Regex Five Key Points

Content Prepared By: Chandra Lingam, Cotton Cola Designs LLC

Regex Engine - Overview

Python, Perl, .Net Regular Expression Engines belong to a class of implementation known as Nondeterministic Finite Automation

Pattern drives how the engine processes input text and what path to take

Remembers previous successful state and incase of incomplete match, it can go back to previous successful state and try a different path

Regex Engine is generic in nature and takes the path as instructed in pattern.

It puts the responsibility on developer to define efficient patterns to guide the engine

Five Key Points

One Character at a time

Left to Right

Greedy, Lazy and Backtracking

Groups

Look ahead and Look behind

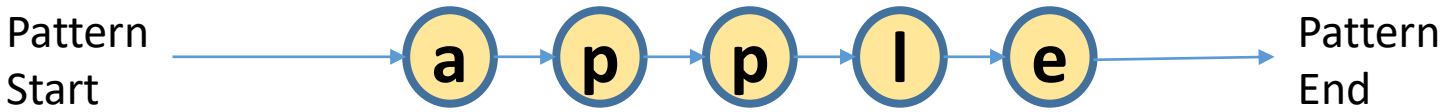
#1 - One Character at a time

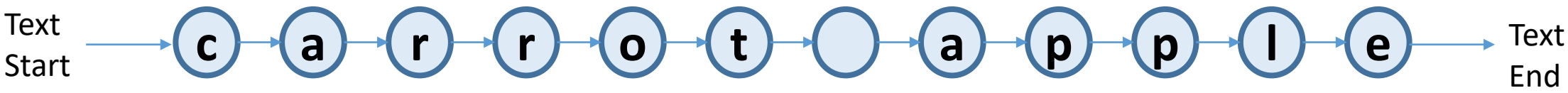
Pattern and Text gets evaluated one character at a time

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

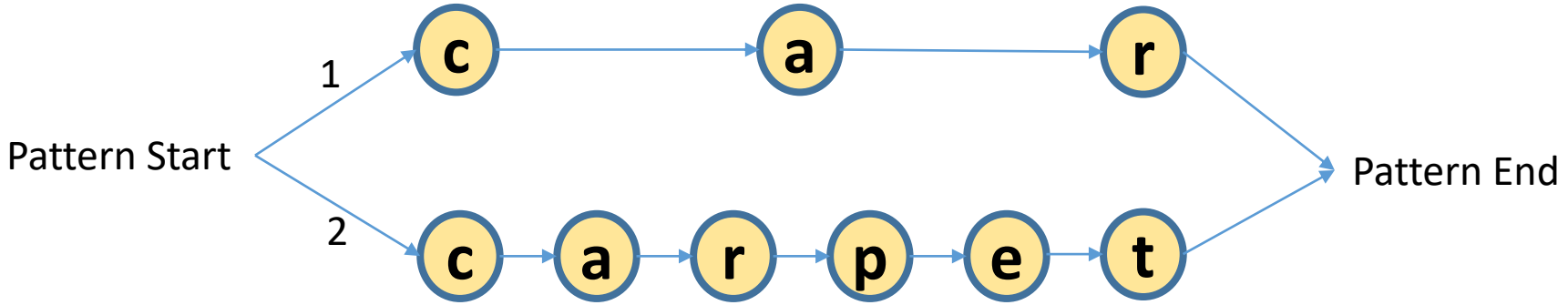
Pattern is traversed left to right

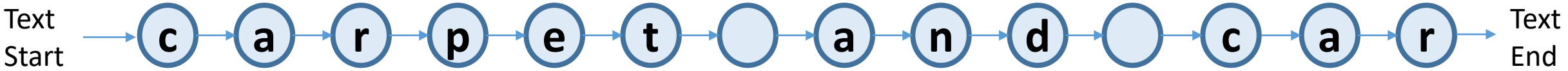
- Pattern defined left most is attempted first and gradually moves right to attempt other patterns
- All viable Patterns are evaluated before proceeding with next character in the text.

