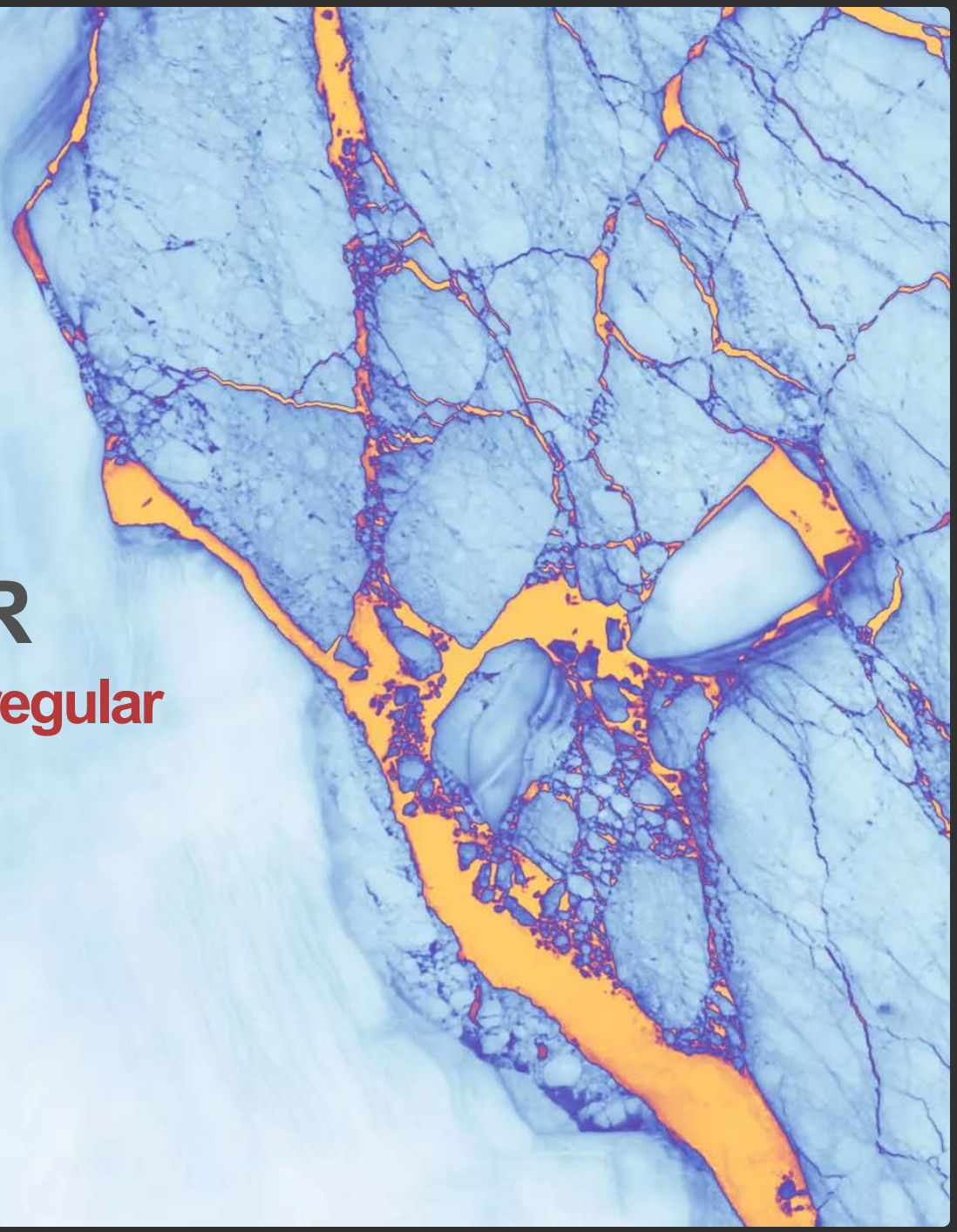
Marine Data Science

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

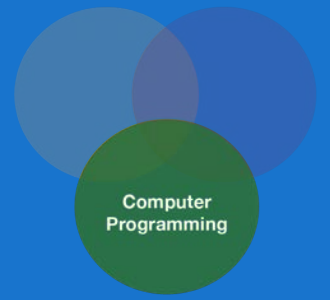


Data Analysis with R

16 - String manipulation and regular expressions

Saskia A. Otto
Postdoctoral Researcher

Strings in R



What is a string again?

