Regex Engine – Five Key Points

Regex Engine Overview

Regex Engine is Generic

Engine follows the instructions in your pattern

Incase of incomplete match, engine can try alternate paths

You are responsible for writing efficient patterns

Five Key Points

- One Character at a time
- Left to Right
- Greedy, Lazy, and Backtracking
- Groups
- Look ahead and Look behind

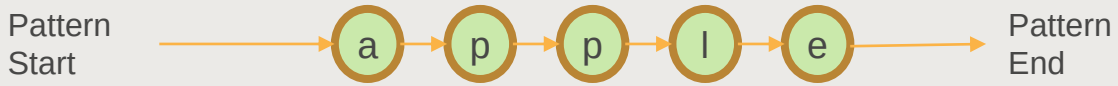
#1 - One Character at a time

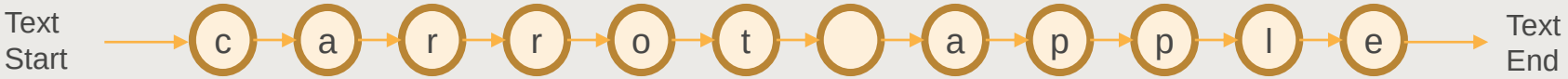
Pattern and Text are evaluated one character at a time

Path taken depends on results of the match

One Character at a time

```
Problem: Find all occurrences of apple
Pattern: apple
Text: carrot apple
```





#2 - Left to Right

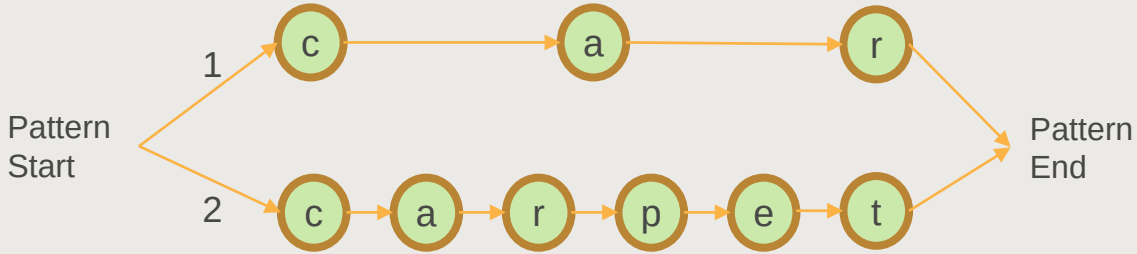
Pattern and text are evaluated left to right

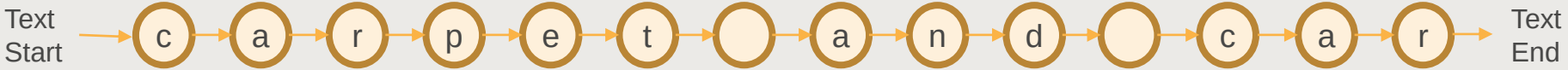
Left most pattern is attempted first and gradually moves right to attempt other patterns

When left most pattern is not viable, other patterns are evaluated

Left to Right

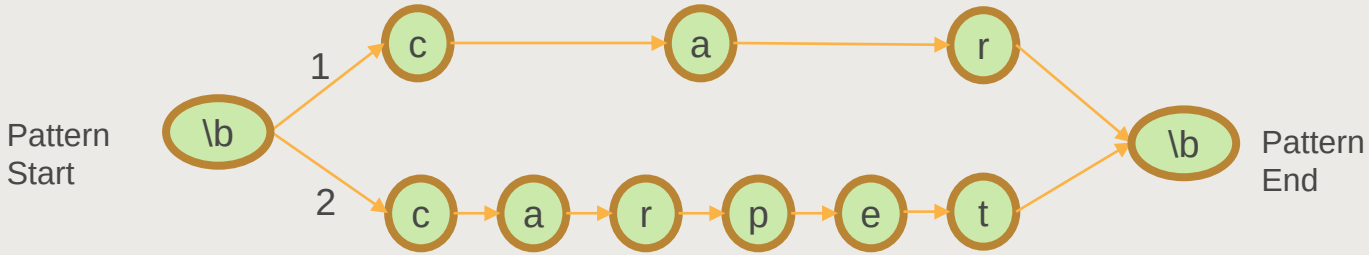
```
Problem: Find all matches for the words - car, carpet
Pattern: car|carpet
Text: carpet and car
```

