Working with Data: Day 2

May 3-4, 2018

Outline

- tidy data
- manipulating text in R

Tidy data

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In tidy data:

- Each variable forms a column.
- Each observation forms a row.
- Each value has its own cell
- Each type of observational unit has its table.

All datasets we used so far have been "tidy", and notice how neatly they slotted into *dplyr* verbs. The same would apply for plotting (be it ggplot2, base, or lattice), as well as statistical modelling functions in R. As you might imagine, we don't always have tidy data – either because those collecting it didn't know better; or it was easier to enter this way or more efficient to process and store; it is following a discipline-specific format; or because what "tidy" exactly mean can vary depending on the intended analysis.

Whatever the reason, untidy data will be with us for a long time, and so "tidying" will be an important component of data cleaning you do prior to even beginning your analyses.

And before you even begin tidying, you will need to identify what the values and observations are in the table(s) you have. You will probably find one (or more) of the following problems:

- variable is spread out across multiple columns (e.g., `table4a/b`)
- observations are scattered across multiple rows (e.g., `table2`)
- cells encode multiple variables (e.g., `table3`)

Luckily, the *tidyr* dataset provides a simple set of verbs that each deal with one of these problems.

Gathering

In `table4a`, some column names are really values of a variable:

```
library(tidyr)
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

Columns `1999` and `2000` are really values of `year`, and each row has two observations of the number of cases. To tidy such a table, we need to **gather** the "value" columns `1999` and `2000` into a single variable `year`. For this, we will need to specify

- which columns are the value columns (`1999` and `2000`)
- what the name, or **key**, of their variable should be (`year`)
- what variable has its values spread in the cells. This is the **value**, and for `table4a` it is "`cases`".

```
table4a %>%
  gather(`1999`, `2000`, key = 'year', value = 'cases')
```

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

Note the backticks around `1999` and `2000`. This makes R interpret them as "symbols", i.e., the names of columns, rather than numbers. This will frequently be the case for value columns, and there can be many of them. Typing them all in would be extremely tedious and error-prone; luckily, we can use any of the column-selecting notations from `select`. In this case, the easiest is to just say "everything except `country`":

```
table4a %>%
  gather(-country, key = 'year', value = 'cases')
```

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

Spreading

In `table2`, the observation for each country in a year is spread across two rows, one for `cases` and one of `population`.

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
```

```
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

This situation calls for the **spreading** operation, which is the opposite of gathering. To tidy such a table we will need to specify:

- which column is the **key** that contains the variable names (`type`)
- which is the **value** column and contains values that really belong to multiple variables (`count`)

```
table2 %>%
  spread(key = type, value = count)
```

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

Separating

In `table3`, the value for both `cases` and `population` is stored in a single column, `rate`, using a particular encoding of "*cases* `/` *population*". To fix this, we need to **separate** the two sides of the "`/`".

```
table3 %>%
  separate(rate, into = c('cases', 'population'))
```

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

In this simple example, we didn't specify how to separate `rate`, and *dplyr* splits on every non-alphanumeric character it encounters. If we want to be safer, we could have told it to only use "/":

```
table3 %>%
  separate(rate,
           into = c('cases', 'population'),
           sep = '/')
```

```
## # A tibble: 6 x 4
##   country      year cases  population
## * <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
```

```
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

We could have also given a vector of integers to separate on specific character positions in the value. E.g., to split the year into two-digit components, we could do:

```
table3 %>%
  separate(year,
           into = c('cen', 'yr'),
           sep=2)
```

```
## # A tibble: 6 x 4
##   country     cen   yr    rate
## * <chr>       <chr> <chr> <chr>
## 1 Afghanistan 19    99    745/19987071
## 2 Afghanistan 20    00    2666/20595360
## 3 Brazil      19    99    37737/172006362
## 4 Brazil      20    00    80488/174504898
## 5 China       19    99    212258/1272915272
## 6 China       20    00    213766/1280428583
```

