Working with Strings in R

Stats 102A

Miles Chen

Department of Statistics and Data Science

UCLA

Introduction

Most of statistical computing involves working with numeric data. However, many modern applications have considerable amounts of data in the form of text.

There are whole areas of statistics and machine learning devoted to organizing and interpreting text-based data, such as textual data analysis, linguistic analysis, text mining, sentiment analysis, and natural language processing (NLP).

For more information and resources:

- https://cran.r-project.org/web/views/NaturalLanguageProcessing.html
- https://www.tidytextmining.com/

Text-based analyses are beyond the scope of this course.

Resources

We will discuss the most common syntax and functions for string manipulation and regular expressions in R. For more information and resources:

Books and Articles

- Gaston Sanchez's "Handling Strings with R": https://www.gastonsanchez.com/r4strings/
- Garrett Grolemund and Hadley Wickham's "R for Data Science": http://r4ds.had.co.nz/strings.html
- https://en.wikibooks.org/wiki/R_Programming/Text_Processing
- https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html

Cheat Sheets for `stringr` and Regular Expressions

- Strings Cheat Sheet at: https://posit.co/resources/cheatsheets/
- https://www.cheatography.com//davechild/cheat-sheets/regular-expressions/pdf/

Sites for Testing Regular Expressions

- https://regex101.com/
- https://regexr.com/

Section 1

Characters in R

Characters in R

Symbols in R that represent text or words are called **characters**.

A **string** is a character variable that contains one or more characters, but we often will use "character" and "string" interchangeably.

Values that are stored as characters have base type `character` and are typically printed with quotation marks.

```
x <- "Hello World"
x
```

```
[1] "Hello World"
```

```
typeof(x)
```

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

Creating Character Strings

Characters can be created using single or double quotation marks.

Single quotation marks can be used within double quotation marks and vice versa, but you cannot directly insert single quotes within single quotes or double quotes within double quotes.

```
"This is the 'R' Language"
```

```
[1] "This is the 'R' Language"
```

```
'This is the "R" Language'
```

```
[1] "This is the \"R\" Language"
```

```
"This is an "error""
```

```
## Error: unexpected symbol in ""This is an "error"
```

Note: The double quotation inside a string is a special character, which is why it prints with a backslash \".

Empty Characters

The `character()` function creates a character vector of a specified length. The default value in each element of the vector is the **empty character** `""`.

```
character(5)
```

```
[1] "" "" "" "" ""
```

Note: The empty character `""` is **not the same** as `character(0)` which is a character vector of length 0. `c("")` is identical to `character(1)`

The paste() Function

The **paste()** function is one of the most important functions for creating and building strings.

The paste() function inputs one or more R objects, converts them to character, and then concatenates (pastes) them to form one or several character strings.

The basic syntax is:

```
paste(..., sep = " ", collapse = NULL)
```

- The ... argument means the input can be any number of objects.
- The optional **sep** argument specifies the separator between characters after pasting. The default is a single whitespace " ".
- The optional **collapse** argument specifies characters to separate the result.

The **paste0()** function is equivalent to using paste(..., sep = "", collapse)

The paste() Function

```
paste("I ate some", pi, "and it was delicious.")
```

```
[1] "I ate some 3.14159265358979 and it was delicious."
```

```
paste("Bears", "Beets", "Battlestar Galactica", sep = ", ")
```

```
[1] "Bears, Beets, Battlestar Galactica"
```

```
paste("h", c("a", "e", "i"), sep = "") # No collapsing produces vector
```

```
[1] "ha" "he" "hi"
```

```
paste(c("a", "b", "c"), 1:3, sep = "") # pastes element-wise
```

```
[1] "a1" "b2" "c3"
```

```
paste("a", 1:3, sep = "", collapse = ", ") # recycles + collapses
```

```
[1] "a1, a2, a3"
```

Note: Partial recycling will not throw a warning.

Print Functions for Characters

There are several functions to output strings:

- `print()` is for generic printing.
- `cat()` is for concatenation.
- `format()` is for (pretty) printing with special formatting.

The `print()` and `noquote()` Functions

The **`print()`** function (technically the `print.default()` method) has an optional logical `quote` argument that specifies whether to print characters with or without quotation marks.

A similar output can be produced using **`noquote()`**.

```
print(x, quote = FALSE)
```

```
[1] Hello World
```

```
noquote(x)
```

```
[1] Hello World
```

Note: While the output appears identical, the commands are not the same. The `noquote()` function outputs a `noquote` class object, which is then inputted into the `print.noquote()` method.

The cat() Function

The **cat()** funtion concatenates multiple character vectors into a single vector, adds a specified separator, and outputs the result (without quotations).

```
cat(x, "Hello Universe", sep = ", ")
```

```
Hello World, Hello Universe
```

The printing is slightly different from that of `noquote()`. In particular, the printed output does not have the vector index, and the `cat()` function returns an invisible `NULL` (meaning assigning the printed output to a variable does not work).

The cat() Function

One benefit of `cat()` is that the printed output can be saved to an external file using the **`file`** argument:

```
cat(x, "Hello Universe",
    sep = ", ", file = "hello.txt"
)
```

When `file` is specified, an optional logical argument **`append`** specifies whether the result should be appended to or overwrite an existing file.

Side Note: There are a few other optional arguments that are useful for longer text strings. Consult the R documentation for more information.

The `format()` Function

The **`format()`** function formats an R object for "pretty" printing.

Some useful arguments used in `format()`:

- **`width`** specifies the (minimum) width of strings produced.
- **`trim`** specifies whether there should be no padding with spaces (`TRUE`).
- **`justify`** controls how padding takes place for strings. Takes the values `"left"`, `"right"`, `"centre"`, or `"none"`.

For controling the printing of numbers:

- **`digits`** specifies the number of significant digits to use.
- **`nsmall`** specifies the minimum number of digits to the right of the decimal point to include.
- **`scientific`** specifies whether to use scientific notation (`TRUE`) or standard notation (`FALSE`).

The format() Function

```
format(1 / (1:5), digits = 2)
```

```
[1] "1.00" "0.50" "0.33" "0.25" "0.20"
```

```
format(1 / (1:5), digits = 2, scientific = TRUE)
```

```
[1] "1.0e+00" "5.0e-01" "3.3e-01" "2.5e-01" "2.0e-01"
```

```
format(c("Hello", "world", "Hello", "Universe"),
  width = 10, justify = "left"
)
```

```
[1] "Hello     " "world     " "Hello     " "Universe  "
```

Section 2

Basic String Manipulation

Functions for Basic String Manipulation

There are many functions in base R for basic string manipulation.