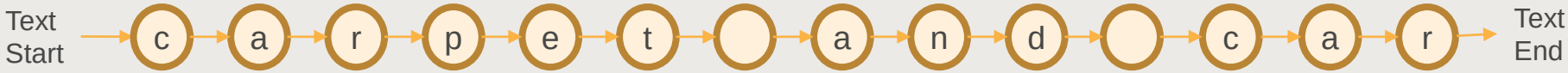




Left to Right – Example 2

```
Problem: Find all matches for the words - car, carpet
Pattern: \b(car|carpet)\b
Text: carpet and car
```





Extract common sub-expression

```
Pattern: car(pet)?
```

Summary

Pattern and Text are evaluated Left to Right

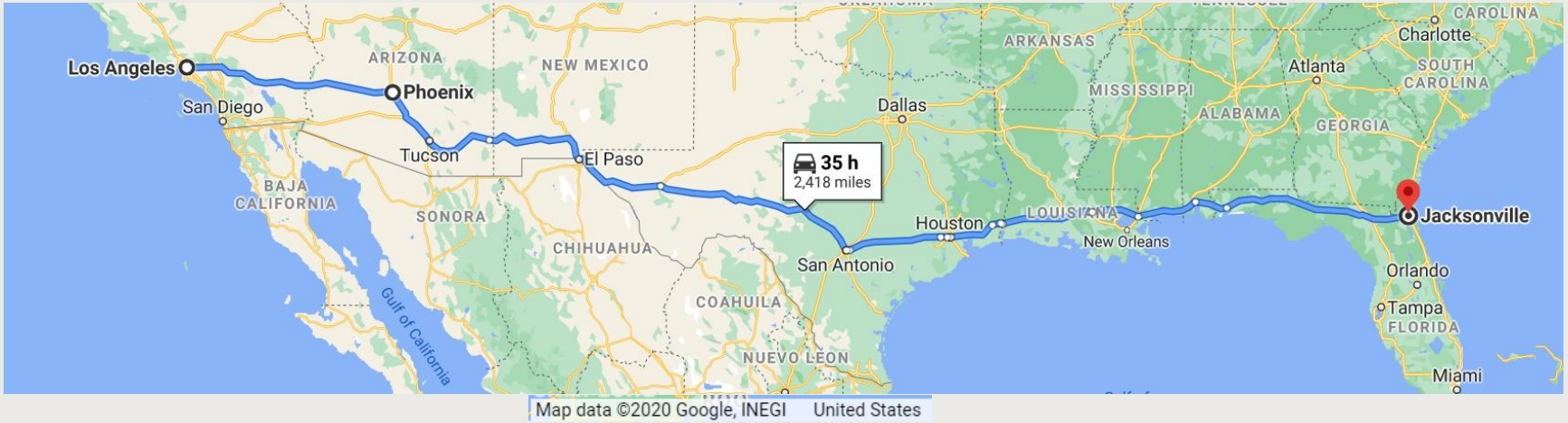
Regex engine uses backtracking to evaluate other paths

