CTIS 256 Web Technologies II

Note # 6

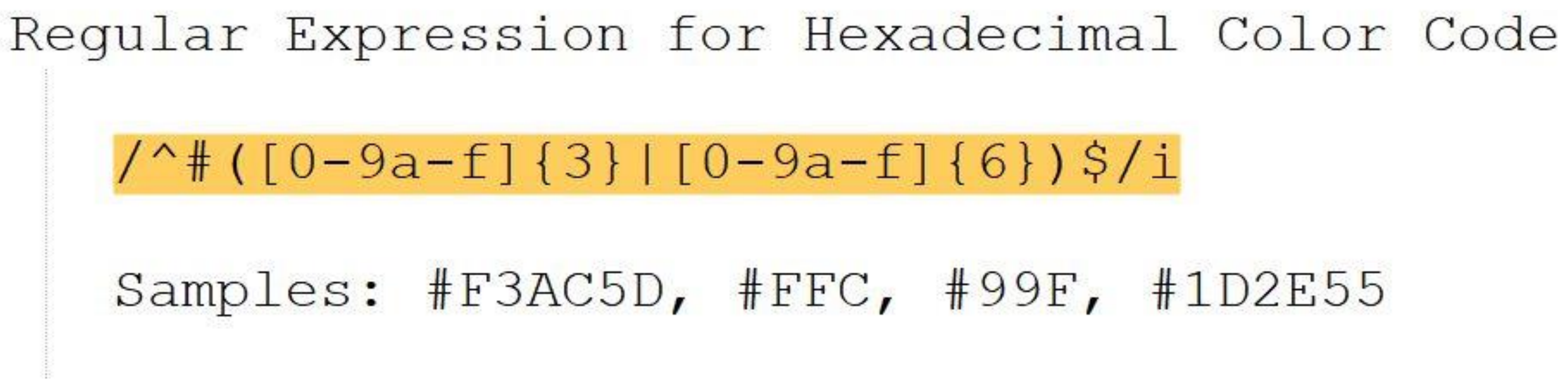
Serkan GENÇ

Regular Expression

- Regular expression (regex) is a powerful tool to define string patterns in a formal way.
- String patterns can be phone number format, date, time, email addresses, urls, zip codes, and custom defined patterns such as the format of a flight ticket , serial no of an item.
- It is used for searching complex patterns and/or replacing a pattern with a new one.
- One line of “regex” is worth tens of lines of codes.
- It is supported by almost all languages (javascript, php, java, C, C++, etc.)

A Sample RegEx

Formal Way:



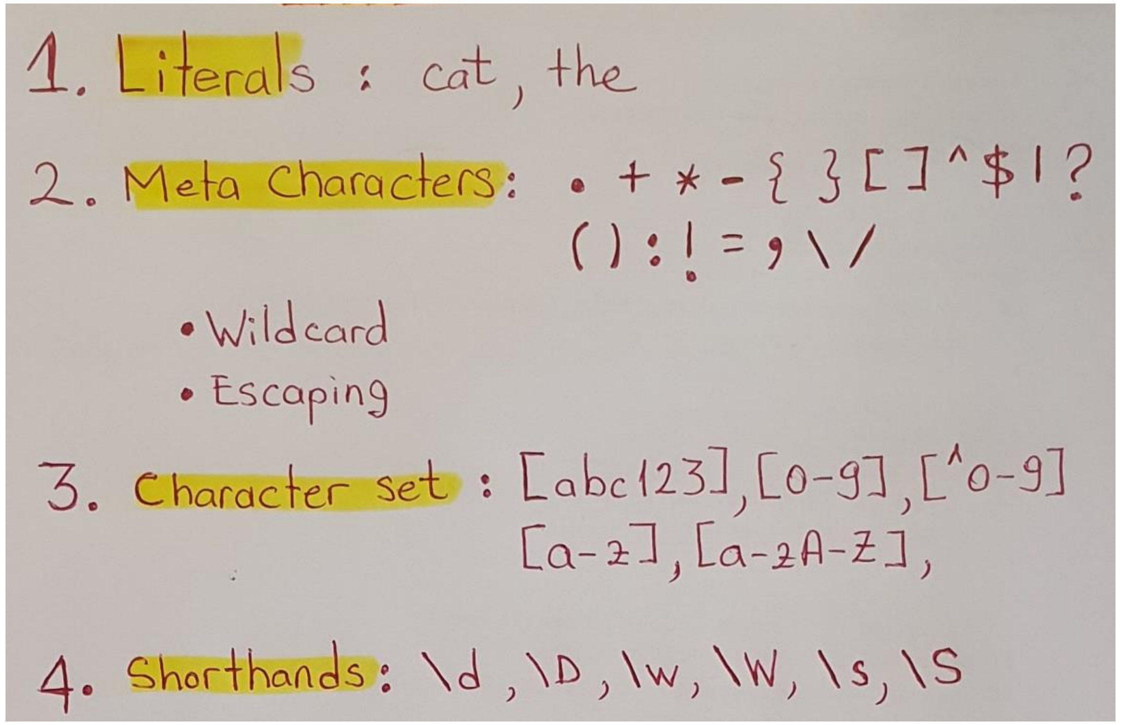
Regular Expression for Hexadecimal Color Code

```
/^#([0-9a-f]{3}|[0-9a-f]{6})$/i
```

