Lecture 25: Nov 9, 2018

Regular Expressions

- *Regular Expressions*
- *Using Regex*
  - *Literal Characters*, *Metacharacters*, *Character Classes*, *Quantifiers*, *Groups*, *Backreferences*, *Anchors*
- *Resources*

James Balamuta
STAT 385 @ UIUC

Announcements

- **Group Proposal** due **Friday, November 16th** at **11:59 PM**
- **hw08** due **Friday, November 16th** at **6:00 PM**
- **Quiz 12** covers Week 11 contents @ CBTF.
  - Window: Nov 13th - Nov 15th
  - Sign up: https://cbtf.engr.illinois.edu/sched
- Want to review your homework or quiz grades?
  Schedule an appointment.

Last Time

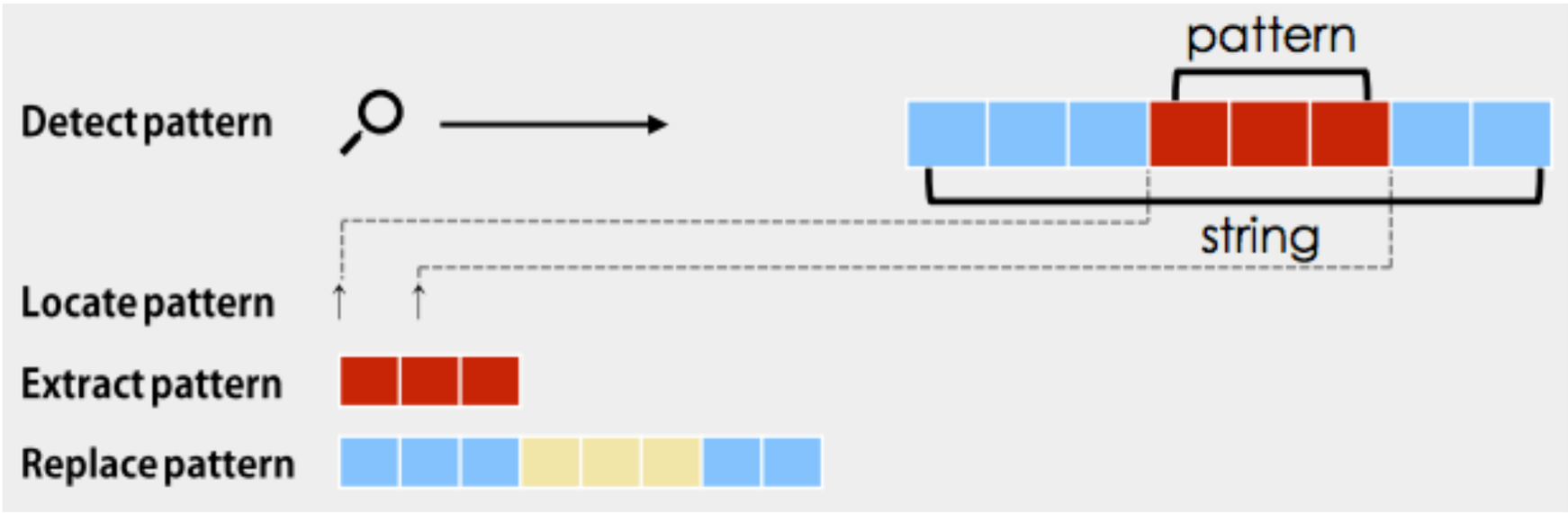
- **Ubiquitousness of Functions**
  - *R* is a functional language
  - Functions are objects
- **Environments**
  - Indicate where information is stored.
- **Functionals**
  - Functionals use a function input with a vector.
  - Functionals can be used in place of loops when there is no dependency between iteration
- **An Odyssey with purrr**
  - Type stable function output is preferred.

Lecture Objectives

- **Explain** how regular expressions can be used to identify patterns in text.
- **Apply** regular expressions to extract and replace values in text.

Regular Expressions

Definition:
Regular Expressions ("regex") are sequences of characters that defines a search pattern to find or replace values contained in a collection of character (string) values.



