post01-arushi-desai

Arushi Desai

10/28/2017

Post 1: Data Manipulation (Cleaning)

Specific Emphasis on Regular Expressions (regex)

What are regular expressions?

A regular expression is a pattern that describes a set of strings. These patterns are expressed with metacharacters that describe digits, characters, spaces, and more. There are two types of regular expressions used in R: **extended regular expressions** (the default), and **Perl-like regular expressions** (induced with `perl = TRUE`). These two different types of regular expressions differentiate regex in R than regex in other programming languages, such as Python. We will stick with the default for this introductory course and for subsequent examples.

Purpose

Regular expressions are essential in cleaning data. They can be used to make data (particular long strings) much more readable. It is most used for string matching and replacing. Regex can also be used to split up text by tab characters, new lines, etc. so it is incredibly useful. Data from text files is what regex is most useful for. Today, we will use data from Gapminder, a Swedish non-partisan foundation that is a powerhouse for reliable statistics that produces free teaching resources for the world, to demonstrate the use of regular expressions in R.

Packages and Loading Data

To start out, we have to load the `stringr` package (to read in a .txt file), read in the Gapminder data, and define a vector of strings of hypothetical filenames. The `stringr` package is essential to using regular expressions in R because you need it for most string operations, and regex manipulates strings. However, you can use string functions in `base` R to use regular expressions.

```
library(stringr)
gDat <- read.delim("gapminderDataFiveYear.txt")
str(gDat)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

```
files <- c("block0_dplyr-fake.rmd", "block000_dplyr-fake.rmd.txt", "gapminderDataFiveYear.txt",
"regex.html", "regex.md", "regex.R", "regex.Rmd", "regex.Rpres",
"xblock000_dplyr-fake.rmd")
```

Basic Functions to Use with RegEx

We will only be using `base` R in this tutorial, but I will cover the `stringr` functions that do the same.
- **Identify match to a pattern**: `grep(..., value = FALSE)`, `stringr::str_detect()`
- **Extract match to a pattern**: `grep(..., value = TRUE)`, `stringr::str_extract()`, `stringr::str_extract_all()`
- **Locate pattern within a string (give start position of matched patterns)**: `regexpr()`, `gregexpr()`, `stringr::str_locate()`, `string::str_locate_all()`
- **Replace a pattern**: `gsub()`, `stringr::str_replace()`, `stringr::str_replace_all()`
- **Split a string using a pattern**: `strsplit()`, `stringr::str_split()`

Regular Expression Syntax

The most important aspect of regex we have not covered thus far is the syntax. This syntax is essential to specify characters for regex to seek out, sometimes with information about repeats and location within the string. Regex across programming languages only changes slightly with the syntax. All of them use *metacharacters* with specific meaning: `$ * + . ? [ ] ^ { } | ( ) \`. We will cover what these metacharacters mean throughout the rest of our tutorial.

Part 1: Escape Sequences We might want to match with characters in R that we cannot directly because they are “special” in R, and cannot be coded directly in a string. For example, if we want an apostrophe in an R string, you need to use the “escape” character, a backlash `\`, to include the apostrophe. For example, instead of defining a string as `last_name <- 'O'Malley'`, you would have to do define it as `last_name <- 'O\'Malley'`.
These are the characters in R that require escaping. Keep in mind that this rule applies to all string functions in R, not just regular expressions.
- `\'`: single quote; you don’t need to escape single quote inside a double-quoted string, so we can also use “‘” in the previous example
- `\"`: double quote; similarly, double quotes can be used inside a single-quoted string, i.e.’”’. - `\n`: newline
- `\r`: carriage return
- `\t`: tab character

EXAMPLE: Search for country names with an apostrophe

```
grep('\'', levels(gDat$country))
```

```
## [1] 31
```

```
grep('\'', levels(gDat$country), value = TRUE)
```

```
## [1] "Cote d'Ivoire"
```

Part 2: Quantifiers Quantifiers are a powerful tool in regex. They indicate how many times (repetitions) a pattern should be matched. Here is a list of quantifiers.
- `*` : matches at least 0 times
- `+` : matches at least 1 times
- `?` : matches at most 1 times (0 or 1 times, non-greedy)
- `{n}` : matches exactly n times
- `{n,}` : matches at least n times
- `{n,m}` : matches between n and m times

Here is an example that doesn't use the Gapminder data:

```
strings <- c("a", "ab", "acb", "accb", "acccb", "accccb")
strings
```

```
## [1] "a"      "ab"     "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac*b", strings, value = TRUE)
```

```
## [1] "ab"     "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac+b", strings, value = TRUE)
```

```
## [1] "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac?b", strings, value = TRUE)
```

```
## [1] "ab"  "acb"
```

```
grep("ac{2}b", strings, value = TRUE)
```

```
## [1] "accb"
```

```
grep("ac{2,}b", strings, value = TRUE)
```

```
## [1] "accb"   "acccb"  "accccb"
```

```
grep("ac{2,3}b", strings, value = TRUE)
```

```
## [1] "accb"  "acccb"
```

Not so tricky!

EXAMPLE: Find all countries with `ee` but not `eee`

