**Introduction**

In this post, we will learn about using regular expressions in R. While it is  
aimed at absolute beginners, we hope experienced users will find it useful as  
well. The post is broadly divided into 3 sections. In the first section, we  
will introduce the pattern matching functions such as grep, grepl etc. in  
base R as we will be using them in the rest of the post. Once the reader is  
comfortable with the above mentioned pattern matching functions, we will  
learn about regular expressions while exploring the names of R packages by  
probing the following:

* how many package names include the letter r?
* how many package names begin or end with the letter r?
* how many package names include the words data or plot?

In the final section, we will go through 4 case studies including simple email  
validation. If you plan to try the case studies, please do not skip any of the  
topics in the second section.

**What**



**Why**

Regular expressions can be used for:

* search
* replace
* validation
* extraction



**Use Cases**

Regular expressions have applications in a wide range of areas. We list some of  
the most popular ones below:

* email validation
* password validation
* date validation
* phone number validation
* search and replace in text editors
* web scraping

**Learning**

The below steps offer advice on the best way to learn or use regular expressions:

* describe the pattern in layman terms
* break down the pattern into individual components
* match each component to a regular expression
* build incrementally
* test

**Libraries**

We will use the following libraries in this post:

library(dplyr)

library(readr)

**Data**

We will use two data sets in this post. The first one is a list of all R  
packages on CRAN and is present in the package\_names.csv file.In the first week of May 2019, and extracted using the cranlogs pacakge.

**R Packages Data**

**Library(cranlogs)**

read\_csv("package\_names.csv") %>%

pull(1) -> r\_packages

**Top R Packages**

top\_downloads <- c("devtools", "rlang", "dplyr", "Rcpp", "tibble",

"ggplot2", "glue", "pillar", "cli", "data.table")

top\_downloads

## [1] "devtools" "rlang" "dplyr" "Rcpp" "tibble"

## [6] "ggplot2" "glue" "pillar" "cli" "data.table"

**Pattern Matching Functions**

Before we get into the nitty gritty of regular expressions, let us explore a  
few functions from base R for pattern matching. We will learn about using  
regular expressions with the stringr package in an upcoming post.

**grep**

The first function we will learn is grep(). It can be used to find elements  
that match a given pattern in a character vector. It will return the elements  
or the index of the elements based on your specification. In the below example,  
grep() returns the index of the elements that match the given pattern.

grep(x = top\_downloads, pattern = "r")

## [1] 2 3 8

Now let us look at the inputs:

* pattern
* x
* ignore.case
* value
* invert

**grep – Value**

If you want grep() to return the element instead of the index, set the  
value argument to TRUE. The default is FALSE. In the below example,  
grep() returns the elements and not their index.



grep(x = top\_downloads, pattern = "r", value = TRUE)

## [1] "rlang" "dplyr" "pillar"

**grep – Ignore Case**

If you have carefully observed the previous examples, have you noticed that  
the pattern r did not match the element Rcpp i.e. regular expressions are  
case sensitive. The ignore.case argument will ignore case while matching the  
pattern as shown below.



grep(x = top\_downloads, pattern = "r", value = TRUE, ignore.case = TRUE)

## [1] "rlang" "dplyr" "Rcpp" "pillar"

grep(x = top\_downloads, pattern = "R", value = TRUE)

## [1] "Rcpp"

grep(x = top\_downloads, pattern = "R", value = TRUE, ignore.case = TRUE)

## [1] "rlang" "dplyr" "Rcpp" "pillar"

**grep – Invert**

In some cases, you may want to retrieve elements that do not match the pattern  
specified. The invert argument will return the elements that do not match  
the pattern. In the below example, the elements returned do not match the  
pattern specified by us.



grep(x = top\_downloads, pattern = "r", value = TRUE, invert = TRUE)

## [1] "devtools" "Rcpp" "tibble" "ggplot2" "glue"

## [6] "cli" "data.table"

grep(x = top\_downloads, pattern = "r", value = TRUE,

invert = TRUE, ignore.case = TRUE)

## [1] "devtools" "tibble" "ggplot2" "glue" "cli"

## [6] "data.table"

**grepl**

grepl() will return only logical values. If the elements match the pattern  
specified, it will return TRUE else FALSE.



grepl(x = top\_downloads, pattern = "r")

## [1] FALSE TRUE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

**Ignore Case**

To ignore the case, use the ignore.case argument and set it to TRUE.



grepl(x = top\_downloads, pattern = "r", ignore.case = TRUE)

## [1] FALSE TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE

The next 3 functions that we explore differ from the above 2 in the format of  
and amount of details in the results. They all return the following additional  
details:

* the starting position of the first match
* the length of the matched text
* whether the match position and length are in chracter or bytes

**regexpr**

rr\_pkgs <- c("purrr", "olsrr", "blorr")

regexpr("r", rr\_pkgs)

## [1] 3 4 4

## attr(,"match.length")

## [1] 1 1 1

## attr(,"index.type")

## [1] "chars"

## attr(,"useBytes")

## [1] TRUE

**gregexpr**

gregexpr("r", rr\_pkgs)