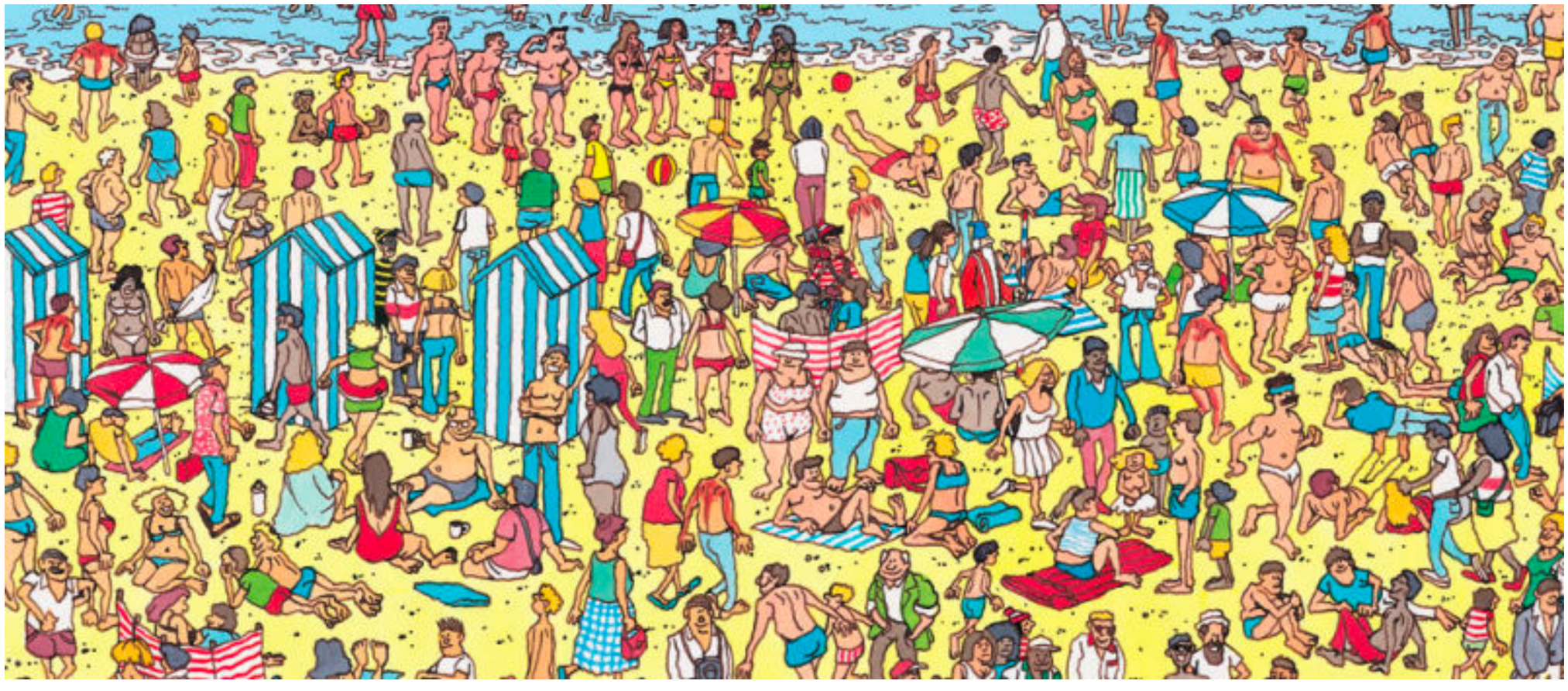
Source: Ian Kopacka

Meet Waldo

… Search Pattern …

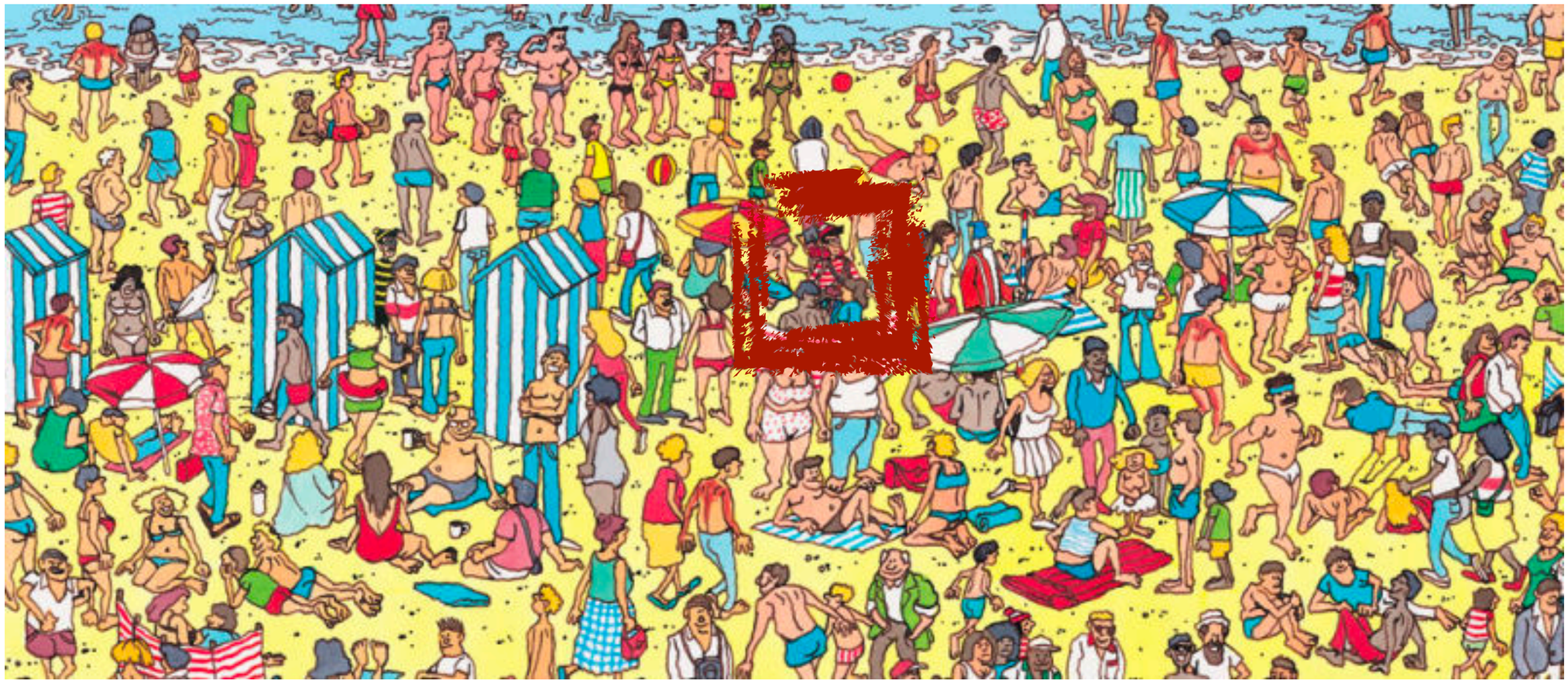
Search Area

… Place to Look for Waldo …



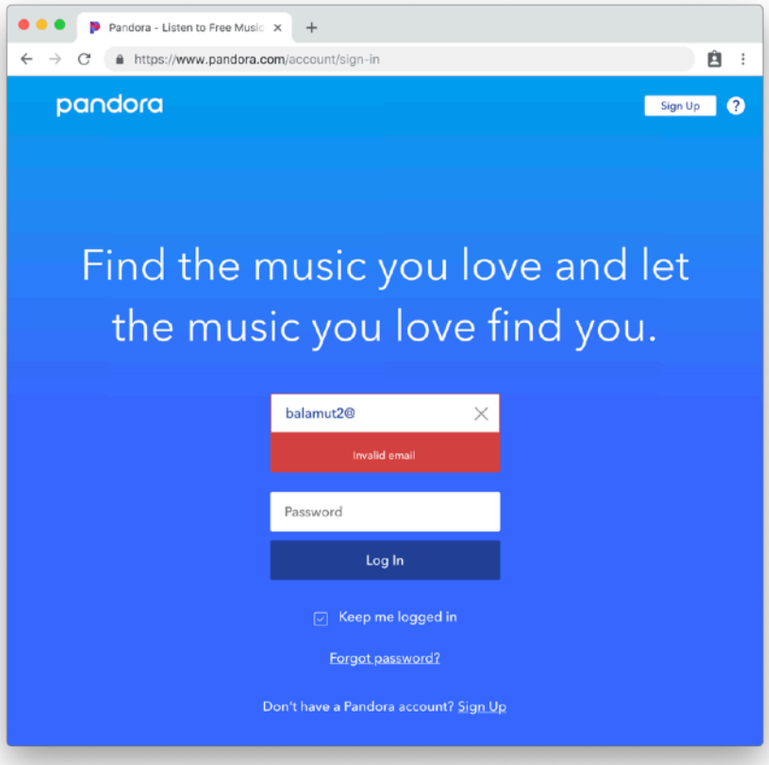
Pattern Found

... We've found Waldo !!!



