

15 How to expand data frames and create complete combinations of values

i What will this tutorial cover?

In this tutorial you will find out how to create complete sets of values using **complete**, **expand** and **crossing**. I will try to show you how these functions differ and for which use cases they are useful.

💡 Who do I have to thank?

For this tutorial, I relied on [the official documentation](#). Kudos to the developers Hadley Wickham and Maximilian Girlich.

I've always found it difficult to distinguish the functions **complete**, **expand**, **nesting**, and **crossing** from another. In a sense, they do similar things. They find combinations of values in vectors or columns. I originally thought of writing a separate tutorial for each function, but digesting them all at once makes it easier to tell the difference between them. Let's take some time to look into these functions. And let's start with an overview of what they do:

Function	Explanation
complete	Turn implicit missing values into explicit values. The function completes combinations of values from columns that exist in a data frame and/or from vectors. complete is a shortcut version of expand .
expand	Creates a new tibble with all possible combinations of values from a data frame. The function is often used with joins.
expand with nesting	Create a new tibble with the unique combinations of column values that exist in a data frame.
crossing	Create a tibble with all combinations of values from vectors.

If you look at this table, you will notice a couple of things. First, **complete** makes an existing data frame longer by converting implicit values to existing values. This means that combinations of values that are not present in the data are created as new rows.

`expand`, in contrast, creates a new tibble. The tibble represents either all possible combinations of values or the unique combinations of values (with `nesting`). Suppose you have an incomplete tibble with months and years, in which the combination of the month “February” and the year 2013 is missing. You can use `expand` to create a complete set of years and months, including the combination of February and 2013. `crossing` works similarly to `expand`, except that it uses *vectors* to create combinations of values in a data frame. However, as we will see later, `expand` can also do this.

For this tutorial, we will use a made-up data set of running events. Suppose the following data frame shows the running races a runner has completed since 2010. The minutes show the time it took this person to complete the races. Let’s call her Anna.

```
running_races_anna<- tribble(
  ~year, ~race,           ~minutes,
  2010,   "half marathon", 110,
  2011,   "marathon",      230,
  2013,   "half marathon", 105,
  2016,   "10km",          50,
  2018,   "10km",          45,
  2018,   "half marathon", 100,
  2022,   "marathon",      210
)
```

You can see that some years are missing. Anna didn’t run a race in 2012. Also, she did not run a half marathon in 2016. In the next chapters, we will try to complete this data frame with the four functions. Let’s start with `complete`.

15.1 complete

Let’s assume Anna has only run 10Ks, half marathons, and marathons in recent years. Let’s further assume that she could have participated in every race each year. How many running races could she have participated in then?

A first approach could be to convert the implicit combinations into explicit combinations using `complete`. This essentially means nothing more than creating new rows representing the runs in which it did not participate. For these runs, `complete` sets the values of the `minutes` column to NA:

```
running_races_anna %>%
  complete(year, race)
```

```
# A tibble: 18 x 3
  year race      minutes
  <dbl> <chr>      <dbl>
1  2010 10km          NA
2  2010 half marathon 110
3  2010 marathon       NA
4  2011 10km          NA
5  2011 half marathon  NA
6  2011 marathon    230
7  2013 10km          NA
8  2013 half marathon 105
9  2013 marathon       NA
10 2016 10km          50
11 2016 half marathon  NA
12 2016 marathon       NA
13 2018 10km          45
14 2018 half marathon 100
15 2018 marathon       NA
16 2022 10km          NA
17 2022 half marathon  NA
18 2022 marathon    210
```

Looking at the number of rows she could have participated in 18 competitions. But could she? You might see that we are missing some years. There is no data from 2012 or 2014. This is because `complete` only works with the values that are already in the data. Since she never participated in a race in 2012, we don't see these races.