## [[1]]

## [1] 3 4 5

## attr(,"match.length")

## [1] 1 1 1

## attr(,"index.type")

## [1] "chars"

## attr(,"useBytes")

## [1] TRUE

##

## [[2]]

## [1] 4 5

## attr(,"match.length")

## [1] 1 1

## attr(,"index.type")

## [1] "chars"

## attr(,"useBytes")

## [1] TRUE

##

## [[3]]

## [1] 4 5

## attr(,"match.length")

## [1] 1 1

## attr(,"index.type")

## [1] "chars"

## attr(,"useBytes")

## [1] TRUE

**regexec**

regexec("r", rr\_pkgs)

## [[1]]

## [1] 3

## attr(,"match.length")

## [1] 1

## attr(,"index.type")

## [1] "chars"

## attr(,"useBytes")

## [1] TRUE

##

## [[2]]

## [1] 4

## attr(,"match.length")

## [1] 1

## attr(,"index.type")

## [1] "chars"

## attr(,"useBytes")

## [1] TRUE

##

## [[3]]

## [1] 4

## attr(,"match.length")

## [1] 1

## attr(,"index.type")

## [1] "chars"

## attr(,"useBytes")

## [1] TRUE

**sub**

sub() will perform replacement of the first match. In the below example,  
you can observe that only the first match of r is replaced by s while  
the rest remain the same.

rr\_pkgs <- c("purrr", "olsrr", "blorr")

sub(x = rr\_pkgs, pattern = "r", replacement = "s")

## [1] "pusrr" "olssr" "blosr"

**gsub**

gsub() will perform replacement of all the matches. In the below example,  
all the s are replaced by r. Compare the below output with the output from  
sub() to understand the difference between them.

gsub(x = rr\_pkgs, pattern = "r", replacement = "s")

## [1] "pusss" "olsss" "bloss"

**Regular Expressions**

So far we have been exploring R functions for pattern matching with a very  
simple pattern i.e. a single character. From this section, we will start  
exploring different scenarios and the corresponding regular expressions. This  
section is further divided into 5 sub sections:

* anchors
* metacharacters
* quantifiers
* sequences
* and character classes

**Anchors**

Anchors do not match any character. Instead, they match the pattern supplied to  
a position before, after or between characters i.e. they are used to anchor the  
regex or pattern at a certain position. Anchors are useful when we are searching  
for a pattern at the beggining or end of a string.

**Caret Symbol (^)**

The caret **^** matches the position before the first character in the string.  
In the below example, we want to know the R packages whose names begin with  
the letter r. To achieve this, we use ^, the caret symbol, which specifies  
that the pattern must be present at the beginning of the string.

grep(x = top\_downloads, pattern = "^r", value = TRUE)

## [1] "rlang"

It has returned only one package, rlang but if you look at the package names  
even Rcpp begins with r but has been ignored. Let us ignore the case of the  
pattern and see if the results change.

grep(x = top\_downloads, pattern = "^r", value = TRUE, ignore.case = TRUE)

## [1] "rlang" "Rcpp"

**Dollar Symbol ($)**

The dollar $ matches right after the last character in the string. Let us  
now look at packages whose names end with the letter r. To achieve this, we  
use $, the dollar symbol. As you can see in the below example, the $ is  
specified at the end of the pattern we are looking for.

From our sample data set, we can see that there are 2 packages that end with  
the letter r, dplyr and pillar.

grep(x = top\_downloads, pattern = "r$", value = TRUE)

## [1] "dplyr" "pillar"

**Meta Characters**

Meta characters are a special set of characters not captured by regular  
expressions i.e. if these special characters are present in a string, regular  
expressions will not detect them. In order to be detected, they must be  
prefixed by double backslash (\). The below table displays the metacharacters:

Now that we know the meta characters, let us look at some examples. In the first  
example, we want to detect package names separated by a dot.

grep(x = r\_packages, pattern = ".", value = TRUE)[1:60]

## [1] "A3" "abbyyR" "abc"

## [4] "abc.data" "ABC.RAP" "ABCanalysis"

## [7] "abcdeFBA" "ABCoptim" "ABCp2"

## [10] "abcrf" "abctools" "abd"

## [13] "abe" "abf2" "ABHgenotypeR"

## [16] "abind" "abjutils" "abn"

## [19] "abnormality" "abodOutlier" "ABPS"

## [22] "AbsFilterGSEA" "AbSim" "abstractr"

## [25] "abtest" "abundant" "Ac3net"

## [28] "ACA" "acc" "accelerometry"

## [31] "accelmissing" "AcceptanceSampling" "ACCLMA"

## [34] "accrual" "accrued" "accSDA"

## [37] "ACD" "ACDm" "acebayes"

## [40] "acepack" "ACEt" "acid"

## [43] "acm4r" "ACMEeqtl" "acmeR"

## [46] "ACNE" "acnr" "acopula"

## [49] "AcousticNDLCodeR" "acp" "aCRM"

## [52] "AcrossTic" "acrt" "acs"

## [55] "ACSNMineR" "acss" "acss.data"

## [58] "ACSWR" "ACTCD" "Actigraphy"

If you look at the output, it includes names of even those package names which  
are not separated by dot. Why is this happening? A dot is special character in  
regular expressions. It is also known as wildcard character i.e. it is used to  
match any character other than \n (new line). Now let us try to escape it  
using the double backslash (\\).

grep(x = r\_packages, pattern = "\\.", value = TRUE)[1:50]