- `nchar()` Returns number of characters
- `tolower()` Converts to lower case
- `toupper()` Converts to upper case
- `chartr()` Translates characters
- `substr()` Extracts substrings of a character vector
- `strsplit()` Splits strings into substrings

Examples of Basic String Manipulation

```
y <- c("Hello", "World", "Hello", "Universe")
nchar(y)
```

```
[1] 5 5 5 8
```

```
tolower(y)
```

```
[1] "hello"    "world"    "hello"    "universe"
```

```
toupper(y)
```

```
[1] "HELLO"    "WORLD"    "HELLO"    "UNIVERSE"
```

Examples of Basic String Manipulation

```
chartr("H", "y", y)
```

```
[1] "yello"    "World"    "yello"    "Universe"
```

```
chartr("elio", "31!0", y)
```

```
[1] "H3110"    "W0r1d"    "H3110"    "Un!v3rs3"
```

Examples of Basic String Manipulation

```
substr(x, 2, 9)
```

```
[1] "ello Wor"
```

```
strsplit(x, split = " ")
```

```
[[1]]
[1] "Hello" "World"
```

```
strsplit(x, split = "o")
```

```
[[1]]
[1] "Hell" " W"   "rld"
```

Section 3

stringr

The stringr Package

The **stringr** package is part of the core tidyverse: it is automatically loaded by typing `library(tidyverse)` or alone with `library(stringr)`.

```
library(tidyverse)
```

The functions in `stringr` are intended to make R's string functions more consistent, simpler, and easier to use. The primary ways it does this are:

- Function and argument names (and positions) are consistent.
- All functions deal with `NA`s and zero length characters appropriately.
- The output data structures from each function matches the input data structures of other functions.

Basic stringr Functions

The stringr package has functions for basic manipulation and for regular expression operations.

The main stringr functions for basic manipulation are shown below.

str_c() String concatenation, similar to paste() **str_length()** Number of characters, similar tonchar() **str_sub()** Extracts substrings similar to substr() **str_dup()** Duplicates characters **str_trim()** Removes leading and trailing whitespace **str_pad()** Pads a string **str_wrap()** Wraps a string paragraph **str_trim()** Trims a string

Examples and documentation can be found in:

- https://www.gastonsanchez.com/r4strings/stringr-basics.html
- https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html

The str_c() Function

The **str_c()** function automatically removes NULL arguments.

```
paste(x, NULL, c("Hello Universe"),
      sep = ", "
)
```

```
[1] "Hello World, , Hello Universe"
```

```
str_c(x, NULL, c("Hello Universe"),
      sep = ", "
)
```

```
[1] "Hello World, Hello Universe"
```

Note: The default separator for paste() is whitespace (sep=" ") and for str_c() is the empty string (sep="").

The str_length() Function

The **str_length()** function works with factors, whereas nchar() does not.

```
nchar(factor(month.name))
```

```
Error in `nchar()`:
! 'nchar()' requires a character vector
```

```
str_length(factor(month.name))
```

```
 [1] 7 8 5 5 3 4 4 6 9 7 8 8
```

Section 4

Regular Expressions

Regular Expressions

One main application of string manipulation is pattern matching. Finding patterns in text are useful for data validation, data scraping, text parsing, filtering search results, etc.

A **regular expression** (or **regex**) is a set of symbols that describes a text pattern. More formally, a regular expression is a pattern that describes a set of strings.

Regular expressions are a formal language in the sense that the symbols have a defined set of rules to specify the desired patterns. The best way to learn the syntax and become fluent with regular expressions is to practice.

Applications of Regular Expressions

Some common applications of regular expressions:

- Test if a phone number has the correct number of digits
- Test if a date follows a specifc format (e.g. mm/dd/yy)
- Test if an email address is in a valid format
- Test if a password has numbers and special characters
- Search a document for gray spelled either as “gray” or “grey”
- Search a document and replace all occurrences of “Will”, “Bill”, or “W.” with “William”
- Count the number of times in a document that the word “analysis” is immediately preceded by the words “data”, “computer”, or “statistical”
- Convert a comma-delimited file into a tab-delimited file
- Find duplicate words in a text

stringr Functions for Regular Expressions

R has native regex handling capabilities (e.g. **grep()**), but stringr has made their usage easier and more consistent.

Function	Description
`str_detect(str, pattern)`	Detect the presence of a pattern and returns TRUE if it is found
`str_locate(str, pattern)`	Locate the 1st position of a pattern and return a matrix with start & end.
`str_extract(str, pattern)`	Extracts text corresponding to the first match.
`str_match(str, pattern)`	Extracts capture groups formed by () from the first match.
`str_split(str, pattern)`	Splits string into pieces and returns a list of character vectors.

Many of these functions have variants with an _all suffix which will match more than one occurrence of the pattern in a given string.

Matching Literal Character Strings

The most basic type of regular expressions are **literal character strings**, which are strings that match themselves.

A literal character match is one in which a given character such as the letter `"R"` matches the letter R. This type of match is the most basic type of regular expression operation: just matching plain text.

All the letters and digits in the English alphabet (i.e., alphanumeric characters) are considered literal characters because, as regular expressions, they match themselves.

Matching Literal Characters

Literal character matching is case sensitive.

```
str_locate("I love stats", "stat")
```

Matching Literal Characters

Literal character matching is case sensitive.

```
str_locate("I love stats", "stat")
```

```
     start end
[1,]     8  11
```

Matching Literal Characters

Literal character matching is case sensitive.

```
str_locate("I love stats", "stat")
```

```
     start end
[1,]     8  11
```

```
str_locate("I love Stats", "stat")
```

Matching Literal Characters

Literal character matching is case sensitive.

```
str_locate("I love stats", "stat")
```

```
     start end
[1,]     8  11
```

```
str_locate("I love Stats", "stat")
```

```
     start end
[1,]    NA  NA
```

Matching Literal Characters

The `str_locate()` function only returns the first occurrence of a match. To find all matches, use `str_locate_all()`.

```
love_stats <- "I love statistics, so I am a stats major."
```

```
str_locate(love_stats, "stat")
```

Matching Literal Characters

The `str_locate()` function only returns the first occurrence of a match. To find all matches, use `str_locate_all()`.

```
love_stats <- "I love statistics, so I am a stats major."
```

```
str_locate(love_stats, "stat")
```

```
     start end
[1,]     8  11
```

Matching Literal Characters

The `str_locate()` function only returns the first occurrence of a match. To find all matches, use `str_locate_all()`.

```
love_stats <- "I love statistics, so I am a stats major."
```

```
str_locate(love_stats, "stat")
```

```
     start end
[1,]     8  11
```

```
str_locate_all(love_stats, "stat")
```

Matching Literal Characters

The `str_locate()` function only returns the first occurrence of a match. To find all matches, use `str_locate_all()`.

```
love_stats <- "I love statistics, so I am a stats major."
```

```
str_locate(love_stats, "stat")
```

```
     start end
[1,]     8  11
```

```
str_locate_all(love_stats, "stat")
```

```
[[1]]
     start end
[1,]     8  11
[2,]    30  33
```

Metacharacters

Not all characters match themselves. Any character that is not a literal character is a **metacharacter**.

The power of regular expressions comes from the ability to use a number of special metacharacters that modify how the pattern matching is performed.

