R – Regular Expressions

Regular Expression

- A pattern of special characters used to match strings in a search
- Typically made up from special characters called metacharacters
- "grep" command:
  searches for text in file(s)

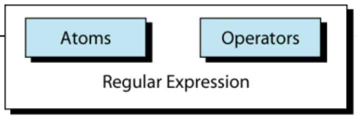
Metacharacters

RE Metacharacter	Matches…
.	**Any one character, except new line**
[a-z]	**Any one of the enclosed characters (e.g. a-z)**
*****	**Zero or more of preceding character**
? or \?	**Zero or one of the preceding characters**
+ or \+	**One or more of the preceding characters**

more Metacharacters

RE Metacharacter	Matches…
^	**beginning of line**
$	**end of line**
\char	**Escape the meaning of *char* following it**
[^]	**One character not in the set**
\\<	**Beginning of word anchor**
\\>	**End of word anchor**
( ) or \\(\\)	**Tags matched characters to be used later (max = 9)**
| or \\\|	**Or grouping**
x\\{m\\}	**Repetition of character x, m times (x,m = integer)**
x\\{m,\\}	**Repetition of character x, at least m times**
x\\{m,n\\}	**Repetition of character x between m and m times**

Regular Expression

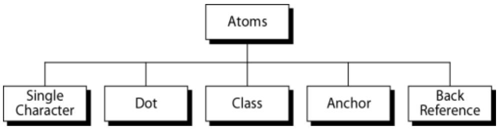


An atom specifies what text is to be matched and where it is to be found.

An operator combines regular expression atoms.

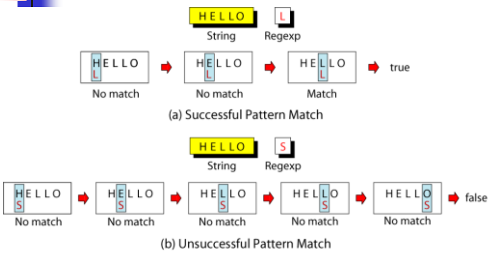
Atoms

An atom specifies what text is to be matched and where it is to be found.



Single-Character Atom

A single character matches itself

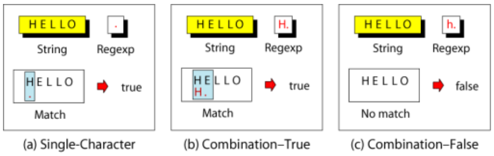


(a) Successful Pattern Match

(b) Unsuccessful Pattern Match

Dot Atom

matches any single character except for a new line character (\n)



(a) Single-Character

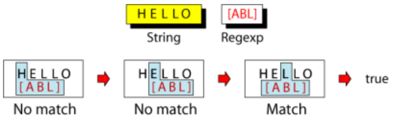
(b) Combination–True

(c) Combination–False

Class Atom

matches only single character that can be any of the characters defined in a set:

Example: [ABC] matches either A, B, or C.



Notes:

1) A range of characters is indicated by a dash, e.g. [A-Q]
2) Can specify characters to be excluded from the set, e.g. [^0-9] matches any character other than a number.

Example: Classes

RegExpr	Means	RegExpr	Means
[A-H]	[ABCDEFGH]	[^AB]	Any character except A or B
[A-Z]	Any uppercase alphabetic	[A-Za-z]	Any alphabetic
[0-9]	Any digit	[^0-9]	Any character except a digit
[[a]	[or a	[]a]	] or a
[0-9\-]	digit or hyphen	[^\^]	Anything except^

short-hand classes

- [:alnum:]
- [:alpha:]
- [:upper:]
- [:lower:]
- [:digit:]
- [:space:]

Anchors

Anchors tell where the next character in the pattern must be located in the text data.

