Introduction to Regular Expressions

Matthias Braun

Today's Plan

- What are regular expressions?

Today's Plan

- What are regular expressions?
- Why are they so useful?

Regular Languages

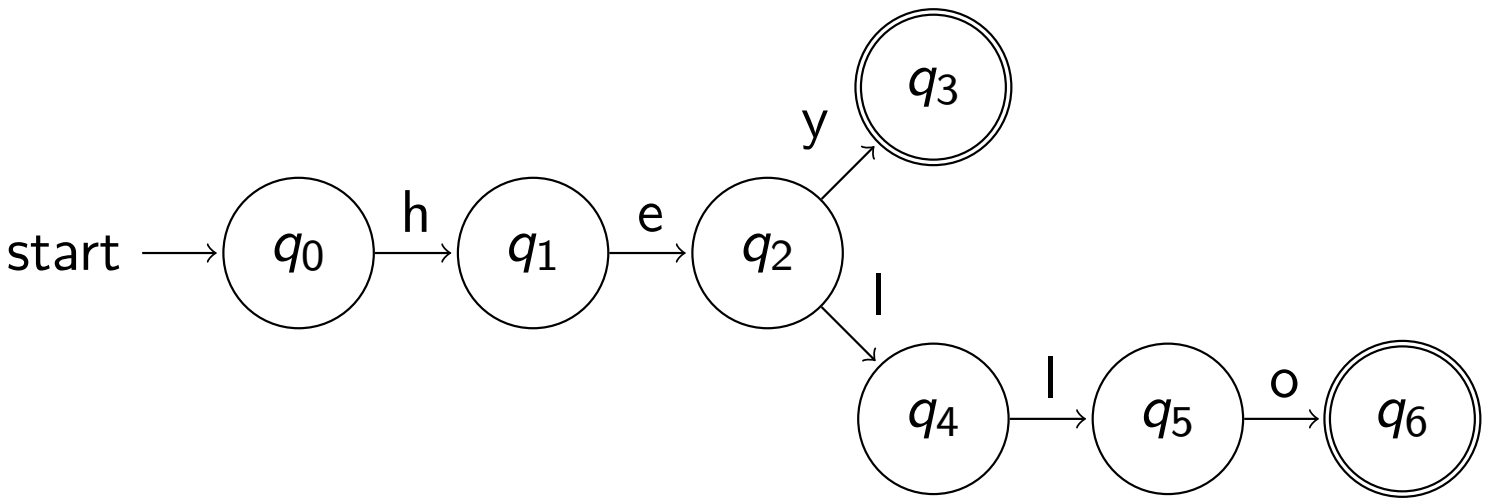
A reminder

- Languages are called regular if there's an automaton that accepts all the language's sentences

Regular Languages

A reminder

- Languages are called regular if there's an automaton that accepts all the language's sentences
- For example, the language $\{hey, hello\}$ is regular since the following automaton accepts all of its two sentences:



Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:
  - A word spelled the American or English way: color/colour, license/licence

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:
  - A word spelled the American or English way: color/colour, license/licence
  - Two digits, followed by a vowel (aeiou)

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:
  - A word spelled the American or English way: color/colour, license/licence
  - Two digits, followed by a vowel (aeiou)
  - One or more whitespace (tab, space)

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:
  - A word spelled the American or English way: color/colour, license/licence
  - Two digits, followed by a vowel (aeiou)
  - One or more whitespace (tab, space)
  - Dates of a certain format, like 2020-08-24

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:
  - A word spelled the American or English way: color/colour, license/licence
  - Two digits, followed by a vowel (aeiou)
  - One or more whitespace (tab, space)
  - Dates of a certain format, like 2020-08-24
- Using regexes, we can search text

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:
  - A word spelled the American or English way: color/colour, license/licence
  - Two digits, followed by a vowel (aeiou)
  - One or more whitespace (tab, space)
  - Dates of a certain format, like 2020-08-24
- Using regexes, we can search text
- We can also replace text that matches the regex (e.g., replace all tabs with spaces)