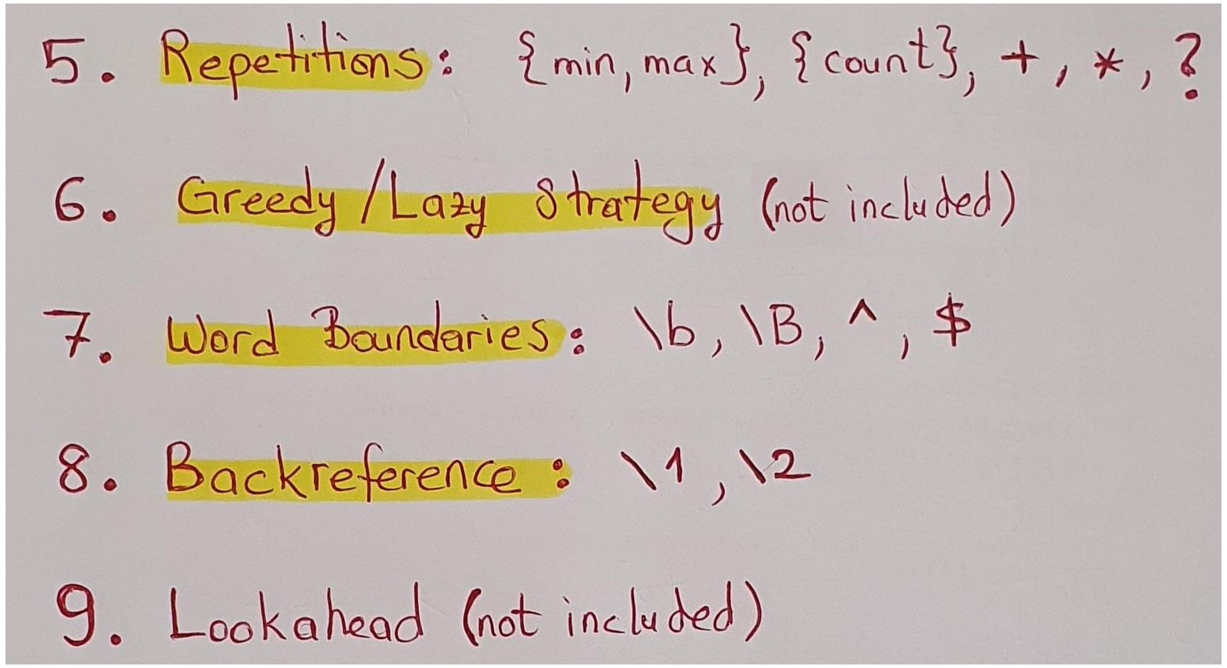
Samples: #F3AC5D, #FFC, #99F, #1D2E55

Informal Way: A hexadecimal color code starts with a “#” hashmark, and it is followed by 3 or 6 hexadecimal digits. A hexadecimal digit is represented by any one of “0” to “9” and “a” to “f” in lower or uppercase symbols. It represents a digit in 16 base.

Outline



1. Literals : cat, the
2. Meta Characters: . + * - { } [ ] ^ $ | ?
   ( ) : ! = , \ /
   - Wildcard
   - Escaping
3. Character set : [abc123], [0-9], [^0-9]
   [a-z], [a-zA-Z],
4. Shorthands: \d , \D , \w, \W, \s, \S



5. Repetitions: {min, max}, {count}, +, *, ?
6. Greedy / Lazy Strategy (not included)
7. Word Boundaries: \b, \B, ^, $
8. Backreference : \1 , \2
9. Lookahead (not included)

RegEx Syntax

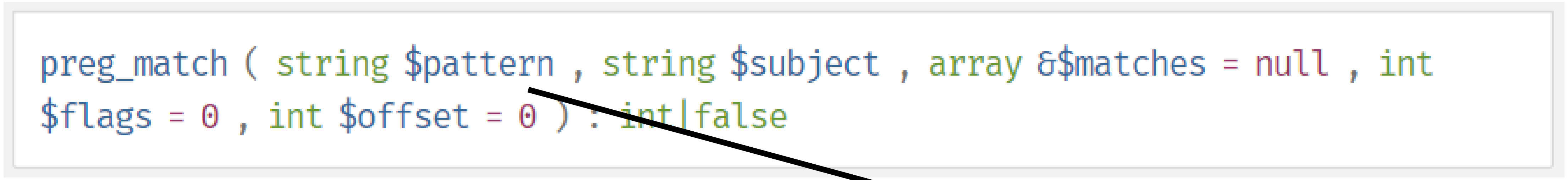
/ *Regular Expression Pattern* / *flags*