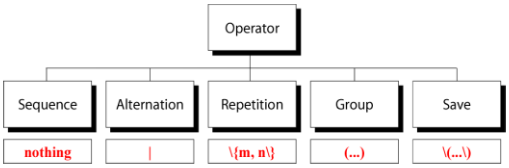
Anchor	Means	Example
^	Beginning of line	One line of text.\n
$	End of line	One line of text.\n
\<	Beginning of word	One line of text.\n
\>	End of word	One line of text.\n

Back References: \n

- used to retrieve saved text in one of nine buffers
- can refer to the text in a saved buffer by using a back reference:

  ex.: \1 \2 \3 ...\9

Operators



Sequence Operator

In a sequence operator, if a series of atoms are shown in a regular expression, there is no operator between them.

dog	→	matches the pattern "dog"
a..b	→	matches "a" , any two characters, and "b"
[2-4][0-9]	→	matches a number between 20 and 49
[0-9][0-9]	→	matches any two digits
^$	→	matches a blank line
^.$	→	matches a one-character line
[0-9]-[0-9]	→	matches two digits separated by a "-"

Alternation Operator: | or \|

operator (| or \|) is used to define one **or** more alternatives

UNIX|unix → matches "UNIX" or "unix"

Ms|Miss|Mrs → matches "Ms" or "Miss" or "Mrs"

Note: depends on version of “grep”

Repetition Operator: \{...\}

The repetition operator specifies that the atom or expression immediately before the repetition may be repeated.

\{m , n\}

matches previous character m to n times.

A\{3 , 5\}	→	matches "AAA" , "AAAA", or "AAAAA"
BA\{3 , 5\}	→	matches "BAAA" , "BAAAA", or "BAAAAA"

Basic Repetition Forms

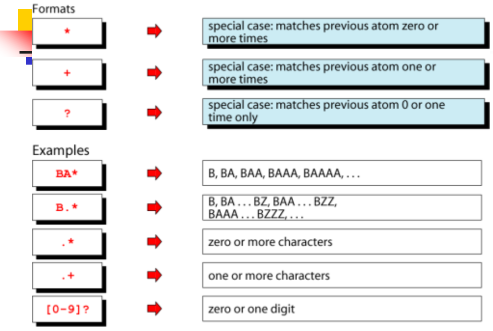
Formats

\{m\}	→	matches previous atom exactly m times
\{m, \}	→	matches previous atom m times or more
\{, n\}	→	matches previous atom n times or less

Examples

CA\{5\}	→	CAAAAA
CA\{3,\}	→	CAAA, CAAAA, CAAAAA, …
CA\{,2\}	→	C, CA, CAA

Short Form Repetition Operators: *



Formats

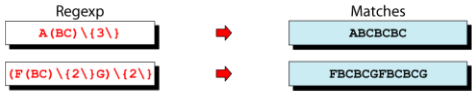
`*`	special case: matches previous atom zero or more times
`+`	special case: matches previous atom one or more times
`?`	special case: matches previous atom 0 or one time only

Examples

`BA*`	B, BA, BAA, BAAA, BAAAA, . . .
`B.*`	B, BA . . . BZ, BAA . . . BZZ, BAAA . . . BZZZ, . . .
`.*`	zero or more characters
`.+`	one or more characters
`[0-9]?`	zero or one digit

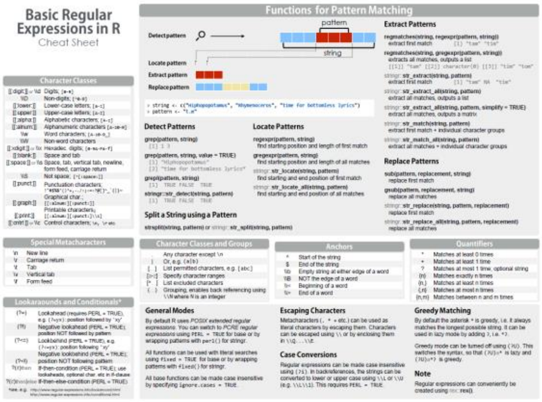
Group Operator

In the group operator, when a group of characters is enclosed in parentheses, the next operator applies to the whole group, not only the previous characters.