Regular Expressions

- Regular expressions (regexes) also describe regular languages: `he(y|llo)` is equivalent to the previous automaton
- Even more useful, regular expressions can describe patterns in text, for example:
  - A word spelled the American or English way: color/colour, license/licence
  - Two digits, followed by a vowel (aeiou)
  - One or more whitespace (tab, space)
  - Dates of a certain format, like 2020-08-24
- Using regexes, we can search text
- We can also replace text that matches the regex (e.g., replace all tabs with spaces)
- Regexes are a pattern language for text

Elements of a Regular Expression

- given an alphabet $\{a, b\}$, there are the primitive regular expressions `a` and `b`, they match the letters “a” and “b”

Elements of a Regular Expression

- given an alphabet $\{a, b\}$, there are the primitive regular expressions `a` and `b`, they match the letters “a” and “b”
- we can combine those primitives:

Elements of a Regular Expression

- given an alphabet $\{a, b\}$, there are the primitive regular expressions `a` and `b`, they match the letters “a” and “b”
- we can combine those primitives:
  - Choice: `a|b`

Elements of a Regular Expression

- given an alphabet $\{a, b\}$, there are the primitive regular expressions `a` and `b`, they match the letters “a” and “b”
- we can combine those primitives:
  - Choice: `a|b`
  - Concatenation: `ab`

Elements of a Regular Expression

- given an alphabet $\{a, b\}$, there are the primitive regular expressions `a` and `b`, they match the letters “a” and “b”
- we can combine those primitives:
  - Choice: `a|b`
  - Concatenation: `ab`
  - Repetition (zero or more): `a*`

Elements of a Regular Expression

- given an alphabet $\{a, b\}$, there are the primitive regular expressions `a` and `b`, they match the letters “a” and “b”
- we can combine those primitives:
  - Choice: `a|b`
  - Concatenation: `ab`
  - Repetition (zero or more): `a*`

regex	matched string
`a|b`	“a”
`a|b`	“b”
`ab`	“ab”
`a*`	“”
`a*`	“a”
`a*`	“aa”
`a*`	“aaa”

Elements of a Regular Expression

Precedence and grouping

- The rules of operator precedence from lowest to highest:

Elements of a Regular Expression

Precedence and grouping

- The rules of operator precedence from lowest to highest:
  - Choice: `a|b`

Elements of a Regular Expression

Precedence and grouping

- The rules of operator precedence from lowest to highest:
  - Choice: `a|b`
  - Concatenation: `ab`

Elements of a Regular Expression

Precedence and grouping

- The rules of operator precedence from lowest to highest:
  - Choice: `a|b`
  - Concatenation: `ab`
  - Repetition: `a*`

Elements of a Regular Expression

Precedence and grouping

- The rules of operator precedence from lowest to highest:
  - Choice: `a|b`
  - Concatenation: `ab`
  - Repetition: `a*`
- We can group subexpressions in regexes with parentheses to override the precedence rules, for example:

regex	matched string
`ab*`	"abbbb"
`(ab)*`	"abababab"
`0123*`	"012"
`0123*`	"01233333"
`0(123)*`	"0123123123"
`0(123)*`	"0"

Your Turn!

Part 1: Does it match?

1. Which of these strings does the regex `ab*` match?
    1. "a"
    2. "abab"
    3. "ab"
    4. "abb"
    5. "b"
2. Which of these strings does the regex `XY|(ab)*` match?
    1. ""
    2. "XY"
    3. "XYXY"
    4. "aba"
    5. "abab"

Your Turn!

Part 2: Gotta match 'em all

Given these strings, create *one* regex that matches them all:

1. YRPGX
2. YX
3. YRPGRPGRPGX
4. YRPGRPGX

Regular Expressions in Practice

- Command-line tools like `grep` let us use regexes to search text inside files (if you're on Windows, you can use `grep` via Cygwin, WSL, or Linux in the browser)

Regular Expressions in Practice

- Command-line tools like `grep` let us use regexes to search text inside files (if you're on Windows, you can use `grep` via Cygwin, WSL, or Linux in the browser)
- These tools extend the original idea of regular expressions from formal language theory. They have extensions that enable them to describe a language that's *not* regular: a language that *can't* be defined using an automaton. Still, we call this regular expressions and not something like "regex++"