Lastly, note that the new columns are character, just as the original one. We'd prefer them to be numbers, and we can instruct `separate` to do so with the `convert` argument:

```
table3 %>%
  separate(rate,
           into = c('cases', 'population'),
           sep = '/',
           convert = TRUE)
```

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

Text values in R

We have already worked with text values in previous sessions. For example:

```
my_name <- "Biljana"
plot(wt ~ mpg, data = mtcars, xlab = "Miles per galon")
```

Informally, you may have heard (us) refer to such text values as *strings* or *text strings*. This is a common term in programming, although strictly speaking in R these values are *character vectors*.

```
is.vector("Biljana")
```

```
## [1] TRUE
```

```
length("Biljana")
```

```
## [1] 1
```

```
class("Biljana")
```

```
## [1] "character"
```

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
length(letters)
```

```
## [1] 26
```

For convenience, when we talk about strings in this workshop, we'll mean a "scalar" text value, similarly to the way we called a scalar numeric value like "1" an integer. So you may think of a character vector as a sequence (or, if you're used to other programming languages, an array) of strings.

Creating character vectors

As we've seen, one way to create a string is to put some text inside quotes (single or double, doesn't matter). If you need a quote as part of the string itself, you can either *escape* it with a backslash, or simply use the other quote to delimit the string:

```
str1="I'm getting this!"
str2="He said: \"I like this!!\""

length(str1)
```

```
## [1] 1
```

```
length(str2)
```

```
## [1] 1
```

There are many special characters that can be entered with a backslash-escape; the most common are newline `\n`, tab `\t`, and the backslash itself `\\`.

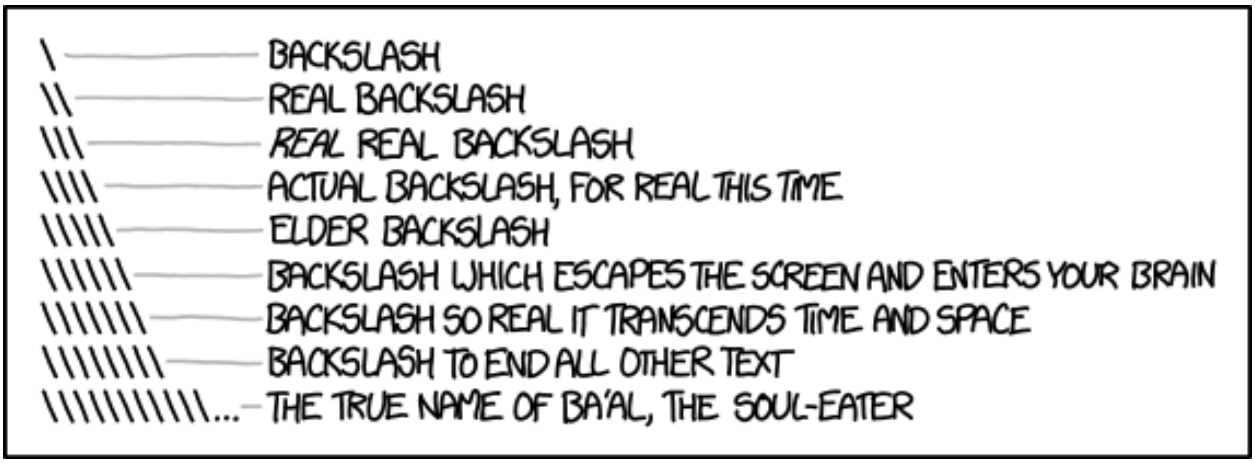

Figure 1: Backslashes

You can also enter any unicode character using the `\u` escape and its code:

