to clarity

```
diamonds %>%
  select(-(cut:clarity))
```

Select Helpers

The `select()` function also has helper functions that make selecting variables by name more convenient. The most common ones are:

- `starts_with()`: Select names that start with a prefix.
- `ends_with()`: Select names that end with a prefix.
- `contains()`: Select names that contain a specific string.
- `matches()`: Selects names that match a regular expression.
- `num_range()`: Selects names with a numeric range.

The full list can be found in the R Documentation for `?select_helpers`.

Select Helpers

Used inside a `select()` function, the helpers all input a string/character value (values in quotations).

```
diamonds %>%
  select(starts_with("c"))
```

```
diamonds %>%
  select(ends_with("e"))
```

```
diamonds %>%
  select(contains("l"), contains("th"))
```

```
diamonds %>%
  select(matches(".r."))
```

Select Helpers

```
diag_df <- data.frame(diag(5))
diag_df
```

```
##   X1 X2 X3 X4 X5
## 1  1  0  0  0  0
## 2  0  1  0  0  0
## 3  0  0  1  0  0
## 4  0  0  0  1  0
## 5  0  0  0  0  1
```

```
# Select the columns X1, X2, X3
diag_df %>%
  select(num_range("X", 1:3)) %>%
  names()
```

```
## [1] "X1" "X2" "X3"
```

The arrange() Function

The arrange() function sorts rows by values of a column in increasing order.

Within arrange(), the desc() function reverses the sort to decreasing order.

```
# Arrange the rows by increasing carat
diamonds %>%
  arrange(carat)
```

```
# Arrange the rows by decreasing carat
diamonds %>%
  arrange(desc(carat))
```

Notice that missing values are always sorted at the end.

The arrange() Function

The arrange() function can sort rows by more than one column.

```
# Arrange the rows by carat then color
diamonds %>%
  arrange(carat, color)
```

```
# Arrange the rows by color then carat
diamonds %>%
  arrange(color, carat)
```

Notice that the order of the variable names matter.

The mutate() Function

The mutate() function computes a new variable from the existing ones and adds it to the input data frame.

Suppose we want to compute the price per carat for each diamond in the diamonds data. We will call this new variable ppc.

```
diamonds %>%
  mutate(ppc = price / carat)
```

The transmute() Function

The mutate() function adds the new variable to the input data frame. To only return the new variable (and not the original data), we can use the transmute() function with the same syntax.

```
diamonds %>%
  transmute(ppc = price / carat, color)
```

The summarize() Function

The summarize() function, typically used with the group_by() function, makes summarizing by groups easier.

The group_by() function adds grouping data to a data frame. The summarize() function can then apply a summary function to summarize a specified variable by group.

Side Note: Because Hadley Wickham is from New Zealand, the British spelling summarise() also works.

The summarize() Function

For example, to compute the mean price split by color:

```
diamonds %>%
  group_by(color) %>%
  summarize(mean_price = mean(price), n_price = n())
```

Note that we assigned the name `mean_price` to the column of mean prices in the output object.

Caution: The `summarize()` function must output a single value (a vector of length 1).

The summarize() Function

To group by multiple variables, we can include more arguments in the group_by() function.

We can also include more arguments in summarize() to compute multiple summary functions.

```
diamonds %>%
  group_by(color, cut) %>%
  summarize(mean_price = mean(price), n_price = n())
```

Note: The n() function computes the number of observations in each group. It can only be used inside summarize(), mutate(), and filter().

Missing Values

```
library(nycflights13)
```

```
head(flights %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay)))
```

```
## `summarise()` has grouped output by 'year', 'month'. You ca
```

```
## # A tibble: 6 x 4
## # Groups:   year, month [1]
##    year month   day  mean
##   <int> <int> <int> <dbl>
## 1  2013     1     1    NA
## 2  2013     1     2    NA
## 3  2013     1     3    NA
## 4  2013     1     4    NA
## 5  2013     1     5    NA
## 6  2013     1     6    NA
```

Missing Values (Cont.)