The list of metacharacters used in regular expressions is given below:

```
. ^ $ * + ? { } [ ] \ | ( )
```

The Wild Metacharacter

The **dot** (or **period**) **.** is called the **wild** metacharacter (sometimes the **wildcard**). This metacharacter is used to match ANY (single) character except a new line.

```
not <- c("not", "note", "knot", "nut")
```

```
str_detect(not, "n.t")
```

The Wild Metacharacter

The **dot** (or **period**) **.** is called the **wild** metacharacter (sometimes the **wildcard**). This metacharacter is used to match ANY (single) character except a new line.

```
not <- c("not", "note", "knot", "nut")
```

```
str_detect(not, "n.t")
```

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

The Wild Metacharacter

The **dot** (or **period**) **.** is called the **wild** metacharacter (sometimes the **wildcard**). This metacharacter is used to match ANY (single) character except a new line.

```
not <- c("not", "note", "knot", "nut")
```

```
str_detect(not, "n.t")
```

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

```
str_detect(not, "no.")
```

The Wild Metacharacter

The **dot** (or **period**) **.** is called the **wild** metacharacter (sometimes the **wildcard**). This metacharacter is used to match ANY (single) character except a new line.

```
not <- c("not", "note", "knot", "nut")
```

```
str_detect(not, "n.t")
```

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

```
str_detect(not, "no.")
```

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

Question: Consider the following vector.

```
fives <- c("5.00", "5100", "5-00", "5 00")
```

What will `str_detect(fives,"5.00")` return?

Escaping Metacharacters

Because of their special properties, metacharacters cannot be matched directly as literal characters.

To do a literal match, we need to **escape** the metacharacter by adding a backslash \ in front of the metacharacter.

In R, however, since the backlash \ itself is a metacharacter for normal strings, we need a **double backlash** \\ to escape metacharacters in regular expressions.

```
str_detect("abc[def", "[")
```

```
## Error in stri_detect_regex(string, pattern, negate = negate, opts_regex =
opts(pattern)) :    Missing closing bracket on a bracket expression.
(U_REGEX_MISSING_CLOSE_BRACKET)
```

```
str_detect("abc[def", "\\[")
```

```
[1] TRUE
```

Escaping The Wild Metacharacter

Consider the following vector.

```
fives <- c("5.00", "5100", "5-00", "5 00")
str_detect(fives, "5.00")
```

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

To match the literal dot `.` and match only `5.00`, we need to escape the wild metacharacter:

```
str_detect(fives, "5\\.00")
```

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

Character Sets

Square brackets `[ ]` indicate a **character set**, which will match any one of the characters that are inside the set.

A character set will match only one character. The order of the characters inside the set does not matter.

For example, the character set defined by `[aeiou]` will match any one lower case vowel.

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d", "happiness")
```

```
str_detect(pnx, "p[aeiou]n")
```

Character Sets

Square brackets `[ ]` indicate a **character set**, which will match any one of the characters that are inside the set.

A character set will match only one character. The order of the characters inside the set does not matter.

For example, the character set defined by `[aeiou]` will match any one lower case vowel.

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d", "happiness")
```

```
str_detect(pnx, "p[aeiou]n")
```

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

Character Ranges

To match all capital letters (in English), we can define the character set `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]`.

However, this expression is long and inconvenient. Fortunately, the **dash** `-` metacharacter is a shortcut to indicate a **range** of characters.

Some examples:

Pattern	Character Range
`[a-q]`	All lower case letters from a to q
`[A-Q]`	All upper case letters from A to Q
`[a-zA-Z]`	All 52 ASCII letters
`[0-7]`	All digits from 0 to 7

Character Ranges

Character ranges are useful to match various occurrences of a certain type of character.

Question: Consider the following vector.

```
triplets <- c("123", "abc", "ABC", ":-)", "ab12a", "a8908ab")
```

What pattern can be defined to match 3 adjacent digits?

Character Ranges

Character ranges are useful to match various occurrences of a certain type of character.

Question: Consider the following vector.

```
triplets <- c("123", "abc", "ABC", ":-)", "ab12a", "a8908ab")
```

What pattern can be defined to match 3 adjacent digits?

```
str_detect(triplets, "[0-9][0-9][0-9]")
```

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

Similarly, the pattern `[a-z][a-z][a-z]` matches three adjacent lower case letters.

Negative Character Sets

A common situation when working with regular expressions consists of matching characters that are NOT part of a certain set.

This type of matching can be done using a **negative character set**: by matching any one character that is not in the set.

The **caret** `^` metacharacter is used to create negative character sets.

If a caret `^` is placed in the first position inside a character set, it means **negation** (similar to the negative sign in numeric indices or the exclamation point in logical expressions).

For example, the pattern `[^aeiou]` means "not any one of lower case vowels."

Note: The caret `^` is a metacharacter that has more than one meaning depending on where it appears in a pattern.

Negative Character Sets

For example, the pattern `[^A-Z]` will match any character that is NOT an upper case letter.

```
basic <- c("1", "a", "A", "&", "-", "^")
```

```
str_detect(basic, "[^A-Z]")
```

Negative Character Sets

For example, the pattern `[^A-Z]` will match any character that is NOT an upper case letter.

```
basic <- c("1", "a", "A", "&", "-", "^")
```

```
str_detect(basic, "[^A-Z]")
```

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

Negative Character Sets

Caution: It is important that the caret ^ is the first character inside the character set, otherwise the set is not a negative one.

For example, the pattern `[A-Z^]` mean any one upper case letter or the caret character.

```
basic
```

```
[1] "1" "a" "A" "&" "-" "^"
```

```
str_detect(basic, "[A-Z^]")
```

Negative Character Sets

Caution: It is important that the caret ^ is the first character inside the character set, otherwise the set is not a negative one.

For example, the pattern `[A-Z^]` mean any one upper case letter or the caret character.

```
basic
```

```
[1] "1" "a" "A" "&" "-" "^"
```

```
str_detect(basic, "[A-Z^]")
```

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

Question: What pattern means "anything except the caret?"

Negative Character Sets

The pattern `[^\\^]` can be used to mean anything except the caret.

```
basic
```

```
[1] "1" "a" "A" "&" "-" "^"
```

```
str_detect(basic, "[^\\^]")
```

Negative Character Sets

The pattern `[^\\^]` can be used to mean anything except the caret.

```
basic
```

```
[1] "1" "a" "A" "&" "-" "^"
```

```
str_detect(basic, "[^\\^]")
```

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

Metacharacters Inside Character Sets

Most metacharacters inside a character set are already escaped. This implies that you do not need to escape them using double backslashes.

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p[ae.iou]n")
```

Metacharacters Inside Character Sets

Most metacharacters inside a character set are already escaped. This implies that you do not need to escape them using double backslashes.

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p[ae.iou]n")
```

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

Metacharacters Inside Character Sets

Most metacharacters inside a character set are already escaped. This implies that you do not need to escape them using double backslashes.