However, we can add these values if we use vectors instead of columns. Suppose we want to ensure that the data frame includes all years between 2010 and 2022 and all three running events that are already present in the data:

```
running_races_anna %>%
  complete(year = 2010:2022, race)
```

```
# A tibble: 39 x 3
  year race      minutes
  <dbl> <chr>      <dbl>
1  2010 10km          NA
2  2010 half marathon 110
3  2010 marathon       NA
4  2011 10km          NA
5  2011 half marathon  NA
```

```

6 2011 marathon      230
7 2012 10km          NA
8 2012 half marathon NA
9 2012 marathon      NA
10 2013 10km          NA
# ... with 29 more rows

```

All combinations of values that were already present in the data did not change. However, the code added rows that were not present. In other words, it added rows with years and races that were not present in the original data frame.

We could even go so far as to include new races to the data frame (i.e. ultra marathons):

```

running_races_anna %>%
  complete(year = 2010:2022, race = c(race, "ultra marathons"))

```

```

# A tibble: 52 x 3
   year race      minutes
  <dbl> <chr>      <dbl>
1  2010 10km          NA
2  2010 half marathon  110
3  2010 marathon      NA
4  2010 ultra marathons NA
5  2011 10km          NA
6  2011 half marathon  NA
7  2011 marathon      230
8  2011 ultra marathons NA
9  2012 10km          NA
10 2012 half marathon  NA
# ... with 42 more rows

```

Look how we created a vector that includes the races already present in the data plus ultra marathons.

15.2 expand

The `expand` function does something very similar. However, instead of adding new rows with the complete set of values, a new data frame is created only for the columns you specify in the function (compared to `complete`, where we kept the `minutes` column).

First, let's create a simple example. Let's create a complete combination of years and races from the existing data frame:

```
running_races_anna %>%  
  expand(year, race)
```

```
# A tibble: 18 x 2  
  year race  
  <dbl> <chr>  
1  2010 10km  
2  2010 half marathon  
3  2010 marathon  
4  2011 10km  
5  2011 half marathon  
6  2011 marathon  
7  2013 10km  
8  2013 half marathon  
9  2013 marathon  
10 2016 10km  
11 2016 half marathon  
12 2016 marathon  
13 2018 10km  
14 2018 half marathon  
15 2018 marathon  
16 2022 10km  
17 2022 half marathon  
18 2022 marathon
```

The result is a new data frame. You can see that the column `minutes` is missing. Similar to `complete` we can specify a vector instead of a column, for example to make sure that the data frame covers all years from 2010 to 2022:

```
running_races_anna %>%  
  expand(year = 2010:2022, race)
```

```
# A tibble: 39 x 2  
  year race  
  <int> <chr>  
1  2010 10km  
2  2010 half marathon  
3  2010 marathon
```

```

4 2011 10km
5 2011 half marathon
6 2011 marathon
7 2012 10km
8 2012 half marathon
9 2012 marathon
10 2013 10km
# ... with 29 more rows

```

A neat trick to complete the years is the `full_seq` function:

```

running_races_anna %>%
  expand(year = full_seq(year, 1), race)

```

```

# A tibble: 39 x 2
  year race
  <dbl> <chr>
1 2010 10km
2 2010 half marathon
3 2010 marathon
4 2011 10km
5 2011 half marathon
6 2011 marathon
7 2012 10km
8 2012 half marathon
9 2012 marathon
10 2013 10km
# ... with 29 more rows

```

In this case `full_seq` generated the complete set of years, starting with the lowest year in the data frame and ending with the highest year. The 1 indicates that the years should be incremented by 1 each time.

So we have a handle on all the combinations of years and races in our data frame. But we are missing the actual data, namely the minutes Anna took for these races. To add this data to the data frame, we combine `expand` with `full_join`:

```

(all_running_races_anna <- running_races_anna %>%
  expand(year = full_seq(year, 1), race) %>%
  full_join(running_races_anna, by = c("year", "race")))

```

```
# A tibble: 39 x 3
  year race      minutes
  <dbl> <chr>    <dbl>
1  2010 10km      NA
2  2010 half marathon 110
3  2010 marathon      NA
4  2011 10km      NA
5  2011 half marathon  NA
6  2011 marathon    230
7  2012 10km      NA
8  2012 half marathon  NA
9  2012 marathon      NA
10 2013 10km      NA
# ... with 29 more rows
```

This data frame includes all 39 races that Anna could have participated in between 2010 and 2022.