- Any value written within a **pair of** single quote or **double quotes** in R is treated as a string and stored in a **character vector** (within double quotes).
- Lets look at a famous quote made by Albert Einstein:

```
einstein <- c("The difference", "between stupidity",
  "and genius is that", "genius has its limits.")
```

→ The character vector `einstein` contains 4 elements or more precisely **4 strings**.

Manipulation of strings

- R may not be as rich and diverse as other scripting languages when it comes to string manipulation, but it can take you very far if you know how.
- This tutorial gives you only a short introduction into some functions for basic manipulations.
- Some of these functions require **regular expressions** (*regex* or *regexpr* in short), which are a concise **language for describing patterns in strings** that typically contain unstructured or semi-structured data.
- To learn more about regex I recommend the excellent website http://www.regular-expressions.info. It contains many different topics, resources, examples, and tutorials at both beginner and advanced levels.

Manipulation of strings

Even if you don't plan to do text analysis, text mining, or natural language processing, it is useful to have some knowledge on handling and processing strings in R for the following reasons:

- Your dataset most likely will contain some text, e.g. stations names, species names, etc.
  - you might want to **remove a given character** in the names of your variables or in the entire dataset
  - you might want to **convert labels** to upper case (or lower case)
  - you might want to **replace** an **outdated species names** with the new name
  - you might want to **re-classify** certain categories, e.g. group different life stages together
  - you want to **abbreviate names**

- Your dataset most likely will contain some text, e.g. stations names, species names, etc.
    - you might want to **remove a given character** in the names of your variables or in the entire dataset
    - you might want to **convert labels** to upper case (or lower case)
    - you might want to **replace** an **outdated species names** with the new name
    - you might want to **re-classify** certain categories, e.g. group different life stages together
    - you want to **abbreviate names**
- You want to **extract data** from the web (**web-scraping**) and remove irrelevant information.
- You want to **remove** all unnecessary **metadata** that your imported dataset contains.
- You want to **iterate** the data import and processing for 100 data files that have slightly different file names.

Some useful base functions

base functions	description
`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.
`grep(pattern, x)`	detects patterns in a string, output is a logical vector
`gsub(pattern, replacement, x)`	performs replacement of all matches

Some useful base functions

base functions	description
`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.
`grep(pattern, x)`	detects patterns in a string, output is a logical vector
`gsub(pattern, replacement, x)`	performs replacement of all matches

Most of the times these functions are enough and they will allow you to get your job done. However, they have some drawbacks when it comes to **handling NAs** or pasting elements with **zero length**.

A **nice package** that solves these problems and provides several functions for carrying out consistent string processing comes again from **tidyverse**...

The *stringr* package

- *stringr* adds more functionality to the base functions for handling strings in R.
- In *stringr*,
  - **argument names** (and positions) are **consistent**,
  - all functions deal with **NA's** and **zero length** character appropriately, and
  - the **output** data structures from each function **matches the input** data structures of **other functions**
  - all functions start with `str_` so you can quickly select the appropriate one from the dropdown list displayed by R Studio
- to access these function load stringr or tidyverse:

```
library(stringr)
library(tidyverse)
```

A quick comparison of *base* and *stringr* functions

base functions	description	stringr functions
`paste(x, y, sep = ' ')`	Join multiple vectors together.	`str_c(x, y, sep = ' ')`
`paste(x, collapse = ' ')`	Join elements of a vector together.	`str_c(x, collapse = ' ')`
`toupper(x)`	Convert to uppercase.	`str_to_upper(x)`
`tolower(x)`	Convert to lowercase.	`str_to_lower(x)`
`nchar(x)`	Number of characters in a string.	`str_length(x)`
`grep(pattern, x)`	detects patterns in a string, output is a logical vector	`str_detect(x, pattern)`
`gsub(pattern, replacement, x)`	performs replacement of all matches	`str_replace_all(x, pattern, replacement)`

`str_c` makes difference between NA (or NA_character_) and "NA", whereas `paste` treats them all the same!

A quick comparison of *base* and *stringr* functions

base functions	description	stringr functions
`paste(x, y, sep = ' ')`	Join multiple vectors together.	`str_c(x, y, sep = ' ')`
`paste(x, collapse = ' ')`	Join elements of a vector together.	`str_c(x, collapse = ' ')`
`toupper(x)`	Convert to uppercase.	`str_to_upper(x)`
`tolower(x)`	Convert to lowercase.	`str_to_lower(x)`
`nchar(x)`	Number of characters in a string.	`str_length(x)`
`grep(pattern, x)`	detects patterns in a string, output is a logical vector	`str_detect(x, pattern)`
`gsub(pattern, replacement, x)`	performs replacement of all matches	`str_replace_all(x, pattern, replacement)`