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p[ae.iou]n")
```

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

The dot `.` inside the character set now represents the literal dot character rather than the wildcard character.

Note: Not all metacharacters become literal characters when they appear inside a character set. The exceptions are the opening bracket `[`, the closing bracket `]`, the dash `-`, the caret `^` (if at the front or by itself), and the backslash `\`.

Character Classes

Closely related to character sets and character ranges are **character classes**, which are used to match a certain class of characters.

The most common character classes in most regex engines are:

Pattern	Matches	Same as
`\\d`	Any digit	`[0-9]`
`\\D`	Any non-digit	`[^0-9]`
`\\w`	Any word character	`[a-zA-Z0-9_]`
`\\W`	Any non-word character	`[^a-zA-Z0-9_]`
`\\s`	Any whitespace character	`[\f\n\r\t\v]`
`\\S`	Any non-whitespace character	`[^\f\n\r\t\v]`

Character classes can be thought of as another type of metacharacter or as shortcuts for special character sets.

Whitespace

There are several types of whitespace characters, shown in the following table:

Character	Description
`\f`	Form feed (page break)
`\n`	Line feed (new line)
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab

For situations with non-printing whitespace characters, it can be difficult to determine which exact character it is, so the whitespace class `\\s` is a useful way to match with all of them.

Character Classes

For example:

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p\\d") # p followed by digit
```

Character Classes

For example:

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p\\d") # p followed by digit
```

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

Character Classes

For example:

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p\\d") # p followed by digit
```

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

```
str_detect(pnx, "p\\D") # p followed by non-digit
```

Character Classes

For example:

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p\\d") # p followed by digit
```

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

```
str_detect(pnx, "p\\D") # p followed by non-digit
```

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

Character Classes

For example:

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p\\d") # p followed by digit
```

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

```
str_detect(pnx, "p\\D") # p followed by non-digit
```

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

```
str_detect(pnx, "p\\W") # p followed by non-word character
```

Character Classes

For example:

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
```

```
str_detect(pnx, "p\\d") # p followed by digit
```

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

```
str_detect(pnx, "p\\D") # p followed by non-digit
```

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

```
str_detect(pnx, "p\\W") # p followed by non-word character
```

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

POSIX Character Classes

There is another type of character classes known as **POSIX character classes** that is supported by the regex engine in R.

The main POSIX classes are:

Class	Description	Same as
`[:alnum:]`	Any letter or digit	`[a-zA-Z0-9]`
`[:alpha:]`	Any letter	`[a-zA-Z]`
`[:digit:]`	Any digit	`[0-9]`
`[:lower:]`	Any lower case letter	`[a-z]`
`[:upper:]`	Any upper case letter	`[A-Z]`
`[:space:]`	Any whitespace, inluding space	`[\f\n\r\t\v ]`
`[:punct:]`	Any punctuation symbol	
`[:print:]`	Any printable character	
`[:graph:]`	Any printable character excluding space	
`[:xdigit:]`	Any hexadecimal digit	`[a-fA-F0-9]`
`[:cntrl:]`	ASCII control characters	

POSIX Character Classes

To use POSIX classes in R, the class needs to be wrapped inside a regex character class, i.e., the class needs to be inside a second set of square brackets.

For example:

```
pnx <-
  c("pan", "pen", "pin", "p0n", "p.n", "paun", "pwn3d")
str_detect(pnx, "[[:alpha:]]") # has any letter
```

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

```
str_detect(pnx, "[[:digit:]]") # has any digit
```

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

Anchors

An **anchor** is a pattern that does not match a character but rather a position before, after, or between characters. Anchors are used to "anchor" a match at a certain position.

Pattern	Meaning
`^` or `\A`	Start of string
`$` or `\Z`	End of string
`\b`	Word boundary (i.e., the edge of a word)
`\B`	Not a word boundary

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

```
[1] "- quick brown fox jumps over - lazy dog dog"
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

```
[1] "- quick brown fox jumps over - lazy dog dog"
```

```
str_replace_all(text, "^the", "-") # 'the' only at the start
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

```
[1] "- quick brown fox jumps over - lazy dog dog"
```

```
str_replace_all(text, "^the", "-") # 'the' only at the start
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

```
[1] "- quick brown fox jumps over - lazy dog dog"
```

```
str_replace_all(text, "^the", "-") # 'the' only at the start
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "\\Athe", "-") # same thing
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

```
[1] "- quick brown fox jumps over - lazy dog dog"
```

```
str_replace_all(text, "^the", "-") # 'the' only at the start
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "\\Athe", "-") # same thing
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

```
[1] "- quick brown fox jumps over - lazy dog dog"
```

```
str_replace_all(text, "^the", "-") # 'the' only at the start
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "\\Athe", "-") # same thing
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the$", "-") # 'the' only at the end
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the", "-") # 'the' anywhere
```

```
[1] "- quick brown fox jumps over - lazy dog dog"
```

```
str_replace_all(text, "^the", "-") # 'the' only at the start
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "\\Athe", "-") # same thing
```

```
[1] "- quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "the$", "-") # 'the' only at the end
```

```
[1] "the quick brown fox jumps over the lazy dog dog"
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "dog", "-") # 'dog' anywhere
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "dog", "-") # 'dog' anywhere
```

```
[1] "the quick brown fox jumps over the lazy - -"
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "dog", "-") # 'dog' anywhere
```

```
[1] "the quick brown fox jumps over the lazy - -"
```

```
str_replace_all(text, "dog$", "-") # 'dog' only at the end
```

Anchor Examples

```
text <- "the quick brown fox jumps over the lazy dog dog"
```

```
str_replace_all(text, "dog", "-") # 'dog' anywhere
```

```
[1] "the quick brown fox jumps over the lazy - -"
```

```
str_replace_all(text, "dog$", "-") # 'dog' only at the end
```

```
[1] "the quick brown fox jumps over the lazy dog -"
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "(\\b.|.\\b)", "-") # word boundaries
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "(\\b.|.\\b)", "-") # word boundaries
```

```
[1] "-ord---um---umpin---mpir---um---mpteent---ump-"
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "(\\b.|.\\b)", "-") # word boundaries
```

```
[1] "-ord---um---umpin---mpir---um---mpteent---ump-"
```

```
str_replace_all(text, "\\B.\\B", "-") # non-word-boundaries
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "(\\b.|.\\b)", "-") # word boundaries
```

```
[1] "-ord---um---umpin---mpir---um---mpteent---ump-"
```

```
str_replace_all(text, "\\B.\\B", "-") # non-word-boundaries
```

```
[1] "w---s j--p j-----g u----e p--p u-------h l---s"
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

```
[1] "words jump jumping -ire pump -teenth lumps"
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

```
[1] "words jump jumping -ire pump -teenth lumps"
```

```
str_replace_all(text, "\\Bump", "-") # 'ump' not at the beginning of a word
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

```
[1] "words jump jumping -ire pump -teenth lumps"
```

```
str_replace_all(text, "\\Bump", "-") # 'ump' not at the beginning of a word
```

```
[1] "words j- j-ing umpire p- umpteenth l-s"
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

```
[1] "words jump jumping -ire pump -teenth lumps"
```

```
str_replace_all(text, "\\Bump", "-") # 'ump' not at the beginning of a word
```

```
[1] "words j- j-ing umpire p- umpteenth l-s"
```

```
str_replace_all(text, "ump\\b", "-") # 'ump' at the end of a word
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

```
[1] "words jump jumping -ire pump -teenth lumps"
```

```
str_replace_all(text, "\\Bump", "-") # 'ump' not at the beginning of a word
```