You may wonder why you should use `expand` instead of `complete` at all? The result is the same we got with `complete`. And the code it is more complicated.

If you take a look at the document, you will see that `complete` is actually a wrapper around `expand`. In other words, it is `expand` combined with `full_join` (see the [official code on GitHub](#)). Essentially, it is a shortcut for the more complicated code we just used. We will show this in the upcoming examples.

15.3 expand/complete with group_by

Now let's imagine Anna is running in a club with three other runners. Eva, John and Leonie.

```
running_races_club <- tribble(
  ~year, ~runner, ~race, ~minutes,
  2012, "Eva", "half marathon", 109,
  2013, "Eva", "marathon", 260,
  2022, "Eva", "half marathon", 120,
  2018, "John", "10km", 51,
  2019, "John", "10km", 49,
  2020, "John", "10km", 50,
  2019, "Leonie", "half marathon", 45,
  2020, "Leonie", "10km", 45,
  2021, "Leonie", "half marathon", 102,
  2022, "Leonie", "marathon", 220
```

```
)
```

Again, you want to find all races that each runner could have participated in since joining the club. If we used the same `expand` technique we just did, we will run into problems:

```
(all_running_races_club <- running_races_club %>%  
  expand(year = full_seq(year, 1), race, runner))
```

```
# A tibble: 99 x 3  
   year race      runner  
   <dbl> <chr>    <chr>  
1  2012 10km      Eva  
2  2012 10km      John  
3  2012 10km      Leonie  
4  2012 half marathon Eva  
5  2012 half marathon John  
6  2012 half marathon Leonie  
7  2012 marathon   Eva  
8  2012 marathon   John  
9  2012 marathon   Leonie  
10 2013 10km      Eva  
# ... with 89 more rows
```

Take John, for example:

```
all_running_races_club %>%  
  filter(runner == "John")
```

```
# A tibble: 33 x 3  
   year race      runner  
   <dbl> <chr>    <chr>  
1  2012 10km      John  
2  2012 half marathon John  
3  2012 marathon   John  
4  2013 10km      John  
5  2013 half marathon John  
6  2013 marathon   John  
7  2014 10km      John  
8  2014 half marathon John  
9  2014 marathon   John  
10 2015 10km      John
```



```
# ... with 23 more rows
```

He joined the club in 2019. However, the data frame shows missed races from 2012. This is because the data frame contains the races of Eva, who joined in 2012.

We can fix this problem by grouping the data frame by runners.

```
(all_running_races_club_correct <- running_races_club %>%  
  group_by(runner) %>%  
  expand(year = full_seq(year, 1), race = c("10km", "half marathon",  
                                           "marathon")) %>%  
  ungroup())
```

```
# A tibble: 54 x 3  
  runner year race  
  <chr>   <dbl> <chr>  
1 Eva     2012 10km  
2 Eva     2012 half marathon  
3 Eva     2012 marathon  
4 Eva     2013 10km  
5 Eva     2013 half marathon  
6 Eva     2013 marathon  
7 Eva     2014 10km  
8 Eva     2014 half marathon  
9 Eva     2014 marathon  
10 Eva    2015 10km  
# ... with 44 more rows
```

With `group_by` we expand the rows only within the runners. If you now take a look at the data, you will notice that John has no races before 2018, which is exactly what we want.

```
all_running_races_club_correct %>%  
  filter(runner == "John")
```

```
# A tibble: 9 x 3  
  runner year race  
  <chr>   <dbl> <chr>  
1 John   2018 10km  
2 John   2018 half marathon  
3 John   2018 marathon  
4 John   2019 10km
```

```

5 John    2019 half marathon
6 John    2019 marathon
7 John    2020 10km
8 John    2020 half marathon
9 John    2020 marathon