`str_c` makes difference between NA (or NA_character_) and "NA", whereas `paste` treats them all the same!

```
x <- c("Shark", "Whale", "Ray")
str_length(x)
```

```
## [1] 5 5 3
```

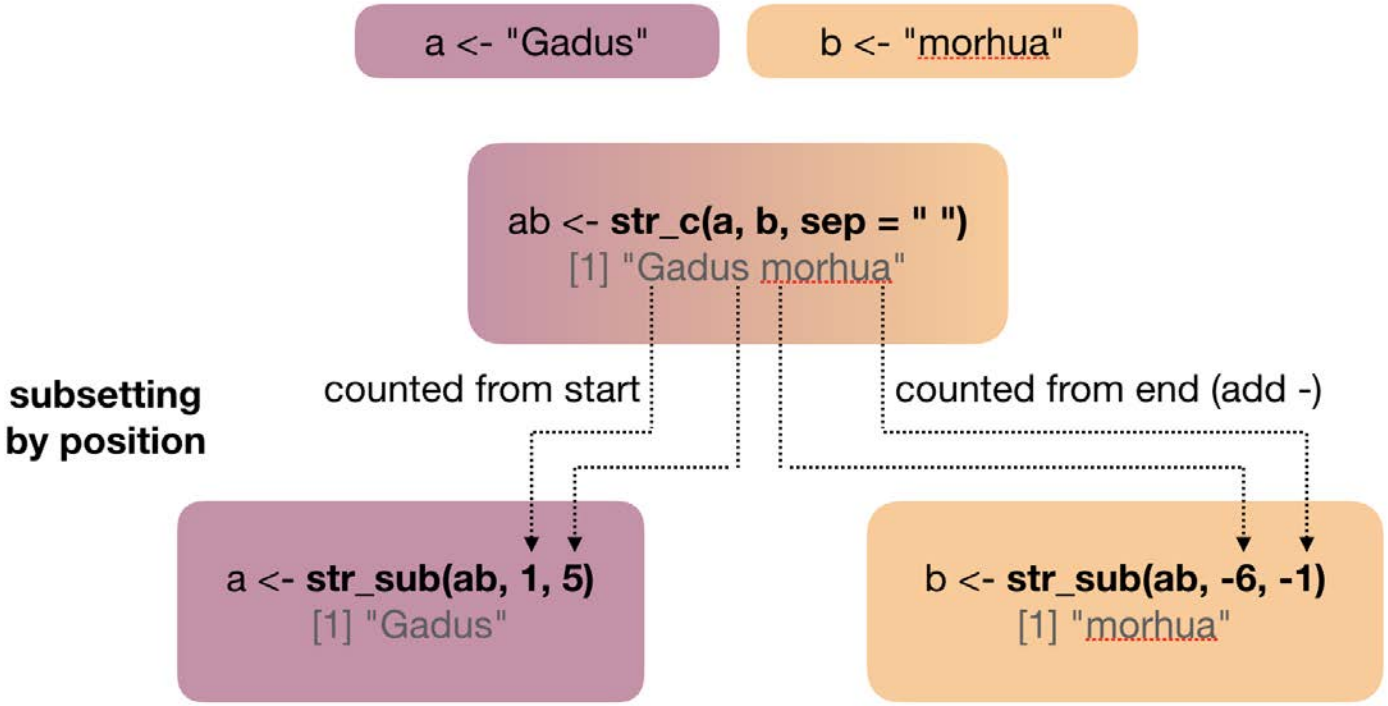
```
str_to_lower(x)
```

```
## [1] "shark" "whale" "ray"
```

```
str_to_upper(x)
```

```
## [1] "SHARK" "WHALE" "RAY"
```

Combining and subsetting strings using *stringr*



Combining with `str_c()`

```r
# Args 'sep' for strings of DIFFERENT vectors
str_c("a", "b", sep = "<")
```

```
## [1] "a<b"
```

```r
# Args 'collapse' for strings within the SAME vector
str_c(c("a","b"), collapse = "-")
```

```
## [1] "a-b"
```

```r
# Both args for doing both (first sep, than collapse applied)
str_c(c("a","b"), c(1,2), sep = "<", collapse = "-")
```

```
## [1] "a<1-b<2"
```

```r
# The recycling rule also applies here:
str_c("a", 1:10, sep = "_")
```

```
##  [1] "a_1"  "a_2"  "a_3"  "a_4"  "a_5"  "a_6"  "a_7"  "a_8"  "a_9"  "a_10"
```

