6 How to use extract to create multiple columns from one column

> **What will this tutorial cover?**
>
> In this tutorial you will learn how to extract a character column into multiple columns using `extract`. The `extract` function is basically the `separate` function with super powers and works with groups instead of separators.

> **Who do I have to thank?**
>
> I have to thank two people in particular for this tutorial. First, many thanks to Tom Henry who covered this topic in his fantastic YouTube video tidyverse tips & tricks. Thanks also to Mark Dulhunty who showed me in this post that `extract` is much more powerful than I initially thought. Thanks again to the team working on stringr, who taught me about the concept of non-grouping parentheses.

`extract` is the more powerful version of `separate`. The `separate` function allows you to split a character variable into multiple variables. Suppose we want to split the `variable` column in this dataset into two columns:

```
tibble(
  variable = c("a-b", "a-d", "b-c", "d-e")
) %>%
  separate(
    variable,
    into = c("a", "b"),
    sep = "-",
    remove = FALSE
  )
```

```
# A tibble: 4 x 3
  variable a     b
  <chr>    <chr> <chr>
```

```
1 a-b      a     b
2 a-d      a     d
3 b-c      b     c
4 d-e      d     e
```

This approach reaches its limits quite quickly. Especially if we don't have a clear separator to distinguish the columns we want to create. For these use cases we have `extract`.

The key difference between `separate` and `extract` is that `extract` works with groups within its regular expressions. We will see many examples of this in a minute. For now, let's just say that a group is indicated by two parentheses in regular expressions: `()`. We define groups in `extract` to tell the function which parts of a column should represent a new column.

6.1 How to extract a simple character column

A hyphen

Let's start with the simplest example, a column separated by a hyphen:

```
tibble(
  variable = c("a-b", "a-d", "b-c", "d-e")
) %>%
  extract(
    col = variable,
    into = c("a", "b"),
    regex = "([a-z])-([a-z])",
    remove = FALSE
  )
```

```
# A tibble: 4 x 3
  variable a     b
  <chr>    <chr> <chr>
1 a-b      a     b
2 a-d      a     d
3 b-c      b     c
4 d-e      d     e
```

`extract` takes a few arguments:

- `col` specifies the character column to be split into several columns.

- `into` specifies the name of the columns to be created
- `regex` defines the regular expression in which we capture the groups that will represent the new columns
- `remove` tells the function if the original column should be removed (by default `TRUE`)

The interesting thing about this code are the two groups: `([a-z])` and `([a-z])`. They are separated by a hyphen (`-`). Each captured group is converted into a new column. So instead of thinking of the separator in `separate` with `extract`, we think of groups.

A letter and a number

Let's try this concept on another example. In this data frame, we want to extract two columns from one column. The first one should be the first letter, the second one the number. In fact, in this example we don't even have a separator anymore:

```
tibble(
  variable = c("x1", "x2", "y1", "y2")
)
```

```
# A tibble: 4 x 1
  variable
  <chr>
1 x1
2 x2
3 y1
4 y2
```

This is how we would extract both columns with `extract`:

```
tibble(
  variable = c("x1", "x2", "y1", "y2")
) %>%
  extract(
    variable,
    into = c("letter", "number"),
    regex = "([xy])(\\d)",
    remove = FALSE
  )
```

```
# A tibble: 4 x 3
  variable letter number
  <chr>    <chr>  <chr>
1 x1       x      1
2 x2       x      2
3 y1       y      1
4 y2       y      2
```

Again, we have defined two groups for the two columns. The first group captures the letters `x` or `y` marked by square brackets (`([xy])`), the second group captures a single number (`(\\d)`).

First name and last name

Now suppose you want to extract the first and last names of famous people:

```
tibble(
  variable = c("David Jude Heyworth Law", "Elton Hercules John",
               "Angelina Jolie Voight", "Jennifer Shrader Lawrence")
) %>%
  extract(
    variable,
    into = c("first_name", "last_name"),
    regex = "(\\w+) .* (\\w+)",
    remove = FALSE
  )
```

```
# A tibble: 4 x 3
  variable                  first_name last_name
  <chr>                     <chr>      <chr>
1 David Jude Heyworth Law   David      Law
2 Elton Hercules John       Elton      John
3 Angelina Jolie Voight     Angelina   Voight
4 Jennifer Shrader Lawrence Jennifer   Lawrence
```

What we want to say with the regular expression is the following:

- The first group captures at least 1 letter (`(\\w+)`.
- The column is then followed by a space, and all characters in between are followed by another space: `.*`
- The last group again contains at least 1 letter: `(\w+)`

You can see that this also works for people who have more than one middle name. This is because the regular expression only captures the last word preceded by a space.

6.2 How to extract more complicated character column

Non-grouping parentheses

To extract columns that are more complicated and confusing, we need to learn the concept of **non-grouping parentheses**. Non-grouping parentheses define groups that are not captured. In other words, these groups are not converted into a new column. But they allow us to extract columns that have some inconsistencies.