```
"M\u{00f6}tley Cr\u{00fc}e"
```

```
## [1] "Mötley Crüe"
```

In the session on graphics, we saw the use of empty string, `""`, to hide a label: `plot(wt ~ mpg, data=mtcars, xlab="")`. As an aside, the empty string is still a vector of length 1!! If, for some reason, you want a truly empty character vector, create it with `character(0)`:

```
length("")
```

```
## [1] 1
```

```
character(0)
```

```
## character(0)
```

```
length(character(0))
```

```
## [1] 0
```

Reading and writing character vectors

When we work interactively in the R console, results are automatically printed out for us. This won't be the case when you're running R code non-interactively (for example, by "source"-ing it). Then, you can use `print`:

```
print(1+1)
```

```
## [1] 2
```

```
print("Hello there")
```

```
## [1] "Hello there"
```

Printing with `print` will "decorate" the character vector it outputs: it will try to line up elements into columns, surround each value with double quotes, and put the index of the first value in each printed row in square brackets at the left margin:

```
print(state.name)
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

You can tweak `print`'s output with additional arguments; for example specifying `quote = FALSE` will omit the quotes around elements. But if you want to R to print *only* the text and *exactly* as you gave it, you will need to use `cat`. You can embed newlines ("`\n`") to break the next into lines:

```
cat(1+1)
```

```
## 2
```

```
cat("Hello there\nBiljana")
```

```
## Hello there
## Biljana
```

Vectors and multiple arguments to `cat` are printed an element at a time, left to right, with elements separated by space. You can change the separator with the `sep` argument:

```
cat(1:4, letters[10:15])
```

```
## 1 2 3 4 j k l m n o
```

```
cat(1:4, letters[10:15], sep="")
```

```
## 1234jklmno
```

```
cat(1:4, letters[10:15], sep=" - ")
```

```
## 1 - 2 - 3 - 4 - j - k - l - m - n - o
```

Lastly, you can send `cat`'s output to a file with the `file` argument, which can be useful if you're constructing:

```
cat(1:4, letters[10:15], file="test-cat.txt")
getwd()
```

```
## [1] "C:/Users/Biljana/Nextcloud/2018-04_Ecoscope_data"
```

Reading data in tabular format should already be familiar to you from past sessions, see functions `read.delim`, `read.csv`, and the more general `read.table`. To read a file without trying to interpret its contents into rows and columns, you can use `readLines`. It returns a character vector where each element contains one line of the input file.

```
readLines("test-cat.txt")
```

```
## [1] "1 2 3 4 j k l m n o"
```

As a nice bonus, if you give it a URL, it will read the file directly from the web!

```
gettysburg <- readLines("https://www.clear.rice.edu/comp200/resources/texts/Gettysburg%20Address.txt")
length(gettysburg)
```

```
## [1] 5
```

```
head(gettysburg)
```

```
## [1] "Four score and seven years ago our fathers brought forth on this continent, a new nation, conce
## [2] ""
## [3] "Now we are engaged in a great civil war, testing whether that nation, or any nation so conceive
## [4] ""
## [5] "But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallow -- th
```

Manipulating text using the *stringr* package

While R has built-in functions for manipulating text and they do the job just fine, they often have their idiosyncracies that make it harder for non-expert R users to remember how to use them correctly. (For example, the naming of functions, as well as their argument order, treatment of multiple vector arguments

and NA values, can significantly differ from function to function.) For that reason, we'll use the *stringr* package. One of the most popular R packages, *stringr* offers a cleaned-up and modernized collection of common string operations. If you want, or have to, use the base R functions to work with text data, a table at the end of this document includes a list of *stringr* functions with their base R equivalents.

```
library(stringr)
```

String length

Let's start of with something simple: how do we determine the length of a string? We know `length()` isn't going to do it; with *stringr*, we can use `str_length()`, which given a character vector `s` returns an integer vector with the number of characters in each element of `s`. Like so:

```
str_length(c("Biljana", "Miles per gallon", NA, ""))
```

```
## [1]  7 16 NA  0
```