## [1] "abc.data" "ABC.RAP" "acss.data"

## [4] "aire.zmvm" "AMAP.Seq" "anim.plots"

## [7] "ANOVA.TFNs" "ar.matrix" "archivist.github"

## [10] "aroma.affymetrix" "aroma.apd" "aroma.cn"

## [13] "aroma.core" "ASGS.foyer" "assertive.base"

## [16] "assertive.code" "assertive.data" "assertive.data.uk"

## [19] "assertive.data.us" "assertive.datetimes" "assertive.files"

## [22] "assertive.matrices" "assertive.models" "assertive.numbers"

## [25] "assertive.properties" "assertive.reflection" "assertive.sets"

## [28] "assertive.strings" "assertive.types" "auto.pca"

## [31] "AWR.Athena" "AWR.Kinesis" "AWR.KMS"

## [34] "aws.alexa" "aws.cloudtrail" "aws.comprehend"

## [37] "aws.ec2metadata" "aws.iam" "aws.kms"

## [40] "aws.lambda" "aws.polly" "aws.s3"

## [43] "aws.ses" "aws.signature" "aws.sns"

## [46] "aws.sqs" "aws.transcribe" "aws.translate"

## [49] "bea.R" "benford.analysis"

When we use \\., it matches the dot. Feel free to play around with other  
special characters mentioned in the table but ensure that you use a different  
data set.

**Quantifiers**

Quantifiers are very powerful and we need to be careful while using them. They  
always act on items to the immediate left and are used to specify the number of  
times a pattern must appear or be matched. The below table shows the different  
quantifiers and their description:

**Dot**

The . (dot) is a wildcard character as it will match any character except a  
new line (). Keep in mind that it will match only 1 character and if you want  
to match more than 1 character, you need to specify as many dots. Let us look  
at a few examples.

# extract package names that include the string data

data\_pkgs <- grep(x = r\_packages, pattern = "data", value = TRUE)

head(data\_pkgs)

## [1] "abc.data" "acss.data" "adeptdata" "adklakedata"

## [5] "archdata" "assertive.data"

# package name includes the string data followed by any character and then the letter r

grep(x = data\_pkgs, pattern = "data.r", value = TRUE, ignore.case = TRUE)

## [1] "datadr" "dataframes2xls" "dataPreparation"

# package name includes the string data followed by any 3 characters and then the letter r

grep(x = data\_pkgs, pattern = "data...r", value = TRUE, ignore.case = TRUE)

## [1] "data.world" "datadogr" "dataRetrieval"

## [4] "datasauRus" "rdataretriever" "WikidataQueryServiceR"

# package name includes the string data followed by any 3 characters and then the letter r

grep(x = r\_packages, pattern = "data(.){3}r", value = TRUE, ignore.case = TRUE)

## [1] "data.world" "datadogr" "DataEntry"

## [4] "dataRetrieval" "datasauRus" "rdataretriever"

## [7] "RWDataPlyr" "WikidataQueryServiceR"

# package name includes the string stat followed by any 2 characters and then the letter r

grep(x = r\_packages, pattern = "stat..r", value = TRUE, ignore.case = TRUE)

## [1] "DistatisR" "snpStatsWriter" "StatPerMeCo"

**Optional Character**

?, the optional character is used when the item to its left is optional and  
is matched at most once.

In this first example, we are looking for package names that include the  
following pattern:

* includes the letter r
* includes the string data
* there may be zero or one character between r and data

grep(x = data\_pkgs, pattern = "r(.)?data", value = TRUE)

## [1] "cluster.datasets" "ctrdata" "dplyr.teradata"

## [4] "engsoccerdata" "historydata" "icpsrdata"

## [7] "mldr.datasets" "prioritizrdata" "qrmdata"

## [10] "rdatacite" "rdataretriever" "rqdatatable"

## [13] "smartdata"

In the below example, we are looking for package names that include the  
following pattern:

* includes the letter r
* includes the string data
* there may be zero or one dot between r and data

grep(x = data\_pkgs, pattern = "r(\\.)?data", value = TRUE)

## [1] "cluster.datasets" "ctrdata" "engsoccerdata"

## [4] "icpsrdata" "mldr.datasets" "prioritizrdata"

## [7] "rdatacite" "rdataretriever"

In the next example, we are looking for package names that include the  
following pattern:

* includes the letter r
* includes the string data
* there may be zero or one character between r and data
* and the character must be any of the following:
  + m
  + y
  + q

grep(x = data\_pkgs, pattern = "r(m|y|q)?data", value = TRUE)

## [1] "ctrdata" "engsoccerdata" "historydata" "icpsrdata"

## [5] "prioritizrdata" "qrmdata" "rdatacite" "rdataretriever"

## [9] "rqdatatable"

In the last example, we are looking for package names that include the  
following pattern:

* includes the letter r
* includes the string data
* there may be zero or one character between r and data
* and the character must be any of the following:
  + m
  + y
  + q
  + dot

grep(x = data\_pkgs, pattern = "r(\\.|m|y|q)?data", value = TRUE)

## [1] "cluster.datasets" "ctrdata" "engsoccerdata"

## [4] "historydata" "icpsrdata" "mldr.datasets"

## [7] "prioritizrdata" "qrmdata" "rdatacite"

## [10] "rdataretriever" "rqdatatable"

**Asterik Symbol**

\*, the asterik symbol is used when the item to its left will be matched zero  
or more times.

