WORKING WITH STRINGS

Sources: H. Wickham, G.
Sanchez, & UC Programming Guide

Dealing with Characters

- Character string basics
- String manipulation with base R
- String manipulation with stringr
- Set operatons for character strings

Character string basics

- How to create, convert and print character strings
- How to count the number of elements and characters in a string

Creating Strings

- use quotation marks and assign a string to an object

```
a <- "learning to create"    # create string a

b <- "character strings"     # create string b
```

Creating Strings

- paste() function provides a versatile means for creating and building strings

```
# paste together string a & b
paste(a, b)
## [1] "learning to create character strings"

# paste character and number strings (converts numbers to character class)
paste("The life of", pi)
## [1] "The life of 3.14159265358979"
```

Creating Strings

- paste() function provides a versatile means for creating and building strings

```
# paste multiple strings
paste("I", "love", "R")
## [1] "I love R"

# paste multiple strings with a separating character
paste("I", "love", "R", sep = "-")
## [1] "I-love-R"
```

Creating Strings

- paste() function provides a versatile means for creating and building strings

```
# use paste0() to paste without spaces btwn characters
paste0("I", "love", "R")
## [1] "IloveR"

# paste objects with different lengths
paste("R", 1:5, sep = " v1.")
## [1] "R v1.1" "R v1.2" "R v1.3" "R v1.4" "R v1.5"
```

Converting to Strings

- Test if strings are characters with is.character() and convert strings to character with as.character() or with toString()

```
a <- "The life of"
b <- pi

is.character(a)
## [1] TRUE

is.character(b)
## [1] FALSE
```

Converting to Strings

- Test if strings are characters with is.character() and convert strings to character with as.character() or with toString()

```
c <- as.character(b)
is.character(c)
## [1] TRUE

toString(c("Aug", 24, 1980))
## [1] "Aug, 24, 1980"
```

Printing Strings

- Common printing methods include:
  - *print(): generic printing*
  - *noquote(): print with no quotes*
  - *cat(): concatenate and print with no quotes*

Printing Strings

- The primary printing function in R is print()

```
x <- "learning to print strings"

# basic printing
    print(x)
## [1] "learning to print strings"

# print without quotes
    print(x, quote = FALSE)
## [1] learning to print strings
```

Printing Strings

- An alternative to printing a string without quotes is to use noquote()

```
noquote(x)

## [1] learning to print strings
```

Printing Strings

- cat() function:
  - *allows us to concatenate objects and print them either on screen or* ***to a file.***

    a<- as.character(c(1,2,3))

    cat(a, file = "catoutput.txt")

  - *Very similar to noquote();*
    - however, cat() does not print the numeric line indicator.

More cat() examples…

```
# basic printing (similar to noquote)
cat(x)
## learning to print strings

# combining character strings
cat(x, "in R")
## learning to print strings in R
```

More cat() examples…

```
# basic printing of alphabet
cat(letters)
## a b c d e f g h i j k l m n o p q r s t u v w x y z

# specify a seperator between the combined characters
cat(letters, sep = "-")
## a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z

# collapse the space between the combine characters
cat(letters, sep = "")
## abcdefghijklmnopqrstuvwxyz
```

Counting string elements and characters

- To count the number of characters in a string use nchar():

```
nchar("How many characters are in this string?")
## [1] 39

nchar(c("How", "many", "characters", "are", "in", "this", "string?"))
## [1]  3  4 10  3  2  4  7
```

String manipulation with base R

- Can use base R for:
  - *case conversion,*
  - *abbreviating,*
  - *substring replacement,*
  - *adding/removing whitespace, and*
  - *performing set operations to compare similarities and differences between two character vectors*

Case conversion

- To convert all upper case characters to lower case use tolower():

```
x <- "Learning To MANIPULATE strinGS in R"

tolower(x)
## [1] "learning to manipulate strings in r"
```

Case conversion

- To convert all lower case characters to upper case use toupper():

```
toupper(x)
```

```
## [1] "LEARNING TO MANIPULATE STRINGS IN R"
```

Simple Character Replacement

- To replace a character (or multiple characters) in a string you can use chartr():

```
# replace 'A' with 'a'
x <- "This is A string."
chartr(old = "A", new = "a", x)
## [1] "This is a string."

# multiple character replacements
# replace any 'd' with 't' and any 'z' with 'a‘

y <- "Tomorrow I plzn do lezrn zbout dexduzl znzlysis."

chartr(old = "dz", new = "ta", y)
## [1] "Tomorrow I plan to learn about textual analysis."
```

Simple Character Replacement

- To replace a character (or multiple characters) in a string you can use chartr():

Note: Replaces every identified letter for replacement

*Use it is when change **every** possible occurrence of a letter.*

String Abbreviations

- To abbreviate strings you can use abbreviate()
  - *Abbreviates strings to at least minlength characters, such that they remain unique*

```
streets <- c("Main", "Elm", "Riverbend", "Mario", "Frederick")

# default abbreviations
abbreviate(streets)
##      Main       Elm Riverbend     Mario Frederick
##    "Main"     "Elm"    "Rvrb"    "Mari"    "Frdr"

# set minimum length of abbreviation
abbreviate(streets, minlength = 2)
##      Main       Elm Riverbend     Mario Frederick
##      "Mn"      "El"      "Rv"      "Mr"      "Fr"
```

Extract/Replace Substrings

- Three primary base R functions to use:
  - *substr(),*
  - *substring(), and*
  - *strsplit()*

Extract/Replace Substrings

- The purpose of substr() is to extract and replace substrings with specified starting and stopping characters:

```
alphabet <- paste(LETTERS, collapse = "")

# extract 18th character in string
substr(alphabet, start = 18, stop = 18)
## [1] "R"

# extract 18-24th characters in string
substr(alphabet, start = 18, stop = 24)
## [1] "RSTUVWX"
```