Subsetting with `str_sub()`

```
x <- c("Shark", "Whale", "Ray")
str_sub(string = x, start = 1, end = 3) # extract 1st to 3rd
```

```
## [1] "Sha" "Wha" "Ray"
```

```
str_sub(string = x, end = 1) # extract 1st
```

```
## [1] "S" "W" "R"
```

```
str_sub(x, -1) # extract last using negative index
```

```
## [1] "k" "e" "y"
```

```
# Replacing values in each string with str_sub
str_sub(x, 1, 1) <- "A"; x
```

```
## [1] "Ahark" "Ahale" "Aay"
```

```
# Combine str_sub with str_to_upper
str_sub(x, -1) <- str_to_upper(str_sub(x, -1)); x
```

```
## [1] "AharK" "AhalE" "AaY"
```

Other useful *stringr* functions (1)

`str_sort()` and `str_order()`: sort character vectors using the current locale (= ISO 639 language code)

```
x <- c("Shark", "Whale", "Ray")
str_sort(x) # returns sorted character vector
```

```
## [1] "Ray"   "Shark" "Whale"
```

```
str_order(x) # returns index vector of sorted strings
```

```
## [1] 3 1 2
```

Other useful *stringr* functions (2)

`str_trim()`: removes whitespace from start and end of string

```
str_trim("  String with trailing and leading white space\t")
```

```
## [1] "String with trailing and leading white space"
```

`str_pad()`: adds single padding character (default is whitespace) (args 'width' indicates the total string length INCLUDING the existing characters)

```
str_pad("a", width = 5, side = "both")
```

```
## [1] "  a  "
```

```
str_pad("a", 6, "both", pad = "-")
```

```
## [1] "--a---"
```

Other useful *stringr* functions (3)

`str_wrap()`: wrap strings into formatted paragraphs (based on a specific algorithm)

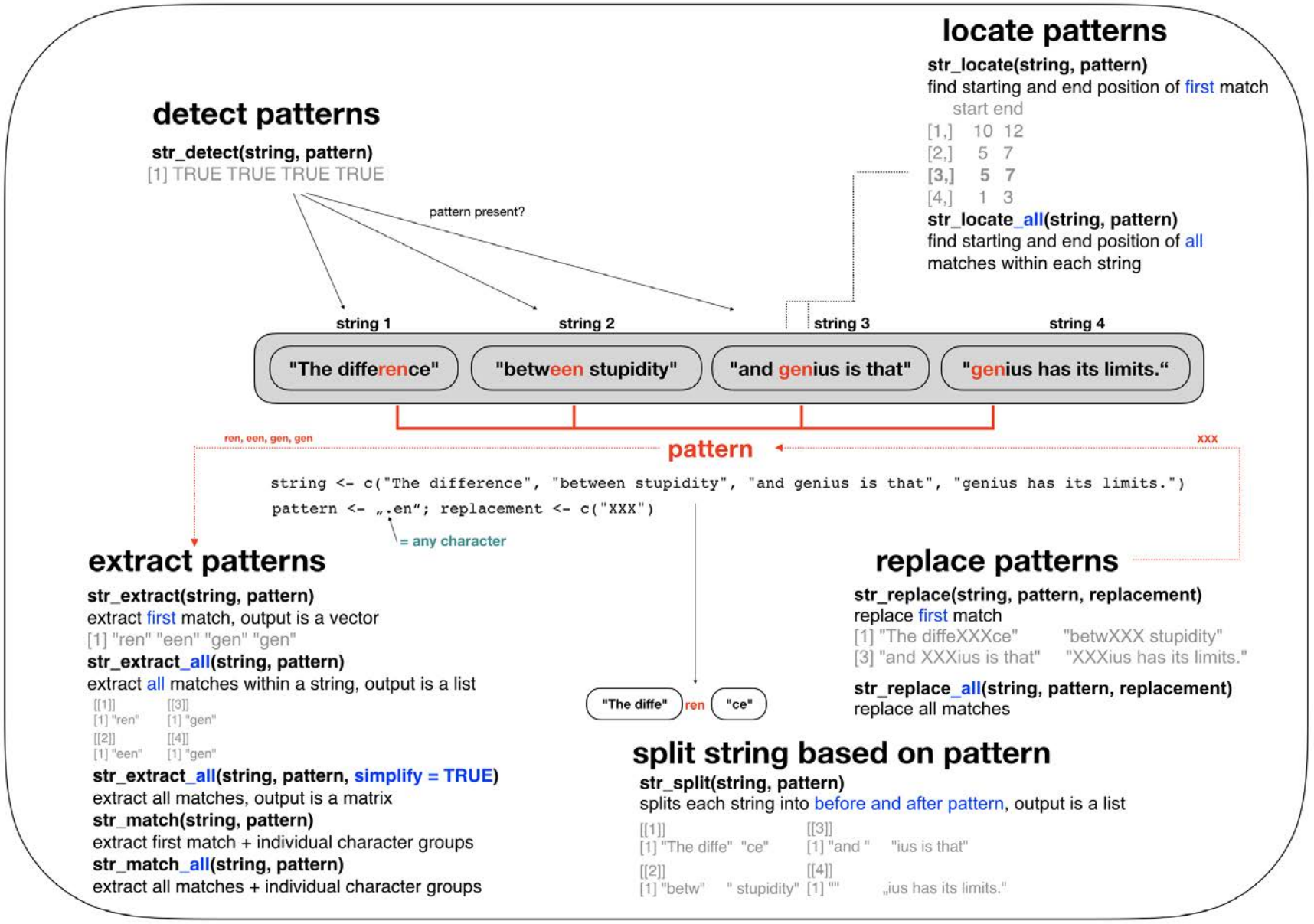
```
x <- "This is a wrapper around stri_wrap which implements a wrapping algorithm."
str_wrap(x, width=10)
```

```
## [1] "This is\na wrapper\naround\nstri_wrap\nwhich\nimplements\na wrapping\nalgorithm."
```

```
cat(str_wrap(x, width=10))
```

```
## This is
## a wrapper
## around
## stri_wrap
## which
## implements
## a wrapping
## algorithm.
```