```
[1] "words j- j-ing umpire p- umpteenth l-s"
```

```
str_replace_all(text, "ump\\b", "-") # 'ump' at the end of a word
```

```
[1] "words j- jumping umpire p- umpteenth lumps"
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

```
[1] "words jump jumping -ire pump -teenth lumps"
```

```
str_replace_all(text, "\\Bump", "-") # 'ump' not at the beginning of a word
```

```
[1] "words j- j-ing umpire p- umpteenth l-s"
```

```
str_replace_all(text, "ump\\b", "-") # 'ump' at the end of a word
```

```
[1] "words j- jumping umpire p- umpteenth lumps"
```

```
str_replace_all(text, "ump\\B", "-") # 'ump' not at the end of a word
```

Anchor Examples

```
text <- "words jump jumping umpire pump umpteenth lumps"
```

```
str_replace_all(text, "\\bump", "-") # 'ump' at the beginning of a word
```

```
[1] "words jump jumping -ire pump -teenth lumps"
```

```
str_replace_all(text, "\\Bump", "-") # 'ump' not at the beginning of a word
```

```
[1] "words j- j-ing umpire p- umpteenth l-s"
```

```
str_replace_all(text, "ump\\b", "-") # 'ump' at the end of a word
```

```
[1] "words j- jumping umpire p- umpteenth lumps"
```

```
str_replace_all(text, "ump\\B", "-") # 'ump' not at the end of a word
```

```
[1] "words jump j-ing -ire pump -teenth l-s"
```

The Caret Metacharacter Revisited

Question: What is the difference between `^[0-9]`, `[^0-9]`, and `[0-9^]`?

The Caret Metacharacter Revisited

Question: What is the difference between `^[0-9]`, `[^0-9]`, and `[0-9^]`?

The caret `^` outside of the character set is an anchor, so `^[0-9]` matches strings that begin with a digit.

The caret `^` at the start of the character set is a negation, so `[^0-9]` matches a character that is not a digit.

The caret `^` inside a character set but not at the start is the literal caret character, so `[0-9^]` matches a character that is a digit or the caret.

Quantifiers

Quantifiers can be attached to literal characters, character classes, or groups to match repeats.

Pattern	Meaning
`*`	Match 0 or more (is greedy)
`+`	Match 1 or more (is greedy)
`?`	Match 0 or 1
`{3}`	Match Exactly 3
`{3,}`	Match 3 or more
`{3,5}`	Match 3, 4 or 5

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

```
[1] "words-or-numbers-9,876-and-combos123-like-password_1234"
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

```
[1] "words-or-numbers-9,876-and-combos123-like-password_1234"
```

```
str_replace_all(text, "\\S", "-") # anything but whitespace
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

```
[1] "words-or-numbers-9,876-and-combos123-like-password_1234"
```

```
str_replace_all(text, "\\S", "-") # anything but whitespace
```

```
[1] "----- -- ------- ----- --- --------- ---- -------------"
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

```
[1] "words-or-numbers-9,876-and-combos123-like-password_1234"
```

```
str_replace_all(text, "\\S", "-") # anything but whitespace
```

```
[1] "----- -- ------- ----- --- --------- ---- -------------"
```

```
str_replace_all(text, "\\S+", "-") # one or more non-whitespace
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

```
[1] "words-or-numbers-9,876-and-combos123-like-password_1234"
```

```
str_replace_all(text, "\\S", "-") # anything but whitespace
```

```
[1] "----- -- ------- ----- --- --------- ---- -------------"
```

```
str_replace_all(text, "\\S+", "-") # one or more non-whitespace
```

```
[1] "- - - - - - - -"
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

```
[1] "words-or-numbers-9,876-and-combos123-like-password_1234"
```

```
str_replace_all(text, "\\S", "-") # anything but whitespace
```

```
[1] "----- -- ------- ----- --- --------- ---- -------------"
```

```
str_replace_all(text, "\\S+", "-") # one or more non-whitespace
```

```
[1] "- - - - - - - -"
```

```
str_replace_all(text, "\\w+", "-") # one or more word characters
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\s", "-") # any whitespace
```

```
[1] "words-or-numbers-9,876-and-combos123-like-password_1234"
```

```
str_replace_all(text, "\\S", "-") # anything but whitespace
```

```
[1] "----- -- ------- ----- --- --------- ---- -------------"
```

```
str_replace_all(text, "\\S+", "-") # one or more non-whitespace
```

```
[1] "- - - - - - - -"
```

```
str_replace_all(text, "\\w+", "-") # one or more word characters
```

```
[1] "- - - -,- - - - -"
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

```
[1] "words or numbers -,--- and combos--- like password_----"
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

```
[1] "words or numbers -,--- and combos--- like password_----"
```

```
str_replace_all(text, "\\D", "-") # any non-digit
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

```
[1] "words or numbers -,--- and combos--- like password_----"
```

```
str_replace_all(text, "\\D", "-") # any non-digit
```

```
[1] "-----------------9-876-----------123---------------1234"
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

```
[1] "words or numbers -,--- and combos--- like password_----"
```

```
str_replace_all(text, "\\D", "-") # any non-digit
```

```
[1] "-----------------9-876-----------123---------------1234"
```

```
str_replace_all(text, "\\d+", "-") # one or more digits
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

```
[1] "words or numbers -,--- and combos--- like password_----"
```

```
str_replace_all(text, "\\D", "-") # any non-digit
```

```
[1] "-----------------9-876-----------123---------------1234"
```

```
str_replace_all(text, "\\d+", "-") # one or more digits
```

```
[1] "words or numbers -,- and combos- like password_-"
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

```
[1] "words or numbers -,--- and combos--- like password_----"
```

```
str_replace_all(text, "\\D", "-") # any non-digit
```

```
[1] "-----------------9-876-----------123---------------1234"
```

```
str_replace_all(text, "\\d+", "-") # one or more digits
```

```
[1] "words or numbers -,- and combos- like password_-"
```

```
str_replace_all(text, "\\D+", "-") # one or more nondigits
```

Quantifier Examples

```
text <- "words or numbers 9,876 and combos123 like password_1234"
```

```
str_replace_all(text, "\\d", "-") # any digit
```

```
[1] "words or numbers -,--- and combos--- like password_----"
```

```
str_replace_all(text, "\\D", "-") # any non-digit
```

```
[1] "-----------------9-876-----------123---------------1234"
```

```
str_replace_all(text, "\\d+", "-") # one or more digits
```

```
[1] "words or numbers -,- and combos- like password_-"
```

```
str_replace_all(text, "\\D+", "-") # one or more nondigits
```

```
[1] "-9-876-123-1234"
```

Quantifier Examples

```
text <-
"year 1996 area code 310 combo123 password_1234 singledigit5"
```

```
str_replace_all(text, "\\d{3}", "-") # 3 adjacent digits. reads left to right
```

Quantifier Examples

```
text <-
"year 1996 area code 310 combo123 password_1234 singledigit5"
```

```
str_replace_all(text, "\\d{3}", "-") # 3 adjacent digits. reads left to right
```

```
[1] "year -6 area code - combo- password_-4 singledigit5"
```

Quantifier Examples

```
text <-
"year 1996 area code 310 combo123 password_1234 singledigit5"
```

```
str_replace_all(text, "\\d{3}", "-") # 3 adjacent digits. reads left to right
```

```
[1] "year -6 area code - combo- password_-4 singledigit5"
```

```
str_replace_all(text, "\\d{2,4}", "-") # 2 to 4 adjacent digits
```

Quantifier Examples

```
text <-
"year 1996 area code 310 combo123 password_1234 singledigit5"
```

```
str_replace_all(text, "\\d{3}", "-") # 3 adjacent digits. reads left to right
```

```
[1] "year -6 area code - combo- password_-4 singledigit5"
```

```
str_replace_all(text, "\\d{2,4}", "-") # 2 to 4 adjacent digits
```

```
[1] "year - area code - combo- password_- singledigit5"
```

Quantifier Examples for ?