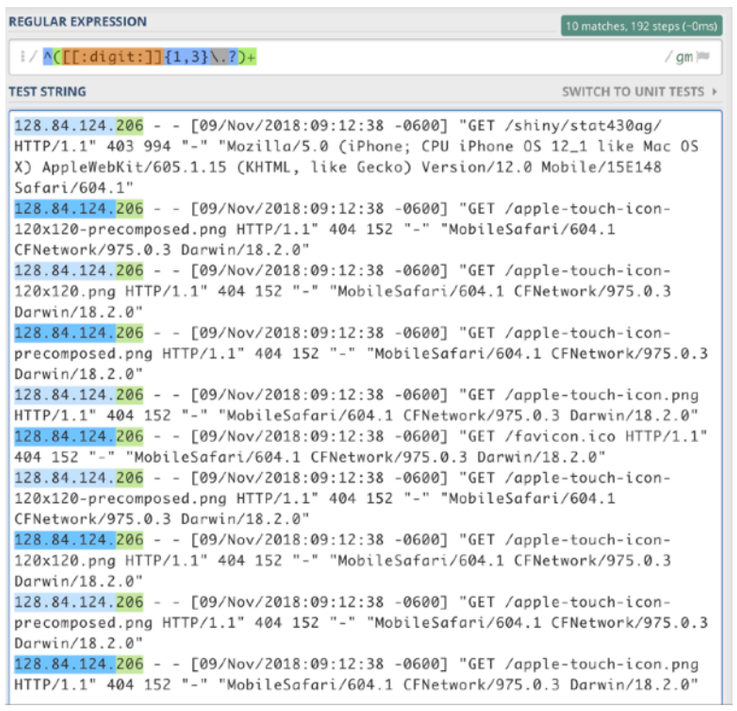
Why Regex?

Validate Data



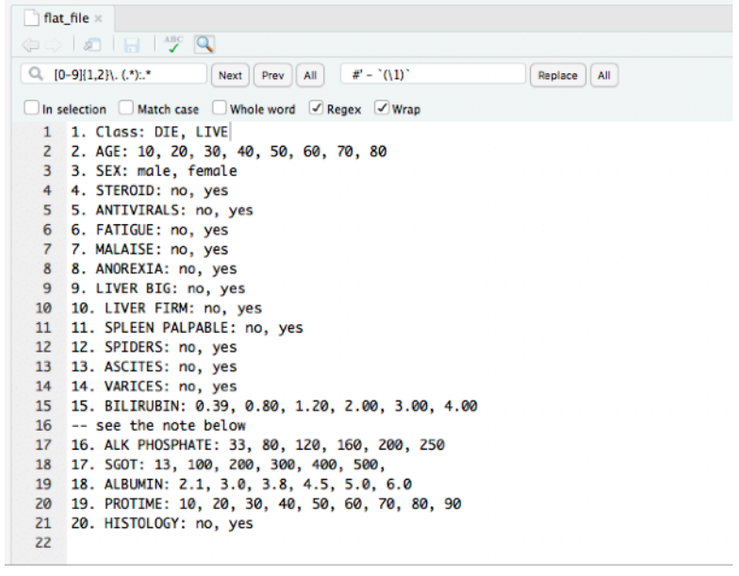
... email, credit card, phone number ...

Search and Extract



... key words or phrases in logs ...

Find and Replace



... mass name changes ...

Using Regex

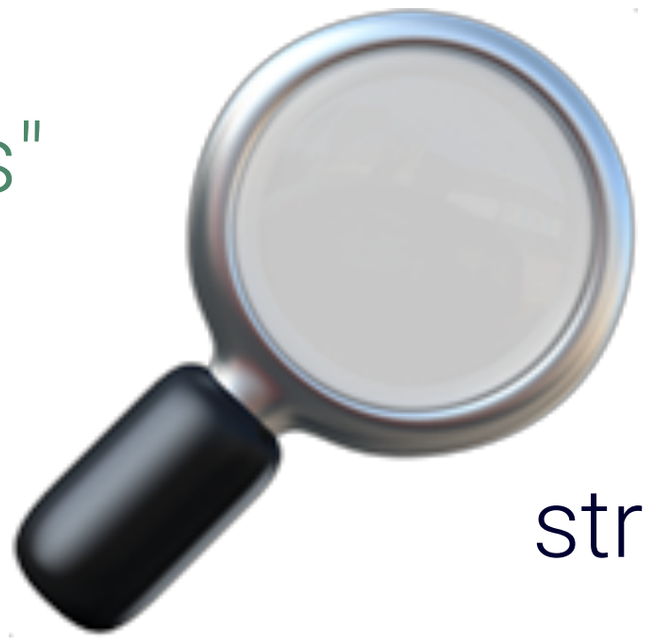
… a much nicer way to work with regex …

```
install.packages("stringr")
library("stringr")
```

Function	Description
str_detect(x, pattern)	Any pattern matches?
str_count(x, pattern)	How many matches?
str_subset(x, pattern)	What are the matching patterns?
str_locate(x, pattern)	Where are the matching patterns?
str_extract(x, pattern)	What is the matching value?
str_match(x, pattern)	What is the matching group?
str_replace(x, pattern, replacement)	What should replace the pattern?
str_split(x, pattern)	How should the string be split?

Definition:
Literal Characters are string values that should be matched *exactly*.

pattern = "s"



string = "cat**s** and dog**s**"

Detecting Patterns

… determining if a pattern is in a string …

String Data — Strings to check against pattern

Pattern — Value to search for in the string

```
str_detect(x = <data>, pattern = <match-this>)
```

Searching for **Patterns**

```
# Get the required library
install.packages("stringr")
library("stringr")

# Sample String Data
x = c("did you lie to me?",
      "all lies!",
      "are you lying?",
      "lying on the couch")

str_detect(x, pattern = "lie")
# [1]  TRUE  TRUE FALSE FALSE

str_detect(x, pattern = "you")
# [1]  TRUE FALSE  TRUE FALSE
```

How the Matches Happened

pattern = "lie"

"did you **lie** to me?"
"all **lie**s"
"are you lying?"
"lying on the couch"

pattern = "you"

"did **you** lie to me?"
"all lies"
"are **you** lying?"
"lying on the couch"

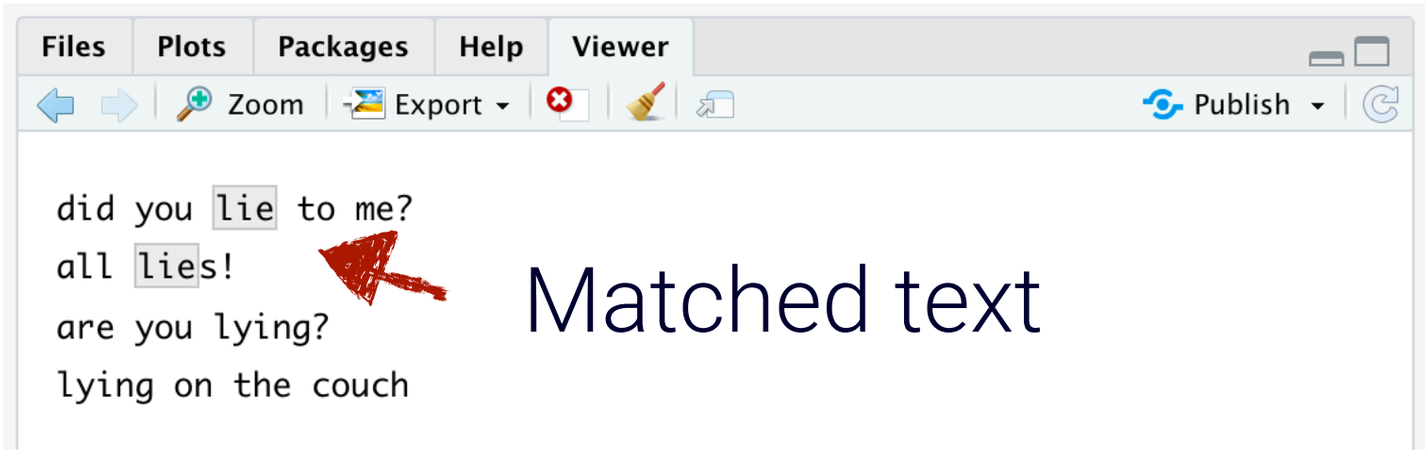
Viewing Matches

… seeing matches if a pattern is in a string …

String Data
Strings to check against pattern

Pattern
Value to search for in the string

str_**view_all**(x = <data>, pattern = <match-this>)