Functions for pattern matching in *stringr*


locate patterns
str_locate(string, pattern)
find starting and end position of first match
start end
[1,] 10 12
[2,] 5 7
[3,] 5 7
[4,] 1 3
str_locate_all(string, pattern)
find starting and end position of all matches within each string
detect patterns
str_detect(string, pattern)
[1] TRUE TRUE TRUE TRUE
pattern present?
string 1
string 2
string 3
string 4
"The difference"
"between stupidity"
"and genius is that"
"genius has its limits."
ren, een, gen, gen
pattern
XXX
string <- c("The difference", "between stupidity", "and genius is that", "genius has its limits.")
pattern <- „.en“; replacement <- c("XXX")
= any character
extract patterns
str_extract(string, pattern)
extract first match, output is a vector
[1] "ren" "een" "gen" "gen"
str_extract_all(string, pattern)
extract all matches within a string, output is a list
[[1]] [1] "ren"
[[2]] [1] "een"
[[3]] [1] "gen"
[[4]] [1] "gen"
str_extract_all(string, pattern, simplify = TRUE)
extract all matches, output is a matrix
str_match(string, pattern)
extract first match + individual character groups
str_match_all(string, pattern)
extract all matches + individual character groups
"The diffe"
ren
"ce"
replace patterns
str_replace(string, pattern, replacement)
replace first match
[1] "The diffeXXXce" "betwXXX stupidity"
[3] "and XXXius is that" "XXXius has its limits."
str_replace_all(string, pattern, replacement)
replace all matches
split string based on pattern
str_split(string, pattern)
splits each string into before and after pattern, output is a list
[[1]] [1] "The diffe" "ce"
[[2]] [1] "betw" " stupidity"
[[3]] [1] "and " "ius is that"
[[4]] [1] "" „ius has its limits."

Overview of regular expressions

Character classes	
[[:digit:]] or \\d	Digits; [0-9]
\\D	Non-digits; [^0-9]
[[:lower:]]	Lower-case letters; [a-z]
[[:upper:]]	Upper-case letters; [A-Z]
[[:alpha:]]	Alphabetic characters; [A-z]
[[:alnum:]]	Alphanumeric characters [A-z0-9]
\\w	Word characters; [A-z0-9_]
\\W	Non-word characters
[[:xdigit:]] or \\x	Hexadec. digits; [0-9A-Fa-f]
[[:blank:]]	Space and tab
[[:space:]] or \\s	Space, tab, vertical tab, newline, form feed, carriage return
\\S	Not space; [^[:space:]]
[[:punct:]]	Punctuation characters; !"#$%&'()*+,-./:;<=>? @[]^_`{|}~
[[:graph:]]	Graphical char.; [[:alnum:][:punct:]]
[[:print:]]	Printable characters; [[:alnum:][:punct:]\\s]
[[:cntrl:]] or \\c	Control characters; \n, \r etc.
Character classes and groups	
.	Any character except \n
|	Or, e.g. (a|b)
[…]	List permitted characters, e.g. [abc]
[a-z]	Specify character ranges
[^…]	List excluded characters
(…)	Grouping, enables back referencing using \\N where N is an integer