```
text <- c("Momma","Mama","Mamma","Mommy","Mom","Mother")
```

```
str_match(text, "M[ao]m{1,2}[ay]") |> as.vector() # [ay] required as last character
```

Quantifier Examples for ?

```
text <- c("Momma","Mama","Mamma","Mommy","Mom","Mother")
```

```
str_match(text, "M[ao]m{1,2}[ay]") |> as.vector() # [ay] required as last character
```

```
[1] "Momma" "Mama"  "Mamma" "Mommy" NA      NA
```

Quantifier Examples for ?

```
text <- c("Momma","Mama","Mamma","Mommy","Mom","Mother")
```

```
str_match(text, "M[ao]m{1,2}[ay]") |> as.vector() # [ay] required as last character
```

```
[1] "Momma" "Mama"  "Mamma" "Mommy" NA      NA
```

```
str_match(text, "M[ao]m{1,2}[ay]?") |> as.vector() #? makes [ay] optional as last character
```

Quantifier Examples for ?

```
text <- c("Momma","Mama","Mamma","Mommy","Mom","Mother")
```

```
str_match(text, "M[ao]m{1,2}[ay]") |> as.vector() # [ay] required as last character
```

```
[1] "Momma" "Mama"  "Mamma" "Mommy" NA      NA
```

```
str_match(text, "M[ao]m{1,2}[ay]?") |> as.vector() #? makes [ay] optional as last character
```

```
[1] "Momma" "Mama"  "Mamma" "Mommy" "Mom"   NA
```

Greedy vs Ungreedy Matching

Quantifiers are by default **greedy** in the sense that they will return the longest match.

Adding **?** to a quantifier will make it ungreedy (or **lazy**), so it will return the shortest match.

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract(text, "P.*r") # 'P' to 'r' anything in between greedy
```

Greedy vs Ungreedy Matching

Quantifiers are by default **greedy** in the sense that they will return the longest match.

Adding **?** to a quantifier will make it ungreedy (or **lazy**), so it will return the shortest match.

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract(text, "P.*r") # 'P' to 'r' anything in between greedy
```

```
[1] "Peter Piper picked a peck of pickled pepper"
```

Greedy vs Ungreedy Matching

Quantifiers are by default **greedy** in the sense that they will return the longest match.

Adding **?** to a quantifier will make it ungreedy (or **lazy**), so it will return the shortest match.

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract(text, "P.*r") # 'P' to 'r' anything in between greedy
```

```
[1] "Peter Piper picked a peck of pickled pepper"
```

```
str_extract(text, "P.*?r") # 'P' to 'r' anything in between ungreedy
```

Greedy vs Ungreedy Matching

Quantifiers are by default **greedy** in the sense that they will return the longest match.

Adding **?** to a quantifier will make it ungreedy (or **lazy**), so it will return the shortest match.

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract(text, "P.*r") # 'P' to 'r' anything in between greedy
```

```
[1] "Peter Piper picked a peck of pickled pepper"
```

```
str_extract(text, "P.*?r") # 'P' to 'r' anything in between ungreedy
```

```
[1] "Peter"
```

Greedy vs Ungreedy Matching, str_extrac_all()

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract_all(text, "P.*?r") # ungreedy
```

Greedy vs Ungreedy Matching, str_extrac_all()

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract_all(text, "P.*?r") # ungreedy
```

```
[[1]]
[1] "Peter" "Piper"
```

Greedy vs Ungreedy Matching, str_extrac_all()

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract_all(text, "P.*?r") # ungreedy
```

```
[[1]]
[1] "Peter" "Piper"
```

```
str_extract_all(text, "[Pp].*?r") # ungreedy
```

Greedy vs Ungreedy Matching, str_extrac_all()

```
text <- "Peter Piper picked a peck of pickled peppers"
```

```
str_extract_all(text, "P.*?r") # ungreedy
```

```
[[1]]
[1] "Peter" "Piper"
```

```
str_extract_all(text, "[Pp].*?r") # ungreedy
```

```
[[1]]
[1] "Peter"                              "Piper"
[3] "picked a peck of pickled pepper"
```

Grouping and Capturing

Parentheses `( )` define a **group** that groups together parts of a regular expression.

Besides grouping part of a regular expression together, parentheses also create a numbered **capturing group**: Any matches to the part of the pattern defined by the parentheses can be referenced by group number, either for modification or replacement.

By including `?:` after the opening parenthesis, the group becomes a **non-capturing group**.

For example, in the pattern `(abc)(def)(?:ghi)`, the pattern `(abc)` creates capturing group 1, `(def)` creates capturing group 2, and `(ghi)` is a group that is not captured.

Groups are used in conjunction with `str_match()` and `str_match_all()`.

Grouping and Capturing

Some examples of the common syntax for groups:

Pattern	Meaning
`a(bc)d`	Match the text `abcd`, capture the text in the group `bc`
`(?:abc)`	Non-capturing group
`(abc)def(ghi)`	Match `abcdefghi`, group `abc` and `ghi`
`(Mrs|Ms|Mr)`	`Mrs` or `Ms` or `Mr` (preference in the order given)
\1, \2, etc.	The first, second, etc. matched group (for `str_replace()`)

Note: Notice that the vertical line | is used for "or", just like in logical expressions. The vertical line | is called the **alternation** operator.

