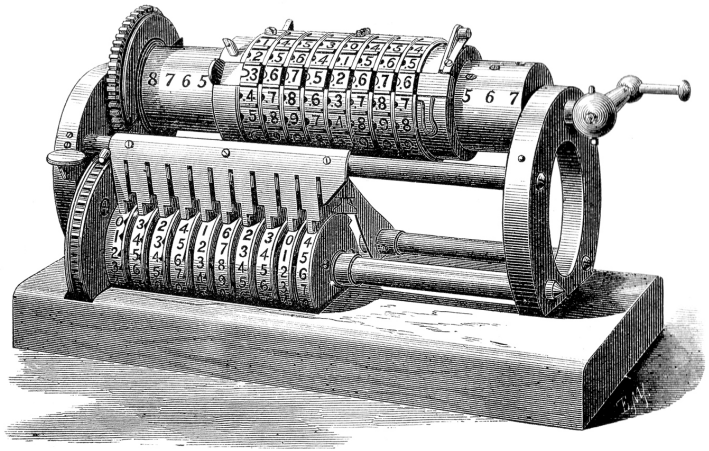
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Programming in R

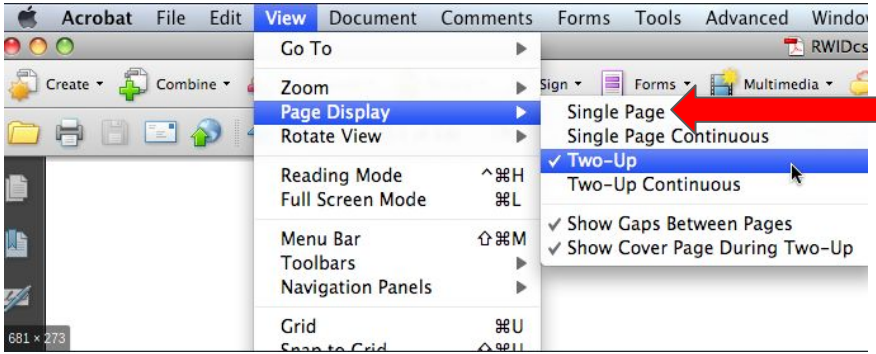
Binder

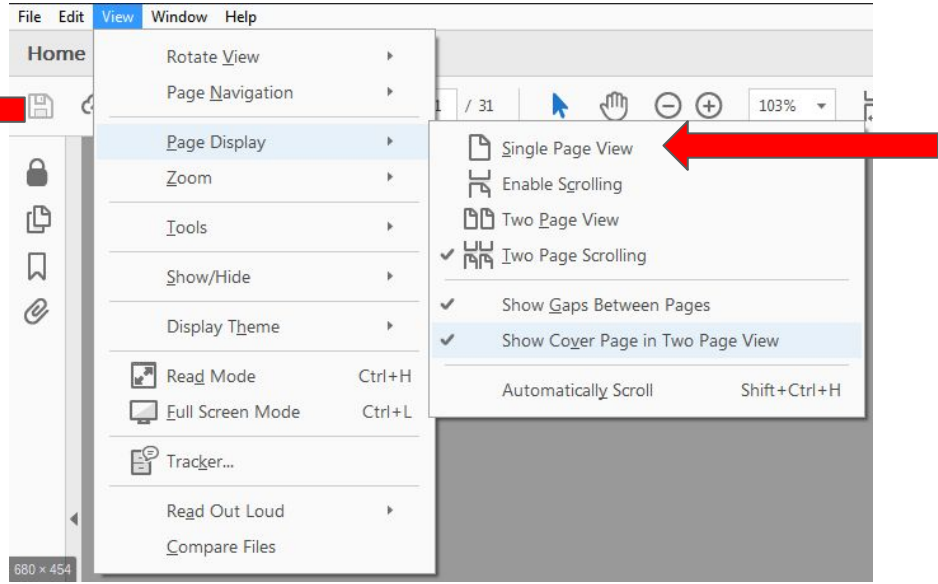


Unit 5:
Strings and Regex

About these Slides

The best way to view these slides (and to get the most out of the animations) is to view it in "presentation mode" or "single page view". See the images below where to set up single page view in your system.