Extract/Replace Substrings

- The purpose of substr() is to extract and replace substrings with specified starting and stopping characters:

```
# replace 1st-17th characters with `R`
substr(alphabet, start = 19, stop = 24) <- "RRRRRR"
alphabet
## [1] "ABCDEFGHIJKLMNOPQRRRRRRRYZ"
```

Extract/Replace Substrings

- The purpose of substring() is to extract and replace substrings with only a specified starting point.
- allows you to extract/replace in a recursive fashion:

```
# extract 18th through last character
substring(alphabet, first = 18)
## [1] "RSTUVWXYZ"
```

Extract/Replace Substrings

```
# recursive extraction; specify start position only

substring(alphabet, first = 18:24)

## [1] "RSTUVWXYZ" "STUVWXYZ"  "TUVWXYZ"   "UVWXYZ"
"VWXYZ"     "WXYZ"

## [7] "XYZ"


# recursive extraction; specify start and stop positions

substring(alphabet, first = 1:5, last = 3:7) #1:3, 2:4, 3:5, etc.

## [1] "ABC" "BCD" "CDE" "DEF" "EFG"
```

Extract/Replace Substrings

- To split the elements of a character string use strsplit():

```
z <- "The day after I will take a break and drink a beer."
strsplit(z, split = " ")
## [[1]]
##  [1] "The"   "day"   "after" "I"     "will"  "take"  "a"     "break"
##  [9] "and"   "drink" "a"     "beer."

a <- "Alabama-Alaska-Arizona-Arkansas-California"
strsplit(a, split = "-")
## [[1]]
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
```

Extract/Replace Substrings

- To convert the output to a simple vector simply wrap in unlist():

```
unlist(strsplit(a, split = "-"))
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
```

String manipulation with stringr

- The stringr package was developed by Hadley Wickham to act as simple wrappers that make R’s string functions more consistent, simple, and easier to use.
  - *Concatenate with str_c()*
  - *Number of characters with str_length()*
  - *Substring with str_sub()*
- Functions also extend base R in different ways.

String manipulation with stringr

- str_c() is equivalent to the paste() functions:

```
# same as paste0()
str_c("Learning", "to", "use", "the", "stringr", "package")
## [1] "Learningtousethestringrpackage"

# same as paste()
str_c("Learning", "to", "use", "the", "stringr", "package", sep = " ")
## [1] "Learning to use the stringr package"
```

String manipulation with stringr

- str_c() is equivalent to the paste() functions:

```
# can combine objects w/ diff’t lengths
str_c(letters, " is for", "…")
##  [1] "a is for…" "b is for…" "c is for…" "d is for…" "e is for…"
##  [6] "f is for…" "g is for…" "h is for…" "i is for…" "j is for…"
## [11] "k is for…" "l is for…" "m is for…" "n is for…" "o is for…"
## [16] "p is for…" "q is for…" "r is for…" "s is for…" "t is for…"
## [21] "u is for…" "v is for…" "w is for…" "x is for…" "y is for…"
## [26] "z is for…"
```

String manipulation with stringr

- str_length() is similiar to the nchar() function; however, str_length() ***behaves more appropriately with missing (‘NA’) values***:

```
# some text with NA
text = c("Learning", "to", NA, "use", "the", NA, "stringr", "package")

# compare `str_length()` with `nchar()`
nchar(text)
## [1] 8 2 2 3 3 2 7 7

str_length(text)
## [1]  8  2 NA  3  3 NA  7  7
```

String manipulation with stringr

- str_sub() is similar to substr();
  - *however, it returns a zero length vector if any of its inputs are zero length*

```
x <- "Learning to use the stringr package"
# alternative indexing
    str_sub(x, start = 1, end = 15)
    ## [1] "Learning to use"

    str_sub(x, end = 15)
    ## [1] "Learning to use"

    str_sub(x, start = 17)
    ## [1] "the stringr package"

    str_sub(x, start = c(1, 17), end = c(15, 35))
    ## [1] "Learning to use"     "the stringr package"
```

String manipulation with stringr

- str_sub() is similar to substr();
  - *It also accepts negative positions, which are calculated from the left of the last character.*

```
# using negative indices for start/end points from end of string
str_sub(x, start = -1)
## [1] "e"

str_sub(x, start = -19)
## [1] "the stringr package"

str_sub(x, end = -21)
## [1] "Learning to use"
```

String manipulation with stringr

- str_sub() is similar to substr();
  - *It can also be used to replace text*

```
# Replacement
str_sub(x, end = 15) <- "I know how to use"
x
## [1] "I know how to use the stringr package"
```

String manipulation with stringr

- Stringr also let’s us duplicate characters within a string with str_dup()

```
str_dup("beer", times = 3)
## [1] "beerbeerbeer"

str_dup("beer", times = 1:3)
## [1] "beer"         "beerbeer"     "beerbeerbeer"

# use with a vector of strings
states_i_luv <- state.name[c(6, 23, 34, 35)]
str_dup(states_i_luv, times = 2)
## [1] "ColoradoColorado"         "MinnesotaMinnesota"
## [3] "North DakotaNorth Dakota" "OhioOhio"
```

String manipulation with stringr

- Or trim whitespace with str_trim()

```
text <- c("Text ", "  with", " whitespace ", " on", "both ", " sides ")

# remove whitespaces on the left side
str_trim(text, side = "left")
## [1] "Text "       "with"        "whitespace " "on"          "both "
## [6] "sides "

# remove whitespaces on the right side
str_trim(text, side = "right")
## [1] "Text"        "  with"      " whitespace" " on"         "both"
## [6] " sides"
```