Searching for **Multiple** Patterns

```
# Sample String Data
x = c("did you lie to me?",
      "all lies!",
      "are you lying?",
      "lying on the couch")

str_detect(x, pattern = "lie|you")
# [1]  TRUE  TRUE
# [3]  TRUE  FALSE

str_detect(x, pattern = "(lie)|(you)")
# [1]  TRUE TRUE
# [3]  TRUE FALSE
```

How the Matches Happened

pattern = "**lie**|*you*"

"did *you* **lie** to me?"
"all **lie**s"
"are *you* lying?"
"lying on the couch"

Your Turn

Find all cities that reside in the state of Illinois
e.g. have the acronym of **IL**

```
x = c("Chicago, IL", "San Fran, CA", "Iowa City, IA", "Urbana, IL",
      "Wheaton, IL", "Myrtle Beach, SC")
```

Find all instances of either **UIUC** *or* **UofI**

```
Y = c("UNR", "UNC", "UofI", "UIUC", "UI")
```

Definition:
Metacharacters are values that have a special meaning inside of a pattern that trigger a *dynamic* matching scheme.

Character classes: **[]**
Any character: **.**
End of String: **$**
Beginning of String / Negation: **^**
Quantifiers: **{}, *, +, ?**
Escape sequences: ****

Escaping a Metacharacter

… retaining the original character's meaning …

String	Regex Pattern	Escaped Pattern
.	\.	"\\."
?	\?	"\\?"
(	\(	"\\("

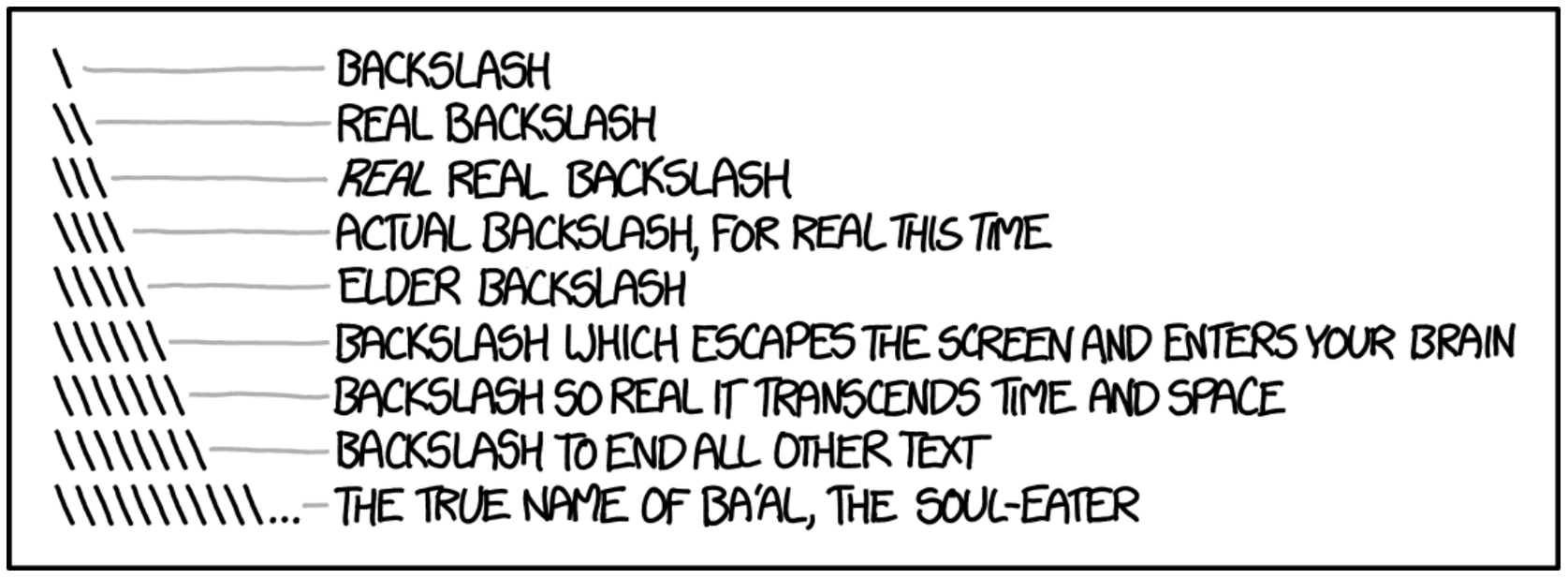
Searching for **Special** Patterns

```
# Sample String Data
x = c("did you lie to me?",
      "all lies!",
      "are you lying?",
      "lying on my couch")

str_detect(x, pattern = "\?")
# Error: Error: '\?' is an
unrecognized escape in character
string starting ""\?"

str_detect(x, pattern = "\\?")
# [1]  TRUE FALSE TRUE FALSE
```

Your Turn

Find all strings with either an "+" or "/"

```
x = c("3 + 4 = 7", "1 / 4 = 0.25", "2 * 4 = 8", "3 * 4",
"Algebra is fun?", "Green Eggs and\\or Ham")
```