start delimeter

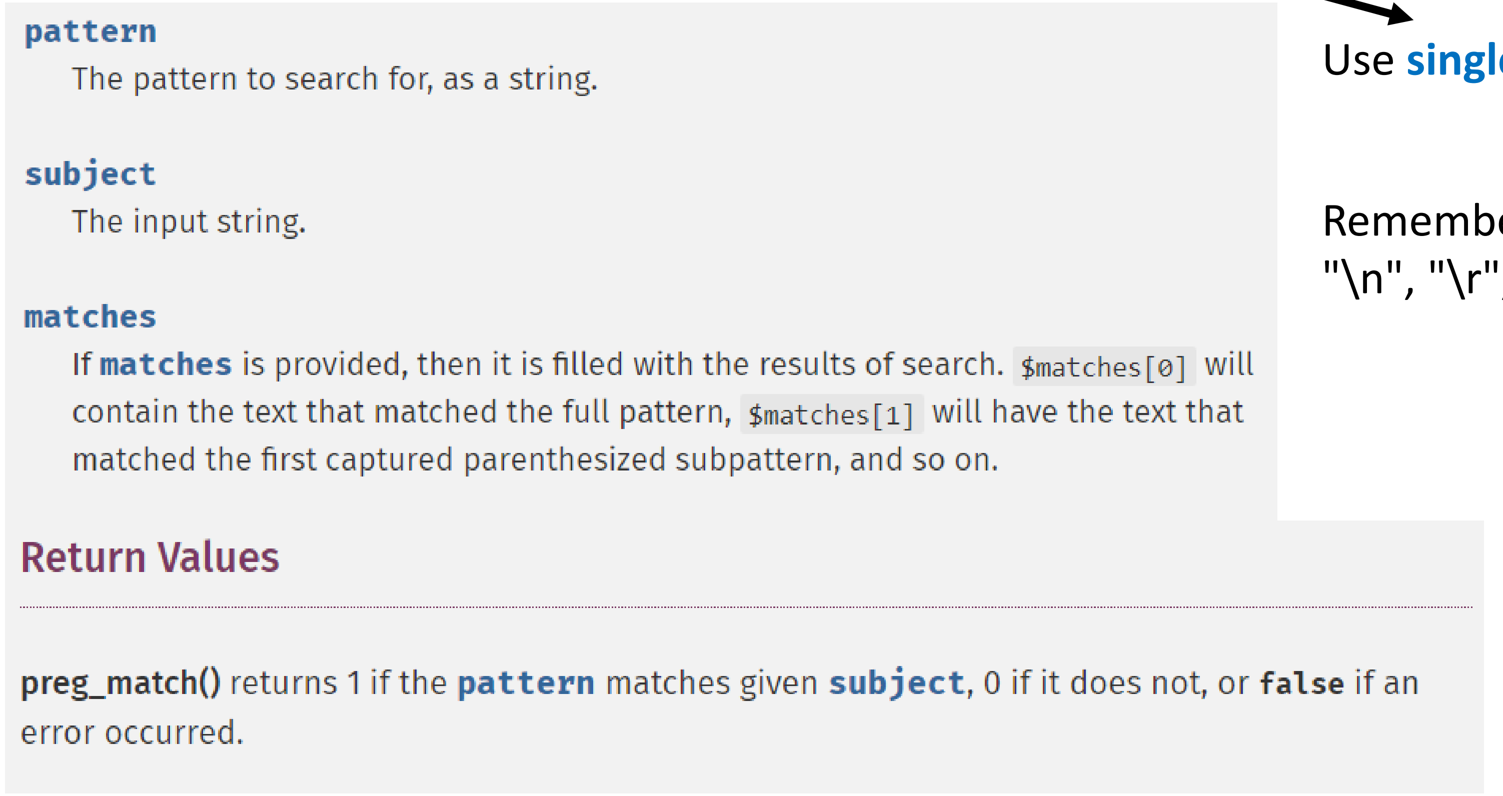
end delimeter

i : case insensitive
m : multi-line
u : unicode support

Matching in PHP



```
preg_match ( string $pattern , string $subject , array &$matches = null , int
$flags = 0 , int $offset = 0 ) : int|false
```



pattern
The pattern to search for, as a string.

subject
The input string.

matches
If **matches** is provided, then it is filled with the results of search. `$matches[0]` will contain the text that matched the full pattern, `$matches[1]` will have the text that matched the first captured parenthesized subpattern, and so on.

Return Values

preg_match() returns 1 if the **pattern** matches given **subject**, 0 if it does not, or **false** if an error occurred.

Use **single quotes** in regular expression pattern

'/regexp/'

Remember double quote converts some escaped chars "\n", "\r", "\x65", "\1", "\$", etc.

Literal

- Fixed strings such as cat , hello, the.
- Since regular expression is slower than other keyword searching functions such as strpos(), or stripos(), don't use regular expression just for literal searching.

