# Video 24: regex character sets

Stats 102A

Miles Chen, PhD

# Regular Expressions

# Resources

Cheat Sheets for `stringr` and Regular Expressions

- Strings Cheat Sheet at: https://posit.cloud/learn/cheat-sheets
- https://www.cheatography.com//davechild/cheat-sheets/regular-expressions/pdf/

Sites for Testing Regular Expressions

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

# 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 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. |

# Literal Characters

The most basic type of regular expressions are **literal characters**, which are characters 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.

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

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

```r
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()`.

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

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

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

```r
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.

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

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

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

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

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

**Question**: Consider the following vector.

```r
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.

```r
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:

```r
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")
```

```
[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.

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

What pattern can be defined to match 3 adjacent digits?

```r
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.

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

```r
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.

```r
basic
```

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

```r
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, "[^\\^]")
```

```
[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.

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

```r
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
```

```
[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.

| 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]` |

# 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:

```r
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
```

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

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