Fortunately, all aggregation functions have an `na.rm` argument which removes the missing values prior to computation:

```
flights %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay, na.rm = TRUE))
```

In this case, where missing values represent cancelled flights, we could also tackle the problem by first removing the cancelled flights.

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))
```

Useful Summary Functions

Just using means, counts, and sum can get you a long way, but R provides many other useful summary functions:

- Measures of location: `mean(x)`, `median(x)`
- Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`
- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`
- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`
- Counts: `n()`, `n_distinct(x)`
- Counts and proportions of logical values: `sum(x > 10)`, `mean(y < 0)`.

Useful Summary Functions (Cont.)

Why is distance to some destinations more variable than to others?

```
not_cancelled %>%
  group_by(dest) %>%
  summarise(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))
```

```
## # A tibble: 104 x 2
##    dest  distance_sd
##    <chr>       <dbl>
##  1 EGE         10.5
##  2 SAN         10.4
##  3 SFO         10.2
##  4 HNL         10.0
##  5 SEA          9.98
##  6 LAS          9.91
##  7 PDX          9.87
##  8 PHX          9.86
```

Useful Summary Functions (Cont.)

How many flights left before 5am?

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(n_early = sum(dep_time < 500))
```

What proportion of flights are delayed by more than an hour?

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(hour_prop = mean(arr_delay > 60))
```

Grouped Mutates and Filters

Find all groups bigger than a thresholds:

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
```

Standardise to compute per group metrics:

```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay),
         sum_delay = sum(arr_delay)) %>%
  select(dest, arr_delay, prop_delay, sum_delay)
```

A grouped filter is a grouped mutate followed by an ungrouped filter.

Section 2

Relational Data with dplyr

Relational Data

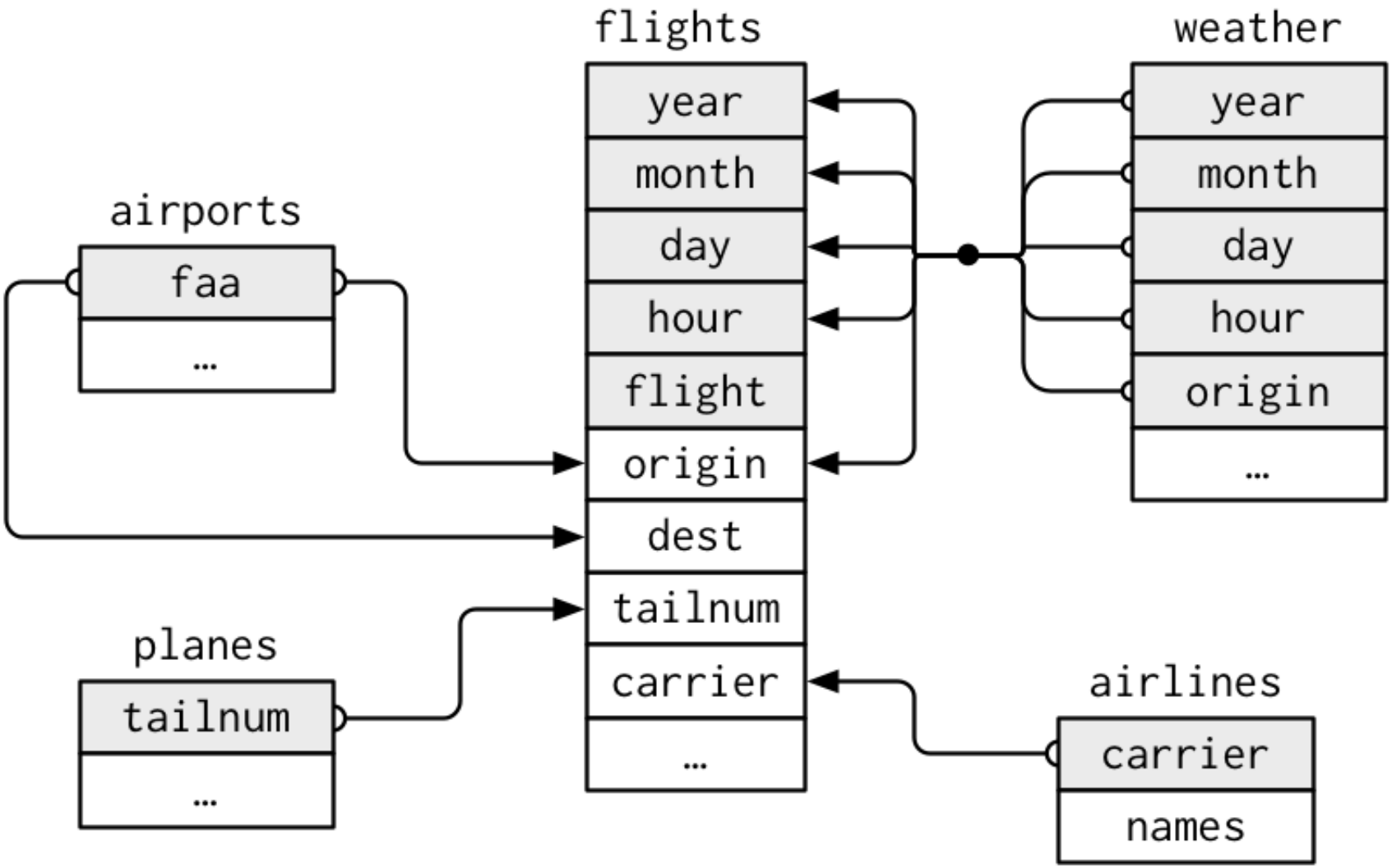
A collection of data tables with pre-defined relationships between them.