Samples:

```
preg_match('/the/', 'The birds run away from them immediately') ;

preg_match('/the/', 'them and therefore') ;

preg_match('/cat/', 'There are three categories.') ;

// it does not find "\" and "n", it looks for new line.
preg_match("/\n/", 'any \n in the string') ;  // Wrong
preg_match("/\\n/", 'any \n in the string') ; // True but confusing
preg_match('/\n/', 'any \n in the string') ; // Use single quote
```

Metacharacter: . *(dot)*

- Dot character “ . ” represents any character except new line \n.
- To search literal dot in a string, use escape character \ before any metacharacter.
- “ \. “ means a dot, not any character.

Samples:

```
preg_match('/.the/', 'the man looking...') ;

preg_match('/.the/', 'It stops there!') ;

preg_match('/c.t/', 'There is a cat in a cotton pillow') ;

preg_match('/\.\.\./', 'There are cats, dogs, ... , and others') ;
```

Metacharacters: [ ] ^ -

- Square brackets are used to define a custom character class such as [aeiou] for vowels.
- ^ (caret) within square bracket as the first character negates the character class. – (hypen) shows a range.

Character Class	Syntax	Negation
Vowels	**[aeiouAEIOU]**	**[^aeiouAEIOU]**
Decimal Digits	**[0123456789]** or **[0-9]** or **\d**	**[^0123456789]** or **[^0-9]** or **\D**
Even Digits	**[02468]**	**[^02468]**
Letters	**[a-zA-Z]**	**[^a-zA-Z]**
Hexadecimal Digits	**[0-9a-zA-Z]**	**[^0-9a-zA-Z]**
Alphanumeric	**[a-zA-Z0-9]**	**[^a-zA-Z0-9]**
Word Characters	**[a-zA-Z0-9_]** or **\w**	**[^a-zA-Z0-9_]** or **\W**
Whitespace	**[\t\n]** or **\s**	**[^ \t\n]** or **\S**

Metacharacters: [ ] ^ -

- Square brackets are used to define a custom character class such as [aeiou] for vowels.
- ^ (caret) within square bracket as the first character negates the character class. – (hypen) shows a range.

Samples:

```
preg_match('/\d\dTR\d\d/', 'Ticket no is 34TR45678') ;

preg_match('/\d\dTR\d\d/', 'Ticket no is 34TR4K5678') ;

preg_match('/[^aeiou]\w\w/', 'cat dog apple egg') ;

preg_match('/#[0-9a-f][0-9a-f]/i', 'A parts are #d3 and #AF and #956') ;

preg_match('/\d\d\.\d\d\.\d\d\d\d/', 'My birthday is 23.12.1998, Friday') ;

preg_match('/[12][0-9]/', 'Some numbers are 45, 56, 15, 29, 78') ;
```

Metacharacters: [ ] ^ -

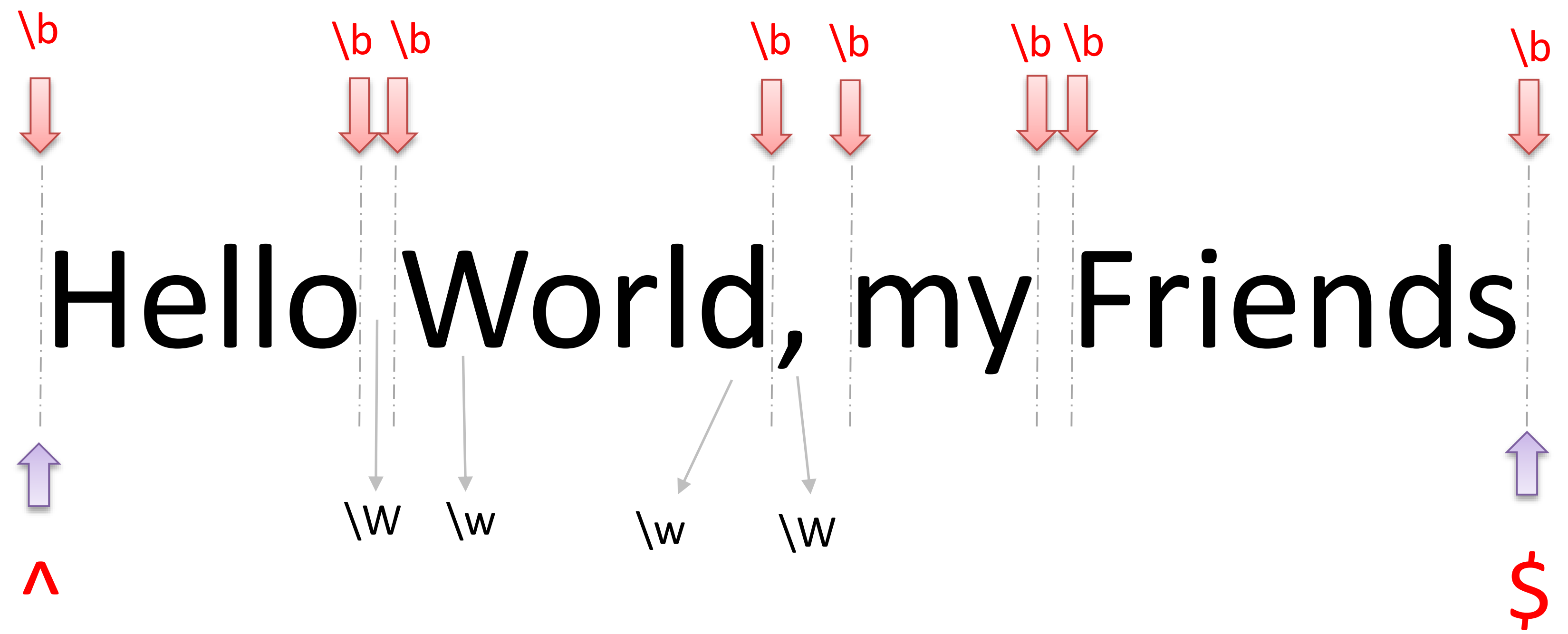
- Square brackets are used to define a custom character class such as [aeiou] for vowels.
- ^ (caret) within square bracket as the first character negates the character class. – (hypen) shows a range.