Text is traversed left to right

Left to Right

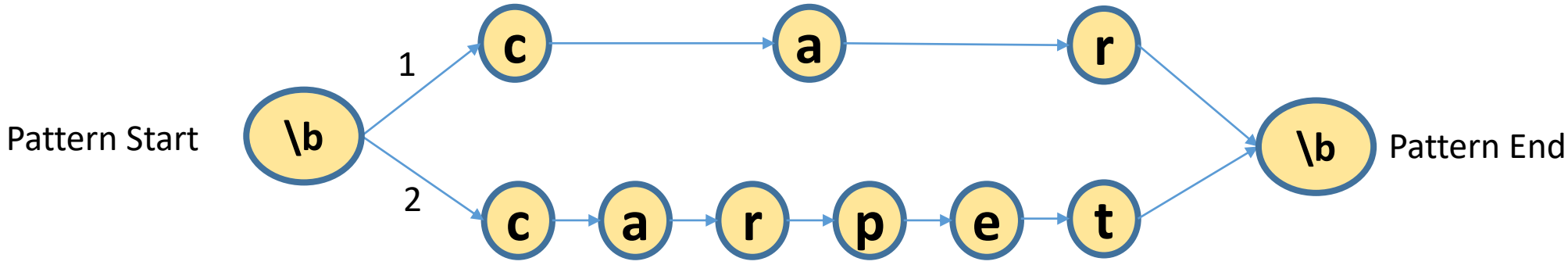
```
Problem: Find all words that contain car or carpet
Pattern: car|carpet
Text: carpet and car
```

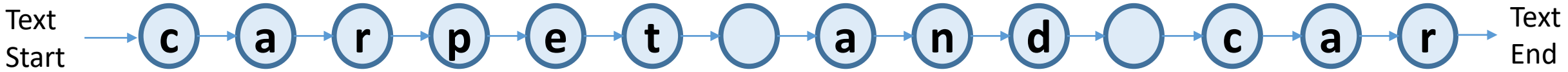




Left to Right – Example 2

```
Problem: Find occurrences of words – car, carpet
Pattern: \b(car|carpet)\b
Text: carpet and car
```





Summary

Pattern and Text are evaluated Left to Right

In-order to minimize backtracking, extract common patterns out

```
car(pet)?
```

Write more precise patterns first followed by more generic patterns

#3 - Greedy, Lazy and Backtracking

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

Sometimes these quantifiers when applied to wildcard like ‘.’ can consume entire text and can starve rest of the pattern

Regex engine evaluates and backtracks one step at a time to see if it can match rest of the pattern

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

Quiz

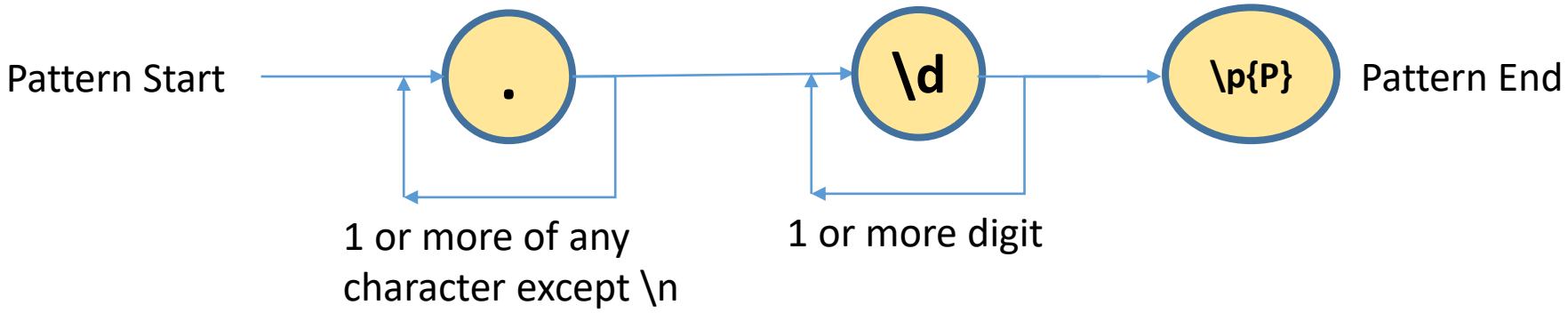
- Problem: Match sentence that ends with number. Extract number.
- Pattern: **.+(\d+)[.!]**
- Text: First 1234. Second 5678!

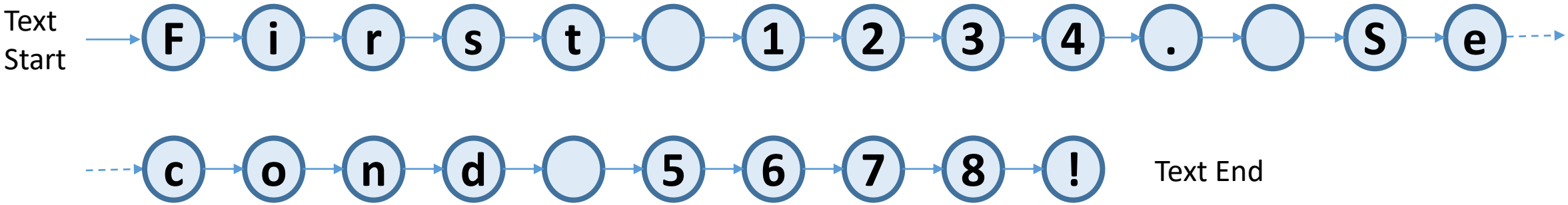
Question: What value would (\d+) capture?