Debugging and Modular Software

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "R" Track: String operations and regex (in base-R)
- "Dev" Track: Modular Programming

R Track

String Operations and Regex

Regular Expressions (RegEx)

Typical use-case when using character strings: does it follow a certain **pattern**, or does it contain something matching that pattern? Examples:

- Given vector of file names:

  ```
  c("analysis.R", "bibliography.doc", "data.zip", "load_data.R", "thesis.doc")
  ```

  which of these filenames have the file-ending ".doc"?
- Analysing the response of a questionnaire, have the respondents entered a valid email-address (one or more numbers / letters / certain special characters, followed by the '@'-sign, followed by more numbers / letters / special characters and at least one dot, etc.*)
- What are all the URLs (starting with "http" and indicating a web-address) contained in a given text?

* actual rules for valid email-addresses are more complicated

Regular Expressions (RegEx)

- **pattern-matching** and related functions in R:

```
grep(),grepl(),

 sub(), regexpr(), regexec(),
gsub(),gregexpr(),gregexec(),

regmatches()
```

- all (except `regmatches()`) have a **`pattern`** and a **`text`** / **`x`** argument
- Search for **`pattern`** in a given **`text`**

Regular Expressions (RegEx)

- Simplest pattern: plain text, e.g. "ab".
- pattern matching functions search for occurrence of *pattern* in *text* and return information about match
  - which element of a vector matched (grep, grepl)
  - position inside a string where a match occurred (regexpr, gregexpr, regexec)
  - positions of subgroups (regexec, gregexec) (we see this later).
- All these functions have extra arguments:
  - Argument '`ignore.case`': count uppercase and lowercase the same
  - Argument '`fixed`': ignore all special regex characters (next slide) and treat all patterns as plain text
  - Argument '`perl`': use extended "perl" regex (see following slides)

grep: which vector element matches with pattern?

grep with 'value = TRUE': return value that matches

grepl ('l' for 'logical'): return truth vector whether element matches

sub: substitute *first* occurrence (in each vector element)

gsub: substitute *all* occurrences ('g' for 'global')

regexpr: find the *position* of the *first* match (2, in this case). The 'match.length' attribute gives the *length* (here 2: two letters). '-1' means 'not found'.

gregexpr: like regexpr, but "global" -- give positions (and match lengths) of *all* matches.
Note: gregexpr returns a **list** with one element for each element of input text (here: 3 elements).

Missing here: **(g)regexec**, we will see it later.

```
> grep("ab", c("xyz", "abc", "abab"))
[1] 2 3
> grep("ab", c("xyz", "abc", "abab"), value = TRUE)
[1] "abc"  "abab"
> grepl("ab", c("xyz", "abc", "abab"))
[1] FALSE  TRUE  TRUE
> sub("ab", "123", c("xyz", "abc", "abab"))
[1] "xyz"   "123c"  "123ab"
> gsub("ab", "123", c("xyz", "abc", "abab"))
[1] "xyz"    "123c"   "123123"
> regexpr("ab", c("xyz", "abc", "abab"))
[1] -1  1  1
attr(,"match.length")
[1] -1  2  2
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
> gregexpr("ab", c("xyz", "abc", "abab"))
[[1]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

[[2]]
[1] 1
attr(,"match.length")
[1] 2
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

[[3]]
[1] 1 3
attr(,"match.length")
[1] 2 2
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

In this example, gregexpr found 2 matches in the 3rd string ("abab"), at position 1 and 3, each of length 2.

Regular Expressions (RegEx)

- Patterns can be more than just text; some characters have *special meaning*
- Example:
  - The dot (".") matches *any character*
  - The asterisk ("*") means: The last character (or group), repeated arbitrarily often (even 0 times)
  - Often combined: ".*" (dot asterisk) means: any character, repeated any number of times
  - Double backslash ("\\"): ignore the special meaning of the next special character.
    Note: R uses double backslash where some online guides for other languages use a single backslash!