Anchors	
^	Start of the string
$	End of the string
\\b	Empty string at either edge of a word
\\B	NOT the edge of a word
\\<	Beginning of a word
\\>	End of a word

Quantifiers	
*	Matches at least 0 times
+	Matches at least 1 time
?	Matches at most 1 time; optional string
{n}	Matches exactly n times
{n,}	Matches at least n times
{,n}	Matches at most n times
{n,m}	Matches between n and m times

Special Metacharacters	
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed

Escaping characters:
Metacharacters (. * + etc.) can be used as literal characters by escaping them. Characters can be escaped using \\ or by enclosing them in \\Q…\\E.

Some examples: `str_detect()` (1)

Identify strings that match a specific pattern:

```
x <- c("shark", "whale shark", "whale", "manta ray", "sting ray")
```

Specific pattern using **anchors**:

```
str_detect(x, "^w") # ^ = start of string
```

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

```
str_detect(x, "y$") # $ = end of string
```

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

```
str_detect(x, "whale") # all strings that contain that word
```

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

```
str_detect(x, "^whale$") # all strings that start end end with this word
```

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

Some examples: `str_detect()` (2)

Identify strings that match a specific pattern:

```
x <- c("shark", "whale shark", "whale", "manta ray", "sting ray")
```

Specific pattern using **character classes**:

```
# Start with a vowel (same as "^[a,e,i,u,o]")
str_detect(string = x, pattern = "^(a|e|i|u|o)")
```

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

```
# End with 'ark' or 'ale'
str_detect(x, pattern = "(ark|ale)$")
```

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

```
# Contains any character, then 'a', then whitespace
str_detect(x, pattern = ".a ")
```

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

Some examples: `str_subset()`

Subset strings that match a specific pattern using `str_detect()` for indexing or the **wrapper function** `str_subset()`:

```
x <- c("shark", "whale shark", "whale", "manta ray", "sting ray")
```

```
# Get all strings in x that start with 'm' or end with 'k'
x[str_detect(x, "^m|k$")]
```

```
## [1] "shark"       "whale shark" "manta ray"
```

```
# same as
str_subset(x, "^m|k$")
```

```
## [1] "shark"       "whale shark" "manta ray"
```

Some examples: `str_split()` (1)

Split a string into pieces based on a specific pattern:

```
x <- c("shark", "whale shark", "whale", "manta ray", "sting ray")
str_split(x, " ", simplify = TRUE)
```

```
##      [,1]    [,2]
## [1,] "shark" ""
## [2,] "whale" "shark"
## [3,] "whale" ""
## [4,] "manta" "ray"
## [5,] "sting" "ray"
```

Some examples: `str_split()` (2)

Split a string into pieces based on a specific pattern:

```
fruits <- c("apples and oranges and pears and bananas",
  "pineapples and mangos and guavas")

str_split(fruits, " and ", simplify = TRUE)
```

```
##      [,1]         [,2]      [,3]     [,4]
## [1,] "apples"     "oranges" "pears"  "bananas"
## [2,] "pineapples" "mangos"  "guavas" ""
```

```
# Specify n to restrict the number of possible matches
str_split(fruits, " and ", n = 2, simplify = TRUE)
```

```
##      [,1]         [,2]
## [1,] "apples"     "oranges and pears and bananas"
## [2,] "pineapples" "mangos and guavas"
```

Your turn...

stringr provides a dataset (vector) called `words`, which contains a selection of 980 words:

```
stringr::words
```

```
##   [1] "a"          "able"       "about"      "absolute"   "accept"
##   [6] "account"    "achieve"    "across"     "act"        "active"
##  [11] "actual"     "add"        "address"    "admit"      "advertise"
##  [16] "affect"     "afford"     "after"      "afternoon"  "again"
##  [21] "against"    "age"        "agent"      "ago"        "agree"
##  [26] "air"        "all"        "allow"      "almost"     "along"
##  [31] "already"    "alright"    "also"       "although"   "always"
##  [36] "america"    "amount"     "and"        "another"    "answer"
##  [41] "any"        "apart"      "apparent"   "appear"     "apply"
##  [46] "appoint"    "approach"   "appropriate" "area"      "argue"
##  [51] "arm"        "around"     "arrange"    "art"        "as"
##  [56] "ask"        "associate"  "assume"     "at"         "attend"
##  [61] "authority"  "available"  "aware"      "away"       "awful"
##  [66] "baby"       "back"       "bad"        "bag"        "balance"
##  [71] "ball"       "bank"       "bar"        "base"       "basis"
##  [76] "be"         "bear"       "beat"       "beauty"     "because"
##  [81] "become"     "bed"        "before"     "begin"      "behind"
##  [86] "believe"    "benefit"    "best"       "bet"        "between"
##  [91] "big"        "bill"       "birth"      "bit"        "black"
##  [96] "bloke"      "blood"      "blow"       "blue"       "board"
## [101] "boat"       "body"       "book"       "both"       "bother"
## [106] "bottle"     "bottom"     "box"        "boy"        "break"
```

Quiz 1: Detect pattern

Now tell me,

1. how many words are longer than 10 characters?