Note ‘.’ is a wildcard character pattern.

Greedy and Backtracking

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





Lazy

Idea behind lazy is to match as few times as possible for quantifiers and proceed to match rest of the pattern

Quantifiers *,+,? can be turned to Lazy by adding a ? After the quantifier

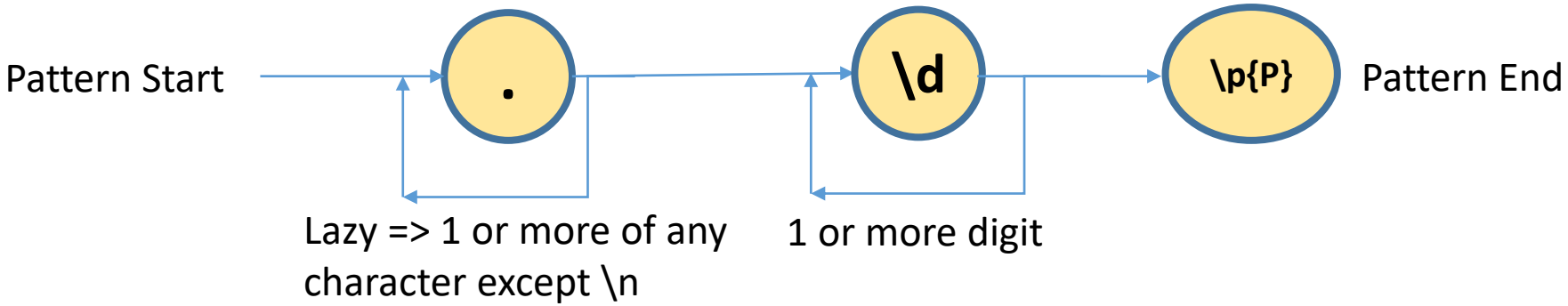
Example: *?, +?, ??

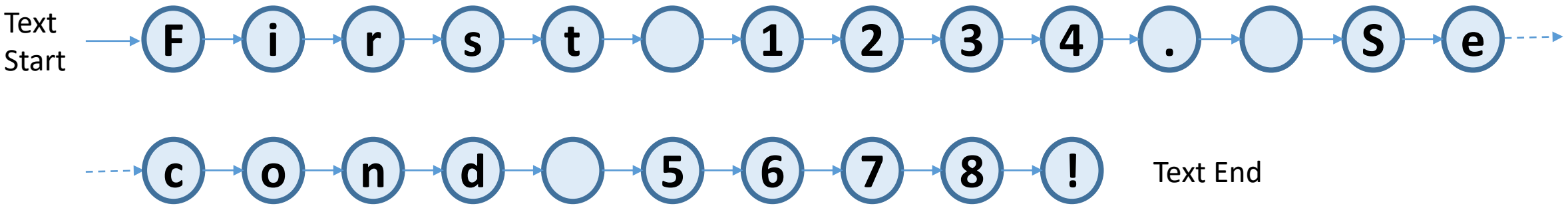
When there is no match for a pattern, lazy mode backtracks on the pattern and expands to match more characters in input.

Lazy – Consumes as few input text as possible and then attempts to match rest of the pattern

Lazy and Backtracking

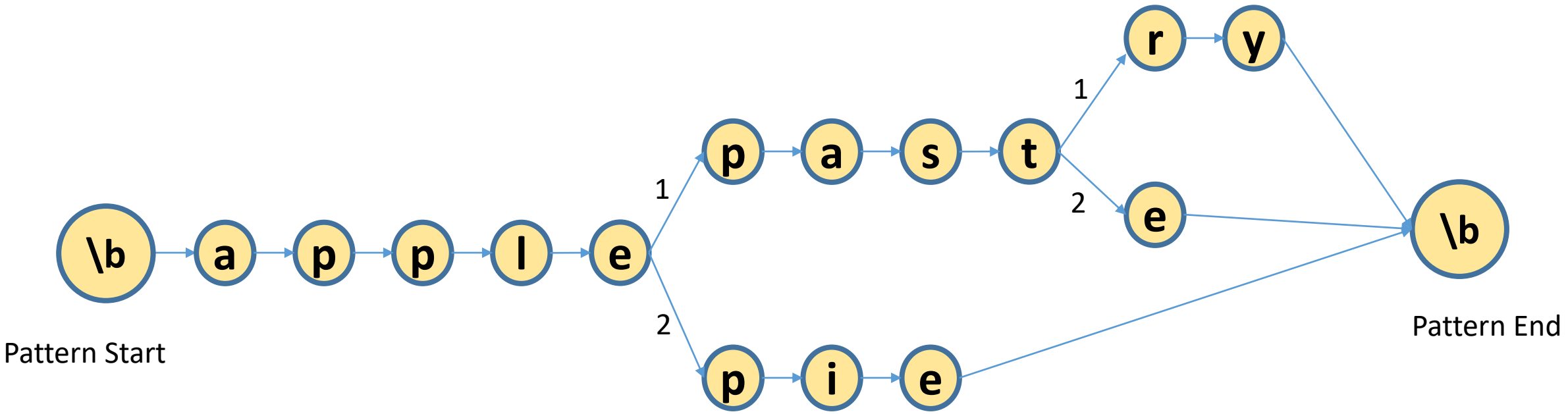
```
Problem: Find sentences ending with a number and extract the number.
Pattern: .+?(\d+)[.!]
Text: First 1234. Second 5678!
```