```
> grep("a.c", c("abc", "axc", "acb"), value = TRUE)
[1] "abc" "axc"
> grep("ab*c", c("abc", "abbbbc", "ac", "ab"), value = TRUE)
[1] "abc"    "abbbbc" "ac"

> grep("a.*b", c("ab", "a<anything>b", "a"), value = TRUE)
[1] "ab"            "a<anything>b"
> grep("a\\.c", c("abc", "axc", "a.c"), value = TRUE)
[1] "a.c"
```

- There is no point in listing all special characters here. Use a resource such as (in no particular order) R help, regex reference, Interactive regex tester, another interactive regex tester, RStudio Cheat Sheet or PCRE Specs.
  (Now is a good time to read some of the material linked in the homework!)

"regmatches()" Function

- **regexpr()**, **gregexpr()** are a bit cumbersome: We get the ***positions*** of matches, but not the *content* of matches.

```
> string <- "This string contains the numbers 3, 13, and 1818"
> gregexpr("[0-9]+", string)
[[1]]
[1] 34 37 45
attr(,"match.length")
[1] 1 2 4
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

"**[0-9]+**": matches any consecutive string of digits (i.e. characters between 0 and 9, inclusive)
➜ positive integer numbers!

"regmatches()" Function

- **regexpr()**, **gregexpr()** are a bit cumbersome: We get the ***positions*** of matches, but not the *content* of matches.
- **regmatches()** solves this for us: It takes the result of regexpr(), gregexpr() and regexec() (i.e., it takes the *positions* of matches), as well as the input string, and returns the ***content*** of matches.

```
> string <- "This string contains the numbers 3, 13, and 1818"
> gregexpr("[0-9]+", string)
[[1]]
[1] 34 37 45
attr(,"match.length")
[1] 1 2 4
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

> regmatches(string, gregexpr("[0-9]+", string))
[[1]]
[1] "3"    "13"   "1818"
```

"**[0-9]+**": matches any consecutive string of digits (i.e. characters between 0 and 9, inclusive)
➜ positive integer numbers!

"regmatches()" Function

- **regexpr()**, **gregexpr()** are a bit cumbersome: We get the ***positions*** of matches, but not the *content* of matches.
- **regmatches()** solves this for us: It takes the result of regexpr(), gregexpr() and regexec() (i.e., it takes the *positions* of matches), as well as the input string, and returns the ***content*** of matches.
- This *vectorizes*: If input (for regexpr(), gregexpr(), or regexec()) has multiple elements, then the output has too.
- Note that, for gregexpr(), regexec(), gregexec(), the output is a *list*!

```
> string <- "This string contains the numbers 3, 13, and 1818"
> gregexpr("[0-9]+", string)
[[1]]
[1] 34 37 45
attr(,"match.length")
[1] 1 2 4
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

"**[0-9]+**": matches any consecutive string of digits (i.e. characters between 0 and 9, inclusive) ➜ positive integer numbers!

```
> regmatches(string, gregexpr("[0-9]+", string))
[[1]]
[1] "3"    "13"   "1818"
```

```
> strings <- c("numbers 10, 20", "number 3")
> regmatches(strings, gregexpr("[0-9]+", strings))
[[1]]
[1] "10" "20"

[[2]]
[1] "3"
```

Both matches from first element of **strings**

Single match from second element of **strings**

"regmatches()" Function

- **regexpr()**, **gregexpr()** are a bit cumbersome: We get the ***positions*** of matches, but not the *content* of matches.
- **regmatches()** solves this for us: It takes the result of regexpr(), gregexpr() and regexec() (i.e., it takes the *positions* of matches), as well as the input string, and returns the ***content*** of matches.
- This *vectorizes*: If input (for regexpr(), gregexpr(), or regexec()) has multiple elements, then the output has too.
- Note that, for gregexpr(), regexec(), gregexec(), the output is a *list*!
- Also does *assignment* to matches!

```
> string <- "This string contains the numbers 3, 13, and 1818"
> gregexpr("[0-9]+", string)
[[1]]
[1] 34 37 45
attr(,"match.length")
[1] 1 2 4
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

"[0-9]+": matches any consecutive string of digits (i.e. characters between 0 and 9, inclusive) ➜ positive integer numbers!

```
> regmatches(string, gregexpr("[0-9]+", string))
[[1]]
[1] "3"    "13"   "1818"
```

```
> strings <- c("numbers 10, 20", "number 3")
> regmatches(strings, gregexpr("[0-9]+", strings))
[[1]]
[1] "10" "20"
```

Both matches from first element of **strings**

```
[[2]]
[1] "3"
```

Single match from second element of **strings**

```
> regmatches(strings, gregexpr("[0-9]+", strings)) <- "000"
> strings
[1] "numbers 000, 000" "number 000"
```

(Groups) in Regex

Groups: delimited by "(" and ")", have multiple purposes:

1.: function as parenthesis for | (alternation).
- "xa|by" matches either "xa" or "by".
- "x(a|b)y" matches either "xay" or "xby".