Regexp	Matches
`A(BC)\{3\}`	ABCBCBC
`(F(BC)\{2\}G)\{2\}`	FBCBCGFBCBCG

Note: depends on version of "grep"
use \(and \) instead



Basic Regular Expressions in R
Cheat Sheet

Character Classes

[[:digit:]] or \\d	Digits; [0-9]
\\D	Non-digits; [^0-9]
[[:lower:]]	Lower-case letters; [a-z]
[[:upper:]]	Upper-case letters; [A-Z]
[[:alpha:]]	Alphabetic characters; [A-z]
[[:alnum:]]	Alphanumeric characters [A-z0-9]
\\w	Word characters; [A-z0-9_]
\\W	Non-word characters
[[:xdigit:]] or \\x	Hexadec. digits; [0-9A-Fa-f]
[[:blank:]]	Space and tab
[[:space:]] or \\s	Space, tab, vertical tab, newline, form feed, carriage return
\\S	Not space; [^[:space:]]
[[:punct:]]	Punctuation characters; !"#$%&'()*+,-./:;<=>?@[]^_`{|}~
[[:graph:]]	Graphical char.; [[:alnum:][:punct:]]
[[:print:]]	Printable characters; [[:alnum:][:punct:]\\s]
[[:cntrl:]] or \\c	Control characters; \n, \r etc.

Special Metacharacters

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

Lookaraounds and Conditionals*

(?=)	Lookahead (requires PERL = TRUE), e.g. (?=yx): position followed by 'xy'
(?!)	Negative lookahead (PERL = TRUE); position NOT followed by pattern
(?<=)	Lookbehind (PERL = TRUE), e.g. (?<=yx): position following 'xy'
(?<!)	Negative lookbehind (PERL = TRUE); position NOT following pattern
?(if)then	If-then-condition (PERL = TRUE); use lookaheads, optional char. etc in if-clause
?(if)then|else	If-then-else-condition (PERL = TRUE)

*see, e.g. http://www.regular-expressions.info/lookaround.html
http://www.regular-expressions.info/conditional.html

Functions for Pattern Matching

```
> string <- c("Hiphopopotamus", "Rhymenoceros", "time for bottomless lyrics")
> pattern <- "t.m"
```

Detect Patterns

grep(pattern, string)
[1] 1 3

grep(pattern, string, value = TRUE)
[1] "Hiphopopotamus"
[2] "time for bottomless lyrics"

grepl(pattern, string)
[1] TRUE FALSE TRUE

stringr::str_detect(string, pattern)
[1] TRUE FALSE TRUE

Locate Patterns

regexpr(pattern, string)
find starting position and length of first match

gregexpr(pattern, string)
find starting position and length of all matches

stringr::str_locate(string, pattern)
find starting and end position of first match

stringr::str_locate_all(string, pattern)
find starting and end position of all matches

Split a String using a Pattern

strsplit(string, pattern) or stringr::str_split(string, pattern)

Extract Patterns

regmatches(string, regexpr(pattern, string))
extract first match [1] "tam" "tim"

regmatches(string, gregexpr(pattern, string))
extracts all matches, outputs a list
[[1]] "tam" [[2]] character(0) [[3]] "tim" "tom"

stringr::str_extract(string, pattern)
extract first match [1] "tam" NA "tim"

stringr::str_extract_all(string, pattern)
extract all matches, outputs a list

stringr::str_extract_all(string, pattern, simplify = TRUE)
extract all matches, outputs a matrix

stringr::str_match(string, pattern)
extract first match + individual character groups

stringr::str_match_all(string, pattern)
extract all matches + individual character groups

Replace Patterns

sub(pattern, replacement, string)
replace first match

gsub(pattern, replacement, string)
replace all matches

stringr::str_replace(string, pattern, replacement)
replace first match

stringr::str_replace_all(string, pattern, replacement)
replace all matches

Character Classes and Groups

.	Any character except \n
|	Or, e.g. (a|b)
[...]	List permitted characters, e.g. [abc]
[a-z]	Specify character ranges
[^...]	List excluded characters
(...)	Grouping, enables back referencing using \\N where N is an integer

Anchors

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

Quantifiers

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

General Modes

By default R uses *POSIX* extended regular expressions. You can switch to *PCRE* regular expressions using PERL = TRUE for base or by wrapping patterns with perl() for stringr.

All functions can be used with literal searches using fixed = TRUE for base or by wrapping patterns with fixed() for stringr.

All base functions can be made case insensitive by specifying ignore.cases = TRUE.