Exhaustive – Backtracking

```
Problem: Find words that match applepastry, applepaste or applepie
Pattern: \bapple(past(ry|e)|pie)\b
```



Text: Twenty popular recipes to make applepaste
Popular applepie recipe

#4 - Groups

Break a pattern into sub-patterns; pinpoint location of matching string and sub-strings

Groups are any pattern specified inside a parenthesis ()

Reuse common patterns, Mark as optional – minimize backtracking

```
car(pet)?
```

Extract Values

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

Groups Examples

Capture Repeating Sub-Patterns

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

```
Matches:
123
123,456
123,456,789
```

Indexed Group

Capture matching substring

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

Access by group number

- Groups are numbered from left-to-right. Every open parenthesis is assigned an increasing number starting from 1
- Group 0 => Refers to whole pattern
- 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

Capture matching substring – Named Group

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

Access by group name

- Groups are numbered from left-to-right. Every open parenthesis is assigned an increasing number starting from 1
- Group 0 => Refers to whole pattern
- 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

Match without group capture

- `Problem: Match number of format 999,999,999.99.`
- `Pattern: \d+(,\d{3})*(\.\d{2})?`
- `Text: 20,160,501.67`

Groups are used here to capture repeating patterns

- `\d+(,\d{3})*(\.\d{2})?`
- We are interested in only complete match and not what got matched inside groups

Group capture is expensive, turn it off when not needed

- Non-Capturing Group (?:)

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

Group - Back reference and Substitution

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

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

- `Problem: Identify repeating letters`
- `Pattern: (?P<letter>\w)(?P=letter)`

Substitution pattern used for text replacement can refer to previously captured group

- `Problem: Identify repeating words and remove repetition`
- `Find Pattern: (?P<word>\w+)\s+(?P=word)\b`
- `Replacement Pattern: \g<word>`

#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
- Look ahead is similar to “if (expression) yes_expression”

Look ahead and Look behind can contain patterns

Allows you to implement more complex conditional logic

Does not backtrack – Once it return a true/false, job is done. If pattern does not match, Look ahead/Look behind would not backtrack to try another match.

Positive Lookahead

- Positive Lookahead (?=expression)
- Proceed only when lookahead expression evaluates to TRUE
  - Problem: Identify if file contains text annotated in language other than English alphabet.
  - Approach: Is character a language character or mark character? Is character not part of English?
  - Pattern: (?i)(?=[^a-z0-9])\w[^\s]*
  - Text: Tamil தமிழ், Hindi हिन्दी, Japanese日本語, 1234.

Negative Lookahead

- Negative Lookahead (?!expression)
- Proceed only when lookahead expression evaluates to FALSE
  - Problem: Identify if file contains text annotated in another language
  - Approach: Is character a language character or mark character? Is character not part of English?
  - Pattern: (?i)(?![a-z0-9])\w[^\s]*
  - Text: Tamil தமிழ், Hindi हिन्दी, Japanese日本語, 1234.

Positive Lookbehind

- Positive Lookbehind (?<=expression)
- Proceed only when lookbehind expression evaluates to TRUE
  - Problem: Match the digits if they are preceded by 4 alpha characters
  - Approach: Pre-condition. See if there are 4 alpha characters before digits
  - Pattern: (?i)(?<=\b[a-z]{4})\d+\b
  - Text: abcd1234 testing1234 EFGH6789

Negative Lookbehind

- Negative Lookbehind (?<!expression)
- Proceed only when lookbehind expression evaluates to FALSE
  - Problem: Find words(ignore single character word) that does not end with lowercase
  - Pattern: \w{2,}(?<![a-z])\b
  - Text: I am in typinG a letter in mixed casE