Regular Expressions in Practice

- Command-line tools like `grep` let us use regexes to search text inside files (if you're on Windows, you can use `grep` via Cygwin, WSL, or Linux in the browser)
- These tools extend the original idea of regular expressions from formal language theory. They have extensions that enable them to describe a language that's *not* regular: a language that *can't* be defined using an automaton. Still, we call this regular expressions and not something like "regex++"
- One of these extensions are backreferences to previous matches: `(the |a |an )\1` matches
  - "the the "
  - "a a "
  - "an an "

Regular Expressions in Practice

- Command-line tools like `grep` let us use regexes to search text inside files (if you're on Windows, you can use `grep` via Cygwin, WSL, or Linux in the browser)
- These tools extend the original idea of regular expressions from formal language theory. They have extensions that enable them to describe a language that's *not* regular: a language that *can't* be defined using an automaton. Still, we call this regular expressions and not something like "regex++"
- One of these extensions are backreferences to previous matches: `(the |a |an )\1` matches
  - "the the "
  - "a a "
  - "an an "
- regex101.com lets us test and inspect our regexes

grep

Putting the “re” into “g/re/p”

- grep searches text **g**lobally in a file using a **r**egular **e**xpression and then **p**rints the line where the regex matched

grep

Putting the “re” into “g/re/p”

- grep searches text **g**lobally in a file using a **r**egular **e**xpression and then **p**rints the line where the regex matched
- Create a file `grep_test.txt`:

grep_test.txt

```
British spelling: colour
Second line
American spelling: color
```

grep

Putting the "re" into "g/re/p"

- grep searches text **g**lobally in a file using a **r**egular **e**xpression and then **p**rints the line where the regex matched
- Create a file `grep_test.txt`:

grep_test.txt

```
British spelling: colour
Second line
American spelling: color
```

- Let's print the line with the American spelling of "color":

```
grep -E "color" grep_test.txt
```

grep

Putting the “re” into “g/re/p”

- The `-E` option specifies which flavor of regular expressions we use: **E**xtended regexes

grep

Putting the “re” into “g/re/p”

- The `-E` option specifies which flavor of regular expressions we use: **E**xtended regexes
- The first argument to `grep` is actually a regex

grep

Putting the “re” into “g/re/p”

- The `-E` option specifies which flavor of regular expressions we use: **E**xtended regexes
- The first argument to `grep` is actually a regex
- Thus, to get both spellings of “color”, we run

```
grep -E "colou*r" grep_test.txt
```

grep

Putting the “re” into “g/re/p”

- The `-E` option specifies which flavor of regular expressions we use: **E**xtended regexes
- The first argument to `grep` is actually a regex
- Thus, to get both spellings of “color”, we run

```
grep -E "colou*r" grep_test.txt
```

- Adding the `-n` option to `grep` adds line numbers to the results:

```
1:British spelling: colour
3:American spelling: color
```

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`
- The dash `-` inside the character class is a special character creating a range of characters

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`
- The dash - inside the character class is a special character creating a range of characters
- Examples of character classes:

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`
- The dash - inside the character class is a special character creating a range of characters
- Examples of character classes:
  - matches lowercase letters: `[a-z]`

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`
- The dash - inside the character class is a special character creating a range of characters
- Examples of character classes:
  - matches lowercase letters: `[a-z]`
  - matches lowercase letters and digits: `[a-z0-9]`

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`
- The dash - inside the character class is a special character creating a range of characters
- Examples of character classes:
  - matches lowercase letters: `[a-z]`
  - matches lowercase letters and digits: `[a-z0-9]`
  - matches the letters b and o: `[bo]`, equivalent to (b|o)

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`
- The dash - inside the character class is a special character creating a range of characters
- Examples of character classes:
  - matches lowercase letters: `[a-z]`
  - matches lowercase letters and digits: `[a-z0-9]`
  - matches the letters b and o: `[bo]`, equivalent to (b|o)
  - matches some punctuation characters: `[;.,!:?]`

Character Classes

- Let's create a regex matching two digits ("01", "89", "22")
- Easy: `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)`
- Character classes simplify this: `[0-9][0-9]`
- The dash - inside the character class is a special character creating a range of characters
- Examples of character classes:
  - matches lowercase letters: `[a-z]`
  - matches lowercase letters and digits: `[a-z0-9]`
  - matches the letters b and o: `[bo]`, equivalent to (b|o)
  - matches some punctuation characters: `[;.,!:?]`
- Matches two digits followed by a vowel: `[0-9][0-9][aeiou]`