In the below example, we are looking for package names that include the  
following pattern:

* includes the letter r
* includes the string data
* there may be zero or more character(s) between r and data

grep(x = data\_pkgs, pattern = "r(.)\*data", value = TRUE)

## [1] "archdata" "assertive.data" "assertive.data.uk"

## [4] "assertive.data.us" "cluster.datasets" "crimedata"

## [7] "cropdatape" "ctrdata" "dplyr.teradata"

## [10] "engsoccerdata" "groupdata2" "historydata"

## [13] "icpsrdata" "igraphdata" "mldr.datasets"

## [16] "nordklimdata1" "prioritizrdata" "qrmdata"

## [19] "radiant.data" "rangeModelMetadata" "rattle.data"

## [22] "rbefdata" "rdatacite" "rdataretriever"

## [25] "rehh.data" "resampledata" "rnaturalearthdata"

## [28] "ropendata" "rqdatatable" "rtsdata"

## [31] "smartdata" "surveydata" "survJamda.data"

## [34] "traitdataform" "vortexRdata" "xmlparsedata"

**Plus Symbol**

+, the plus symbol is used when the item to its left is matched one or more  
times.

In the below example, we are looking for package names that include the  
following pattern:

* includes the string plot
* plot is preceded by one or more g

plot\_pkgs <- grep(x = r\_packages, pattern = "plot", value = TRUE)

grep(x = plot\_pkgs, pattern = "(g)+plot", value = TRUE, ignore.case = TRUE)

## [1] "ggplot2" "ggplot2movies" "ggplotAssist"

## [4] "ggplotgui" "ggplotify" "gplots"

## [7] "RcmdrPlugin.KMggplot2" "regplot"

**Brackets**

**{n}**

{n} is used when the item to its left is matched exactly n times. In the  
below example, we are looking for package names that include the following  
pattern:

* includes the string plot
* plot is preceded by exactly one g

grep(x = plot\_pkgs, pattern = "(g){2}plot", value = TRUE)

## [1] "ggplot2" "ggplot2movies" "ggplotAssist"

## [4] "ggplotgui" "ggplotify" "RcmdrPlugin.KMggplot2"

**{n,}**

{n, } is used when the item to its left is matched n or more times. In the  
below example, we are looking for package names that include the following  
pattern:

* includes the string plot
* plot is preceded by one or more g

grep(x = plot\_pkgs, pattern = "(g){1, }plot", value = TRUE)

## [1] "ggplot2" "ggplot2movies" "ggplotAssist"

## [4] "ggplotgui" "ggplotify" "gplots"

## [7] "RcmdrPlugin.KMggplot2" "regplot"

**{n,m}**

{n, m} is used when the item to its left is matched at least n times but not  
more than m times. In the below example, we are looking for package names that  
include the following pattern:

* includes the string plot
* plot is preceded by 1 or 3 t

grep(x = plot\_pkgs, pattern = "(t){1,3}plot", value = TRUE)

## [1] "forestplot" "limitplot" "nhstplot" "productplots"

## [5] "tttplot" "txtplot"

**OR**



The | (OR) operator is useful when you want to match one amongst the given  
options. For example, let us say we are looking for package names that begin  
with g and is followed by either another g or l.

grep(x = top\_downloads, pattern = "g(g|l)", value = TRUE)

## [1] "ggplot2" "glue"

The square brackets ([]) can be used in place of | as shown in the below  
example where we are looking for package names that begin with the letter  
d and is followed by either e or p or a.

grep(x = top\_downloads, pattern = "d[epa]", value = TRUE)

## [1] "devtools" "dplyr" "data.table"

**Sequences**

**Digit Character**

\\d matches any digit character. Let us use it to find package names that  
include a digit.

grep(x = r\_packages, pattern = "\\d", value = TRUE)[1:50]

## [1] "A3" "ABCp2" "abf2"

## [4] "Ac3net" "acm4r" "ade4"

## [7] "ade4TkGUI" "AdvDif4" "ALA4R"

## [10] "alphashape3d" "alr3" "alr4"

## [13] "ANN2" "aods3" "aplore3"

## [16] "APML0" "aprean3" "AR1seg"

## [19] "arena2r" "arf3DS4" "argon2"

## [22] "ARTP2" "aster2" "auth0"

## [25] "aws.ec2metadata" "aws.s3" "B2Z"

## [28] "b6e6rl" "base2grob" "base64"

## [31] "base64enc" "base64url" "BaTFLED3D"

## [34] "BayClone2" "BayesS5" "bc3net"

## [37] "BCC1997" "BDP2" "BEQI2"

## [40] "BHH2" "bikeshare14" "bio3d"

## [43] "biomod2" "Bios2cor" "bios2mds"

## [46] "biostat3" "bipartiteD3" "bit64"

## [49] "Bolstad2" "BradleyTerry2"

# invert

grep(x = r\_packages, pattern = "\\d", value = TRUE, invert = TRUE)[1:50]