Samples:

```
preg_match('/\d\dTR\d\d/', 'Ticket no is 34TR45678') ; // 34TR45

preg_match('/\d\dTR\d\d/', 'Ticket no is 34TR4K5678') ; // No Match

preg_match('/[^aeiou]\w\w/', 'cat dog apple egg') ; // cat dog

preg_match('/#[0-9a-f][0-9a-f]/i', 'A parts are #d3 and #AF and #956') ; // #d3 #AF #95

preg_match('/\d\d\.\d\d\.\d\d\d\d/', 'My birthday is 23.12.1998, Friday') ; // 23.12.1998

preg_match('/[12][0-9]/', 'Some numbers are 45, 56, 15, 29, 78') ; // 15 29
```

Word Boundaries

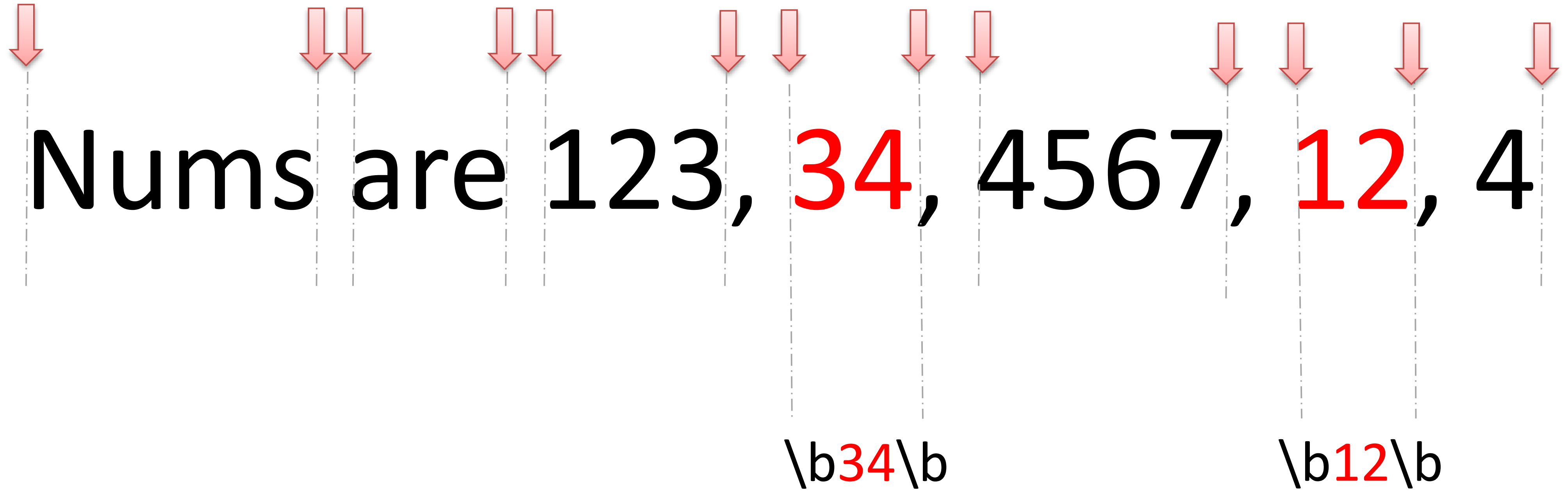
- A word boundary is a position between \w and \W, or at the beginning or end of a string.
- **\b** represents a word boundary in general, **^** is a special word boundary at the beginning, and **$** is a special word boundary at the end of the string (before \n).