String manipulation with stringr

- Or trim whitespace with str_trim()

```
text <- c("Text ", "  with", " whitespace ", " on", "both ", " sides ")

# remove whitespaces on both sides
str_trim(text, side = "both")
## [1] "Text"       "with"       "whitespace" "on"         "both"
## [6] "sides"
```

Set operatons for character strings

- Base R functions that allows for assessing the set…
  - *union,*
  - *intersection,*
  - *difference,*
  - *equality, and*
  - *membership of two vectors*

Set Union

- To obtain the elements of the union between two character vectors use union():

```
set_1 <- c("lagunitas", "bells", "dogfish", "summit",
"odell")
set_2 <- c("sierra", "bells", "harpoon", "lagunitas",
"founders")

union(set_1, set_2)
## [1] "lagunitas" "bells"     "dogfish"   "summit"
"odell"     "sierra"
## [7] "harpoon"   "founders"
```

Set Intersect

- To obtain the common elements of two character vectors use intersect():

```
intersect(set_1, set_2)
## [1] "lagunitas" "bells"
```

Identifying Different Elements

- To obtain the non-common elements, or the difference, of two character vectors use setdiff():

```
# returns elements in set_1 not in set_2
setdiff(set_1, set_2)
## [1] "dogfish" "summit"  "odell"

# returns elements in set_2 not in set_1
setdiff(set_2, set_1)
## [1] "sierra"   "harpoon"  "founders"
```

Testing for Element Equality

- To test if two vectors contain the same elements regardless of order use setequal():

```
set_3 <- c("woody", "buzz", "rex")
set_4 <- c("woody", "andy", "buzz")
set_5 <- c("andy", "buzz", "woody")

setequal(set_3, set_4)
## [1] FALSE

setequal(set_4, set_5)
## [1] TRUE
```

Testing for *Exact* Equality

- To test if two character vectors are equal in content and order use identical():

```
set_6 <- c("woody", "andy", "buzz")
set_7 <- c("andy", "buzz", "woody")
set_8 <- c("woody", "andy", "buzz")

identical(set_6, set_7)
## [1] FALSE

identical(set_6, set_8)
## [1] TRUE
```

Sorting a String

- To sort a character vector use sort():

```
sort(set_8)
## [1] "andy"  "buzz"  "woody"

sort(set_8, decreasing = TRUE)
## [1] "woody" "buzz"  "andy"
```

Dealing with Regular Expressions

- A regular expression (aka regex) is…
  - *a sequence of characters that define a search pattern,*
  - *mainly for use in pattern matching with text strings.*

Dealing with Regular Expressions

- A regular expression (aka regex) is…
  - *a sequence of characters that define a search pattern,*
  - *mainly for use in pattern matching with text strings.*
- Typically, regex patterns consist of a combination of alphanumeric characters as well as special characters

Dealing with Regular Expressions

- To understand how to work with regular expressions in R,
  - *we need to consider two primary features*
    - The syntax, or the way regex patterns are expressed in R.
    - And the functions used for regex matching in R

Syntax used to Find Patterns

- We’ll cover the following:
  - *meta characters,*
  - *character and POSIX classes,*
  - *and quantifiers*

Regex Syntax: Meta Characters

- Metacharacters consist of non-alphanumeric symbols such as:

. \\\ | ( ) [{ $ * + ?

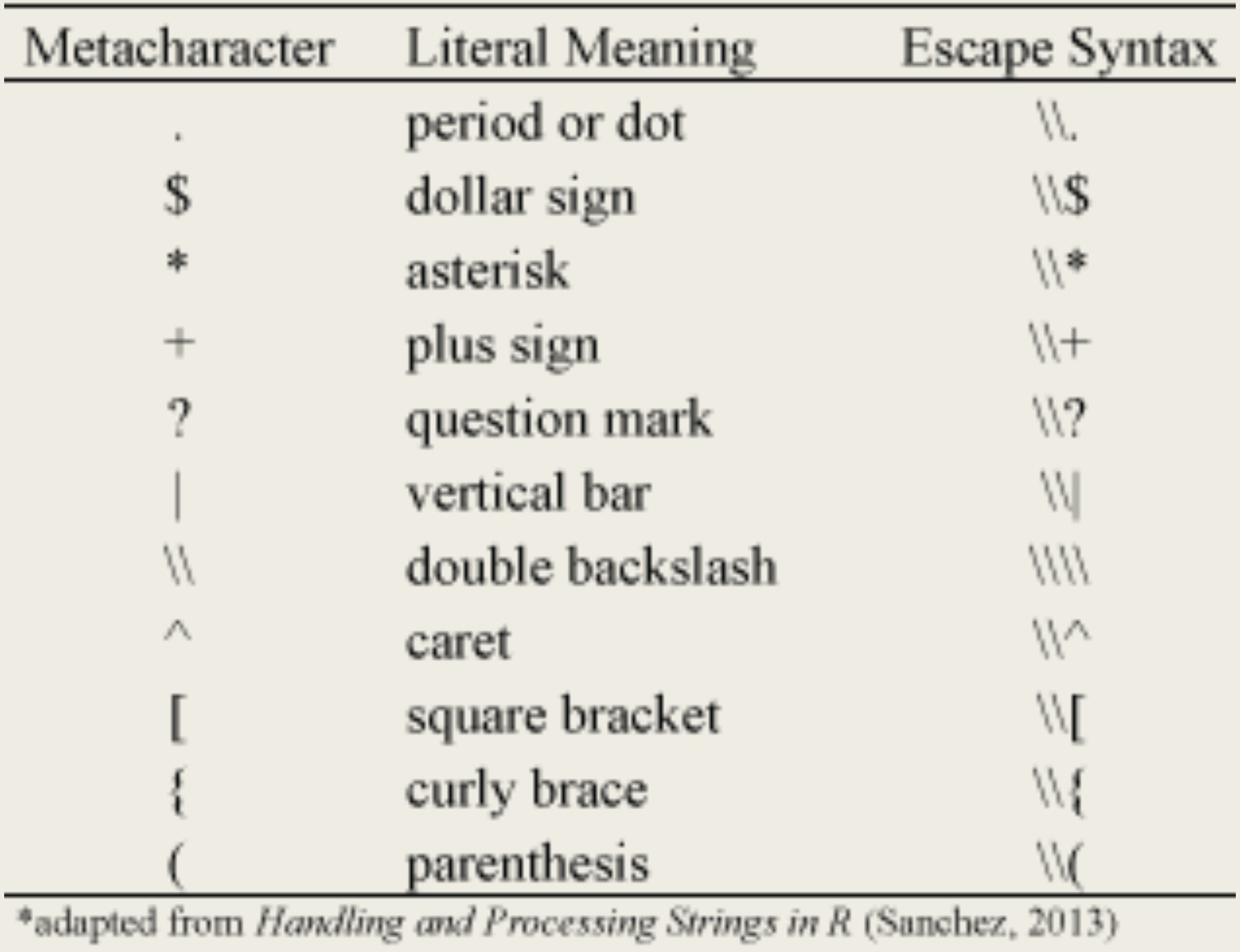
Problem: How do we match these meta characters in R?