## [1] "abbyyR" "abc" "abc.data"

## [4] "ABC.RAP" "ABCanalysis" "abcdeFBA"

## [7] "ABCoptim" "abcrf" "abctools"

## [10] "abd" "abe" "ABHgenotypeR"

## [13] "abind" "abjutils" "abn"

## [16] "abnormality" "abodOutlier" "ABPS"

## [19] "AbsFilterGSEA" "AbSim" "abstractr"

## [22] "abtest" "abundant" "ACA"

## [25] "acc" "accelerometry" "accelmissing"

## [28] "AcceptanceSampling" "ACCLMA" "accrual"

## [31] "accrued" "accSDA" "ACD"

## [34] "ACDm" "acebayes" "acepack"

## [37] "ACEt" "acid" "ACMEeqtl"

## [40] "acmeR" "ACNE" "acnr"

## [43] "acopula" "AcousticNDLCodeR" "acp"

## [46] "aCRM" "AcrossTic" "acrt"

## [49] "acs" "ACSNMineR"

In the next few examples, we will not use R package names data, instead we will  
use dummy data of Invoice IDs and see if they conform to certain rules such as:

* they should include letters and numbers
* they should not include symbols
* they should not include space or tab

**Non Digit Character**

\\D matches any non-digit character. Let us use it to remove invoice ids that  
include only numbers and no letters.

As you can see below, thre are 3 invoice ids that did not conform to the rules  
and have been removed. Only those invoice ids that have both letter and numbers  
have been returned.

invoice\_id <- c("R536365", "R536472", "R536671", "536915", "R536125", "R536287",

"536741", "R536893", "R536521", "536999")

grep(x = invoice\_id, pattern = "\\D", value = TRUE)

## [1] "R536365" "R536472" "R536671" "R536125" "R536287" "R536893" "R536521"

# invert

grep(x = invoice\_id, pattern = "\\D", value = TRUE, invert = TRUE)

## [1] "536915" "536741" "536999"

**White Space Character**

\\s matches any white space character such as space or tab. Let us use it to  
detect invoice ids that include any white space (space or tab).

As you can see below, there are 4 invoice ids that include white space  
character.

grep(x = c("R536365", "R 536472", "R536671", "R536915", "R53 6125", "R536287",

"536741", "R5368 93", "R536521", "536 999"),

pattern = "\\s", value = TRUE)

## [1] "R 536472" "R53 6125" "R5368 93" "536 999"

grep(x = c("R536365", "R 536472", "R536671", "R536915", "R53 6125", "R536287",

"536741", "R5368 93", "R536521", "536 999"),

pattern = "\\s", value = TRUE, invert = TRUE)

## [1] "R536365" "R536671" "R536915" "R536287" "536741" "R536521"

**Non White Space Character**

\\S matches any non white space character. Let us use it to remove any  
invoice ids which are blank or missing.

As you can see below, two invoice ids which were blank have been removed. If you  
observe carefully, it does not remove any invoice ids which have a white space  
character present, it only removes those which are completely blank i.e. those  
which include only space or tab.

grep(x = c("R536365", "R 536472", " ", "R536915", "R53 6125", "R536287",

" ", "R5368 93", "R536521", "536 999"),

pattern = "\\S", value = TRUE)

## [1] "R536365" "R 536472" "R536915" "R53 6125" "R536287" "R5368 93"

## [7] "R536521" "536 999"

# invert

grep(x = c("R536365", "R 536472", " ", "R536915", "R53 6125", "R536287",

" ", "R5368 93", "R536521", "536 999"),

pattern = "\\S", value = TRUE, invert = TRUE)

## [1] " " " "

**Word Character**

\\w matches any word character i.e. alphanumeric. It includes the following:

* a to z
* A to Z
* 0 to 9
* underscore(\_)

Let us use it to remove those invoice ids which include only symbols or special  
characters. Again, you can see that it does not remove those ids which include  
both word characters and symbols as it will match any string that includes  
word characters.

grep(x = c("R536365", "%+$!#@?", "R536671", "R536915", "$%+#!@?", "R5362@87",

"53+67$41", "R536893", "@$+%#!", "536#999"),

pattern = "\\w", value = TRUE)

## [1] "R536365" "R536671" "R536915" "R5362@87" "53+67$41" "R536893"

## [7] "536#999"

# invert

grep(x = c("R536365", "%+$!#@?", "R536671", "R536915", "$%+#!@?", "R5362@87",

"53+67$41", "R536893", "@$+%#!", "536#999"),

pattern = "\\w", value = TRUE, invert = TRUE)

## [1] "%+$!#@?" "$%+#!@?" "@$+%#!"

**Non Word Character**

\\W matches any non-word character i.e. symbols. It includes everything that  
is not a word character.

Let us use it to detect invoice ids that include any non-word character. As you  
can see only 4 ids do not include non-word characters.

grep(x = c("R536365", "%+$!#@?", "R536671", "R536915", "$%+#!@?", "R5362@87",

"53+67$41", "R536893", "@$+%#!", "536#999"),

pattern = "\\W", value = TRUE)

## [1] "%+$!#@?" "$%+#!@?" "R5362@87" "53+67$41" "@$+%#!" "536#999"

# invert

grep(x = c("R536365", "%+$!#@?", "R536671", "R536915", "$%+#!@?", "R5362@87",

"53+67$41", "R536893", "@$+%#!", "536#999"),

pattern = "\\W", value = TRUE, invert = TRUE)

## [1] "R536365" "R536671" "R536915" "R536893"

**Word Boundary**

\\b and \\B are similar to caret and dollar symbol. They match at a position  
called word boundary. Now, what is a word boundary? The following 3 positions  
qualify as word boundaries:

* before the first character in the string
* after the last character in the string
* between two characters in the string

In the first 2 cases, the character must be a word character whereas in the  
last case, one should be a word character and another non-word character. Sounds  
confusing? It will be clear once we go through a few examples.