Let's take this example: Here we want to create two columns separated by a `->` (this example could also be solved with `separate`, but we will get to more complicated examples in a moment):

```
tibble(
  variable = c("x -> 1",
               "y -> 2",
               "p-> 34")
) %>%
  extract(
    variable,
    into = c("letter", "number"),
    remove = FALSE,
    regex = "([a-z])(?: ?-> ?)(\\d+)?"
  )
```

```
# A tibble: 3 x 3
  variable letter number
  <chr>    <chr>  <chr>
1 x -> 1   x      1
2 y -> 2   y      2
3 p-> 34   p      34
```

The most important part here is this: `(?: ?-> ?)`. This is called a non-grouping parenthesis. A non-grouping parenthesis is defined by a group that starts with a question mark and a colon: `(?:)`. The advantage of this method is that we can solve column separation problems caused by messy or inconsistent variables. Let's see what this means in our next example.

Inconsistent separators

In this example, we want to extract columns from a variable that has an arrow as a separator (`->`). However, we might sometimes find two or more arrows in the character string or no arrow at all:

```
df <- tibble(
  variable = c("x ->-> 1",
               "y -> 2",
               "p-> 34",
               "f 4")
)

df %>%
  extract(
    variable,
    into = c("letter", "number"),
    remove = FALSE,
    regex = "([a-z]) ?(?:->){0,} ?(\\d+)?"
  )
```

```
# A tibble: 4 x 3
  variable letter number
  <chr>    <chr>  <chr>
1 x ->-> 1 x      1
2 y -> 2   y      2
3 p-> 34   p      34
4 f 4      f      4
```

Our non-grouping parenthesis looks for any number of arrows between our two new variables: `(?:->){0,}`. The `0` indicates that we can also find a character string that does not contain an arrow. Therefore, we are able to extract the last value that lacks the arrow: `f 4`. The comma `{0,}` indicates that the arrow can be repeated infinitely often.

Non-perfect pattern with unnecessary characters

Let us create an even more complicated example. Suppose that we expect to find some typos after the arrow `->aslkdfj`. This problem can also be solved with non-grouping parentheses:

```
df <- tibble(
  variable = c("x ->aslkdfj 1",
```

```
                "y-> 2",
                "p 34",
                "8")
)

df %>%
  extract(
    variable,
    into = c("letter", "number"),
    remove = FALSE,
    regex = "([a-z])? ?(?:->\\w*)? ?(\\d+)"
  )
```

```
# A tibble: 4 x 3
  variable      letter number
  <chr>         <chr>  <chr>
1 x ->aslkdfj 1 "x"    1
2 y-> 2         "y"    2
3 p 34          "p"    34
4 8             ""     8
```

A few things have been changed here. First, I added the regex `\\w*` to the non-grouping parenthesis. This regex searches for any number of letters. The asterisk `*` indicates that we expect zero to an infinite number of letters.

You can also see that we may not even find a letter for the first newly created column. Here the question mark `([a-z])?` indicates that the first letter is optional.

A hard example

Let's conclude this tutorial with a tricky example. Imagine that our column actually represents four different columns with the following structure:

- First column: A single number (e.g. `3`).
- Separator: A period (e.g. `.`)
- Second column: A number of any length (e.g. `10`).
- Separator: An equal sign with or without an enclosing space (e.g. `=`).
- Fourth column: Any number of letters (e.g. `AX`).
- Separator: A colon (e.g. `:`).
- Fourth column: A number (e.g. `40`)

Let's first look at how this might work, and then go through the regex in more detail:

```
tibble(
  value = c("3.10 = AX",
            "3.1345 = AX:?_40",
            "3.8983 =:$15",
            ".873 = PFS:4")) %>%
  extract(
    value,
    into = c("v0", "v2", "v3", "v4"),
    regex = "(\\d)?\\.(\\d+) ?= ?(?:(\\w+)?:?)?(?:[?_$]*)(\\d+)?",
    remove = FALSE
  )
```

```
# A tibble: 4 x 5
  value            v0    v2    v3    v4
  <chr>            <chr> <chr> <chr> <chr>
1 3.10 = AX        "3"   10    "AX"  ""
2 3.1345 = AX:?_40 "3"   1345  "AX"  "40"
3 3.8983 =:$15     "3"   8983  ""    "15"
4 .873 = PFS:4     ""    873   "PFS" "4"
```

Here is a decomposition of the regex:

- `(\\d)?`: The first column is an optional number indicated by the question mark (0 or 1).
- `\\.`: The number is then followed by a period. The period is not optional. There can be only one period.
- `(\\d+) ?`: The second column is a required number, which can be followed by a white space.
- `= ?`: The second column is followed by an equal sign, which can be followed by one white space
- `(?:(\\w+)?:?)?`: Then we have a non-grouping parenthesis in combination with a group. This actually works. What we want to say here is that the third column could be missing in the column (indicated by the last question mark) and that the third variable is a word (`\\w`). Also the third column can be followed by a colon (`:?`)
- `(?:[?_$]*)`: A number of typos may occur after the optional colon. Especially the characters `?`, `_`, or `$`. There may be zero or many of these characters (indicated by the `*`).
- `(\d+)?`: The last column is an optional number.

From this example you should see that extract can be a very powerful function when you need to work with very messy columns. The most important part is always the regex. I hope we could strengthen your regex muscle a little bit with this tutorial.

> **Summary**
>
> Here is what you can take from this tutorial.
>
> - While `separate` works with separators, `extract` works with groups.
> - A group (`()`) in the regex argument in `extract` represents new columns that will be created.
> - Non-grouping parentheses can be used to work with messy columns and are not converted to new columns.