2.: Repetition operators / quantifiers (i.e. ?, *, +, {}) repeat the whole group, instead of a single character:
- "x(ab)*y" matches "xy", "xaby", "xababy", "xabababy", ...

3.: Back references (\\1, \\2, ...) can refer to the group number 1, number 2, etc. coming before it
- "x(.)y\\1" matches anything that has "x", followed by any character, followed by "y", followed by the same character that came after "x".

"xay" matches, because it contains "xa".

"xaby" matches, it contains both "xa" and "by".

```
> grep("xa|by", c("xa", "xay", "xaby"), value = TRUE)
[1] "xa"   "xay"  "xaby"
> grep("x(a|b)y", c("xa", "xay", "xaby"), value = TRUE)
[1] "xay"
> grep("x(ab)*y", c("xaby", "xababy", "xaabby", "xabby"), value = TRUE)
[1] "xaby"   "xababy"
> grep("x(.)y\\1", c("xaya", "xayb"), value = TRUE)
[1] "xaya"
```

Sometimes you want to use parentheses for function 1. or 2. *without* creating a backreference (function 3.). In this case, you can start a group with `(?:<text>)` instead of `(<text>)`, and it will not take a back-reference slot.

(Groups) and regexec()

- We haven't covered **`regexec()`** yet:
  - It works similar to regexpr(), but it returns the positions of entire match *as well as group matches*
  - Most useful in combination with **regmatches()**.
  - This example extracts the year, month and day from an ISO 8601 date it finds in string:

```
> string <-   "The data was collected on 2021-04-30"
> regmatches(string, regexec("([0-9]{4})-([0-9]{2})-([0-9]{2})", string))
[[1]]
[1] "2021-04-30" "2021"       "04"         "30"
```

{2}: Match exactly two repetitions of **[0-9]**

The whole match to the regex

... followed by the 1st group

... followed by the 2nd etc.

- **`gregexec()`**: finds all matches and their groups. It returns a list of *matrices*, instead of a list of *vectors*. Every column corresponds to a match, the *i*'th row is for the *(i-1)*th group (first row is the entire match).
- If you want to use some of the `(<text>)` for grouping *without* capturing them, use `(?:<text>)` for these (just as for groups without backreferences, see previous slide).

Extended "Perl" Regular Expressions

- By default, pattern matching functions use "POSIX 1003.2 extended regular expressions"
- With argument 'perl = TRUE', uses "Perl-compatible" regular expressions instead.
- Most notable differences:
  - perl = TRUE has **lookahead** & **lookbehind**
  - perl = TRUE has **named groups** and **named backreferences**
  - (See this table and this website for more detailed info)

Named Groups / Backreferences (perl regex)

- Name groups `(`<text to match>`)` by adding '`?<>`': `(?<`<name of the group>`>`<text to match>`)`
- Backreference it, instead by `\\1`, `\\2` etc., by `\\k<`<name of the group>`>`.

First encountered character is group '`x`', second is group '`yz`'. Content of the groups is '`.`' (i.e. arbitrary character).

```
> grepl("(?<x>.)(?<yz>.)\\k<yz>\\k<x>", c("xyyx", "xyxy"), perl = TRUE)
[1]  TRUE FALSE
```

After the groups, the '`yz`'-group should be seen again, then the '`x`' group should be seen.

- Group names are used as vector / row names in **(g)regexec() + regmatches()**

```
> regmatches(string, regexec("([0-9]{4})-([0-9]{2})-([0-9]{2})", string))
[[1]]
[1] "2021-04-30" "2021"       "04"         "30"

> regmatches(string, regexec("(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})", string, perl = TRUE))
[[1]]
                    year       month         day
"2021-04-30"       "2021"        "04"        "30"
```

Lookahead & Lookbehind (perl regex)

- Lookahead / lookbehind: condition on presence of characters, without making them part of the match.
  - Lookahead: `(?=`<text to match>`)` at the *end* of a regex
  - Lookbehind: `(?<=`<text to match>`)` at the *beginning* of a regex

```
> string <- "``Sure,'' he said, ``I will come.''"
> regmatches(string, gregexpr("(?<=``).*?(?='')", string, perl = TRUE))
[[1]]
[1] "Sure,"        "I will come."
```

`*?` is similar to `*`, with the difference: `*` matches as *many* characters as possible, `*?` matches as *few* as possible (called 'non-greedy capture'). Small exercise: What happens if `?` is omitted here, and why?