Pasting strings together

Often times we have multiple strings that we want to combine into a single longer string, for example, a person's full name given the first and last name, or the title for a plot based on the value of input data. We can use `str_c()` function for this purpose:

```
str_c("Biljana", "Jonoska")
```

```
## [1] "BiljanaJonoska"
```

```
str_c("Biljana", NA, "Jonoska")
```

```
## [1] NA
```

```
str_c("Biljana", "", "Jonoska")
```

```
## [1] "BiljanaJonoska"
```

If we want individual pieces to be separated, specify it with the `sep` argument:

```
str_c("Biljana", "Jonoska", sep = " ")
```

```
## [1] "Biljana Jonoska"
```

```
str_c("Jonoska", "Biljana", sep = "; ")
```

```
## [1] "Jonoska; Biljana"
```

What happens when arguments have more than one element? You can think of them as being arranged into a table, with each argument in a column. `str_c` combines each row of this table into a single string, using `sep` to join the pieces:

```
str_c(letters[5:8], 1:4)
```

```
## [1] "e1" "f2" "g3" "h4"
```

```
str_c(letters[5:8], 1:4, sep="\u2192")
```

```
## [1] "e<U+2192>1" "f<U+2192>2" "g<U+2192>3" "h<U+2192>4"
```

(Note: see the wikipedia article for a list of arrow symbol characters.)

We can further use the `collapse` argument to further combine the merged "rows":

```
str_c(letters[5:8], 1:4, collapse="; ")
```

```
## [1] "e1; f2; g3; h4"
```

This can be very handy to insert newlines for printing:

```
cat(str_c(letters[5:8], 1:4, collapse="\n"))
```

```
## e1
## f2
## g3
## h4
```

Exercise: Printing with cat and newlines

Print out the Gettysburg Address with `cat`. Use `str_c` to insert newlines between lines of text.

Working with parts of a string

There are times when you have a string that contains embedded within it some piece of information you really want. For instance, a variable in the dataframe may be used to encode a whole bunch of information about a case – think of mtcars where `am` and `gear` were merged together, so that a car with a 5-gear manual transmission was described as "M5", while a 3-speed automatic would be "A3". To get a car's transmission type, we'd just need the first character of this variable, while the number of gears would be in the remaining characters. (I say "remaining" rather than "second" because there could conceivably be more than nine. I don't know of any cars like that, but trucks can have up to 18.)

We can use `str_sub` to extract a subset of a string just as needed. It takes three arguments: the string from which to extract, the starting position, and the ending position of the substring to extract. (The ending position may be omitted, in which case the substring extends to the end of the original string.)

```
str_sub("M5", 1, 1)
```

```
## [1] "M"
```

```
## If end position is unspecified, extend to the end
str_sub("A10", 2)
```

```
## [1] "10"
```

```
## Negative indices count from the end of string
str_sub(rownames(mtcars), -7, -1)
```

```
##  [1] "zda RX4" "RX4 Wag" "sun 710" "4 Drive" "rtabout" "Valiant" "ter 360"
##  [8] "rc 240D" "erc 230" "erc 280" "rc 280C" "c 450SE" "c 450SL" " 450SLC"
## [15] "eetwood" "inental" "mperial" "iat 128" "a Civic" "Corolla" " Corona"
## [22] "llenger" "Javelin" "aro Z28" "irebird" "at X1-9" "e 914-2" " Europa"
## [29] "ntera L" "ri Dino" "ti Bora" "vo 142E"
```

As usual in R, `str_sub` is vectorized in its arguments:

```
str_sub(rownames(mtcars), 1, 5:8)
```

```
##  [1] "Mazda"    "Mazda "   "Datsun "  "Hornet 4" "Horne"    "Valian"
##  [7] "Duster "  "Merc 240" "Merc "    "Merc 2"   "Merc 28"  "Merc 450"
## [13] "Merc "    "Merc 4"   "Cadilla"  "Lincoln " "Chrys"    "Fiat 1"
## [19] "Honda C"  "Toyota C" "Toyot"    "Dodge "   "AMC Jav"  "Camaro Z"
## [25] "Ponti"    "Fiat X"   "Porsche"  "Lotus Eu" "Ford "    "Ferrar"
## [31] "Maserat"  "Volvo 14"
```