Let us say we are looking for package names beginning with the string stat.  
In this case, we can prefix stat with \\b.

grep(x = r\_packages, pattern = "\\bstat", value = TRUE)

## [1] "haplo.stats" "statar" "statcheck" "statebins"

## [5] "states" "statGraph" "stationery" "statip"

## [9] "statmod" "statnet" "statnet.common" "statnetWeb"

## [13] "statprograms" "statquotes" "stats19" "statsDK"

## [17] "statsr" "statVisual"

Suffix \\b to stat to look at all package names that end with the string  
stat.

If you observe the output, you can find package names that do not end with  
the string stat. spatstat.data, spatstat.local and spatstat.utils do not  
end with stat but satisfy the third condition mentioned aboved for word  
boundaries. They are between 2 characters where t is a word character and dot  
is a non-word character.

grep(x = r\_packages, pattern = "stat\\b", value = TRUE)

## [1] "Blendstat" "costat" "dstat"

## [4] "eurostat" "gstat" "hierfstat"

## [7] "jsonstat" "lawstat" "lestat"

## [10] "lfstat" "LS2Wstat" "maxstat"

## [13] "mdsstat" "mistat" "poolfstat"

## [16] "Pstat" "RcmdrPlugin.lfstat" "rfacebookstat"

## [19] "Rilostat" "rjstat" "RMTstat"

## [22] "sgeostat" "spatstat" "spatstat.data"

## [25] "spatstat.local" "spatstat.utils" "volleystat"

Do package names include the string stat either at the end or in the middle  
but not at the beginning? Prefix stat with \\B to find the answer.

grep(x = r\_packages, pattern = "\\Bstat", value = TRUE)

## [1] "bigstatsr" "biostat3" "Blendstat"

## [4] "compstatr" "costat" "cumstats"

## [7] "curstatCI" "CytobankAPIstats" "dbstats"

## [10] "descstatsr" "DistatisR" "dlstats"

## [13] "dostats" "dstat" "estatapi"

## [16] "eurostat" "freestats" "geostatsp"

## [19] "gestate" "getmstatistic" "ggstatsplot"

## [22] "groupedstats" "gstat" "hierfstat"

## [25] "hydrostats" "jsonstat" "labstatR"

## [28] "labstats" "lawstat" "learnstats"

## [31] "lestat" "lfstat" "LS2Wstat"

## [34] "maxstat" "mdsstat" "mistat"

## [37] "mlbstats" "mstate" "multistate"

## [40] "multistateutils" "ohtadstats" "orderstats"

## [43] "p3state.msm" "poolfstat" "PRISMAstatement"

## [46] "Pstat" "raustats" "RcmdrPlugin.lfstat"

## [49] "readstata13" "realestateDK" "restatapi"

## [52] "rfacebookstat" "Rilostat" "rjstat"

## [55] "RMTstat" "rstatscn" "runstats"

## [58] "scanstatistics" "sgeostat" "sjstats"

## [61] "spatstat" "spatstat.data" "spatstat.local"

## [64] "spatstat.utils" "TDAstats" "tidystats"

## [67] "tigerstats" "tradestatistics" "unsystation"

## [70] "USGSstates2k" "volleystat" "wbstats"

Are there packages whose names include the string stat either at the  
beginning or in the middle but not at the end. Suffix \\B to stat to  
answer this question.

grep(x = r\_packages, pattern = "stat\\B", value = TRUE)

## [1] "bigstatsr" "biostat3" "compstatr"

## [4] "cumstats" "curstatCI" "CytobankAPIstats"

## [7] "dbstats" "descstatsr" "DistatisR"

## [10] "dlstats" "dostats" "estatapi"

## [13] "freestats" "geostatsp" "gestate"

## [16] "getmstatistic" "ggstatsplot" "groupedstats"

## [19] "haplo.stats" "hydrostats" "labstatR"

## [22] "labstats" "learnstats" "mlbstats"

## [25] "mstate" "multistate" "multistateutils"

## [28] "ohtadstats" "orderstats" "p3state.msm"

## [31] "PRISMAstatement" "raustats" "readstata13"

## [34] "realestateDK" "restatapi" "rstatscn"

## [37] "runstats" "scanstatistics" "sjstats"

## [40] "statar" "statcheck" "statebins"

## [43] "states" "statGraph" "stationery"

## [46] "statip" "statmod" "statnet"

## [49] "statnet.common" "statnetWeb" "statprograms"

## [52] "statquotes" "stats19" "statsDK"

## [55] "statsr" "statVisual" "TDAstats"

## [58] "tidystats" "tigerstats" "tradestatistics"

## [61] "unsystation" "USGSstates2k" "wbstats"

Prefix and suffix \\B to stat to look at package names that include the  
string stat but neither in the beginning nor in the end.