The string `` must come before the match

The string '' must come after the match

Result: The direct quotes from the string.

Lookahead & Lookbehind (perl regex)

- Lookahead / lookbehind: condition on presence of characters, without making them part of the match.
  - Lookahead: `(?=`<text to match>`)` at the *end* of a regex
  - Lookbehind: `(?<=`<text to match>`)` at the *beginning* of a regex

```
> string <- "``Sure,'' he said, ``I will come.''"
> regmatches(string, gregexpr("(?<=``).*?(?='')", string, perl = TRUE))
[[1]]
[1] "Sure,"        "I will come."
```

- **Negative** lookahead / lookbehind: condition on *absence* of characters
  - Negative lookahead: `(?!`<text to match>`)` at the *end* of a regex
  - Negative lookbehind: `(?<!`<text to match>`)` at the *beginning* of a regex
  - Example use-case: Detect something when it is an entire word, but not when it is part of a word*:

Remember: '!' usually means 'not' in R

```
> strings <- c("wall", "mall", "allied", "all", "that is all of them")
> grep("(?<![a-z])all(?![a-z])", strings, value = TRUE, perl = TRUE)
[1] "all"                "that is all of them"
```

If we used `[^a-z]all[^a-z]`, then `"all"` wouldn't match!

Must not match with `[a-z]` before or after

* This particular case can also be solved with word boundaries. The regex for that is `"\\ball\\b"`. Try it!

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("**/**")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("**/**")

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("**/**")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("**/**")

Regex:

```
<PROTOCOL><DOMAIN>/<PATH>
```

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("**/**")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("**/**")

Regex:

```
https?://<DOMAIN>/<PATH>
```

's' occurs 0 or 1 times

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("/")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("/")

Regex:

```
(?<![[:alpha:]])https?://<DOMAIN>/<PATH>
```

Negative lookbehind!

[:alpha:] is a POSIX character class of all letters. Use double square brackets, because we use this in a regex character class.

All this means: "not preceded by a letter".

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("/")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("/")

Regex:

```
(?<![[:alpha:]])https?://LABEL(\\.LABEL)+/<PATH>
```

The first **LABEL** occurs exactly once

... followed by at least one **\\.LABEL**

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("/")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("/")

Regex:

```
(?<![[:alpha:]])https?://LABEL(\\.LABEL)+/<PATH>
```

```
LABEL ➜ [[:alnum:]][-[:alnum:]]*
```

Alpha-numerics (letters + numbers)

Alpha-numerics and the minus-sign

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("/")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("/")

Regex:

```
(?<![[:alpha:]])https?://[[:alnum:]][-[:alnum:]]* (\\.[[:alnum:]][-[:alnum:]]* )+/<PATH>
```

```
LABEL ➜ [[:alnum:]][-[:alnum:]]*
```

Alpha-numerics (letters + numbers)

Alpha-numerics and the minus-sign

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("/")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("/")

Regex:

```
(?<![[:alpha:]])https?:// [[:alnum:]][-[:alnum:]]* (\\.[[:alnum:]][-[:alnum:]]* )+/[-/[:alnum:]]*
```

Example: URL

- Example to find: URL (i.e. a typical web address)
- We simplify the conditions here. Our simplified URL:
  - Starts with a "protocol indicator"
    - This is either **http://** or **https://**
    - But not preceded by a character (**ahttp://** does not count, **<http://** does count)
  - Followed by a "domain name"
    - a "domain name" two or more "label"s, separated by a dot ("`\\.`" for regex)
      - a "label" consists of letters, numbers, or minus-signs, but does not start with a minus-sign
  - Followed by a forward slash ("**/**")
  - Followed by a "path"
    - A "path" contains letters, numbers, minus-signs, and forward slashes ("**/**")

Regex:

```
(?<![[:alpha:]])https?:// [[:alnum:]][-[:alnum:]]* (\\.[[:alnum:]][-[:alnum:]]* )+/[-/[:alnum:]]*
```

Example: URL

What if we want to find sub-parts of the URL, such as the "domain name"?