We can also use `str_sub` to *replace* a part of the string. This is done by having `sub_str` on the left side of the assignment and the replacement value on the right. The replacement doesn't have to be the same length as the replaced value, and assigning an empty string will simply remove a chunk of the original string.

```
stat.name.play=state.name
str_sub(stat.name.play, 5, -1) <- "..."
stat.name.play
```

```
##  [1] "Alab..." "Alas..." "Ariz..." "Arka..." "Cali..." "Colo..." "Conn..."
##  [8] "Dela..." "Flor..." "Geor..." "Hawa..." "Idah..." "Illi..." "Indi..."
## [15] "Iowa..." "Kans..." "Kent..." "Loui..." "Main..." "Mary..." "Mass..."
## [22] "Mich..." "Minn..." "Miss..." "Miss..." "Mont..." "Nebr..." "Neva..."
## [29] "New ..." "New ..." "New ..." "New ..." "Nort..." "Nort..." "Ohio..."
## [36] "Okla..." "Oreg..." "Penn..." "Rhod..." "Sout..." "Sout..." "Tenn..."
## [43] "Texa..." "Utah..." "Verm..." "Virg..." "Wash..." "West..." "Wisc..."
## [50] "Wyom..."
```

```
#same as ommiting the end argument and taking all the characters after the position 5
stat.name.play=state.name
str_sub(stat.name.play, 5) <- "..."
stat.name.play
```

```
##  [1] "Alab..." "Alas..." "Ariz..." "Arka..." "Cali..." "Colo..." "Conn..."
##  [8] "Dela..." "Flor..." "Geor..." "Hawa..." "Idah..." "Illi..." "Indi..."
## [15] "Iowa..." "Kans..." "Kent..." "Loui..." "Main..." "Mary..." "Mass..."
## [22] "Mich..." "Minn..." "Miss..." "Miss..." "Mont..." "Nebr..." "Neva..."
## [29] "New ..." "New ..." "New ..." "New ..." "Nort..." "Nort..." "Ohio..."
## [36] "Okla..." "Oreg..." "Penn..." "Rhod..." "Sout..." "Sout..." "Tenn..."
## [43] "Texa..." "Utah..." "Verm..." "Virg..." "Wash..." "West..." "Wisc..."
## [50] "Wyom..."
```

Modifying the string

- changing the case: `str_to_lower`, `str_to_upper`, and `str_to_title`
- re-format a paragraph to fit a width: `str_wrap`
- remove whitepace at the start and/or end: `str_trim`
- add whitepace at the start and/or end: `str_pad`

Finding and matching text

Let's briefly consider how we could choose rows from in the `gapminder` dataset where country is "Canada", "Cambodia", or "Cameroon":

```
library(gapminder)
library(dplyr)
gapminder %>%
  filter(country == "Canada" |
         country == "Cambodia" |
         country == "Cameroon")
```

```
## # A tibble: 36 x 6
##    country  continent  year lifeExp      pop gdpPercap
##    <fct>    <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Cambodia Asia       1952    39.4  4693836      368.
##  2 Cambodia Asia       1957    41.4  5322536      434.
```

```
##  3 Cambodia Asia       1962    43.4  6083619      497.
##  4 Cambodia Asia       1967    45.4  6960067      523.
##  5 Cambodia Asia       1972    40.3  7450606      422.
##  6 Cambodia Asia       1977    31.2  6978607      525.
##  7 Cambodia Asia       1982    51.0  7272485      624.
##  8 Cambodia Asia       1987    53.9  8371791      684.
##  9 Cambodia Asia       1992    55.8 10150094      682.
## 10 Cambodia Asia       1997    56.5 11782962      734.
## # ... with 26 more rows
```