In the below output, you can observe that the string stat must be between  
two word characters. Those examples we showed in the case of \\b where it  
was surrounded by a dot do not hold here.

grep(x = r\_packages, pattern = "\\Bstat\\B", value = TRUE)

## [1] "bigstatsr" "biostat3" "compstatr"

## [4] "cumstats" "curstatCI" "CytobankAPIstats"

## [7] "dbstats" "descstatsr" "DistatisR"

## [10] "dlstats" "dostats" "estatapi"

## [13] "freestats" "geostatsp" "gestate"

## [16] "getmstatistic" "ggstatsplot" "groupedstats"

## [19] "hydrostats" "labstatR" "labstats"

## [22] "learnstats" "mlbstats" "mstate"

## [25] "multistate" "multistateutils" "ohtadstats"

## [28] "orderstats" "p3state.msm" "PRISMAstatement"

## [31] "raustats" "readstata13" "realestateDK"

## [34] "restatapi" "rstatscn" "runstats"

## [37] "scanstatistics" "sjstats" "TDAstats"

## [40] "tidystats" "tigerstats" "tradestatistics"

## [43] "unsystation" "USGSstates2k" "wbstats"

**Character Classes**

A set of characters enclosed in a square bracket ([]). The regular expression  
will match only those characters enclosed in the brackets and it matches only a  
single character. The order of the characters inside the brackets do not matter  
and a hyphen can be used to specify a range of charcters. For example, [0-9]  
will match a single digit between 0 and 9. Similarly, [a-z] will match a single  
letter between a to z. You can specify more than one range as well. [a-z0-9A-Z]  
will match a alphanumeric character while ignoring the case. A caret **^** after  
the opening bracket negates the character class. For example, [^0-9] will match  
a single character that is not a digit.

Let us go through a few examples to understand character classes in more detail.

# package names that include vowels

grep(x = top\_downloads, pattern = "[aeiou]", value = TRUE)

## [1] "devtools" "rlang" "tibble" "ggplot2" "glue"

## [6] "pillar" "cli" "data.table"

# package names that include a number

grep(x = r\_packages, pattern = "[0-9]", value = TRUE)[1:50]

## [1] "A3" "ABCp2" "abf2"

## [4] "Ac3net" "acm4r" "ade4"

## [7] "ade4TkGUI" "AdvDif4" "ALA4R"

## [10] "alphashape3d" "alr3" "alr4"

## [13] "ANN2" "aods3" "aplore3"

## [16] "APML0" "aprean3" "AR1seg"

## [19] "arena2r" "arf3DS4" "argon2"

## [22] "ARTP2" "aster2" "auth0"

## [25] "aws.ec2metadata" "aws.s3" "B2Z"

## [28] "b6e6rl" "base2grob" "base64"

## [31] "base64enc" "base64url" "BaTFLED3D"

## [34] "BayClone2" "BayesS5" "bc3net"

## [37] "BCC1997" "BDP2" "BEQI2"

## [40] "BHH2" "bikeshare14" "bio3d"

## [43] "biomod2" "Bios2cor" "bios2mds"

## [46] "biostat3" "bipartiteD3" "bit64"

## [49] "Bolstad2" "BradleyTerry2"

# package names that begin with a number

grep(x = r\_packages, pattern = "^[0-9]", value = TRUE)

## character(0)

# package names that end with a number

grep(x = r\_packages, pattern = "[0-9]$", value = TRUE)[1:50]

## [1] "A3" "ABCp2" "abf2"

## [4] "ade4" "AdvDif4" "alr3"

## [7] "alr4" "ANN2" "aods3"

## [10] "aplore3" "APML0" "aprean3"

## [13] "arf3DS4" "argon2" "ARTP2"

## [16] "aster2" "auth0" "aws.s3"

## [19] "base64" "BayClone2" "BayesS5"

## [22] "BCC1997" "BDP2" "BEQI2"

## [25] "BHH2" "bikeshare14" "biomod2"

## [28] "biostat3" "bipartiteD3" "bit64"

## [31] "Bolstad2" "BradleyTerry2" "brglm2"

## [34] "bridger2" "c060" "c212"

## [37] "c3" "C443" "C50"

## [40] "cAIC4" "CARE1" "CB2"

## [43] "cec2013" "Census2016" "Chaos01"

## [46] "choroplethrAdmin1" "cld2" "cld3"

## [49] "clogitL1" "CLONETv2"

# package names with only upper case letters

grep(x = r\_packages, pattern = "^[A-Z][A-Z]{1, }[A-Z]$", value = TRUE)[1:50]

## [1] "ABPS" "ACA" "ACCLMA" "ACD" "ACNE" "ACSWR" "ACTCD"

## [8] "ADCT" "ADDT" "ADMM" "ADPF" "AER" "AFM" "AGD"

## [15] "AHR" "AID" "AIG" "AIM" "ALS" "ALSCPC" "ALSM"

## [22] "AMCP" "AMGET" "AMIAS" "AMOEBA" "AMORE" "AMR" "ANOM"

## [29] "APSIM" "ARHT" "AROC" "ART" "ARTIVA" "ARTP" "ASIP"

## [36] "ASSA" "AST" "ATE" "ATR" "AUC" "AUCRF" "AWR"

## [43] "BACA" "BACCO" "BACCT" "BALCONY" "BALD" "BALLI" "BAMBI"

## [50] "BANOVA"

**Case Studies**

Now that we have understood the basics of regular expressions, it is time for  
some practical application. The case studies in this section include validating  
the following:

* blood group
* email id
* PAN number
* GST number

Note, the regular expressions used here are not robust as compared to those  
used in real world applications. Our aim is to demonstrate a general strategy  
to used while dealing with regular expressions.