Grouping and Capturing Examples

The output of `str_match()` is a character matrix whose first column is the complete match, followed by one column for each capturing group.

```
pattern <- "(bc)(def)(?:ghi)" # 'ghi' must be present but do not capture 'ghi
str_match("abcdefghijkl", pattern)
```

Grouping and Capturing Examples

The output of `str_match()` is a character matrix whose first column is the complete match, followed by one column for each capturing group.

```
pattern <- "(bc)(def)(?:ghi)" # 'ghi' must be present but do not capture 'ghi
str_match("abcdefghijkl", pattern)
```

```
     [,1]       [,2] [,3]
[1,] "bcdefghi" "bc" "def"
```

Grouping and Capturing Examples

The output of `str_match()` is a character matrix whose first column is the complete match, followed by one column for each capturing group.

```
pattern <- "(bc)(def)(?:ghi)" # 'ghi' must be present but do not capture 'ghi
str_match("abcdefghijkl", pattern)
```

```
     [,1]       [,2] [,3]
[1,] "bcdefghi" "bc" "def"
```

```
str_match("abcdefghI", pattern)
```

Grouping and Capturing Examples

The output of `str_match()` is a character matrix whose first column is the complete match, followed by one column for each capturing group.

```
pattern <- "(bc)(def)(?:ghi)" # 'ghi' must be present but do not capture 'ghi
str_match("abcdefghijkl", pattern)
```

```
     [,1]       [,2] [,3]
[1,] "bcdefghi" "bc" "def"
```

```
str_match("abcdefghI", pattern)
```

```
     [,1] [,2] [,3]
[1,] NA   NA   NA
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia"
pattern <- "(Mrs|Ms|Mr)" # match one of 'Mrs' or 'Ms' or 'Mr'
```

```
str_match_all(text, pattern)
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia"
pattern <- "(Mrs|Ms|Mr)" # match one of 'Mrs' or 'Ms' or 'Mr'
```

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]  [,2]
[1,] "Mr"  "Mr"
[2,] "Mrs" "Mrs"
[3,] "Ms"  "Ms"
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia"
pattern <- "(Mrs|Ms|Mr)" # match one of 'Mrs' or 'Ms' or 'Mr'
```

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]  [,2]
[1,] "Mr"  "Mr"
[2,] "Mrs" "Mrs"
[3,] "Ms"  "Ms"
```

```
# because Mr is listed before Mrs, it will match Mr and give preference to it
wrong_order <- "(Mr|Mrs|Ms)"
str_match_all(text, wrong_order)
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia"
pattern <- "(Mrs|Ms|Mr)" # match one of 'Mrs' or 'Ms' or 'Mr'
```

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]  [,2]
[1,] "Mr"  "Mr"
[2,] "Mrs" "Mrs"
[3,] "Ms"  "Ms"
```

```
# because Mr is listed before Mrs, it will match Mr and give preference to it
wrong_order <- "(Mr|Mrs|Ms)"
str_match_all(text, wrong_order)
```

```
[[1]]
     [,1] [,2]
[1,] "Mr" "Mr"
[2,] "Mr" "Mr"
[3,] "Ms" "Ms"
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia"
pattern <- "(Mrs|Ms|Mr)" # match one of 'Mrs' or 'Ms' or 'Mr'
```

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]  [,2]
[1,] "Mr"  "Mr"
[2,] "Mrs" "Mrs"
[3,] "Ms"  "Ms"
```

```
# because Mr is listed before Mrs, it will match Mr and give preference to it
wrong_order <- "(Mr|Mrs|Ms)"
str_match_all(text, wrong_order)
```

```
[[1]]
     [,1] [,2]
[1,] "Mr" "Mr"
[2,] "Mr" "Mr"
[3,] "Ms" "Ms"
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia"
short_pattern <- "(Mr?s?)"
str_match_all(text, short_pattern)
```

```
[[1]]
     [,1]  [,2]
[1,] "Mr"  "Mr"
[2,] "Mrs" "Mrs"
[3,] "Ms"  "Ms"
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia, Andy Hope"
capture <- "(Mrs|Ms|Mr)\\. (\\w+)"
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia, Andy Hope"
capture <- "(Mrs|Ms|Mr)\\. (\\w+)"
```

```
str_match_all(text, capture)
```

```
[[1]]
     [,1]          [,2]  [,3]
[1,] "Mr. Smith"   "Mr"  "Smith"
[2,] "Mrs. Lee"    "Mrs" "Lee"
[3,] "Ms. Garcia"  "Ms"  "Garcia"
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia, Andy Hope"
non_capture <- "(?:Mrs|Ms|Mr)\\. (\\w+)"
```

Grouping and Capturing Examples

```
text <- "Mr. Smith, Mrs. Lee, Ms. Garcia, Andy Hope"
non_capture <- "(?:Mrs|Ms|Mr)\\. (\\w+)"
```

```
str_match_all(text, non_capture)
```

```
[[1]]
     [,1]          [,2]
[1,] "Mr. Smith"   "Smith"
[2,] "Mrs. Lee"    "Lee"
[3,] "Ms. Garcia"  "Garcia"
```

Grouping and Capturing Examples: Backreferences

```
text = 'George Washington, John Adams, Thomas Jefferson'
pattern <- "(\\w+) (\\w+),?" # first group becomes \\1, second becomes \\2
```

```
str_match_all(text, pattern)
```

Grouping and Capturing Examples: Backreferences

```
text = 'George Washington, John Adams, Thomas Jefferson'
pattern <- "(\\w+) (\\w+),?" # first group becomes \\1, second becomes \\2
```

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]                 [,2]     [,3]
[1,] "George Washington," "George" "Washington"
[2,] "John Adams,"        "John"   "Adams"
[3,] "Thomas Jefferson"   "Thomas" "Jefferson"
```

Grouping and Capturing Examples: Backreferences

```
text = 'George Washington, John Adams, Thomas Jefferson'
pattern <- "(\\w+) (\\w+),?" # first group becomes \\1, second becomes \\2
```

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]                 [,2]     [,3]
[1,] "George Washington," "George" "Washington"
[2,] "John Adams,"        "John"   "Adams"
[3,] "Thomas Jefferson"   "Thomas" "Jefferson"
```

```
str_replace_all(text, pattern, "\\2, \\1;")
```

Grouping and Capturing Examples: Backreferences

```
text = 'George Washington, John Adams, Thomas Jefferson'
pattern <- "(\\w+) (\\w+),?" # first group becomes \\1, second becomes \\2
```

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]                 [,2]     [,3]
[1,] "George Washington," "George" "Washington"
[2,] "John Adams,"        "John"   "Adams"
[3,] "Thomas Jefferson"   "Thomas" "Jefferson"
```

```
str_replace_all(text, pattern, "\\2, \\1;")
```

```
[1] "Washington, George; Adams, John; Jefferson, Thomas;"
```

Grouping and Capturing Examples: Backreferences

```
text = 'the quick brown fox jumps over the the lazy dog'
pattern <- "\\b(\\w+)\\s+\\1\\b"
```

The pattern says:

- Word boundary
- followed by a capture group of one or more word characters
- followed by one or more spaces
- followed by the group of text that was captured earlier
- followed by a word boundary

```
str_match_all(text, pattern)
```

Grouping and Capturing Examples: Backreferences

```
text = 'the quick brown fox jumps over the the lazy dog'
pattern <- "\\b(\\w+)\\s+\\1\\b"
```

The pattern says:

- Word boundary
- followed by a capture group of one or more word characters
- followed by one or more spaces
- followed by the group of text that was captured earlier
- followed by a word boundary

```
str_match_all(text, pattern)
```

```
[[1]]
     [,1]       [,2]
[1,] "the the" "the"
```

The pattern will match words that are repeated.

Telephone Numbers Example

For a more complicated example, we can define a regular expression to extract phone numbers.