XKCD (1638): Backslashes

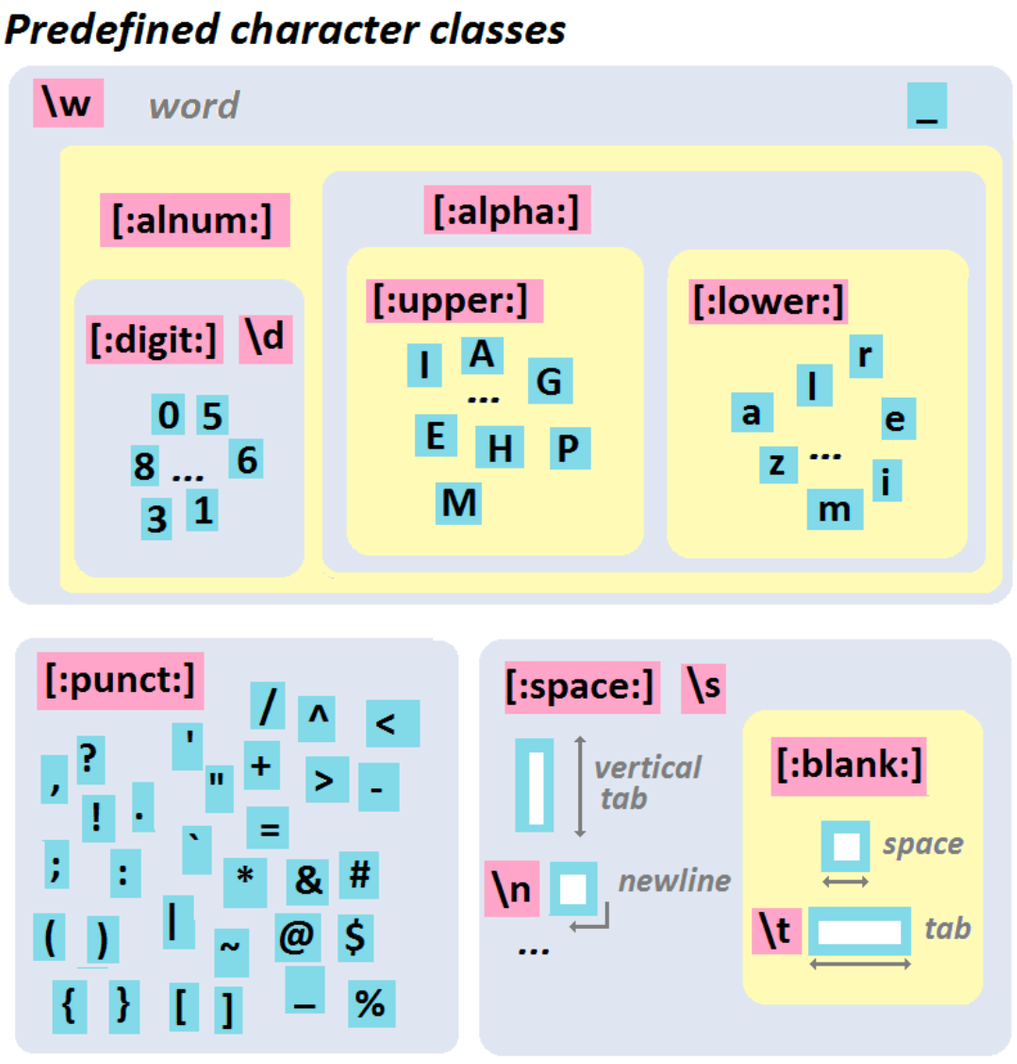
Definition:
Character classes provide ways to match or unmatch a single symbol within the string.

regexp	matches	example
ab|d	or	abcde
[abe]	one of	abcde
[^abe]	anything but	abcde
[a-c]	range	abcde

Character Class Examples

Regex	Description	Example
[aeiou]	Search for a vowel	S**u**p**e**rc**a**l**i**fr**a**g**i**l**i**st**i**c**e**xp**ia**l**i**d**o**c**iou**s
[13579]	Search for odd numbers	**1337**, ph**3**4, m8
[a-z]	Search for *lower* case	M**y name is** J**ames**
[A-Z]	Search for *UPPER* case	**M**y name is **J**ames
[Hh]	Search for variants of *h*	**H**ello World! **H**ow are you? I'm **h**ungry.
[^Hh]	Search for any value but *h* or *H*	H**ello World!** H**ow are you? I'm** h**ungry.**

Character Class Breakdown



Source

Searching for **Dynamic** Values

```r
# Sample String Data
x = c("lower case values",
      "UPPER CASE VALUES",
      "MiXtUrE oF vAlUeS")

# Lower case values for a b c
str_detect(x, pattern = "[abc]")
# [1]  TRUE FALSE FALSE

# Upper case values for A B C
str_detect(x, pattern = "[ABC]")
# [1]  FALSE TRUE TRUE

# Range of lower case values
str_detect(x, pattern = "[a-z]")
# [1]  TRUE FALSE TRUE
```

Matching with Characters

pattern = "gr[ae]y"

"Does the wolf have **gray** or **grey** hair?"

pattern = "a[^n]"

"Do we have **a** to**ad**?"

"He's an **au**thor."

Your Turn

Write a regex that matches a telephone number given as:

###-###-####

Hint: Use the range feature of character classes

```
phone_nums = c("(217) 333-2167", "217-333-2167", "217 244-7190")
```

Replacing Patterns

… replace pattern in a string with a new value …

String Data

Strings to check against pattern

```
str_replace(x = <data>,
            pattern = <match-this>,
            replacement = <replace-with>)
```

---

* str_replace() will only replace the FIRST instance the pattern matches.

** Adding an **_all** to the end of str_replace() will cause ALL matches with the pattern to be replaced.

Replacing **Dynamic** Values

```r
# Sample String Data
x = c("lower case values",
      "UPPER CASE VALUES",
      "MiXtUrE oF vAlUeS")

# Lower case values for a b c
str_replace(x, pattern = "[abc]", replacement = "!")

# Lower case values for a b c
str_replace_all(x, pattern = "[abc]",
                replacement = "!")
# [1]  "lower !!se v!lues" "UPPER CASE VALUES"
# [2] "MiXtUrE oF vAlUeS"

# Replace UPPER case values for A B C
str_replace_all(x, pattern = "[ABC]",
                replacement = "!")
# [1] "lower case values"
# [2] "UPPER !!SE V!LUES"
# [3] "MiXtUrE oF v!lUeS"

# Replace all lower case values
str_replace_all(x, pattern = "[a-z]",
                replacement = "!")
# [1]  "!!!!! !!!! !!!!!!" "UPPER CASE VALUES"
# [3] "M!X!U!E !F !A!U!S"
```

Comparison of Replacements

Single vs. **Global**

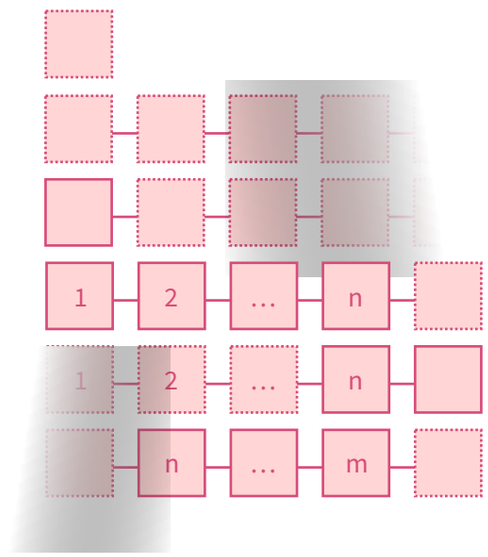
```
# Sample String Data
x = c("I dislike cake. Any cake really.",
      "Cake is okay...",
      "I love cake... Cake... Cake...",
      "I prefer to have pie over cake",
      "Mmmm... Pie.")

# Replacing first instance of cake per string
str_replace(x, pattern = "[Cc]ake",
                    replacement = "Pizza")
# [1] "I dislike Pizza. Any cake really."
# [2] "Pizza is okay..."
# [3] "I love Pizza... Cake... Cake..."
# [4]  "I prefer to have pie over Pizza"
# [5] "Mmmm... Pie."

# Replacing ALL instances of cake
str_replace_all(x, pattern = "[Cc]ake",
                          replacement = "Pizza")
# [1] "I dislike Pizza. Any Pizza really."
# [2] "Pizza is okay..."
# [3] "I love Pizza... Pizza... Pizza..."
# [4] "I prefer to have pie over Pizza"
# [5] "Mmmm... Pie."
```