Because R treats character vectors as sets, we can use the `is.element` function for a more succinct test:

```
gapminder %>%
  filter(is.element(country, c("Canada", "Cambodia", "Cameroon")))
```

```
## # A tibble: 36 x 6
##    country  continent  year lifeExp      pop gdpPercap
##    <fct>    <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Cambodia Asia       1952    39.4  4693836      368.
##  2 Cambodia Asia       1957    41.4  5322536      434.
##  3 Cambodia Asia       1962    43.4  6083619      497.
##  4 Cambodia Asia       1967    45.4  6960067      523.
##  5 Cambodia Asia       1972    40.3  7450606      422.
##  6 Cambodia Asia       1977    31.2  6978607      525.
##  7 Cambodia Asia       1982    51.0  7272485      624.
##  8 Cambodia Asia       1987    53.9  8371791      684.
##  9 Cambodia Asia       1992    55.8 10150094      682.
## 10 Cambodia Asia       1997    56.5 11782962      734.
## # ... with 26 more rows
```

`is.element` can usually written as operator `%in%`:

```
gapminder %>%
  filter(country %in% c("Canada", "Cambodia", "Cameroon"))
```

```
## # A tibble: 36 x 6
##    country  continent  year lifeExp      pop gdpPercap
##    <fct>    <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Cambodia Asia       1952    39.4  4693836      368.
##  2 Cambodia Asia       1957    41.4  5322536      434.
##  3 Cambodia Asia       1962    43.4  6083619      497.
##  4 Cambodia Asia       1967    45.4  6960067      523.
##  5 Cambodia Asia       1972    40.3  7450606      422.
##  6 Cambodia Asia       1977    31.2  6978607      525.
##  7 Cambodia Asia       1982    51.0  7272485      624.
##  8 Cambodia Asia       1987    53.9  8371791      684.
##  9 Cambodia Asia       1992    55.8 10150094      682.
## 10 Cambodia Asia       1997    56.5 11782962      734.
## # ... with 26 more rows
```

Exercise: Select countries that start with "Ca"

Use `str_sub` to select rows from the `gapminder` dataset for countries whose name starts with "Ca".

Regular expressions

What we have seen so far just scratches the surface of text processing because we either had to use the entire string or had to specify the exact characters we wanted to work with. Regular expressions, on the other hand, let us describe *patterns* of strings. These patterns are useful for finding, matching, extracting, and transforming text in ways that complement the functions we have seen so far.

So how could we use a regular expression to select data from `gapminder` for those three countries? The simplest kind of pattern is a sequence of characters that we want the matching strings to contain *anwyhere* inside them. So if we're looking for strings "Canada", "Cameroon", and "Cambodia", they have in common the text "Ca" (case matters!)

We still need to give that pattern to a function. *stringr* includes function `str_detect` that returns `TRUE` where the string matches the pattern. For example:

```
gapminder %>%
  filter(str_detect(country, "Ca"))
```

```
## # A tibble: 36 x 6
##    country  continent  year lifeExp      pop gdpPercap
##    <fct>    <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Cambodia Asia       1952    39.4  4693836      368.
##  2 Cambodia Asia       1957    41.4  5322536      434.
##  3 Cambodia Asia       1962    43.4  6083619      497.
##  4 Cambodia Asia       1967    45.4  6960067      523.
##  5 Cambodia Asia       1972    40.3  7450606      422.
##  6 Cambodia Asia       1977    31.2  6978607      525.
##  7 Cambodia Asia       1982    51.0  7272485      624.
##  8 Cambodia Asia       1987    53.9  8371791      684.
##  9 Cambodia Asia       1992    55.8 10150094      682.
## 10 Cambodia Asia       1997    56.5 11782962      734.
## # ... with 26 more rows
```

Exercise: Select "ana" countries

Select rows from the `gapminder` dataset for countries whose name contains "ana".