```
phone_pattern <- "\\(?([2-9]\\d{2})\\)?[- .]?([2-9]\\d{2}[- .]?\\d{4})"
```

The pattern searches for:

- an optional opening parenthesis
- a capture group consisting of:
  - a digit between 2 and 9, followed by any 2 digits
- an optional closing parenthesis
- an optional character: one of dash, space, or dot
- a capture group consisting of:
  - a digit between 2 and 9, followed by any 2 digits
  - an optional character: one of dash, space, or dot
  - any four digits

Telephone Numbers Example

```
text <- c("apple", "1-800-786-1000", "(310) 209-1626", "310.208.0448",
  "3108258430", "Work: 323 224 2611; Home: (323)224-2621", "123-456-7890")
phone_pattern <- "\\(?([2-9]\\d{2})\\)?[- .]?([2-9]\\d{2}[- .]?\\d{4})"
str_extract(text, phone_pattern)
```

```
[1] NA               "800-786-1000"   "(310) 209-1626" "310.208.0448"
[5] "3108258430"     "323 224 2611"   NA
```

Telephone Numbers Example

```
# text <- c("apple", "1-800-786-1000", "(310) 209-1626", "310.208.0448",
#   "3108258430", "Work: 323 224 2611; Home: (323)224-2621", "123-456-7890")
# phone_pattern <- "\\(?([2-9]\\d{2})\\)?[- .]?([2-9]\\d{2}[- .]?\\d{4})"
str_extract_all(text, phone_pattern)
```

```
[[1]]
character(0)

[[2]]
[1] "800-786-1000"

[[3]]
[1] "(310) 209-1626"

[[4]]
[1] "310.208.0448"

[[5]]
[1] "3108258430"

[[6]]
[1] "323 224 2611"  "(323)224-2621"

[[7]]
character(0)
```

Telephone Numbers Example

```
# text <- c("apple", "1-800-786-1000", "(310) 209-1626", "310.208.0448",
#   "3108258430", "Work: 323 224 2611; Home: (323)224-2621", "123-456-7890")
# phone_pattern <- "\\(?([2-9]\\d{2})\\)?[- .]?([2-9]\\d{2}[- .]?\\d{4})"
str_match(text, phone_pattern)
```

```
     [,1]              [,2]  [,3]
[1,] NA                NA    NA
[2,] "800-786-1000"    "800" "786-1000"
[3,] "(310) 209-1626" "310" "209-1626"
[4,] "310.208.0448"    "310" "208.0448"
[5,] "3108258430"      "310" "8258430"
[6,] "323 224 2611"    "323" "224 2611"
[7,] NA                NA    NA
```

Telephone Numbers Example

```
str_match_all(text, phone_pattern)
```

```
[[1]]
     [,1] [,2] [,3]

[[2]]
     [,1]           [,2]  [,3]
[1,] "800-786-1000" "800" "786-1000"

[[3]]
     [,1]             [,2]  [,3]
[1,] "(310) 209-1626" "310" "209-1626"

[[4]]
     [,1]           [,2]  [,3]
[1,] "310.208.0448" "310" "208.0448"

[[5]]
     [,1]         [,2]  [,3]
[1,] "3108258430" "310" "8258430"

[[6]]
     [,1]            [,2]  [,3]
[1,] "323 224 2611"  "323" "224 2611"
[2,] "(323)224-2621" "323" "224-2621"

[[7]]
     [,1] [,2] [,3]
```

Telephone Numbers Example

Getting the previous results into something workable:

```
phone_list <- str_match_all(text, phone_pattern)
phone_matrix <- matrix(nrow = 0, ncol = 3)
for(i in seq_len(length(phone_list))){
  phone_matrix <- rbind(phone_matrix, phone_list[[i]])
}
phone_matrix
```

```
     [,1]              [,2]  [,3]
[1,] "800-786-1000"    "800" "786-1000"
[2,] "(310) 209-1626" "310" "209-1626"
[3,] "310.208.0448"    "310" "208.0448"
[4,] "3108258430"      "310" "8258430"
[5,] "323 224 2611"    "323" "224 2611"
[6,] "(323)224-2621"   "323" "224-2621"
```

Telephone Example - replacements with capture groups

We might not like the fact the phone numbers in column 3 have a non-standard appearance.

Fixing the problem is not as simple as replacing a dot or space with a dash because some phone numbers don't have either a dot or a space.

I define a new pattern: Three digits as capture group 1; an optional delimiter; then 4 digits which is capture group 2.

My replacement pattern is capture group 1, dash, capture group 2.

```
phone_matrix[,3]
```

```
[1] "786-1000" "209-1626" "208.0448" "8258430"  "224 2611" "224-2621"
```

```
phone_pattern <- "(\\d{3})[- .]?(\\d{4})"
replace_pattern <- "\\1-\\2"
str_replace(phone_matrix[,3], phone_pattern, replace_pattern)
```

```
[1] "786-1000" "209-1626" "208-0448" "825-8430" "224-2611" "224-2621"
```

Lookarounds

Occasionally we want to match characters that have a certain pattern before or after it. There are statements called **lookahead** and **lookbehind**, collectively called **lookarounds**, that look ahead or behind a pattern to check if a pattern does or does not exist.

Pattern	Name
`(?=...)`	Positive lookahead
`(?!...)`	Negative lookahead
`(?<=...)`	Positive lookbehind
`(?<!...)`	Negative lookbehind

The lookbehind patterns must have a bounded length (no $*$ or +).

Positive Lookahead

Positive lookahead with `(?=...)` looks ahead of the current match to ensure that the `...` subpattern matches.

```
text <- "I put a grey hat on my grey greyhound."
pattern <- "grey(?=hound)"
str_locate_all(text, pattern)
```

```
[[1]]
     start end
[1,]    29  32
```

The word `grey` is matched *only* if it is followed by `hound`. Note that `hound` itself is not part of the match.

Negative Lookahead

Negative lookahead with `(?!...)` looks ahead of the current match to ensure that the `...` subpattern does *not* match.

```
text <- "I put a grey hat on my grey greyhound."
pattern <- "grey(?!hound)"
str_locate_all(text, pattern)
```

```
[[1]]
     start end
[1,]     9  12
[2,]    24  27
```

The word `grey` is matched *only* if it is *not* followed by `hound`.

Positive Lookbehind

Positive lookbehind with `(?<=...)` looks behind the current position to ensure that the `...` subpattern immediately precedes the current match.

```
text <-
  "I withdrew 100 $1 bills, 20 $5 bills, and 5 $20 bills."
pattern <- "(?<=\\$)[[:digit:]]+"
str_extract_all(text, pattern)
```

```
[[1]]
[1] "1"  "5"  "20"
```

The digits are matched *only* if they are immediately preceded by a dollar $ sign.

Negative Lookbehind

Positive lookbehind with `(?<!...)` looks behind the current position to ensure that the `...` subpattern does *not* immediately precede the current match.

```
text <-
  "I withdrew 100 $1 bills, 20 $5 bills, and 5 $20 bills."
pattern <- "(?<!\\$)[[:digit:]]+"
str_extract_all(text, pattern)
```

```
[[1]]
[1] "100" "20"  "5"   "0"
```

The digits are matched *only* if they are *not* immediately preceded by a dollar $ sign.