Regex Syntax: Meta Characters

- You need to escape them with a double backslash “\\”.

- The following displays the general escape syntax for the most common metacharacters…

Regex Syntax: Meta Characters



<table>
<tr><th>Metacharacter</th><th>Literal Meaning</th><th>Escape Syntax</th></tr>
<tr><td>.</td><td>period or dot</td><td>\\.</td></tr>
<tr><td>$</td><td>dollar sign</td><td>\\$</td></tr>
<tr><td>*</td><td>asterisk</td><td>*</td></tr>
<tr><td>+</td><td>plus sign</td><td>\\+</td></tr>
<tr><td>?</td><td>question mark</td><td>\\?</td></tr>
<tr><td>|</td><td>vertical bar</td><td>\\|</td></tr>
<tr><td>\\</td><td>double backslash</td><td>\\\\</td></tr>
<tr><td>^</td><td>caret</td><td>\\^</td></tr>
<tr><td>[</td><td>square bracket</td><td>\\[</td></tr>
<tr><td>{</td><td>curly brace</td><td>\\{</td></tr>
<tr><td>(</td><td>parenthesis</td><td>\\(</td></tr>
</table>

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Escaping Meta Characters

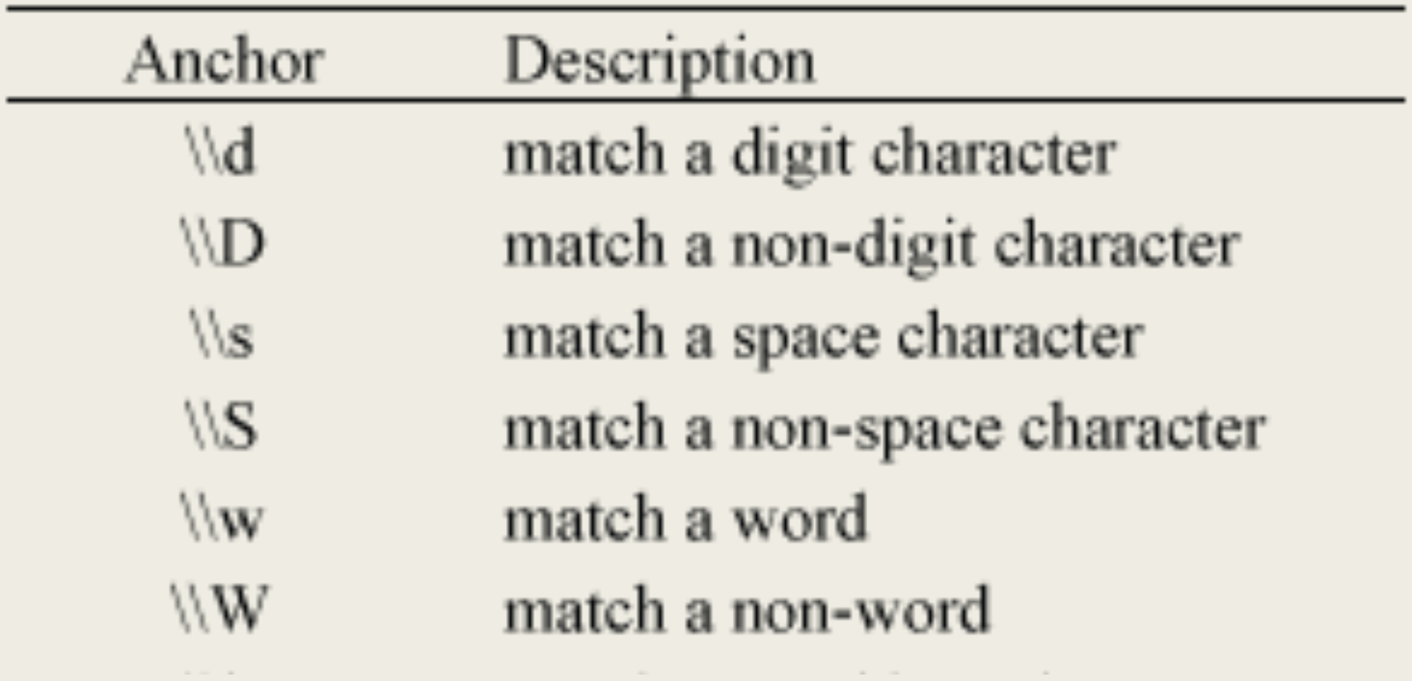
```
# substitute $ with !
sub(pattern = "\\$", "\\!", "I love R$")
## [1] "I love R!"

# substitute ^ with carrot
sub(pattern = "\\^", "carrot", "My daughter has a ^ with almost every meal!")
## [1] "My daughter has a carrot with almost every meal!"

# substitute \\ with whitespace
gsub(pattern = "\\\\", " ", "I\\need\\space")
## [1] "I need space"
```