Escaping Characters

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

Case Conversions

Regular expressions can be made case insensitive using (?i). In backreferences, the strings can be converted to lower or upper case using \\L or \\U (e.g. \\L\\1). This requires PERL = TRUE.

Greedy Matching

By default the asterisk * is greedy, i.e. it always matches the longest possible string. It can be used in lazy mode by adding ?, i.e. *?.

Greedy mode can be turned off using (?U). This switches the syntax, so that (?U)a* is lazy and (?U)a*? is greedy.

Note

Regular expressions can conveniently be created using rex::rex().

Matching Literal Characters

- We're going to start with the simplest match of all: a literal character.
- 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.
- The following examples are extremely basic but they will help you get a good understanding of regex.
- Consider the following text stored in a character vector this_book:
```
this_book <- 'This book is mine'
```
- The first regular expression we are going to work with is "book". This pattern is formed by a letter b, followed by a letter o, followed by another letter o, followed by a letter k. As you may guess, this pattern matches the word book in the character vector this_book.
- To have a visual representation of the actual pattern that is matched, you should use the function str_view() from the package "stringr":
```
str_view(this_book, 'book')
```

Matching Literal Characters

- This pattern, "book" doesn't match the entire content in the vector this_book; it just matches those four letters. It may seem really simple but there are a couple of details to be highlighted.
- The first is that regex searches are case sensitive by default. This means that the pattern "Book" would not match book in this_book.
```
str_view("This Book is mine.", 'book')
```
- You can change the matching task so that it is case insensitive.
- Let's add more text to this_book:
```
this_book <- 'This book is mine. I wrote this book with
bookdown.'
```
- Let's use str_view() to see what pieces of text are matched in this_book with the pattern "book":
```
str_view(this_book, "book")
```

Matching Literal Characters

- Only the first occurrence of book was matched. This is a common behavior of regular expressions in which they return a match as fast possible.
- We can think of this behavior as the "eager principle", that is, regular expressions are eager and they will give preference to an early match. This is a minor but important detail.
- All the letters and digits in the English alphabet are considered literal characters. They are called literal because they match themselves.
```
str_view <- c("I had 3 quesadillas for lunch", "3")
```
- Here is another example:
```
transport <- c("car", "bike", "boat", "airplane")
```
- The first pattern to test is the letter "a":
```
str_view(transport, "a")
```
- When you execute the previous command, we should be able to see that the letter "a" is highlighted in the words car, boat and airplane.

The Wild Metacharacter

- The first metacharacter you should learn about is the dot or period ".", better known as the wild metacharacter. This metacharacter is used to match ANY character except for a new line.
- For example, consider the pattern "p.n", that is, p wildcard n. This pattern will match pan, pen, and pin, but it will not match prun or plan. The dot only matches one single character.
- Let’s see another example using the vector c("not", "note", "knot", "nut") and the pattern "n.t“

```
not <- c("not", "note", "knot", "nut")
str_view(not, "n.t")
```

- The pattern "n.t" matches not in the first three elements, and nut in the last element.

The Wild Metacharacter

- If you specify a pattern "no.", then just the first three elements in not will be matched.

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

- And if you define a pattern "kn.", then only the third element is matched.

```
str_view(not, "kn.")
```

- The wild metacharacter is probably the most used metacharacter, and it is also the most abused one, being the source of many mistakes.
- Here is a basic example with the regular expression formed by "5.00". If you think that this pattern will match five with two decimal places after it, you will be surprised to find out that it not only matches 5.00 but also 5100 and 5-00.

Escaping metacharacters

- What if you just want to match the character dot? For example, say you have the following vector:

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

- If you try the pattern "5.00", it will match all of the elements in fives.

```
str_view(fives, "5.00")
```

- To actually match the dot character, we need to escape the metacharacter.
- In most languages, the way to escape a metacharacter is by adding a backslash character in front of the metacharacter: "\.".
- When you use a backslash in front of a metacharacter you are “escaping” the character, this means that the character no longer has a special meaning, and it will match itself.
- However, R is a bit different. Instead of using a backslash you have to use two backslashes: "5\\.00". This is because the backslash "\", which is another metacharacter, has a special meaning in R.
- Therefore, to match just the element 5.00 in fives in R, you do it like so:

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