Your Turn

1. Find all matches of the word "i" / "I".
2. Remove the word "not".
3. Change the word "Green" to be "Blue".

```
green_eggs = c("I do not like them",
               "Sam-I-am.", "I do not like",
               "Green eggs and ham.")
```

Definition:
Quantifiers specify how many times a character should appear or not.



regexp	matches
a?	zero or one
a*	zero or more
a+	one or more
a{n}	exactly **n**
a{n, }	**n** or more
a{n, m}	between **n** and **m**

Extracting Patterns

… retrieving a pattern found in a string …

String Data — Strings to check against pattern

Pattern — Value to search for in the string

```
str_extract(x = <data>, pattern = <match-this>)
```

---

* str_extract() will only extract the FIRST instance the pattern matches.

** Adding an **_all** to the end of str_extract() will cause ALL matches with the pattern to be retrieved.

Finding **Connected** Similar Values

```r
# Sample String Data
x = c("Hi",
      "Hey",
      "Heyy",
      "Heyyy",
      "Heyyyy")

# Find at least 1 to 3 y's together
str_extract(x, pattern = "y{1,3}")
# [1]  NA   "y"   "yy"  "yyy" "yyy"

# Find one or more "yy" groups
str_extract(x, pattern = "(yy)+")
# [1]  NA    NA    "yy"   "yy"   "yyyy"

# Find zero or more
str_detect(x, pattern = "x*")
# [1] TRUE TRUE TRUE TRUE TRUE

# Find one or more
str_detect(x, pattern = "x+")
# [1] FALSE FALSE FALSE FALSE FALSE
```

Redux Phone Numbers

… character classes with phone numbers …

###-###-####

```
phone_nums = c("(217) 333-2167", "217-333-2167", "217 244-7190")
str_detect(phone_nums, "[[:digit:]]{3}-[[:digit:]]{3}-[[:digit:]]{4}")
```

[[:digit:]] → [0-9] → [0123456789]

Your Turn

What strings are detect given the search pattern?

```
x = c("October 20, 2017", "!Oct 20, 2017",
      "Oct 20, 2017", "Hello STAT 385")
```

Search Pattern
Look for

```
^[[:alpha:]]{3,9}[[:space:]][0-9]{1,2},[[:space:]][[:digit:]]{4}$
```

^ asserts position at start of the string

Match a single character present in the list below [[:alpha:]]{3,9}
{3,9} Quantifier — Matches between 3 and 9 times, as many times as possible, giving back as needed (greedy)
[:alpha:] matches a alphabetic character [a-zA-Z]

Match a single character present in the list below [[:space:]]
[:space:] matches a whitespace character, including a line break [\t\r\n\v\f]

Match a single character present in the list below [0-9]{1,2}
{1,2} Quantifier — Matches between 1 and 2 times, as many times as possible, giving back as needed (greedy)
0-9 a single character in the range between 0 and 9

, matches the character , literally (case sensitive)

Match a single character present in the list below [[:space:]]
[:space:] matches a whitespace character, including a line break [\t\r\n\v\f]

Match a single character present in the list below [[:digit:]]{4}
{4} Quantifier — Matches exactly 4 times
[:digit:] matches a digit [0-9]

$ asserts position at the end of the string, or before the line terminator right at the end of the string (if any)

---

* You should really use a date time parser instead of regular expressions...

Your Turn

Require two consecutive numbers

```
x = c("T-800 Model 101", "Sky Diving", "Coffee&Tea", "STAT 385")
```

Require an upper case followed by a lower case

```
x = c("Up", "i gotta feeling", "skyfall", "R2D2", "down2Night")
```

Greedy vs. Lazy

… to hoard or not to …

Greedy: Match a pattern as *<u>many</u>* times as possible. *(Default)*

… keep searching until pattern cannot be found…

```
str_extract("stackoverflow", pattern = "s.*o")
# [1] "stackoverflo"
```

Lazy: Match a pattern as *<u>few</u>* times as possible.

… stop searching once the pattern is found …

```
str_extract("stacko", pattern = "s.*?o")
# [1] "stacko"
```

Redux Greedy vs. Lazy

… to hoard or not to …

```
html_txt = "<span class='val'> <span> <b> Hi </b> </span> </span>"

# What pattern is this?
str_extract(html_txt, pattern = "<span>(.*)</span>")
# [1] "<span> <b> Hi </b> </span> </span>"

str_extract(html_txt, pattern = "<span>(.*?)</span>")
# [1] "<span> <b> Hi </b> </span>"
```

XKCD (208): Regular Expressions

"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems."

--- Jamie Zawinski in alt.religion.emacs

Definition:
Groups provide a way to order patterns that are found in a string

regexp	matches	example
(ab|d)e	sets precedence	abcde

Definition:
Backreferences provide the ability to reference a group within a replacement option. This allows the pattern match to be included in the replaced value.

string (type this)	**regexp** (to mean this)	**matches** (which matches this)	**example** (String)	(Pattern)	(Replace)	(Output)
\\1	**\1** (etc.)	first () group, etc.	"aabb"	"(a)(b)"	"!\\2!"	a!b!b

Extracting and **Replacing** Capture Group

Target
Extraction Goals

```
00:00:00 - 00:00:05 (5 sec)
         ␣ ␣       ␣ ␣
```

Search Pattern
Look for

```
.*-[[:space:]](.*)[[:space:]]\\(.*
```

Replacement
Replace string

```
\\1
```

Result
Result of the Extraction

```
00:00:05
```

Target
Extraction Goals

```
00:00:00 - 00:00:05 (5 sec)
         ␣ ␣       ␣ ␣
```

Search Pattern
Look for

```
.*\\(([0-9]+).*
```

Replacement
Replace string

```
\\1
```

Result
Result of the Extraction

```
5
```

```r
# Sample String Data
x = c("00:00:00 - 00:00:05 (5 sec)",
      "00:00:05 - 00:00:35 (30 sec)",
      "00:00:35 - 00:00:51 (16 sec)")

# Extract end time stamp and
# replace string with it.
str_replace(x,
  pattern = ".*-[[:space:]](.*)[[:space:]]\\(.*",
  replacement ="\\1")  # ^^ taken from here
# [1] "00:00:05" "00:00:35" "00:00:51"

# Extract time in seconds and
# replace string with it.
str_replace(x,
       pattern = ".*\\(([0-9]+).*",
  replacement ="\\1") # ^^^^ taken from here
# [1] "5" "30" "16"
```