Relations are always defined between a pair of tables.

To work with relational data you need verbs that work with pairs of tables:

- **Mutating joins**, which add new variables to one data frame from matching observations in another.
- **Filtering joins**, which filter observations from one data frame based on whether or not they match an observation in the other table.
- **Set operations**, which treat observations as if they were set elements.

Entity Relationship Diagram (ERD)



Keys

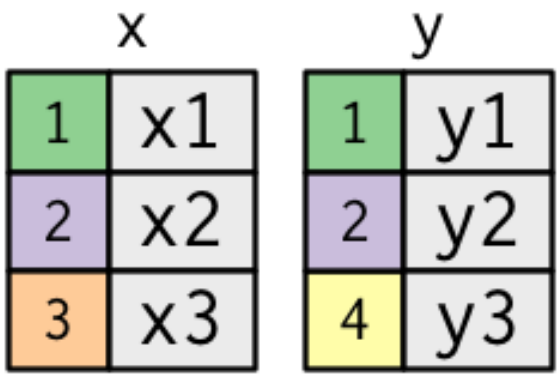
The variables used to connect each pair of tables are called keys. A key is a variable (or set of variables) that uniquely identifies an observation.

There are two types of keys:

- A **primary key** uniquely identifies an observation in its own table. For example, planes$tailnum is a primary key because it uniquely identifies each plane in the planes table.
- A **foreign key** uniquely identifies an observation in another table. For example, flights$tailnum is a foreign key because it appears in the flights table where it matches each flight to a unique plane.

A primary key and the corresponding foreign key in another table form a relation. Relations are typically one-to-many.