Extract common subexpressions

Write more precise patterns first followed by more generic patterns

#3 Greedy, Lazy and Backtracking

Greedy Analogy



LA to Phoenix Direction - Travel on I-10 East and Take Phoenix Exit

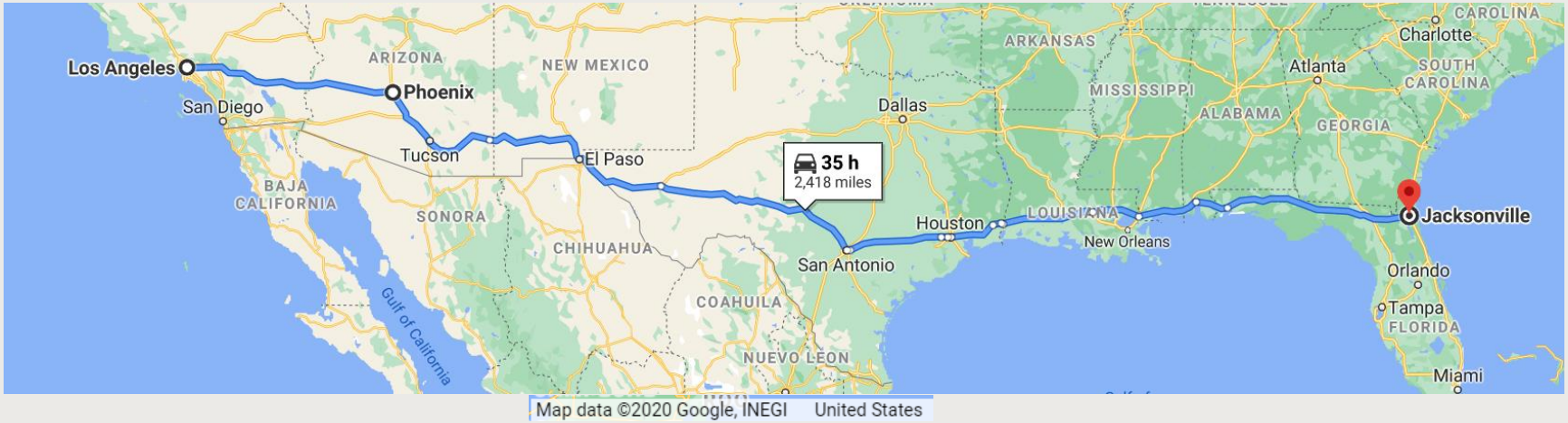
Pattern: (I10 East)+(Phoenix Exit)

(I10 East)+ => + is greedy and it will go all the way to Jacksonville

(I10 East)+**(Phoenix Exit)** => and then backtrack to check for Phoenix exit

Greedy Trip: LA->Jacksonville->Phoenix

Lazy Analogy



LA to Phoenix Direction - Travel on I-10 East and Take Phoenix Exit

```
Pattern: (I10 East)+?(Phoenix Exit)
(I10 East)+? => ? is lazy and it will pause at every exit
(I10 East)+?(Phoenix Exit) => To check for Phoenix exit.
                              If not Phoenix, travel to next exit

Lazy Trip: LA->Phoenix
```

Greedy

Quantifiers such as `*,+` are greedy. They will try to match as much of the input text as possible

Wildcard with greedy quantifiers can consume entire text and can starve rest of the pattern

```
Example: .*(expression), .+(expression)
```

To match rest of the pattern, regex engine backtracks on the captured text

Greedy – Consumes as much of the input text as possible and then gives-up characters to match rest of the pattern

Lazy

Lazy matches as few times as possible and attempts to match rest of the pattern

Quantifiers can be turned to Lazy by adding a ? after the quantifier

```
Example: .*?(expression), +?(expression)
```

When there is no match for a pattern, lazy mode backtracks on the pattern and expands to match more characters in input.

Quiz

Problem: Extract the number at the end of each sentence

Text: First 1234. Second 5678.

Pattern: .+(\d+)[.]

Question: What value would (\d+) capture?