Using a Grouped Pattern Against Itself

```
# Sample String Data
x = c("pineapple",
      "apple",
      "eggplant",
      "blackberry",
      "apricot",
      "nectarine")

# Find consecutively similar letters
str_extract(x,
    pattern = "(.)\\1"
)
# [1] "pp" "pp" "gg" "rr" NA NA

# Find repeated pattern of values
str_extract(x,
    pattern = "(..).*\\1"
)
# [1] NA  NA  NA NA NA  "nectarine"
```

Modifying with Grouped Patterns

```r
# Sample String Data
x = c("STAT 400",
      "MATH 461",
      "CS 225",
      "525")

# Change all courses to STAT
str_replace(x,
    pattern = "([[:upper:]]{2,4}) **([[:digit:]]{3})**",
    replacement = "STAT **\\2**"
)
# [1] "STAT **400**" "STAT **461**" "STAT **225**"  "525"

# Change all course numbers to 410
str_replace(x,
    pattern = "**([[:upper:]]{2,4})** ([[:digit:]]{3})",
    replacement = "**\\1** 410"
)
# [1] "**STAT** 410" "**MATH** 410" "**CS** 410"  "525"
```

Group Matches

… retrieving a **grouped** pattern found in a string …

String Data — Strings to check against pattern

Pattern — Value to search for in the string

```
str_match(x = <data>, pattern = <match-this>)
```

---

* str_match() will only extract the FIRST instance the grouped pattern match.

** Adding an **_all** to the end of str_match() will cause ALL group matches to be retrieved.

Retrieve Group Patterns

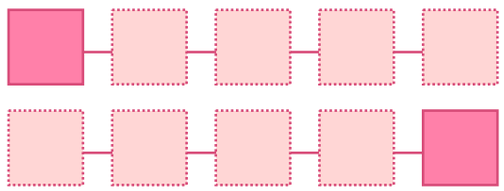
```r
# Sample String Data
x = c("STAT 400",
      "MATH 461",
      "CS 225",
      "525")

# Extract matching pattern separate the group
str_match(x,
    pattern = "([[:upper:]]{2,4}) [[:digit:]]{3}"
)
#      [,1]             [,2]
# [1,] "STAT 400"   "STAT"
# [2,] "MATH 461" "MATH"
# [3,] "CS 225"       "CS"
# [4,] NA                NA

# Extract matching patterns and groups
str_match(x,
    pattern = "([[:upper:]]{2,4}) ([[:digit:]]{3})"
)
#      [,1]             [,2]       [,3]
# [1,] "STAT 400"   "STAT"   "400"
# [2,] "MATH 461" "MATH"  "461"
# [3,] "CS 225"       "CS"      "225"
# [4,] NA                NA         NA
```

Your Turn

1. Retrieve the different portions of a phone number
2. Change the area code of the phone number to 888

```
phone_nums = c("(217) 333-2167", "217-333-2167", "217 244-7190")
```

Definition:
Anchors provide either the beginning or ending of the string.



regexp	matches
^a	start of string
a$	end of string

Bounded Values

```
# Sample String Data
x = c("1 second to 12 AM",
      "15300",
      "19,000",
      "Time to go home",
      "home on the range")

# Must start with a number
str_detect(x, pattern = "^[0-9]")
# [1]  TRUE  TRUE  TRUE FALSE FALSE

# Must end with lower case
str_detect(x, pattern = "[a-z]$")
# [1] FALSE FALSE FALSE  TRUE  TRUE

# Only alphabetic and space characters
str_detect(x, pattern = "^[a-zA-Z[:space:]]+$")
# [1] FALSE FALSE FALSE  TRUE  TRUE

# Only numbers
str_detect(x, pattern = "^[0-9]+$")
# [1] FALSE  TRUE FALSE FALSE FALSE
```

Your Turn

1. Find punctuation **at the end** of a string
2. Find a capital letter **at the start** of the string
3. Combine both 1. and 2.

```
x = c("Today is a good day", "Tomorrow is better!", "Call me!",
      "When can we talk?", "Fly Robbin fly",
       "not really.")
```

Resources

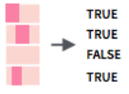
Cheatsheet for Strings

… stringr overview …

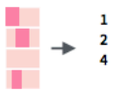
Work with strings with stringr : : **CHEAT SHEET**

The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

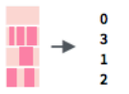
Detect Matches



str_detect(string, **pattern**) Detect the presence of a pattern match in a string. *str_detect(fruit, "a")*



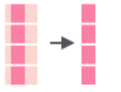
str_which(string, **pattern**) Find the indexes of strings that contain a pattern match. *str_which(fruit, "a")*



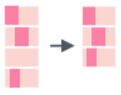
str_count(string, **pattern**) Count the number of matches in a string. *str_count(fruit, "a")*

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**. *str_locate(fruit, "a")*

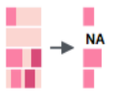
Subset Strings



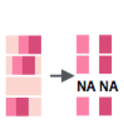
str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector. *str_sub(fruit, 1, 3); str_sub(fruit, -2)*



str_subset(string, **pattern**) Return only the strings that contain a pattern match. *str_subset(fruit, "b")*

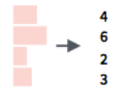


str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*

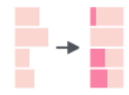


str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all**. *str_match(sentences, "(a|the) ([^]+)")*