```
grep("e{2}", levels(gDat$country), value = TRUE)
```

```
## [1] "Greece"
```

Part 3: Position of Patern Within the String Here is a list of position tools you can use to match strings:
- `^` : matches the start of the string
- `$` : matches the end of the string
- `\b` : matches the empty string at either edge of a *word*
- `\B` : matches the empty string provided it is not at an edge of a word (opposite of)

Here's an example with random strings:

```
strings <- c("abcd", "cdab", "cabd", "c abd")
strings
```

```
## [1] "abcd"  "cdab"  "cabd"  "c abd"
```

```
grep("ab", strings, value = TRUE)
```

```
## [1] "abcd"  "cdab"  "cabd"  "c abd"
```

```
grep("^ab", strings, value = TRUE)
```

```
## [1] "abcd"
```

```
grep("ab$", strings, value = TRUE)
```

```
## [1] "cdab"
```

```
grep("\\bab", strings, value = TRUE) # have to "escape" \b because of the backslash
```

```
## [1] "abcd"  "c abd"
```

EXAMPLE: Find string of country names that start with “South”, end in “land”, have a word in its name that starts with “Ga”

```
grep("^South", levels(gDat$country), value = TRUE)
```

```
## [1] "South Africa"
```

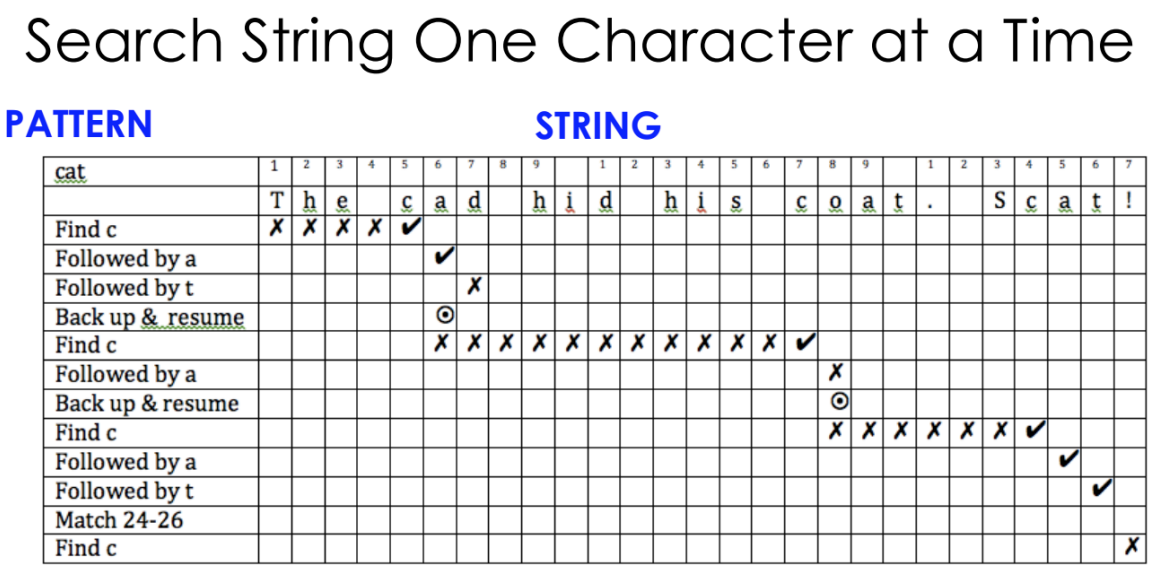
```
grep("land$", levels(gDat$country), value = TRUE)
```

```
## [1] "Finland"     "Iceland"     "Ireland"     "New Zealand" "Poland"
## [6] "Swaziland"   "Switzerland" "Thailand"
```

```
grep("\\bGa", levels(gDat$country), value = TRUE)
```

```
## [1] "Gabon"              "Gambia"             "West Bank and Gaza"
```

Here is a visual representation of a relatively easy regular expression (“cat”). This will help you understand the process R takes when choosing the strings to match with.



Part 4: Character Classes Character classes allow you to specify entire classes of characters, such as numbers, letters, and more. Here is a list of tools you can use:
- `\d` : digits, 0 1 2 3 4 5 6 7 8 9, equivalent to `[0-9]`
- `\D` : non-digits, equivalent to `[^0-9]`
- `[:lower:]` : lower-case letters, equivalent to `[a-z]`
- `[:upper:]` : upper-case letters, equivalent to `[A-Z]`
- `\w` : word characters, equivalent to `[A-z0-9_]`
- `\W` : not word, equivalent to `[^A-z0-9_]`
- `\s` : space, `` `  -` ``: not space

Part 5: Operators Finally, we will cover operators that do a number of different things. Here is a list: - `.` : matches any single character
- `[...]` : a character list, matches any one of the characters inside the square brackets; can also use `-` inside the brackets to specify a range of characters
- `[^...]` : an inverted character list, similar to `[...]`, but matches any characters except those inside the square brackets
- `|` : an “or” operator, matches patterns on either side of the `|`
- `(...)` : grouping in regular expressions; this allows you to retrieve the bits that matched various parts of your regular expression so you can alter them or use them for building up a new string

Here’s some string examples:

```
strings <- c("^ab", "ab", "abc", "abd", "abe", "ab 12")
strings
```

```
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 12"
```

```
grep("ab.", strings, value = TRUE)
```

```
## [1] "abc"   "abd"   "abe"   "ab 12"
```

```
grep("ab[c-e]", strings, value = TRUE)
```

```
## [1] "abc" "abd" "abe"
```

```
grep("ab[^c]", strings, value = TRUE)
```

```
## [1] "abd"   "abe"   "ab 12"
```

```
grep("^ab", strings, value = TRUE)
```

```
## [1] "ab"    "abc"   "abd"   "abe"   "ab 12"
```

```
grep("\\^ab", strings, value = TRUE)
```

```
## [1] "^ab"
```

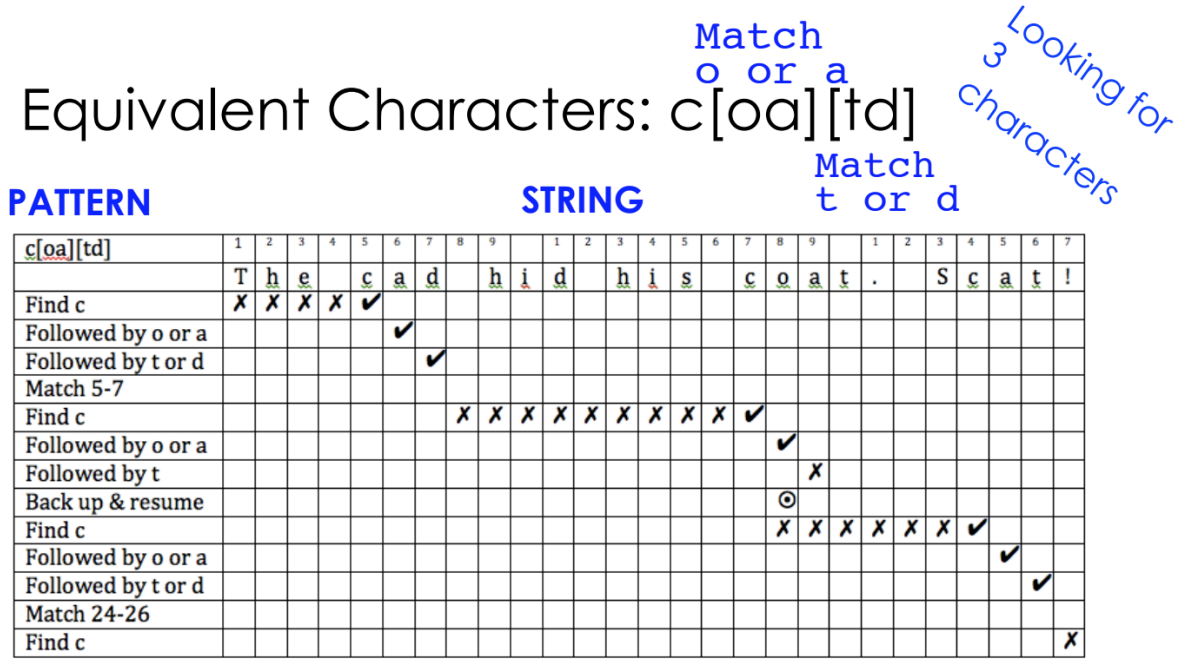
```
grep("abc|abd", strings, value = TRUE)
```

```
## [1] "abc" "abd"
```

```
gsub("(ab) 12", "\\1 34", strings)
```

```
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 34"
```

Here is a visual representation of another regular expression ("c[oa][td]"). This will help you understand the process R takes when choosing the strings to match with, when concerning brackets.



c[oa][td]	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7
	T	h	e		c	a	d		h	i	d		h	i	s		c	o	a	t	.		S	c	a	t	!
Find c	✗	✗	✗	✗	✔																						
Followed by o or a						✔																					
Followed by t or d							✔																				
Match 5-7																											
Find c								✗	✗	✗	✗	✗	✗	✗	✗	✗	✔										
Followed by o or a																		✔									
Followed by t																			✗								
Back up & resume																		⊙									
Find c																		✗	✗	✗	✗	✗	✗	✔			
Followed by o or a																									✔		
Followed by t or d																										✔	
Match 24-26																											
Find c																											✗

Conclusion

And that's it! You are now an expert in regular expressions. Use them for string manipulation and data cleaning, and you will be an effective data cleaner, and thus, a great data scientist.

Resources

Find more information and examples from the resources where I drew my information from:
- R Documentation for regex
- R Documentation for grep
- R Documentation for strsplit
- Basic RegEx in R Cheat Sheet
- Stat 545, Helpful Examples
- Used Data from This Tutorial