Sequences

- To match a sequence of characters

  - *we can apply short-hand notation which captures the fundamental types of sequences…*

To match a sequence of characters



Anchor	Description
\\d	match a digit character
\\D	match a non-digit character
\\s	match a space character
\\S	match a non-space character
\\w	match a word
\\W	match a non-word

To match a sequence of characters

```
# substitute any digit with an underscore

gsub(pattern = "\\d", "_", "I'm working in RStudio v.0.99.484")

## [1] "I'm working in RStudio v._.__.___"


# substitute any non-digit with an underscore

gsub(pattern = "\\D", "_", "I'm working in RStudio v.0.99.484")

## [1] "_________________________0_99_484"
```

To match a sequence of characters

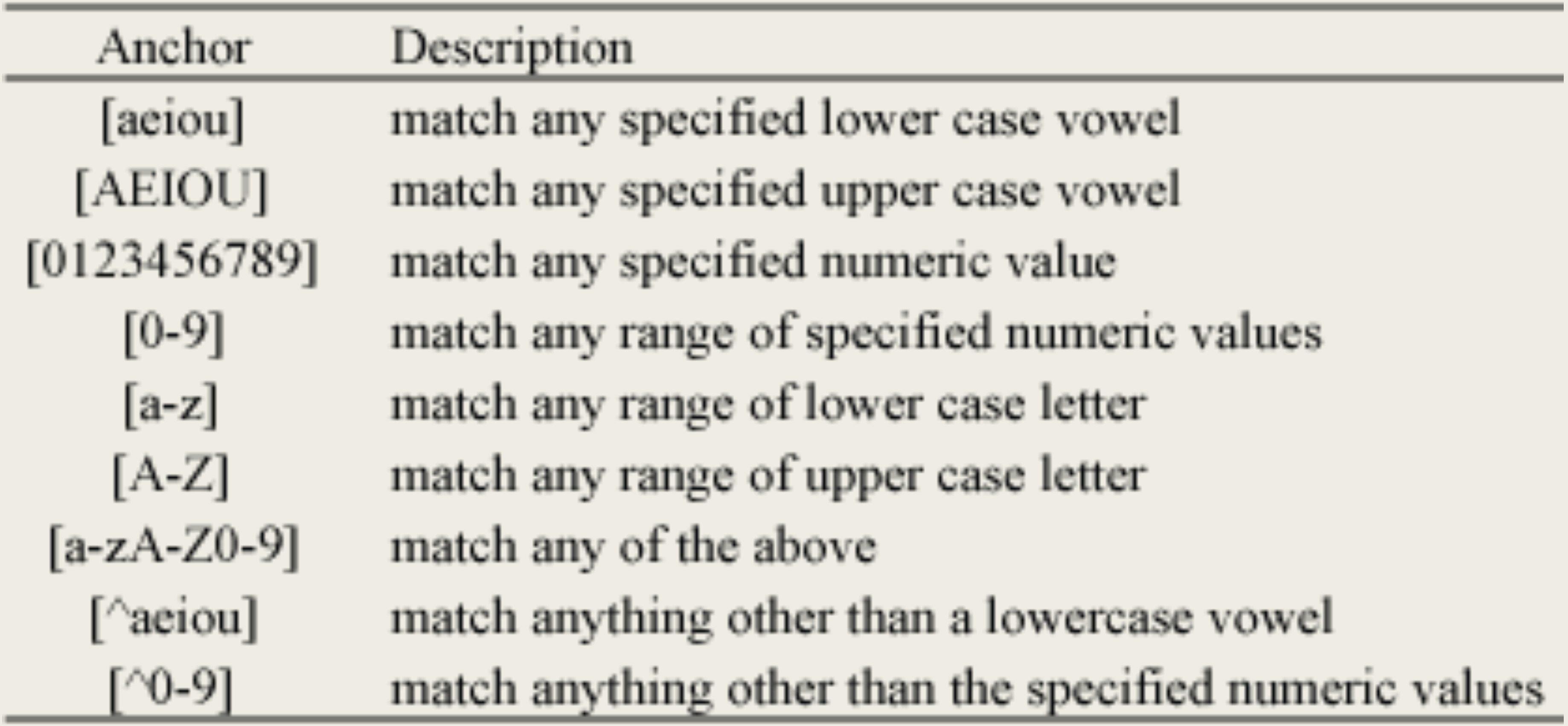
```
# substitute any whitespace with underscore

gsub(pattern = "\\s", "_", "I'm working in RStudio v.0.99.484")

## [1] "I'm_working_in_RStudio_v.0.99.484"


# substitute any wording with underscore

gsub(pattern = "\\w", "_", "I'm working in RStudio v.0.99.484")

## [1] "_'_ _______ __ _______ _._.__.___"
```

Character classes

- To match **one of several characters in a specified set** we can enclose the characters of concern **with square brackets [ ].**

- In addition, to **match any characters not in a specified character set** we **can include the caret ^** at the beginning of the set within the brackets