**Blood Group**

According to Wikipedia, a blood group or type is a classification of blood based  
on the presence and absence of antibodies and inherited antigenic substances on  
the surface of red blood cells (RBCs).



The below table defines the matching pattern for blood group and maps them to  
regular expressions.

* it must begin with A, B, AB or O
* it must end with + or -

Let us test the regular expression with some examples.

blood\_pattern <- "^(A|B|AB|O)[-|+]$"

blood\_sample <- c("A+", "C-", "AB+")

grep(x = blood\_sample, pattern = blood\_pattern, value = TRUE)

## [1] "A+" "AB+"

**email id**

Nowadays email is ubiquitous. We use it for everything from communication to  
registration for online services. Wherever you go, you will be asked for email  
id. You might even be denied a few services if you do not use email. At the same  
time, it is important to validate a email address. You might have seen a message  
similar to the below one when you misspell or enter a wrong email id. Regular  
expressions are used to validate email address and in this case study we will  
attempt to do the same.

First, we will create some basic rules for simple email validation:

* it must begin with a letter
* the id may include letters, numbers and special characters
* must include only one @ and dot
* the id must be to the left of @
* the domain name should be between @ and dot
* the domain extension should be after dot and must include only letters

In the below table, we map the above rules to general expression.Let us now test the regular expression with some dummy email ids.

email\_pattern <- "^[a-zA-Z0-9\\!#$%&'\*+/=?^\_`{|}~-]+@[a-zA-Z0-9-]+\\.[a-z]"

emails <- c("test9+\_A@test.com", "test@test..com", "test-test.com")

grep(x = emails, pattern = email\_pattern, value = TRUE)

## [1] "test9+\_A@test.com"

**PAN Number Validation**

PAN (Permanent Account Number) is an identification number assigned to all  
taxpayers in India. PAN is an electronic system through which, all tax related  
information for a person/company is recorded against a single PAN number.

**Structure**

* must include only 10 characters
* the first 5 characters are letters
* the next 4 characters are numerals
* the last character is a letter
* the first 3 characters are a sequence from AAA to ZZZ
* the 4th character indicates the status of the tax payer and shold be one of  
  A, B, C, F, G, H, L, J, P, T or E
* the 5th character is the first character of the last/surname of the card holder
* the 6th to 10th character is a sequnce from 0001 to 9999
* the last character is a letter

In the below table, we map the pattern to regular expression.

Let us test the regular expression with some dummy PAN numbers.

pan\_pattern <- "^[A-Z]{3}(A|B|C|F|G|H|L|J|P|T|E)[A-Z][0-9]{4}[A-Z]"

my\_pan <- c("AJKNT3865H", "AJKNT38655", "A2KNT3865H", "AJKDT3865H")

grep(x = my\_pan, pattern = pan\_pattern, value = TRUE)

## character(0)

**GST Number Validation**

In simple words, Goods and Service Tax (GST) is an indirect tax levied on the  
supply of goods and services. This law has replaced many indirect tax laws that  
previously existed in India. GST identification number is assigned to every  
GST registed dealer.

**Structure**

Below is the format break down of GST identification number:

* it must include 15 characters only
* the first 2 characters represent the state code and is a sequence from 01 to 35
* the next 10 characters are the PAN number of the entity
* the 13th character is the entity code and is between 1 and 9
* the 14th character is a default alphabet, **Z**
* the 15th character is a random single number or alphabet

In the below table, we map the pattern to regular expression.

Let us test the regular expression with some dummy GST numbers.

gst\_pattern <- "[0-3][1-5][A-Z]{3}(A|B|C|F|G|H|L|J|P|T|E)[A-Z][0-9]{4}[A-Z][1-9]Z[0-9A-Z]"

sample\_gst <- c("22AAAAA0000A1Z5", "22AAAAA0000A1Z", "42AAAAA0000A1Z5",

"38AAAAA0000A1Z5", "22AAAAA0000A0Z5", "22AAAAA0000A1X5",

"22AAAAA0000A1Z$")

grep(x = sample\_gst, pattern = gst\_pattern, value = TRUE)

## [1] "22AAAAA0000A1Z5"

**Other Applications**

* R variable names
* R file names and extensions
* password validation
* camelcase
* currency format
* date of birth
* date validation
* decimal number
* full name / first name
* html tags
* https url
* phone number
* ip address
* month name

**What we have not covered?**

While we have covered a lot, the below topics have been left out:

* flags
* grouping and capturing
* back references
* look ahead and look behind

You may want to explore them to up your regular expressions game.

**Summary**

* a regular expression is a special text for identifying a pattern
* it can be used to search, replace, validate and extract strings matching a  
  given pattern
* use cases include email and password validation, search and replace in text  
  editors, html tags validation, web scraping etc.