```

Yet, we still need the actual data of the three runners. We use `left_join` to add the running times to the expanded data frame:

```

(complete_running_races_club <- all_running_races_club_correct %>%
  left_join(running_races_club, by = c("year", "runner", "race")))

```

```

# A tibble: 54 x 4
  runner year race      minutes
  <chr>  <dbl> <chr>      <dbl>
1 Eva    2012 10km          NA
2 Eva    2012 half marathon  109
3 Eva    2012 marathon       NA
4 Eva    2013 10km          NA
5 Eva    2013 half marathon  NA
6 Eva    2013 marathon    260
7 Eva    2014 10km          NA
8 Eva    2014 half marathon  NA
9 Eva    2014 marathon       NA
10 Eva   2015 10km          NA
# ... with 44 more rows

```

Since we already know that `complete` is a shortcut for such an analysis, we can use it instead:

```

running_races_club %>%
  group_by(runner) %>%
  complete(year = full_seq(year, 1), race = c("10km", "half marathon",
                                             "marathon")) %>%
  ungroup()

```

```

# A tibble: 54 x 4
  runner year race      minutes
  <chr>  <dbl> <chr>      <dbl>
1 Eva    2012 10km          NA
2 Eva    2012 half marathon  109

```

```

3 Eva      2012 marathon      NA
4 Eva      2013 10km          NA
5 Eva      2013 half marathon NA
6 Eva      2013 marathon      260
7 Eva      2014 10km          NA
8 Eva      2014 half marathon NA
9 Eva      2014 marathon      NA
10 Eva     2015 10km          NA
# ... with 44 more rows

```

With this data we can do some interesting analysis. We could visualize the percentage of competitions in which each runner actually participated.

First, we need to find out how many races each runner has completed. To do this, we count the number of races that a runner has or has not completed:

```

(count_races <- complete_running_races_club %>%
  count(runner, race, missed_races = is.na(minutes)))

```

```

# A tibble: 14 x 4
  runner race      missed_races      n
  <chr> <chr>      <lgl>      <int>
1 Eva   10km      TRUE        11
2 Eva   half marathon FALSE        2
3 Eva   half marathon TRUE         9
4 Eva   marathon FALSE         1
5 Eva   marathon TRUE        10
6 John  10km      FALSE        3
7 John  half marathon TRUE         3
8 John  marathon  TRUE         3
9 Leonie 10km      FALSE         1
10 Leonie 10km      TRUE         3
11 Leonie half marathon FALSE        2
12 Leonie half marathon TRUE         2
13 Leonie marathon  FALSE         1
14 Leonie marathon  TRUE         3

```

We see that Eva has not completed a single 10-km run in the years she has been a member of the club, because there is a row missing where the `missed_races` column is set to `FALSE`.

Fortunately, we have learned that we can complete an existing data frame with `complete`. Let's do that:

```
count_races %>%
  complete(runner, race, missed_races, fill = list(n = 0))
```

```
# A tibble: 18 x 4
  runner race      missed_races    n
  <chr> <chr>      <lgl>      <int>
1 Eva   10km        FALSE      0
2 Eva   10km        TRUE       11
3 Eva   half marathon FALSE      2
4 Eva   half marathon TRUE       9
5 Eva   marathon   FALSE      1
6 Eva   marathon   TRUE      10
7 John  10km        FALSE      3
8 John  10km        TRUE       0
9 John  half marathon FALSE      0
10 John half marathon TRUE       3
11 John marathon   FALSE      0
12 John marathon   TRUE       3
13 Leonie 10km        FALSE      1
14 Leonie 10km        TRUE       3
15 Leonie half marathon FALSE      2
16 Leonie half marathon TRUE       2
17 Leonie marathon   FALSE      1
18 Leonie marathon   TRUE       3
```

The code has an interesting addition, the `fill` parameter. The parameter allows us turn NAs to actual values. Since we know that the missing rows represent the number of races that were or were not finish, we can be sure that they represent zero races. For Eva, for example, a row is missing for the 10km races in which she never participated.