a. 1234

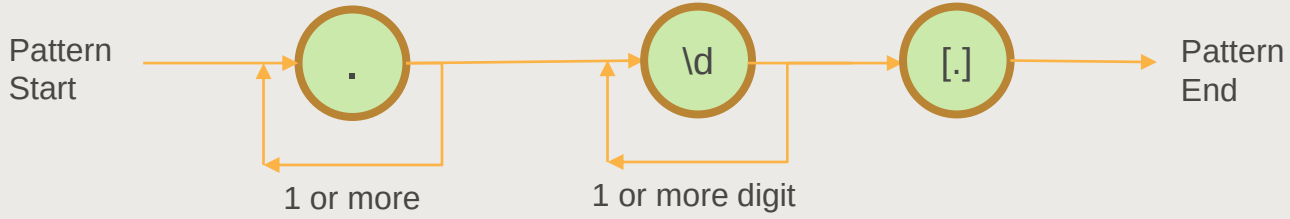
b. 4

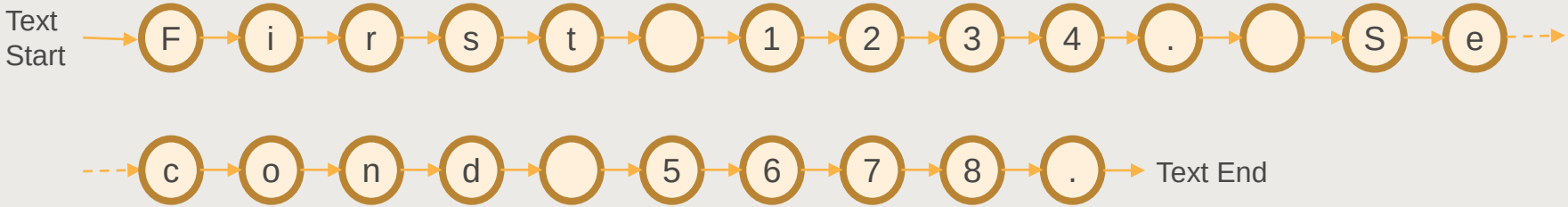
c. 5678

d. 8

Greedy and Backtracking

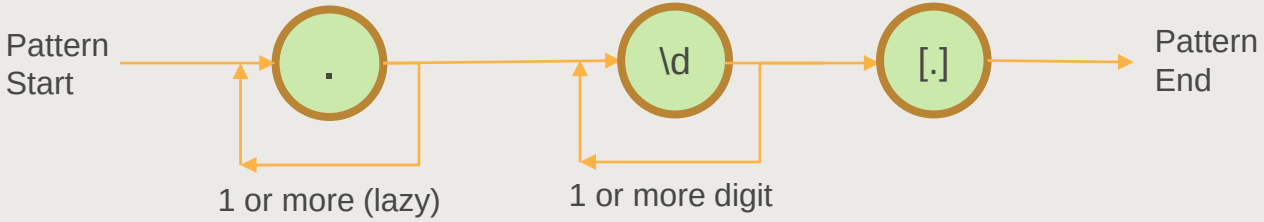
```
Problem: Extract the number at the end of each sentence.
Pattern: .+(\d+)[.]
Text: First 1234. Second 5678.
```

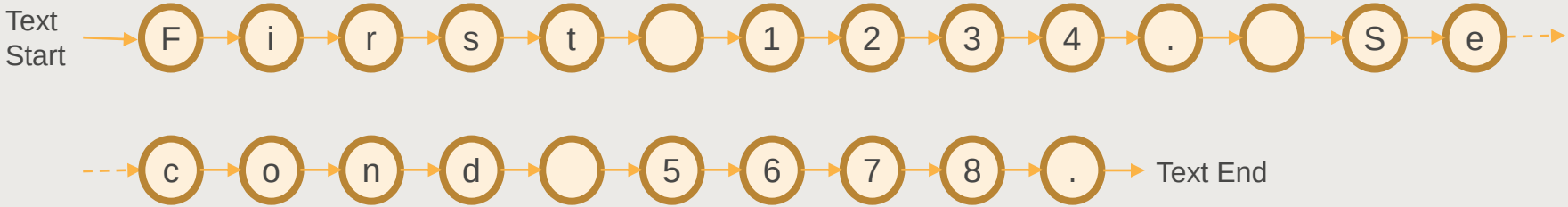




Lazy and Backtracking

```
Problem: Find sentences ending with a number and extract the number.
Pattern: .+?(\d+)[.]
Text: First 1234. Second 5678.
```





One more solution

Problem: Find sentences ending with a number and extract the number.

Pattern: [^\d]*(\d+)[.]