Negated Character Classes

- The caret symbol ^ has special meaning inside a character class: it negates the class

Negated Character Classes

- The caret symbol `^` has special meaning inside a character class: it negates the class
- `[^xyz]` matches a character if it's not an x, y, or z

Negated Character Classes

- The caret symbol `^` has special meaning inside a character class: it negates the class
- `[^xyz]` matches a character if it's not an x, y, or z
- `[0-9][^0-9][0-9]` means "a digit, followed by anything that's not a digit, followed by a digit":

Negated Character Classes

- The caret symbol `^` has special meaning inside a character class: it negates the class
- `[^xyz]` matches a character if it's not an x, y, or z
- `[0-9][^0-9][0-9]` means "a digit, followed by anything that's not a digit, followed by a digit":
  - "8A1"

Negated Character Classes

- The caret symbol `^` has special meaning inside a character class: it negates the class
- `[^xyz]` matches a character if it's not an x, y, or z
- `[0-9][^0-9][0-9]` means "a digit, followed by anything that's not a digit, followed by a digit":
  - "8A1"
  - "0.0"

Negated Character Classes

- The caret symbol `^` has special meaning inside a character class: it negates the class
- `[^xyz]` matches a character if it's not an x, y, or z
- `[0-9][^0-9][0-9]` means "a digit, followed by anything that's not a digit, followed by a digit":
  - "8A1"
  - "0.0"
  - "1/2"

Negated Character Classes

- The caret symbol `^` has special meaning inside a character class: it negates the class
- `[^xyz]` matches a character if it's not an x, y, or z
- `[0-9][^0-9][0-9]` means "a digit, followed by anything that's not a digit, followed by a digit":
  - "8A1"
  - "0.0"
  - "1/2"
  - "1 2"

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces
- `\d` is short for `[0-9]` and `\D` is short for `[^0-9]`. We can simplify our previous regex to `\d\D\d` [1]

[1] Requires option `-P` instead of `-E` for **P**erl-compatible regexes

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces
- `\d` is short for `[0-9]` and `\D` is short for `[^0-9]`. We can simplify our previous regex to `\d\D\d` [1]
- `\w` is short for [a-zA-Z0-9_] which is useful for finding words. The regex `\w\w\w` matches all these:

[1] Requires option `-P` instead of `-E` for **P**erl-compatible regexes

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces
- `\d` is short for `[0-9]` and `\D` is short for `[^0-9]`. We can simplify our previous regex to `\d\D\d` [1]
- `\w` is short for [a-zA-Z0-9_] which is useful for finding words. The regex `\w\w\w` matches all these:
  - "And"

[1] Requires option `-P` instead of `-E` for **P**erl-compatible regexes

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces
- `\d` is short for `[0-9]` and `\D` is short for `[^0-9]`. We can simplify our previous regex to `\d\D\d` [1]
- `\w` is short for [a-zA-Z0-9_] which is useful for finding words. The regex `\w\w\w` matches all these:
  - "And"
  - "the"

[1] Requires option `-P` instead of `-E` for **P**erl-compatible regexes

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces
- `\d` is short for `[0-9]` and `\D` is short for `[^0-9]`. We can simplify our previous regex to `\d\D\d` [1]
- `\w` is short for [a-zA-Z0-9_] which is useful for finding words. The regex `\w\w\w` matches all these:
  - "And"
  - "the"
  - "hat"

[1] Requires option `-P` instead of `-E` for **P**erl-compatible regexes

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces
- `\d` is short for `[0-9]` and `\D` is short for `[^0-9]`. We can simplify our previous regex to `\d\D\d` [1]
- `\w` is short for [a-zA-Z0-9_] which is useful for finding words. The regex `\w\w\w` matches all these:
  - "And"
  - "the"
  - "hat"
  - "DID"

[1] Requires option `-P` instead of `-E` for **P**erl-compatible regexes

Character Class Shorthands

Some classes of characters are so common, there are shorthands for them

- `\s` matches any whitespace characters: tabs and spaces
- `\d` is short for `[0-9]` and `\D` is short for `[^0-9]`. We can simplify our previous regex to `\d\D\d` [1]
- `\w` is short for [a-zA-Z0-9_] which is useful for finding words. The regex `\w\w\w` matches all these:
  - "And"
  - "the"
  - "hat"
  - "DID"
  - "fit"