Word Boundaries

```
preg_match('/\b\d\d/\b', 'Nums are 123, 34, 4567, 12, 4' ) ;
```

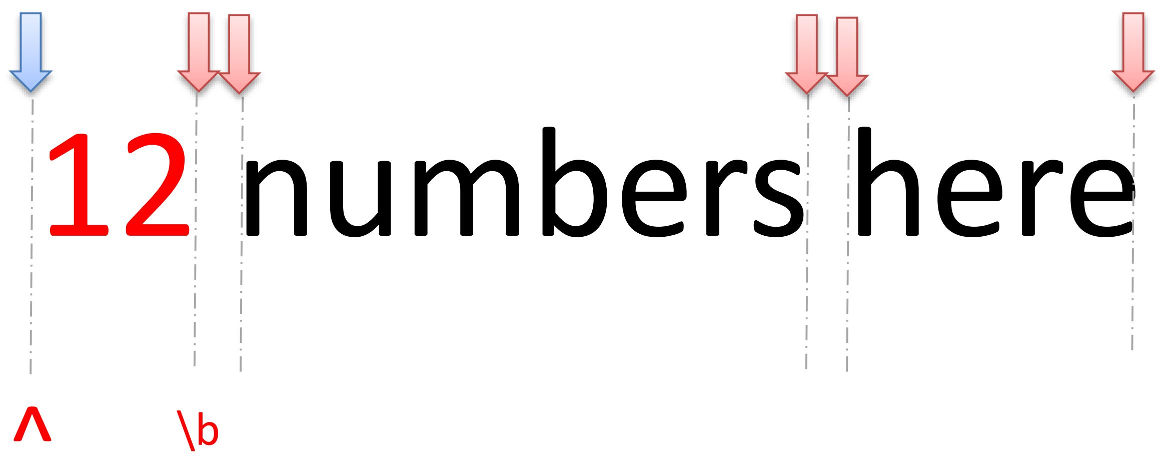
Test if there is any word matching two digits. There are 7 words in this example. Only 34 and 12 are words with two digits.



Starting With

```
preg_match('/^\d\d\b/', '12 numbers here' ) ;
```

^ : starting with (in square bracket it has a distinct meaning)

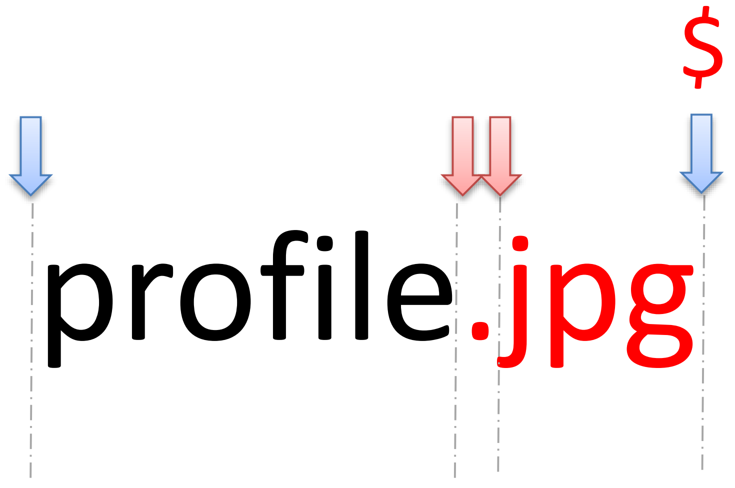


^12\b

Ending With

```
preg_match('/\.jpg$/i, "profile.jpg") ;
```

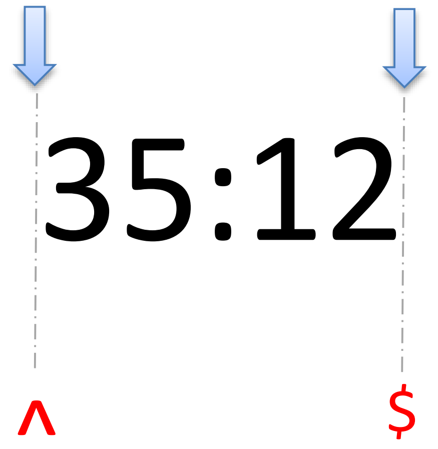
$: ending with



.jpg$

Exact Matching

```
preg_match('/^\d\d:\d\d$/', 'Time is 35:12' ) ; // No Match
preg_match('/^\d\d:\d\d$/', '35:12 today' ) ; // No match
preg_match('/^\d\d:\d\d$/', '35:12' ) ;  // Exact Matching
```