Understanding Joins

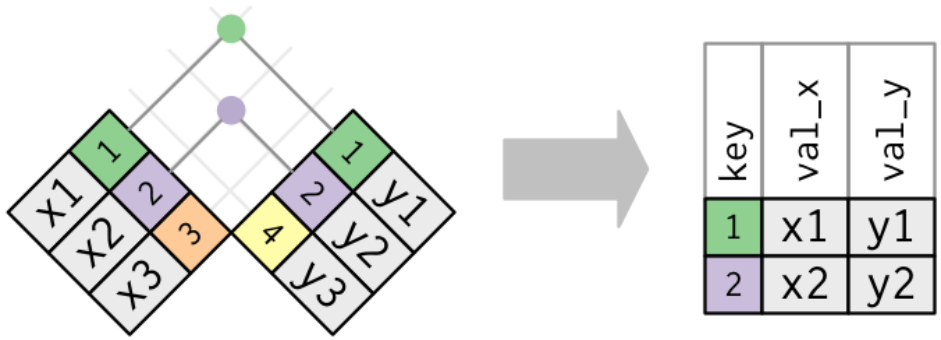



```
x <- tibble(
  key = c(1, 2, 3),
  val_x = c("x1", "x2", "x3")
)
y <- tibble(
  key = c(1, 2, 4),
  val_x = c("y1", "y2", "y3")
)
```

The coloured column represents the "key" variable: these are used to match the rows between the tables. The grey column represents the "value"

Inner Join

The simplest type of join is the inner join. An inner join matches pairs of observations whenever their keys are equal:




```
x %>%
  inner_join(y, by = "key")
```

```
## # A tibble: 2 x 3
##     key val_x.x val_x.y
##   <dbl> <chr>   <chr>
## 1     1 x1      y1
## 2     2 x2      y2
```

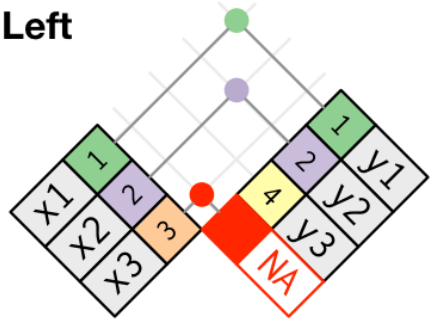
Outer Joins

An outer join keeps observations that appear in at least one of the tables. There are three types of outer joins:

- A **left join** keeps all observations in x.
- A **right join** keeps all observations in y.
- A **full join** keeps all observations in x and y.

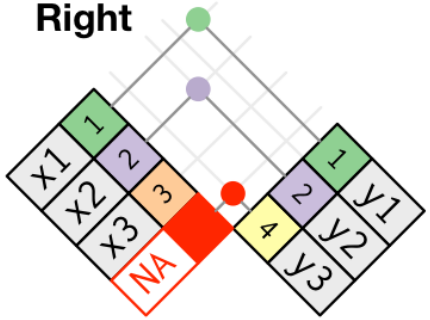
Outer joins (Cont.)

Left



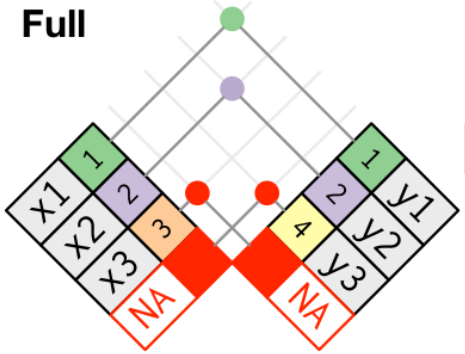
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

Right



key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

Full



key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

Outer Joins (Cont.)

```
x %>%
  left_join(y, by = "key")
```

```
## # A tibble: 3 x 3
##     key val_x.x val_x.y
##   <dbl> <chr>   <chr>
## 1     1 x1      y1
## 2     2 x2      y2
## 3     3 x3      NA
```

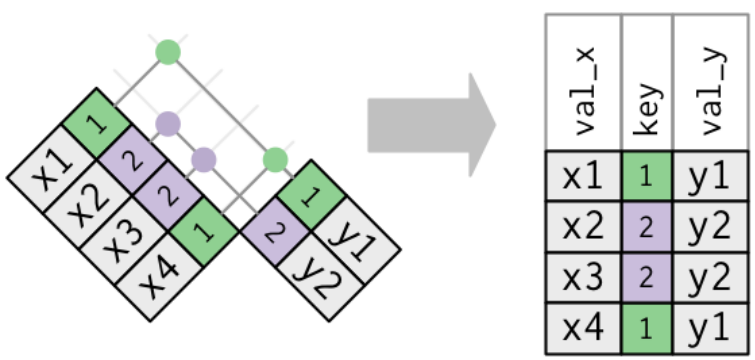
```
x %>%
  right_join(y, by = "key")
```

```
x %>%
  full_join(y, by = "key")
```