[1] Requires option `-P` instead of `-E` for **P**erl-compatible regexes

More Quantifiers

? means zero or one

- `a?` matches zero or one “a”

More Quantifiers

? means zero or one

- `a?` matches zero or one “a”
- Another way to think about `a?`: “a” is optional

More Quantifiers

? means zero or one

- `a?` matches zero or one "a"
- Another way to think about `a?`: "a" is optional
- `colou?r` matches both "color" and "colour": try this with `grep -En "colou?r" grep_test.txt`

More Quantifiers

? means zero or one

- `a?` matches zero or one "a"
- Another way to think about `a?`: "a" is optional
- `colou?r` matches both "color" and "colour": try this with `grep -En "colou?r" grep_test.txt`
- `r?age` matches both "rage" and "age"

More Quantifiers

? means zero or one

- `a?` matches zero or one "a"
- Another way to think about `a?`: "a" is optional
- `colou?r` matches both "color" and "colour": try this with `grep -En "colou?r" grep_test.txt`
- `r?age` matches both "rage" and "age"
- `(semi)?colon` matches both "semicolon" and "colon"

More Quantifiers

? means zero or one

- `a?` matches zero or one "a"
- Another way to think about `a?`: "a" is optional
- `colou?r` matches both "color" and "colour": try this with `grep -En "colou?r" grep_test.txt`
- `r?age` matches both "rage" and "age"
- `(semi)?colon` matches both "semicolon" and "colon"
- `reg(ular)? ?ex(pression)?` matches both "regular expression" and "regex"

More Quantifiers

\+ means one or more

- `a+` matches one or more “a”

More Quantifiers

\+ means one or more

- `a+` matches one or more “a”
- `0+`: at least one zero

More Quantifiers

\+ means one or more

- `a+` matches one or more “a”
- `0+`: at least one zero
- `[a-z]+`: at least one lowercase letter

More Quantifiers

\+ means one or more

- `a+` matches one or more "a"
- `0+`: at least one zero
- `[a-z]+`: at least one lowercase letter
- `(lol)+!`: at least one "lol" followed by a single exclamation mark

More Quantifiers

\+ means one or more

- `a+` matches one or more “a”
- `0+`: at least one zero
- `[a-z]+`: at least one lowercase letter
- `(lol)+!`: at least one “lol” followed by a single exclamation mark
- `int\s+A\s*=\s*42;` matches:

More Quantifiers

\+ means one or more

- `a+` matches one or more “a”
- `0+`: at least one zero
- `[a-z]+`: at least one lowercase letter
- `(lol)+!`: at least one “lol” followed by a single exclamation mark
- `int\s+A\s*=\s*42;` matches:
  - `int A = 42;`

More Quantifiers

\+ means one or more

- `a+` matches one or more “a”
- `0+`: at least one zero
- `[a-z]+`: at least one lowercase letter
- `(lol)+!`: at least one “lol” followed by a single exclamation mark
- `int\s+A\s*=\s*42;` matches:
  - `int A = 42;`
  - `int   A=   42;`

More Quantifiers

\+ means one or more

- `a+` matches one or more “a”
- `0+`: at least one zero
- `[a-z]+`: at least one lowercase letter
- `(lol)+!`: at least one “lol” followed by a single exclamation mark
- `int\s+A\s*=\s*42;` matches:
  - `int A = 42;`
  - `int   A=   42;`
  - `int A=42;`

More Quantifiers

{6} means exactly six

- `a{6}` matches “aaaaaa”

More Quantifiers

{6} means exactly six

- `a{6}` matches “aaaaaa”
- `\d{3}` matches three decimal digits

More Quantifiers

{6} means exactly six

- `a{6}` matches “aaaaaa”
- `\d{3}` matches three decimal digits
- `[a-zA-Z]{4,8}` matches 4 to 8 letters, tries to match as many letters as possible but no more than 8

More Quantifiers

{6} means exactly six

- `a{6}` matches “aaaaaa”
- `\d{3}` matches three decimal digits
- `[a-zA-Z]{4,8}` matches 4 to 8 letters, tries to match as many letters as possible but no more than 8
- `Z{9,}` matches when there are at least nine Zs

More Quantifiers

{6} means exactly six

- `a{6}` matches "aaaaaa"
- `\d{3}` matches three decimal digits
- `[a-zA-Z]{4,8}` matches 4 to 8 letters, tries to match as many letters as possible but no more than 8
- `Z{9,}` matches when there are at least nine Zs
- `\d{6,}` matches a number equal to or greater than a million

Your Turn!