Now that we have the complete count data of races per runner, we can calculate the percentage of races they participated in. To calculate the percentages, we must first put the data into a wide format and then create a column that represents the percentages:

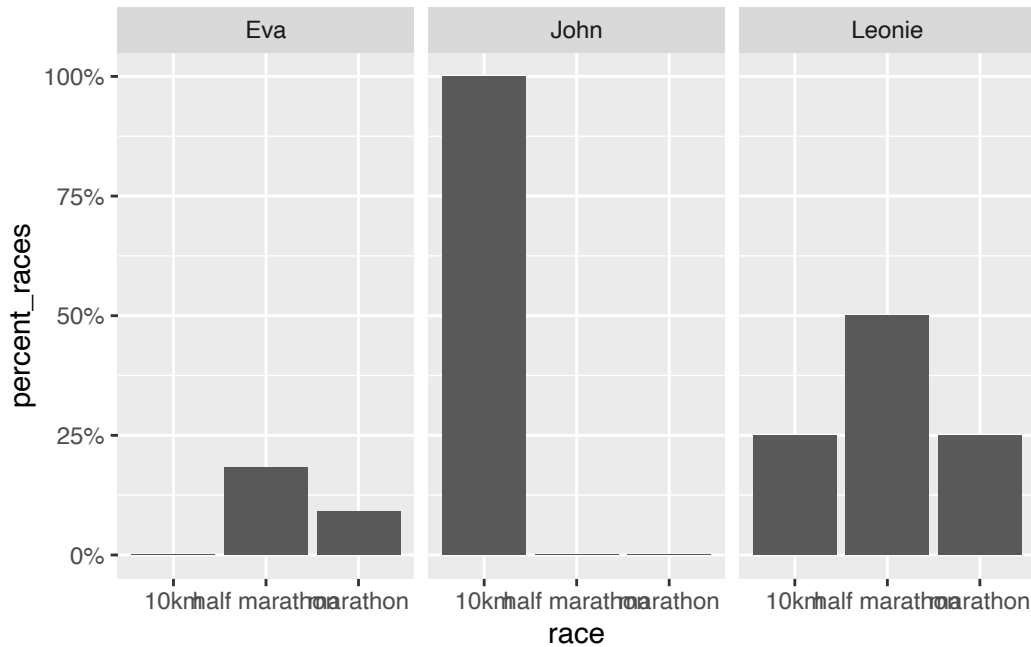
```
count_races %>%
  complete(runner, race, missed_races, fill = list(n = 0)) %>%
  pivot_wider(names_from = missed_races, values_from = n) %>%
  mutate(
    percent_races = (`FALSE` / (`TRUE` + `FALSE`)) * 100
  )
```

```
# A tibble: 9 x 5
  runner race      `FALSE` `TRUE` percent_races
  <chr>   <chr>      <int>  <int>      <dbl>
1 Eva    10km           0    11          0
2 Eva    half marathon  2     9        18.2
3 Eva    marathon       1    10         9.09
4 John   10km           3     0        100
5 John   half marathon  0     3          0
6 John   marathon       0     3          0
7 Leonie 10km           1     3         25
8 Leonie half marathon  2     2         50
9 Leonie marathon     1     3         25
```

Let's talk about Eva again. She participated in 0% of the 10k races and in 18.18% of the possible half marathons. Since she ran only 1 of 11 marathons, she participated in 9% of the marathons.