Character classes



Anchor	Description
[aeiou]	match any specified lower case vowel
[AEIOU]	match any specified upper case vowel
[0123456789]	match any specified numeric value
[0-9]	match any range of specified numeric values
[a-z]	match any range of lower case letter
[A-Z]	match any range of upper case letter
[a-zA-Z0-9]	match any of the above
[^aeiou]	match anything other than a lowercase vowel
[^0-9]	match anything other than the specified numeric values

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Character classes

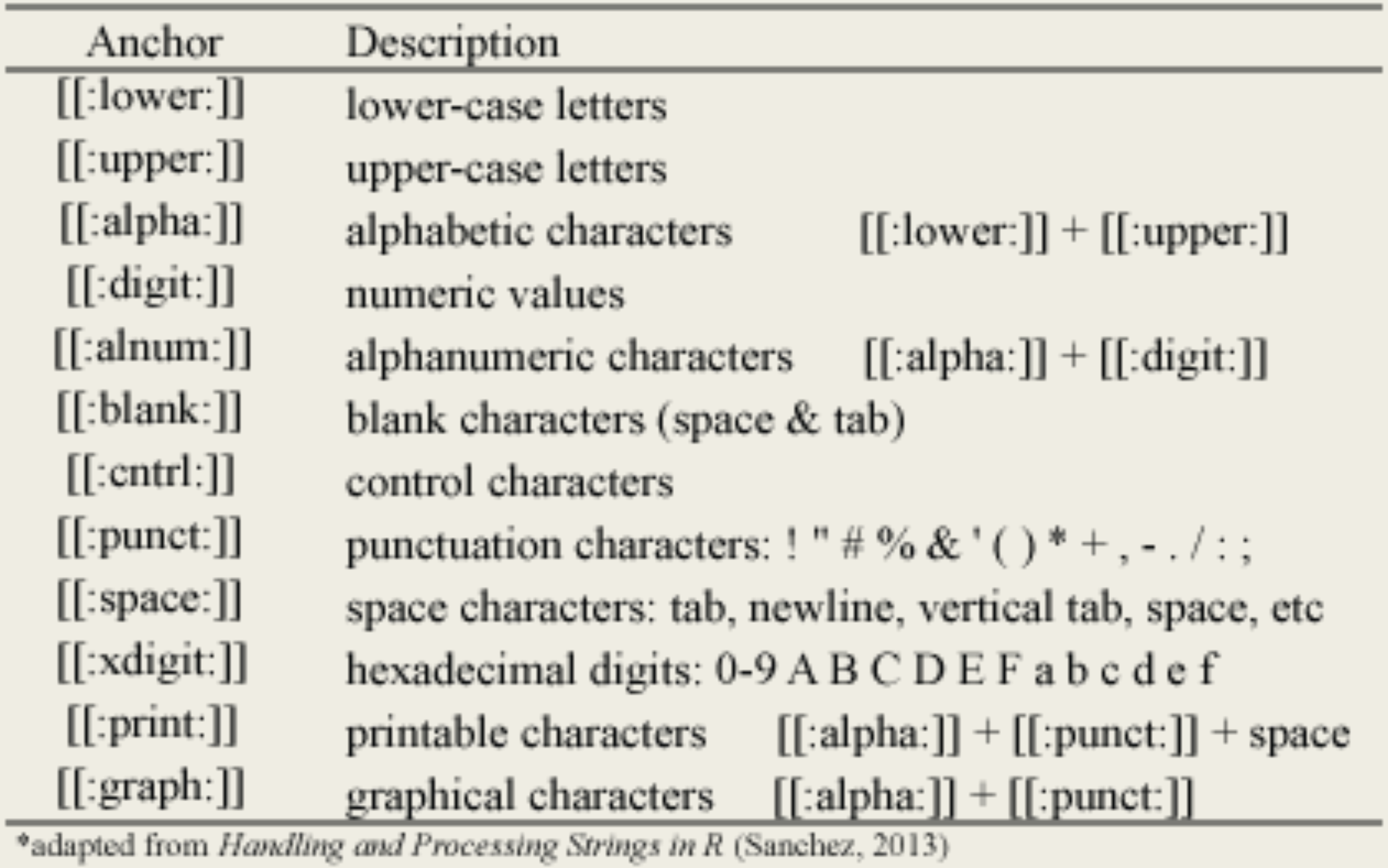
```
x <- c("RStudio", "v.0.99.484", "2015", "09-22-2015", "grep vs. grepl")

# find any strings with numeric values between 0-9
grep(pattern = "[0-9]", x, value = TRUE)
## [1] "v.0.99.484" "2015"       "09-22-2015"

# find any strings with numeric values between 6-9
grep(pattern = "[6-9]", x, value = TRUE)
## [1] "v.0.99.484" "09-22-2015"
```

Character classes

```
x <- c("RStudio", "v.0.99.484", "2015", "09-22-2015", "grep vs. grepl")

# find any strings with the character R or r
grep(pattern = "[Rr]", x, value = TRUE)
## [1] "RStudio"       "grep vs. grepl"

# find any strings that have non-alphanumeric characters
grep(pattern = "[^0-9a-zA-Z]", x, value = TRUE)
## [1] "v.0.99.484"    "09-22-2015"    "grep vs. grepl"
```

POSIX character classes

- Closely related to regex character classes are POSIX character classes which are expressed in double brackets [[ ]].