Finding integers

- Create a regex that matches integers of arbitrary length:
  - "1"
  - "-50"
  - "+961"
  - "2983292032"
- Your regex should *not* match any of these strings:
  - ""
  - "-"
  - "+"
- Use character classes with quantifiers to make the regex as short as you can
- Use regex101.com to test and inspect your regex

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

  "#FF0000"

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

"#FF0000"

"#00FF00"

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

"#FF0000"

"#00FF00"

"#0000FF"

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

"#FF0000"

"#00FF00"

"#0000FF"

"#2E8B57"

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

  "#FF0000"

  "#00FF00"

  "#0000FF"

  "#2E8B57"

- Your tasks:

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

  "#FF0000"

  "#00FF00"

  "#0000FF"

  "#2E8B57"

- Your tasks:
  1. Create a regex to match these strings

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

  "#FF0000"

  "#00FF00"

  "#0000FF"

  "#2E8B57"

- Your tasks:
  1. Create a regex to match these strings
  2. Then, make the "#" optional to also match strings like "FF00FA"

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

  "#FF0000"

  "#00FF00"

  "#0000FF"

  "#2E8B57"

- Your tasks:
  1. Create a regex to match these strings
  2. Then, make the "#" optional to also match strings like "FF00FA"
  3. Finally, make your regex case insensitive to also match strings like "#fF00bC" and "a0eec2"

Your Turn!

Hex colors

- Colors are often represented as three numbers, each number specifying the color's red, green, and blue value (0 to 255)
- On the WWW, these three numbers are commonly specified in hexadecimal, with a leading number sign "#":

  "#FF0000"

  "#00FF00"

  "#0000FF"

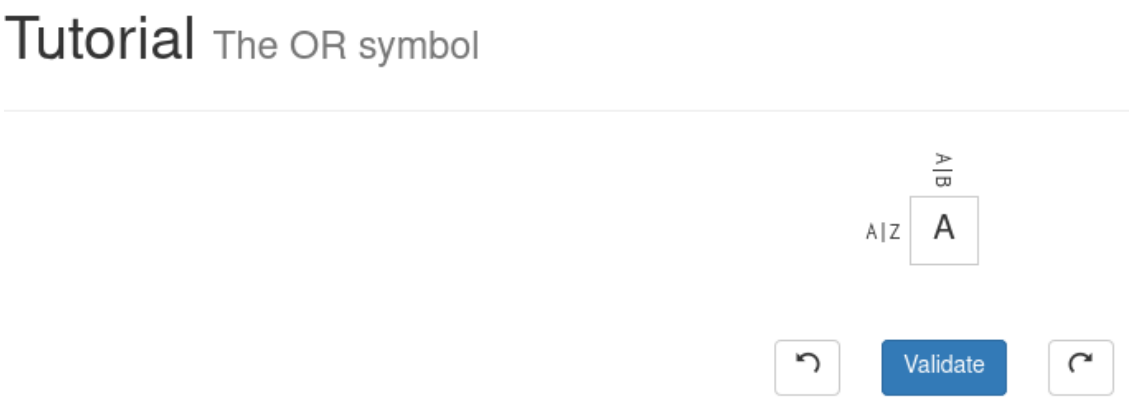
  "#2E8B57"

- Your tasks:
  1. Create a regex to match these strings
  2. Then, make the "#" optional to also match strings like "FF00FA"
  3. Finally, make your regex case insensitive to also match strings like "#fF00bC" and "a0eec2"
- Your regex should *not* match a string like "#G9F1X2" since "G9" and "X2" are not hexadecimal numbers

Your Turn!

Regex Crossword

- Do the tutorial at regexcrossword.com:



- How far can you get with your current regex skills? To intermediate level? For extra credit, send me a screenshot of the hardest puzzle you could solve