This is how it looks for all runners:

```
count_races %>%
  complete(runner, race, missed_races, fill = list(n = 0)) %>%
  pivot_wider(names_from = missed_races, values_from = n) %>%
  mutate(
    percent_races = (`FALSE` / (`TRUE` + `FALSE`)) * 100
  ) %>%
  ggplot(aes(x = race, y = percent_races)) +
  scale_y_continuous(labels = scales::label_percent(scale = 1)) +
  geom_col() +
  facet_wrap(vars(runner))
```



15.4 expand with nesting

So far, we have completed data frames for missing rows. Sometimes, however, we are interested in the unique combinations of values in a data frame. Suppose your running club has 540 members. You want to know in which competitions a runner has participated during her or his time in the club. This is basically the opposite of what we just did. Instead of finding all combinations of values we are looking for the unique combinations; in a given data frame!

To find these combinations we can combine `expand` with `nesting`:

```
running_races_club %>%
  expand(nesting(runner, race))
```

```
# A tibble: 6 x 2
  runner race
  <chr> <chr>
1 Eva   half marathon
2 Eva   marathon
3 John  10km
4 Leonie 10km
5 Leonie half marathon
```

6 Leonie marathon

Once again, you can see that Eva has never run a 10K. John has never run a half marathon or marathon. But we have to infer that information from the data frame. The data shows what happened, not what didn't happen. To find out which runs the runners have never participated in, we can combine the code with `anti_join`:

```
full_combinations_runners <- expand(running_races_club,
  runner, race = c("10km", "half marathon", "marathon"))

full_combinations_runners %>%
  anti_join(running_races_club, by = c("runner", "race"))
```



```
# A tibble: 3 x 2
  runner race
  <chr>   <chr>
1 Eva    10km
2 John   half marathon
3 John   marathon
```

The result of our analysis is now easier to process, as we no longer have to search for the known unknowns and get the desired results directly.

15.5 crossing

Let's talk about tennis. Suppose you want to create a data frame that shows all Grand Slams (Australian Open, French Open, Wimbledon, US Open) from 1905 to 2022 ([1905 was the first year all Grand Slams were held](#)). You don't have an existing data frame at hand, so you need to create one from scratch.

For these cases you need `crossing`. The difference from the other functions is that `crossing` does not need an existing data frame. We use vectors instead:

```
crossing(
  year = 1905:2022,
  major = c("Australien Open", "French Open",
            "Wimbledon", "US Open")
)
```

```
# A tibble: 472 x 2
  year major
  <int> <chr>
1  1905 Australien Open
2  1905 French Open
3  1905 US Open
4  1905 Wimbledon
5  1906 Australien Open
6  1906 French Open
7  1906 US Open
8  1906 Wimbledon
9  1907 Australien Open
10 1907 French Open
# ... with 462 more rows
```

This gives us a total of 472 Grand Slams.

Similarly, we could create a data frame representing the [World Marathon Majors](#), which started in 2006:

```
crossing(
  year = 2006:2022,
  races = c("Tokyo", "Boston", "Chicago",
            "London", "Berlin", "New York")
)
```

```
# A tibble: 102 x 2
  year races
  <int> <chr>
1  2006 Berlin
2  2006 Boston
3  2006 Chicago
4  2006 London
5  2006 New York
6  2006 Tokyo
7  2007 Berlin
8  2007 Boston
9  2007 Chicago
10 2007 London
# ... with 92 more rows
```

The data itself only gives us a complete set of combinations, by itself it is not very meaningful. `crossing` is usually a starting point for further analyses. Imagine if we had a data set with

all the world records set at these majors. We could join the world records to this data frame to determine the percentage of races in which a world record was set at the six majors.

i Summary

- `complete`, `expand` and `crossing` all create complete sets of combinations of values. `complete` and `expand` derive the complete set from values already present in a data frame or vectors, `crossing` from vectors only.
- `complete` is a wrapper around `expand`. It is basically `expand` in combination with `full_join`
- `complete` and `expand` can be used for grouped data frames to complete a set of combinations of values within groups only.
- `expand` and `crossing` create a new data frame, `complete` adds rows to an existing data frame
- `expand` in combination with `nesting` gives you the unique combinations of values in a data frame.