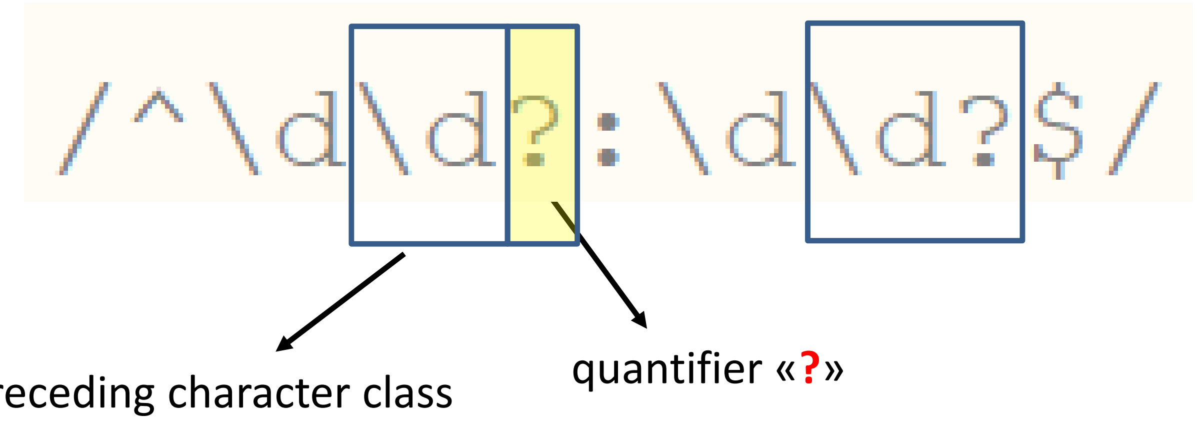


^35:12$

Quantifiers: ? ,+,*, {}

- A quantifier specifies how many instances of the preceeding character, character class or group must be present in the string pattern.
- **?** : zero or one occurrence (meaning optional)
- **+** : one or more occurrence
- ***** : zero or more occurrence
- **{n}** : exactly "n" occurrence
- **{n,}** : minimum "n" occurrence
- **{,n}** : maximum "n" occurrence
- **{m,n}** : minimum "m", maximum "n" occurrence

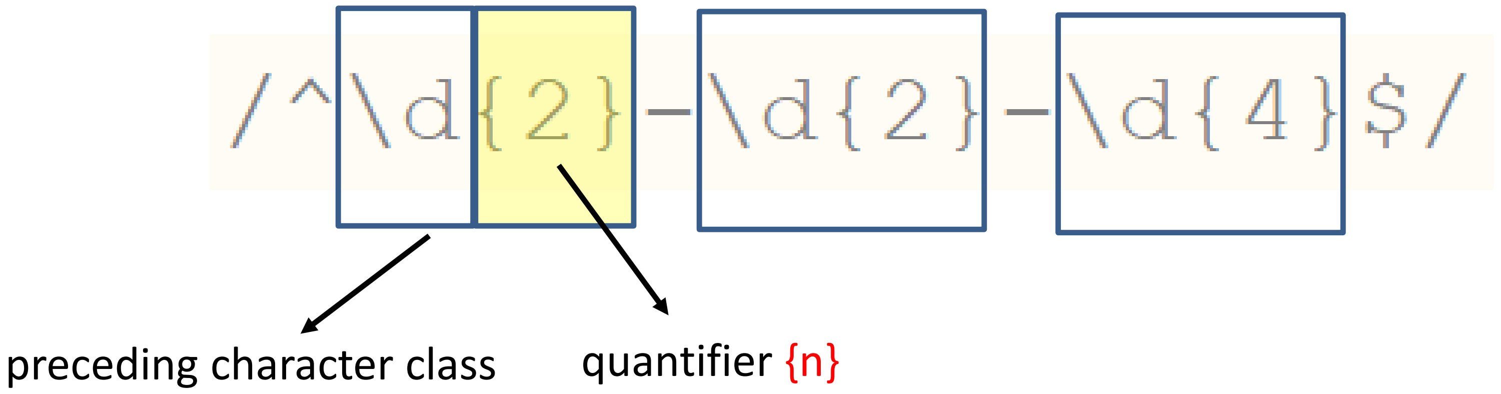
Quantifiers: ? ,+,*, {}



Samples:

```
preg_match('/^\d\d?:\d\d?$/', '13:45' ) ; // Match
preg_match('/^\d\d?:\d\d?$/', '1:45' ) ;  // Match
preg_match('/^\d\d?:\d\d?$/', '1:4' ) ;   // Match
preg_match('/^\d\d?:\d\d?$/', '13:4' ) ;  // Match
preg_match('/^\d\d?:\d\d?$/', '134:4' ) ; // No Match
```