2. how many words are exactly 2 letters long?

3. how many words start end with *p*?

Submit | Show Hint | Show Answer | Clear

Quiz 2: Detect pattern and split strings

1. How many of the 3-letter words start with a consonant?

2. How many words contain 'ee' (as in street)?

3. If you subset all words that contain the pattern 'st' and than split these words by this pattern, how many strings do you get in total?

Submit | Show Hint | Show Answer | Clear

Quiz 3: Subsetting and combining strings

You have the following vector `x`:

```
x <- c("file_001.csv", "file_002.csv", "file_003.csv", "file_004.csv", "file_005.csv
  "file_006.csv", "file_007.csv", "file_008.csv", "file_009.csv", "file_010.csv")
```

1. How can you remove the first 1 or 2 zeros?
2. How could you generate yourself such vector but with 100 elements ("file_1.csv" ... "file_100.csv")?

Solution

```
# 1.You could replace first the '_00' by the underscore and then all remaining '_0'
x %>% str_replace("_00", "_") %>% str_replace("_0", "_")
```

```
##  [1] "file_1.csv"  "file_2.csv"  "file_3.csv"  "file_4.csv"  "file_5.csv"
##  [6] "file_6.csv"  "file_7.csv"  "file_8.csv"  "file_9.csv"  "file_10.csv"
```

```
# 2. Simply take advantage of the recycling rule when using str_c()
x <- str_c("file_", 1:100, ".csv", sep = "")
x[1:15]
```

```
##  [1] "file_1.csv"  "file_2.csv"  "file_3.csv"  "file_4.csv"  "file_5.csv"
##  [6] "file_6.csv"  "file_7.csv"  "file_8.csv"  "file_9.csv"  "file_10.csv"
## [11] "file_11.csv" "file_12.csv" "file_13.csv" "file_14.csv" "file_15.csv"
```

base functions:

`paste()`, `toupper()`, `tolower()`, `nchar()`, `grep()`, `gsub()`

stringr functions:

`str_c()`, `str_to_upper()`, `str_to_lower()`, `str_length()`, `str_sub()`
`str_sort()`, `str_order()`, `str_trim()`, `str_pad()`, `str_wrap()`
`str_detect()`, `str_subset()`, `str_locate()`, `str_locate_all()`,
`str_extract()`, `str_extract_all()`, `str_match()`, `str_match_all()`,
`str_replace()`, `str_replace_all()`, `str_split()`