Regex:

```
(?<![[:alpha:]])https?:// [[:alnum:]][-[:alnum:]]* (\\.[[:alnum:]][-[:alnum:]]* )+/[-/[:alnum:]]*
```

Example: URL

What if we want to find sub-parts of the URL, such as the "domain name"?

- Put group (i.e. parentheses) around domain!

Regex:

```
(?<![[:alpha:]])https?:// ([[:alnum:]][-[:alnum:]]* (\\.[[:alnum:]][-[:alnum:]]* )+) /[-/[:alnum:]]*
```

Example: URL

What if we want to find sub-parts of the URL, such as the "domain name"?

- Put group (i.e. parentheses) around domain!
- The parentheses we use for label are not for extraction, so we turn them into groups without reference using `(?:<text>)`.

Regex:

```
(?<![[:alpha:]])https?:// ([[:alnum:]][-[:alnum:]]* (?:\\.[[:alnum:]][-[:alnum:]]* )+)/[-/[:alnum:]]*
```

Example: URL

What if we want to find sub-parts of the URL, such as the "domain name"?

- Put group (i.e. parentheses) around domain!
- The parentheses we use for label are not for extraction, so we turn them into groups without reference using `(?:<text>)`.
- Use **regexec()**!

Regex:

```
(?<![[:alpha:]])https?:// ([[:alnum:]][-[:alnum:]]* (?:\\.[[:alnum:]][-[:alnum:]]* )+) /[-/[:alnum:]]*
```

```
> string <- "The site is https://www.google.com/search!"
> rex <- "(?<![[:alpha:]])https?://([[:alnum:]][-[:alnum:]]*(?:\\.[[:alnum:]][-[:alnum:]]*)+)/[-/[:alnum:]]*"
> regmatches(string, regexec(rex, string, perl = TRUE))
[[1]]
[1] "https://www.google.com/search" "www.google.com"
```

Example: URL

What if we want to find sub-parts of the URL, such as the "domain name"?

- Put group (i.e. parentheses) around domain!
- The parentheses we use for label are not for extraction, so we turn them into groups without reference using `(?:<text>)`.
- Use **regexec()**!
- You can extend this by:
  - using **gregexec()** to find multiple matches at once
  - using named groups (see before) to get named results

Regex:

```
(?<![[:alpha:]])https?:// ([[:alnum:]][-[:alnum:]]* (?:\\.[[:alnum:]][-[:alnum:]]* )+) /[-/[:alnum:]]*
```

```
> string <- "The site is https://www.google.com/search!"
> rex <- "(?<![[:alpha:]])https?://([[:alnum:]][-[:alnum:]]*(?:\\.[[:alnum:]][-[:alnum:]]*)+)/[-/[:alnum:]]*"
> regmatches(string, regexec(rex, string, perl = TRUE))
[[1]]
[1] "https://www.google.com/search" "www.google.com"
```

How To Build a Regex

- Know the basic building blocks (see next slide!)
- Experiment a lot, with real strings
  - With strings you *want* to match
  - Also with strings you *do not want to match*, to make sure your regex is not too permissive (see also)
- There are some helpful interactive regex experimenting websites, such as https://regexr.com/ or https://www.debuggex.com/
- AI can help: https://www.autoregex.xyz/
- Stackoverflow can always help
  - Especially when trying to match something complicated, like *actual* valid URLs or Email addresses.

What We Expect You to Know

- Pattern matching functions: **grep()**, **grepl()**, **sub()**, **gsub()**, **regexpr()**, **gregexpr()**, **regexec()**, **gregexec()**
- Function of their arguments **ignore.case**, **perl**, **fixed**
- Use **regmatches()** to extract matching strings
  - including (possibly named) groups with **regexec()**
- Regex special symbols: . | + * ? ^ $ \\ (...) [...] [^...] [a-z] [[:...:]] {n} {n,m}
- Groups with (...); named groups: (?<name>:...); groups without reference: (?:...)
- Lookahead and lookbehind, both positive and negative: (?=...), (?!...), (?<=...), (?<!...)