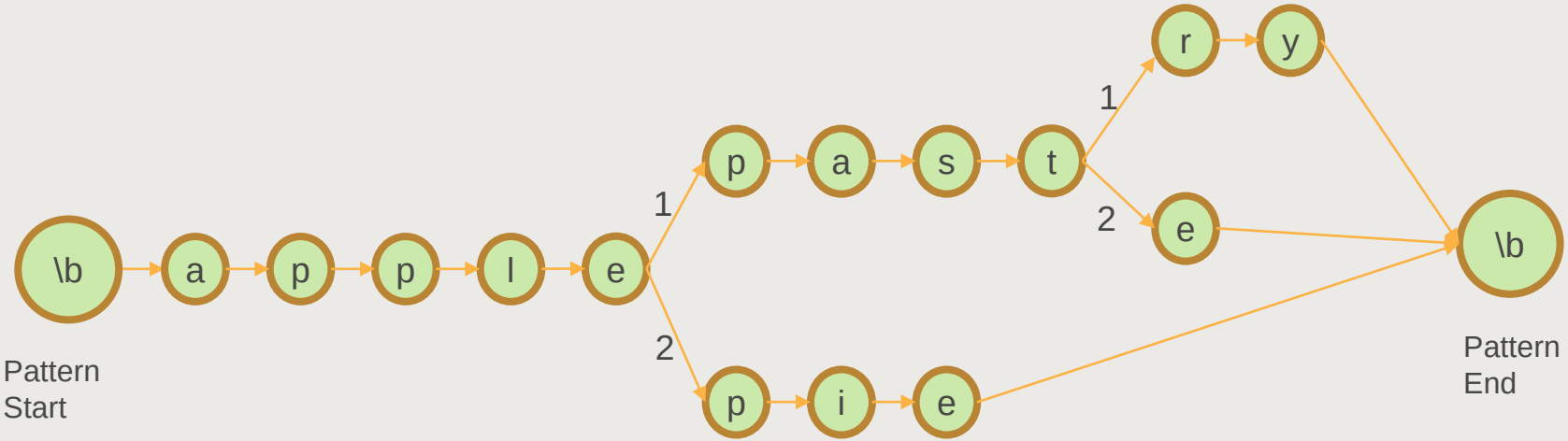
Text: First 1234. Second 5678.

[^\d]* => captures all non-digits.

So, this pattern will stop when it sees a digit and let rest of the pattern to match.

Exhaustive – Backtracking

Problem: Find all matches for applepastry, applepaste or applepie
Pattern: \bapple(past(ry|e)|pie)\b



Text: Twenty popular recipes to make applepaste
Popular applepie recipe

#4 - Groups

Groups are part of the pattern specified inside a parenthesis ()

Break a pattern into sub-patterns

```
(?P<year>\d{4})(?P<month>\d{2})(?P<day>\d{2})
```

Reuse common expression `car(pet)?`

Mark optional expression `car(pet)?`

Capture repeating expression `\d+(,\d{3})*(\.\d{2})?`

Groups – Capture repeating patterns

Problem: Find numbers

Pattern: \d+(,\d{3})*(\.\d{2})?

\d+ => Captures digits

(,\d{3})* => Captures digits with thousand separator ,000,000

(\.\d{2})? => Captures decimal part (optional) .98

Example

123

123**,456**

123**,456,789**

123**,456,789.98**

Indexed Group – Access by number

```
Problem: Extract year, month and day
Pattern: (\d{4})(\d{2})(\d{2})
Text: 20160501
```

Groups are numbered left-to-right

- Group 0 => (\d{4})(\d{2})(\d{2})
- Group 1 => (\d{4})(\d{2})(\d{2})
- Group 2 => (\d{4})(\d{2})(\d{2})
- Group 3 => (\d{4})(\d{2})(\d{2})

Named Groups – Access by name

```
Problem: Extract year, month and day
Pattern:(?P<year>\d{4})(?P<month>\d{2})(?P<day>\d{2})
Text: 20160501
```

Access by name or number

- Group 0 => (\d{4})(\d{2})(\d{2})
- Group 1 or ‘year’ => (\d{4})(\d{2})(\d{2})
- Group 2 or ‘month’ => (\d{4})(\d{2})(\d{2})
- Group 3 or ‘day’ => (\d{4})(\d{2})(\d{2})

Non-Capturing Groups

Group capture is expensive, turn it off when not needed

Non-Capturing Group Syntax (?:)

```
Pattern: \d+(?:,\d{3})*(?:\.\d{2})?
```

Groups are used here to capture repeating patterns

```
20,160,501.67
```

Back reference

Back reference refers to a group that was captured earlier and used subsequently in the pattern

```
Syntax: (?P=name)
```

```
Problem: Identify repeating words
Pattern: (?P<word>\w+)\s+(?P=word)
Text: capture duplicate duplicate words

(?P=word) => refers to the value captured by (?P<word>\w+)
```