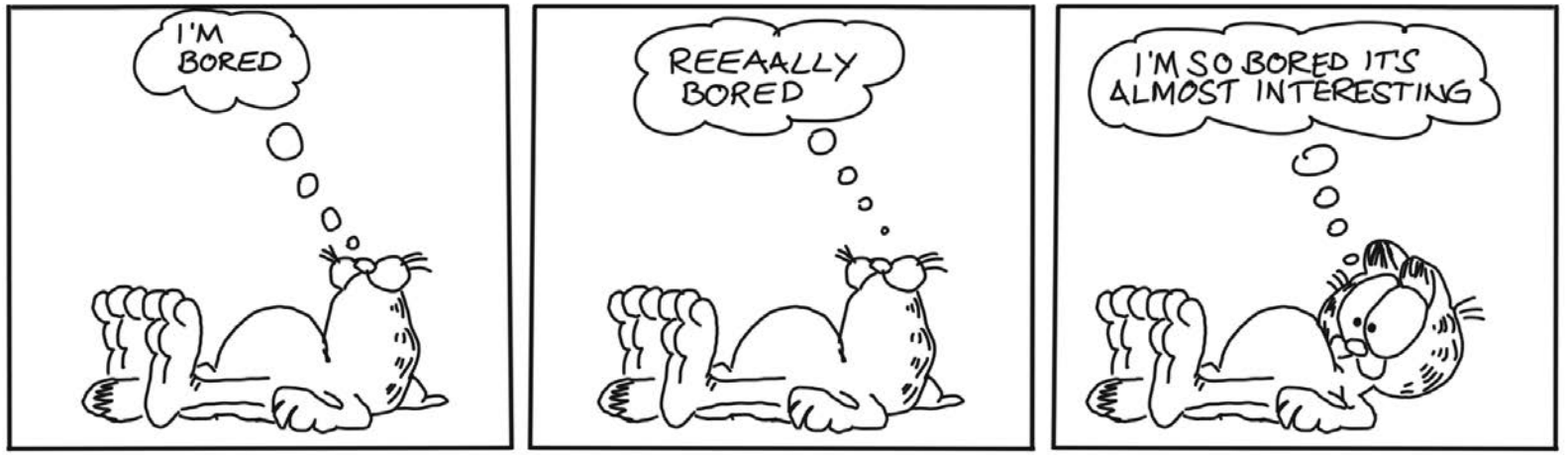
Overview of functions you learned today

How do you feel now.....?

Totally confused?

Chapter 14 on strings is worth reading with good exercises to practise regular expressions as well as the website http://www.regular-expressions.info. See also the stringr cheatsheet for a function overview.

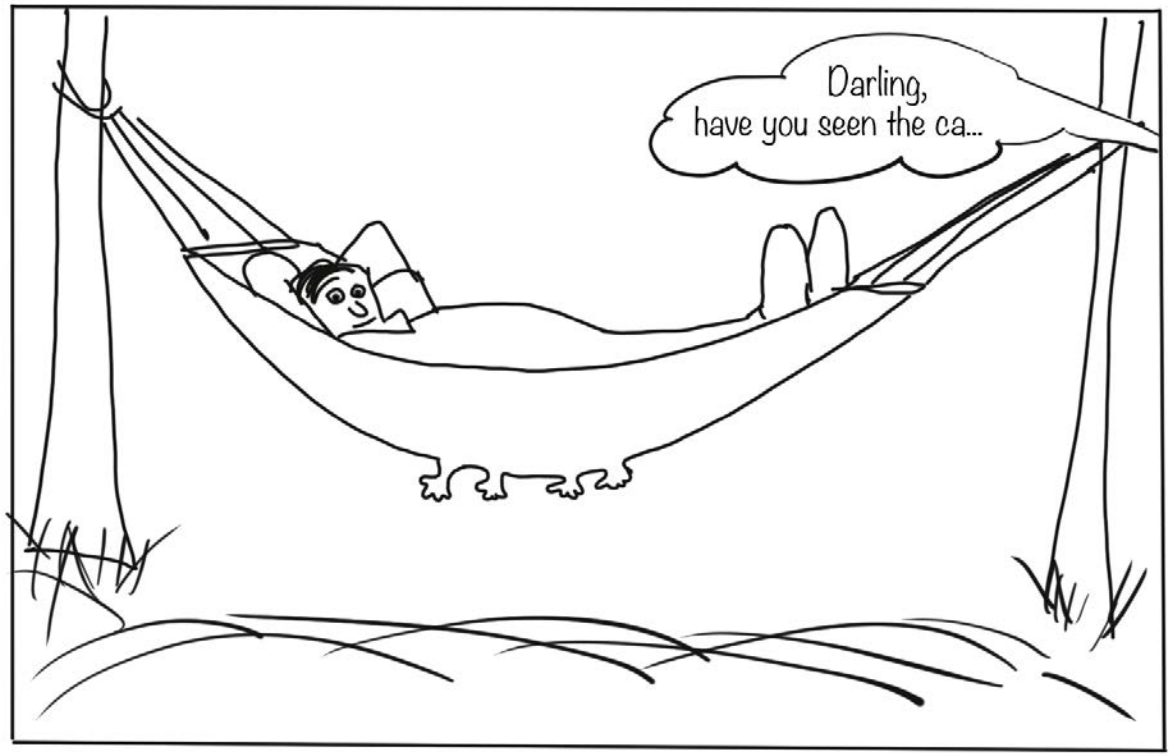
Totally bored?



Keep on working on your case study!

Totally content?

Then go grab a coffee, lean back and enjoy the rest of the day...!



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Thank You

For more information contact me: saskia.otto@uni-hamburg.de

http://www.researchgate.net/profile/Saskia_Otto
http://www.github.com/saskiaotto

Image on title and end slide: Section of an infrared satallite image showing the Larsen C ice shelf on the Antarctic Peninsula - USGS/NASA Landsat: A Crack of Light in the Polar Dark, Landsat 8 - TIRS, June 17, 2017
(under CC0 license)