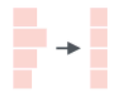
Manage Lengths



str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). *str_length(fruit)*



str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. *str_pad(fruit, 17)*



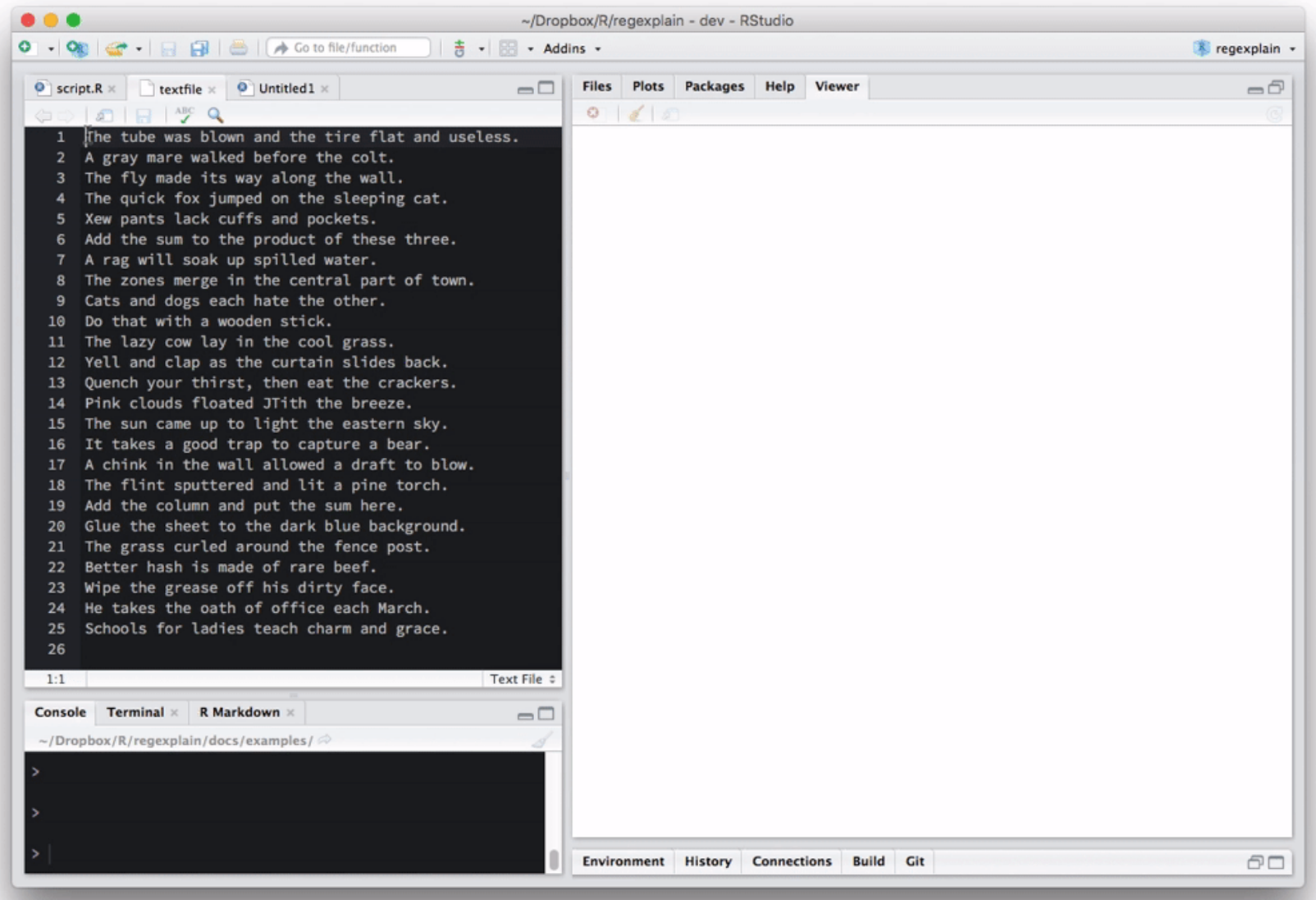
str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. *str_trunc(fruit, 3)*

str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. *str_trim(fruit)*

https://github.com/rstudio/cheatsheets/raw/master/strings.pdf

RStudio Regex Helper

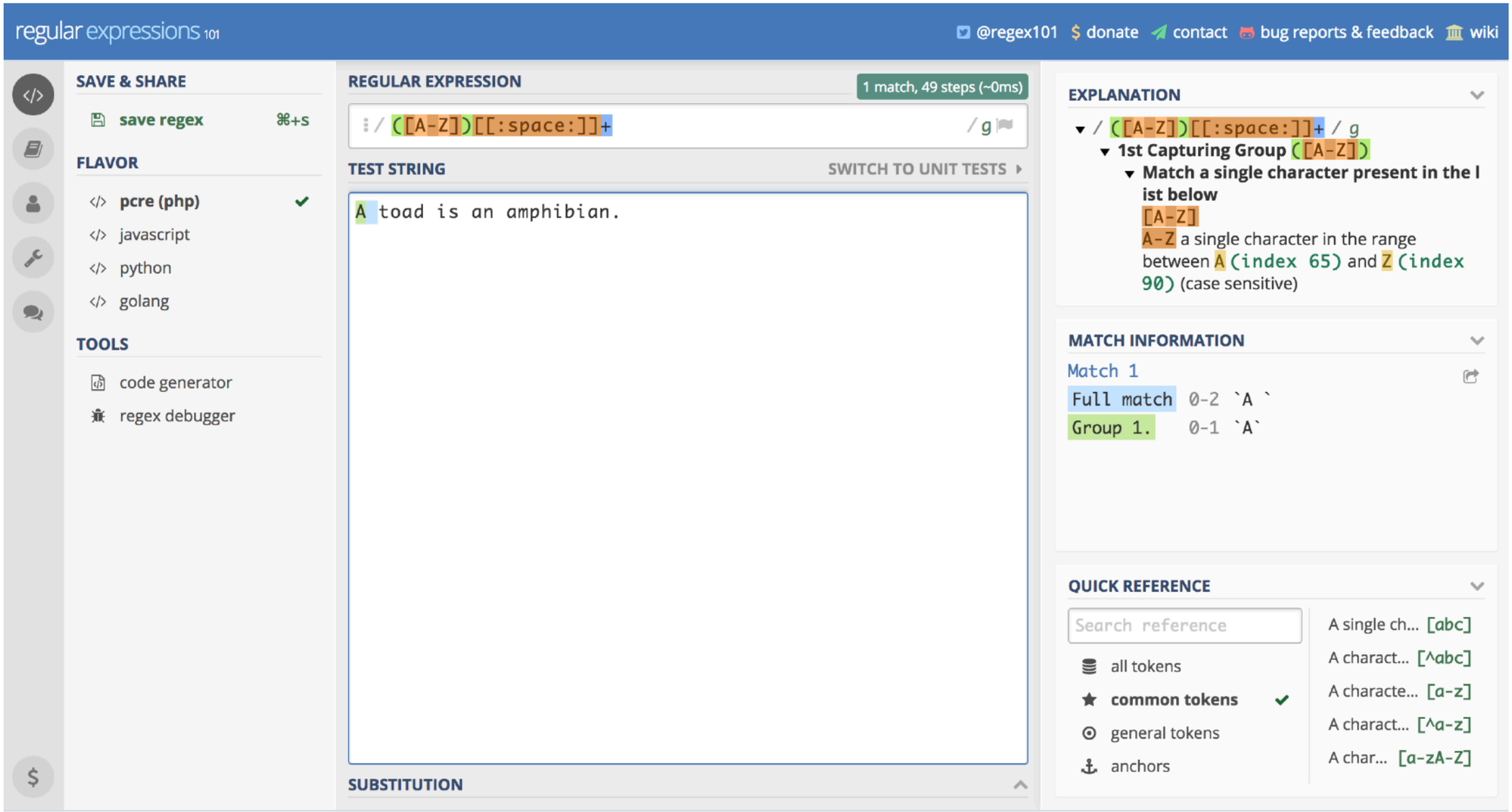
… offline help with generating regex …



https://github.com/gadenbuie/regexplain

Regex101

… online regex helper …



https://regex101.com/

Recap

- **Regular Expressions ("regex")**
  - Find, extract, or replace patterns in strings.
- **Using Regex**
  - Metacharacters provide ways to generate dynamic patterns to match inside a string.
  - Be wary of *greedy* vs. *lazy* capture
- **Resources**
  - Cheatsheets, regex pattern builders, and more!

Acknowledgements

Acknowledgements

- Style of the RStudio cheatsheet for strings

- R-atic who inspired the RStudio cheatsheet style for strings.