One-to-many Relationship

```
x <- tibble(
  key = c(1, 2, 2, 1),
  val_x = c("x1", "x2", "x3", "x4")
)
y <- tibble(
  key = c(1, 2),
  val_y = c("y1", "y2")
)
```

```
left_join(x, y, by = "key")
```



val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1

Combining Tidied Tibbles

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
```

```
left_join(tidy4a, tidy4b, by = c("country", "year"))
```

```
## # A tibble: 6 x 4
##   country     year   cases population
##   <chr>       <chr>  <int>      <int>
## 1 Afghanistan 1999     745   19987071
## 2 Brazil      1999   37737  172006362
## 3 China       1999  212258 1272915272
## 4 Afghanistan 2000    2666   20595360
## 5 Brazil      2000   80488  174504898
## 6 China       2000  213766 1280428583
```

Section 3

Functional Programming with purrr

The apply Family

One of the most widely used features of R is the **apply** family of functions. The apply family in base R consists of vectorized functions that minimize the need to use loops or repetitive code.

The main ones are:

- `apply()` is used to apply a function to the dimensions (the margins) of matrices, arrays, or data frames. The output is a vector or matrix/array.
- `lapply()` is used to apply a function to each component of a list. The output is a list.
- `sapply()` is a wrapper for `lapply()` that simplifies the output to a vector or matrix if possible.
- `tapply()` is used to apply a function to subsets of a vector (usually subsets based on levels of a factor).

The map Functions

The purrr package contains the **map** family of functions that also apply a function to each element of a vector (atomic or generic). The map family is meant to be more consistent in syntax and output so that it is easier to use and learn than the apply family.

There is a map function for each type of desired output:

- **map()** outputs a list.
- **map_lgl()** outputs a logical vector.
- **map_int()** outputs an integer vector.
- **map_dbl()** outputs a double vector.
- **map_chr()** outputs a character vector.
- **map_dfc()** outputs data frames (tibbles).

Each function takes a vector (i.e., atomic vector or list) as input, applies a function to each piece, and then returns a new vector that is the same length (and has the same names) as the input. The type of the output vector is determined by the suffix to the map function.

map Examples

For example:

```
trees %>% map(mean)
```

```
## $Girth
## [1] 13.24839
##
## $Height
## [1] 76
##
## $Volume
## [1] 30.17097
```

```
trees %>% map_dbl(mean)
```

```
##    Girth   Height   Volume
## 13.24839 76.00000 30.17097
```

map Examples

Specifying the wrong type of output will produce an error. This gives you more control and predictability of your code.

```
trees %>% map_dbl(range)
```

```
## Error: Result 1 must be a single double, not a double vecto
```

```
trees %>% map_dfc(range)
```

```
## # A tibble: 2 x 3
##   Girth Height Volume
##   <dbl>  <dbl>  <dbl>
## 1   8.3     63   10.2
## 2  20.6     87   77
```

Using Inline Functions

As with all functions that input function arguments (these types of functions are called **functionals**), functions can be defined inline.

```
trees %>% map_dbl(function(x){sum((x - mean(x))^2)})
```

```
##      Girth     Height     Volume
##   295.4374 1218.0000 8106.0839
```

Formula Shortcut with map Functions

The syntax for writing an inline function can be cumbersome, so the map functions allow for a shortcut using one-sided formula notation.

```
trees %>% map_dbl(~ sum((. - mean(.))^2))
```

```
##     Girth    Height    Volume
##  295.4374 1218.0000 8106.0839
```

The . in the formula refers to the current vector element, replacing the need to create a formal function with a dummy variable x.

Using this shortcut notation is a stylistic choice, but it is helpful to know it exists to understand and read code written by others.