Replacement (Substitution)

Replacement pattern can use groups captured in find pattern

```
Syntax: \g<name>
```

```
    Problem: Delete repeating word
    Find Pattern: (?P<word>\w+)\s+(?P=word)
    Replacement Pattern: \g<word>
    Text: capture duplicate duplicate words
```

```
    \g<word> => refers to value captured in the find pattern
```

#5 - Look ahead and Look behind

Look ahead – Peek at what is coming up next without consuming the characters

Look behind – Look at what came before current character

Both are called zero width assertions

- Returns True or False
- Does not consume any characters

Allows you to implement more complex conditional logic

IF (pre-condition) THEN (expression)

Example: Password is valid only if it has a mix of upper-case, lower-case, digit and special character

Positive Look ahead

IF (pre-condition) THEN (expression)

Pre-condition looks at the characters coming up next (without consuming them)

Expression is evaluated only when the pre-condition is met

```
Positive Look Ahead Syntax: (?=precondition)
```

Example - Positive Look ahead

Problem: Identify Text in Non-English language

Text: Tamil தமிழ், Hindi हिन्दी, Japanese 日本語, 1234.

\w+ => match a word characters in any language (excludes mark-character)

\w[^\s]* => match a word in any language (including mark-character)

To exclude English characters, the pre-condition is [^A-Za-z0-9]

Demo - Positive Look ahead

Problem: Identify Text in Non-English language

Text: Tamil தமிழ், Hindi हिन्दी, Japanese 日本語, 1234.

Pattern: (?=[^A-Za-z0-9])\w[^\s]*

Negative Look ahead

IF NOT (pre-condition) THEN (expression)

Pre-condition looks at the characters coming up next (without consuming them)

Expression is evaluated only when the pre-condition is NOT met

`Negative Look Ahead Syntax: (?!precondition)`

Example - Negative Look ahead

Problem: Identify Text in Non-English language

Text: Tamil தமிழ், Hindi हिन्दी, Japanese 日本語, 1234.

Pattern: `(?![A-Za-z0-9])\w[^\s]*`

Positive Look behind

IF (pre-condition) THEN (expression)

Pre-condition goes back and checks characters before the current one (without consuming them)

Expression is evaluated only when the pre-condition is met

```
Positive Look Behind Syntax: (?<=precondition)
```

Example – Positive Look Behind

Problem: Extract product-numbers that have a four-letter product code

Text: AAAA1234 BBBBBB1234 CCCC5679 TTTTTTT3444 ZZ88191 YAYY616222

\d+ => match digits

\b[A-Z]{4}\d+\b => Will return the entire product identifier- AAAA1234

However, we need only the number. So, we need to rewrite the pattern with \b[A-Z]{4} as a pre-condition

Demo – Positive Look Behind

Problem: Extract product-numbers that have a four-letter product code

Text: AAAA1234 BBBBBB1234 CCCC5679 TTTTTTT3444 ZZ88191 YAYY616222

Pattern: (?<=\b[A-Z]{4})\d+\b

Negative Look behind

IF NOT (pre-condition) THEN (expression)

Pre-condition goes back and checks characters before the current one (without consuming them)

Expression is evaluated only when the pre-condition is NOT met

```
Negative Look Behind Syntax: (?<!precondition)
```

Example - Negative Lookbehind

Problem: List items with price NOT ending in .99

Text:

```
chair 4.98
coffee 1.99
fan 10.97
car 11499.59
banana 0.09
```

.+ => will capture each product and its price

.+(\.99)$ => will capture product that ends in .99

But we don’t want it to end in .99. So, we need to rewrite the pattern with (\.99) as a negative pre-condition

Demo - Negative Lookbehind

Problem: List items with price NOT ending in .99

Text:

```
chair 4.98
coffee 1.99
fan 10.97
car 11499.59
banana 0.09
```

Pattern: `(?m).+(?<!\.99)$`

Chandra Lingam

57,000+ Students

For AWS self-paced video courses, visit:

https://www.cloudwavetraining.com/

Cloud Wave LLC