Anchor	Description	
[[:lower:]]	lower-case letters	
[[:upper:]]	upper-case letters	
[[:alpha:]]	alphabetic characters	[[:lower:]] + [[:upper:]]
[[:digit:]]	numeric values	
[[:alnum:]]	alphanumeric characters	[[:alpha:]] + [[:digit:]]
[[:blank:]]	blank characters (space & tab)	
[[:cntrl:]]	control characters	
[[:punct:]]	punctuation characters: ! " # % & ' ( ) * + , - . / : ;	
[[:space:]]	space characters: tab, newline, vertical tab, space, etc	
[[:xdigit:]]	hexadecimal digits: 0-9 A B C D E F a b c d e f	
[[:print:]]	printable characters	[[:alpha:]] + [[:punct:]] + space
[[:graph:]]	graphical characters	[[:alpha:]] + [[:punct:]]

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

POSIX character classes

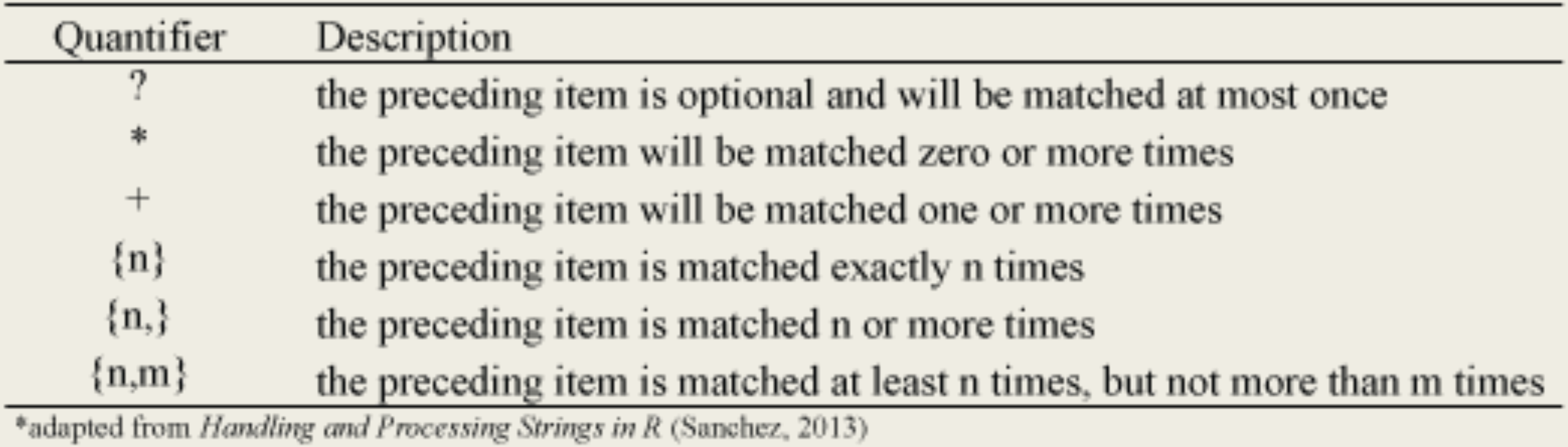
```
x <- "I like beer! #beer, @wheres_my_beer, I like R (v3.2.2) #rrrrrrr2015"

# remove space or tabs
gsub(pattern = "[[:blank:]]", replacement = "", x)
## [1] "Ilikebeer!#beer,@wheres_my_beer,IlikeR(v3.2.2)#rrrrrrr2015"

# replace punctuation with whitespace
gsub(pattern = "[[:punct:]]", replacement = " ", x)
## [1] "I like beer   beer   wheres my beer  I like R  v3 2 2   rrrrrrr2015"
```

POSIX character classes

```
x <- "I like beer! #beer, @wheres_my_beer, I like R (v3.2.2) #rrrrrrr2015"

# remove alphanumeric characters
gsub(pattern = "[[:alnum:]]", replacement = "", x)
## [1] "  ! #, @__,    (..) #"

#Use caret ^ for negation outside first bracket
gsub(pattern = "[^[:alnum:]]", replacement = "", x)
##[1] "IlikebeerbeerwheresmybeerIlikeRv322rrrrrrr2015"
```

Quantifiers

- When we want to match a certain number of characters
  - *apply quantifiers to our pattern searches.*



Quantifier	Description
?	the preceding item is optional and will be matched at most once
*	the preceding item will be matched zero or more times
+	the preceding item will be matched one or more times
{n}	the preceding item is matched exactly n times
{n,}	the preceding item is matched n or more times
{n,m}	the preceding item is matched at least n times, but not more than m times

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Quantifiers

```
# match states that contain z
grep(pattern = "z+", state.name, value = TRUE)
## [1] "Arizona"

# match states with two s
grep(pattern = "s{2}", state.name, value = TRUE)
## [1] "Massachusetts" "Mississippi"   "Missouri"      "Tennessee"
```

Quantifiers

```
# match states with one or two s
grep(pattern = "s{1,2}", state.name, value = TRUE)
##  [1] "Alaska"        "Arkansas"      "Illinois"      "Kansas"
##  [5] "Louisiana"     "Massachusetts" "Minnesota"     "Mississippi"
##  [9] "Missouri"      "Nebraska"      "New Hampshire" "New Jersey"
## [13] "Pennsylvania"  "Rhode Island"  "Tennessee"     "Texas"
## [17] "Washington"    "West Virginia" "Wisconsin"
```

Regex Functions in Base R

R contains a set of functions in the base package that we can use to find pattern matches.

Base R functions provide:

- pattern finding,
- pattern replacement, and
- string splitting capabilities.