Overview of patterns

We will use `str_view` function to show us a graphical overview of where the pattern matches a given string. (It also requires the *htmlwidgets* package to run, so install that if you don't have it.)

We'll use the following simple character vector for our examples:

```
library(htmlwidgets)
x <- c('apple', 'banana', 'pear')
```

```
#- exact string:
str_view(x, 'an')
```

```
#- single-character wildcard with `.`:
str_view(x, '.a.')
```

```
# (Use backslash to indicate a period: `'\\.'`
#- beginning of string with `^`:
str_view(x, '^a')
```

```r
#- end of string with `$`:
str_view(x, 'a$')
```

```r
#- word boundary with `\\b`:
str_view(c('apple', 'b.anana', 'pear'), '\\b.a')
```

```r
#- any digit with `\\d`:
str_view('604-822-1515', '\\d')
```

```r
#- not digits with `\\D`:
str_view('604-822-1515', '\\D')
```

```r
#- any white space with `\\s` (space, tab, newline)
str_view('604-822 1515', '\\s')
```

```r
#- not white space with `\\S`
str_view('604-822-1515', '\\S')
```

```r
#- any of the specified characters (e.g., 'a', 'b', 'c'):
str_view(x, '[abc]')
```

```r
#- any in the range of specified characters (e.g., 'a' to 'e'):
str_view(x, '[a-e]')
```

```r
#- any except one of the specified characters:
str_view(x, '[^abc]')
```

```r
#- either of the two patterns with `|`:
str_view(x, 'an|.ar')
```

```r
#- pattern repeating 0 or 1 times with `?`:
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, 'CC?')
```

```r
#- pattern repeating 0 or many times with `*`:
str_view(x, 'CC*')
```

```r
#- pattern repeating 1 or many times with `+`:
str_view(x, 'CC+')
```

```r
#match exactly two occurences
str_view(x, "C{2}")
```

```r
#match two occurences and more
str_view(x, "C{2,}")
```

```r
#match between two and three occurences
str_view(x, "C{2,3}")
```

Functions working with patterns

We have seen `str_detect`, which returns TRUE/FALSE of the entries that match and so is most useful to use as a logical index to filtering functions (such as *dplyr*'s 'filter).

If we want to get the actual values, we can use `str_extract`:

```r
str_extract(x, 'an')
```

```
## [1] "an"
```

It will extract text corresponding to the *first* match, returning a character vector of the same length as the input. The result has NAs in the places where the input elements didn't match the pattern.

Because `str_extract` extracts only the first match, you will need to use wildcards for the case where you want to extend the match to the surrounding character or a repeating pattern:

```
str_extract(x, '(an)+')
```

```
## [1] "an"
```

```
str_extract(x, 'an.*')
```

```
## [1] "an numerals: MDCCCLXXXVIII"
```

stringr has two functions that replace pattern matches instead of just printing them:

str_replace(*string*, *pattern*, *replacement*)

str_replace_all(*string*, *pattern*, *replacement*)

If you want to replace matching text with something else, there are two useful functions, `str_replace` and `str_replace_all`.

`str_replace(string, pattern, replacement)` will replace *first* occurrence of *pattern* in *string* with *replacement*. `str_replace_all(string, pattern, replacement)` will replace *each* occurrence of *pattern* in *string* with *replacement*.

When you're replacing, it's useful to include the matching text in the replacement. This is done by using backreferences, which will contain the value of the matching text that was "grouped" in parenthesis. `\\1` will refer to the first group, `\\2` to the second and so on up to `\\9`. For example:

```
str_replace(x, '(an).*(.)$', '\\1x\\2')
```

```
## [1] "1888 is the longest year in RomanxI"
```