Pattern Finding Functions

To find a pattern in a character vector and to have the element values or indices as the output use **grep():**

```
# use the built in data set `state.division`
head(as.character(state.division))
## [1] "East South Central" "Pacific" "Mountain"
## [4] "West South Central" "Pacific" "Mountain"


# find the elements which match the patter
grep("North", state.division)
##  [1] 13 14 15 16 22 23 25 27 34 35 41 49
```

Pattern Finding Functions

```
# use 'value = TRUE' to show the element value

grep("North", state.division, value = TRUE)
##  [1] "East North Central" "East North Central" "West North Central"
##  [4] "West North Central" "East North Central" "West North Central"
##  [7] "West North Central" "West North Central" "West North Central"
## [10] "East North Central" "West North Central" "East North Central"
```

```
# can use the 'invert' argument to show the non-matching elements

grep("North | South", state.division, invert = TRUE)
##  [1]  2  3  5  6  7  8  9 10 11 12 19 20 21 26 28 29 30 31 32 33 37 38 39
## [24] 40 44 45 46 47 48 50
```

Pattern Finding Functions

```
# Wrap text in ^ and $ to find exact match

grep("^North$", state.division, value = TRUE)
##  character(0)  # no matched result

#Find last value using $, ends with…
grep("Central$", state.division, value = TRUE)
##  character(0)  # no matched results
```

Pattern Finding Functions

grepl( )

To find a pattern in a character vector and to have logical (TRUE/FALSE) outputs use grepl():

Pattern Finding Functions

```
grepl("North | South", state.division)

##  [1]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
## [23]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [34]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
## [45] FALSE FALSE FALSE FALSE  TRUE FALSE

# wrap in sum() to get the count of matches
sum(grepl("North | South", state.division))
## [1] 20
```

Pattern Replacement Functions

- To replace the first matching occurrence of a pattern use sub():

```
new <- c("New York", "new new York", "New New New York")


# Default is case sensitive
sub("New", replacement = "Old", new)
## [1] "Old York"       "new new York"     "Old New New York"
```

Pattern Replacement Functions

- To replace all matching occurrences of a pattern use gsub():

```
# Default is case sensitive
gsub("New", replacement = "Old", new)
## [1] "Old York"        "new new York"     "Old Old Old York"

# use 'ignore.case = TRUE' to ignore case sensitivity
gsub("New", replacement = "Old", new, ignore.case = TRUE)
## [1] "Old York"        "Old Old York"     "Old Old Old York"
```

Regex Functions with stringr

- We’ll focus on detecting,
- locating,
- extracting, and
- replacing patterns

Detecting Patterns

- To detect whether a pattern is present (or absent) in a string vector use the str_detect().

- This function is a wrapper for grepl().

Detecting Patterns

```
# use the built in data set 'state.name'
head(state.name)
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"
"California"
## [6] "Colorado"

str_detect(state.name, pattern = "New")
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE

# count the total matches by wrapping with sum
sum(str_detect(state.name, pattern = "New"))
## [1] 4
```

Locating Patterns

- Two options: i) locate the first matching occurrence or ii) locate all occurrences.

- To locate the position of the first occurrence of a pattern in a string vector use str_locate()

Locating Patterns

```
x <- c("abcd", "a22bc1d", "ab3453cd46", "a1bc44d")


# locate 1st sequence of 1 or more consecutive numbers
str_locate(x, "[0-9]+") ##      start end
## [1,]    NA  NA
## [2,]     2   3
## [3,]     3   6
## [4,]     2   2
```

Locating Patterns

- To locate the positions of all pattern match occurrences in a character vector use str_locate_all()

Locating Patterns

```
# locate all sequences of 1 or more consecutive numbers
x <- c("abcd", "a22bc1d", "ab3453cd46", "a1bc44d")
str_locate_all(x, "[0-9]+")

## [[1]]
##      start end
##
## [[2]]
##      start end
## [1,]     2   3
## [2,]     6   6…
```

Locating Patterns

```
# locate all sequences of 1 or more consecutive numbers
x <- c("abcd", "a22bc1d", "ab3453cd46", "a1bc44d")
    str_locate_all(x, "[0-9]+")
    ##
    ## [[3]]
    ##      start end
    ## [1,]     3   6
    ## [2,]     9  10
    ##
    ## [[4]]
    ##      start end
    ## [1,]     2   2
    ## [2,]     5   6
```

Extracting Patterns

- Two primary options:

  - *i) extract the first matching occurrence or*

  - *ii) extract all occurrences*

Extracting Patterns

- To extract the first occurrence of a pattern in a character vector use str_extract()

- The output will be the same length as the string and if no match is found the output will be NA for that element.

Extracting Patterns

```
y <- c("I use R #useR2014", "I use R and love R #useR2015", "Beer")


str_extract(y, pattern = "R")
## [1] "R" "R" NA
```

Extracting Patterns

- To extract all occurrences of a pattern in a character vector use str_extract_all()

```
y <- c("I use R #useR2014", "I use R and love R #useR2015", "Beer")

str_extract_all(y, pattern = "[[:punct:]]*[a-zA-Z0-9]*R[a-zA-Z0-9]*")

## [[1]]

## [1] "R"         "#useR2014"

##

## [[2]]

## [1] "R"         "R"         "#useR2015"

##

## [[3]]

## character(0)
```

Replacing Patterns

- two options:

  - *i) replace the first matching occurrence or*
  - *ii) replace all occurrences*

Replacing Patterns

- two options:
  - *i) replace the first matching occurrence or*
  - *ii) replace all occurrences*

Replacing Patterns

- To replace the first occurrence of a pattern in a character vector use str_replace().

- This function is a wrapper for sub().

Replacing Patterns

```
cities <- c("New York", "new new York", "New New New York")

cities

## [1] "New York"         "new new York"     "New New New York"


# case sensitive

str_replace(cities, pattern = "New", replacement = "Old")

## [1] "Old York"         "new new York"     "Old New New York"
```

Replacing Patterns

- To extract all occurrences of a pattern in a character vector use str_replace_all().

- This function is a wrapper for gsub().

Replacing Patterns

```
cities <- c("New York", "new new York", "New
New New York")
```

```
str_replace_all(cities, pattern = "New",
replacement = "Old")
```

```
#[1] "Old York"        "new new York"     "Old Old
Old York"
```