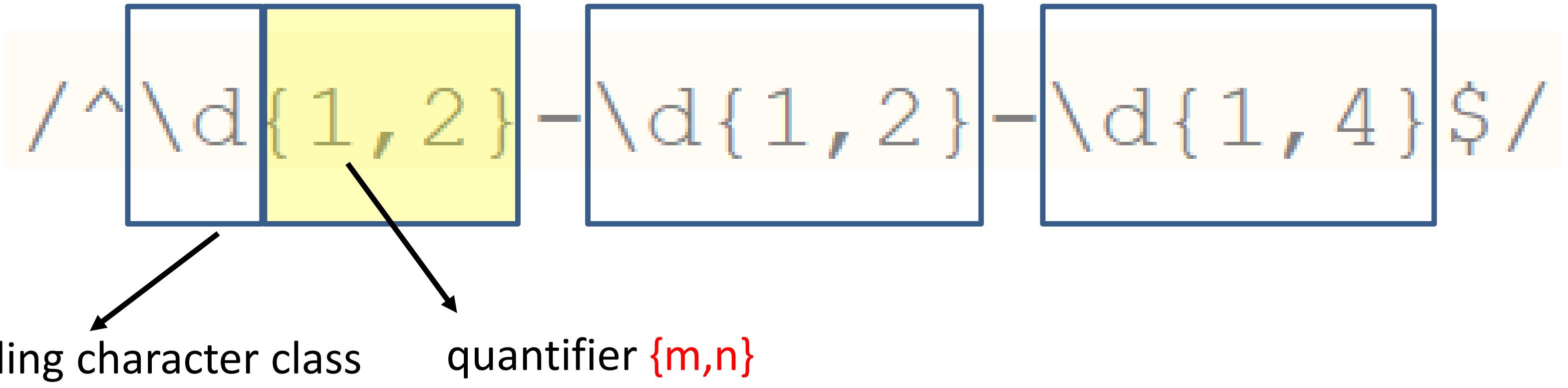
Quantifiers: ? ,+,*, {}



Samples:

```
preg_match('/^\d{2}-\d{2}-\d{4}$/', '12-09-1945') ; // Match
preg_match('/^\d{2}-\d{2}-\d{4}$/', '1-09-1945') ;  // No Match
preg_match('/^\d{2}-\d{2}-\d{4}$/', '12-09-435') ;  // No Match
```

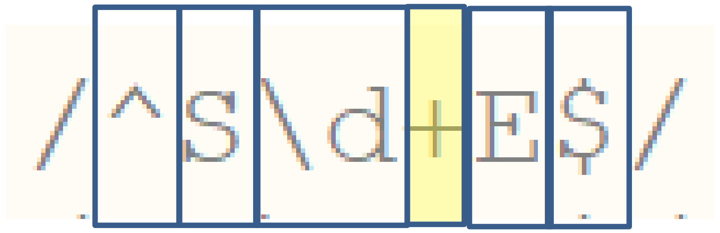
Quantifiers: ? ,+,*, {}



Samples:

```
preg_match('/^\d{1,2}-\d{1,2}-\d{1,4}$/', '12-09-1945') ; // Match
preg_match('/^\d{1,2}-\d{1,2}-\d{1,4}$/', '1-09-1945') ;  // Match
preg_match('/^\d{1,2}-\d{1,2}-\d{1,4}$/', '12-09-435') ;  // Match
preg_match('/^\d{1,2}-\d{1,2}-\d{1,4}$/', '0-0-0') ;      // Match
```

Quantifiers: ? ,+,*, {}



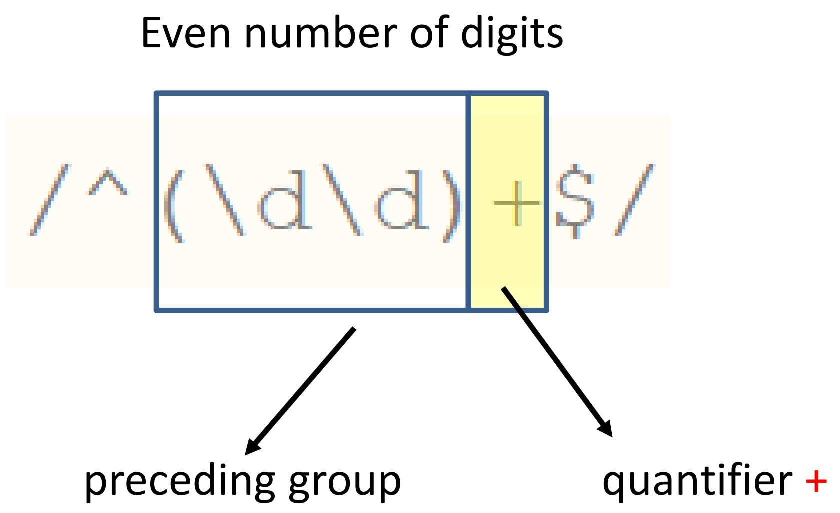
```
/^S\d+E$/
```

Samples:

```
preg_match('/^S\d+E$/', 'SE' ) ;      // No Match
preg_match('/^S\d+E$/', 'S1E' ) ;     // Match
preg_match('/^S\d+E$/', 'S12345E' ) ; // Match

preg_match('/^S\d*E$/', 'SE' ) ;      // Match
preg_match('/^S\d*E$/', 'S1E' ) ;     // Match
preg_match('/^S\d*E$/', 'S12345E' ) ; // Match
```

Grouping: ()

- Parentheses are used to group characters to apply a quantifier to the entire group or to restrict alternation to part of the regex.
- Normally, the characters within the group is saved(captured) to be used in backreference.
- If you don't need the group to capture, you can optimize the regular expression with **(?:)** instead of **( ).**



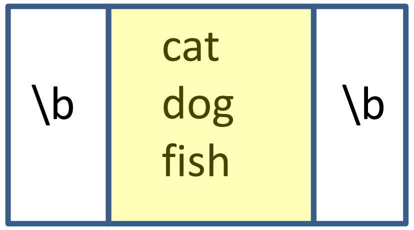
```
preg_match('/^(\d\d)+$/','1') ;    // No Match
preg_match('/^(\d\d)+$/','12') ;   // Match
preg_match('/^(\d\d)+$/','123') ;  // No Match
preg_match('/^(\d\d)+$/','1234') ; // Match
```

Vertical Bar: |

- Vertical bar is used for alternation and it works like «OR» operator.

Samples:

```
preg_match('/\b(cat|dog|fish)\b/', 'My cat staring at the fish.') ; // cat fish
```



Vertical Bar: |

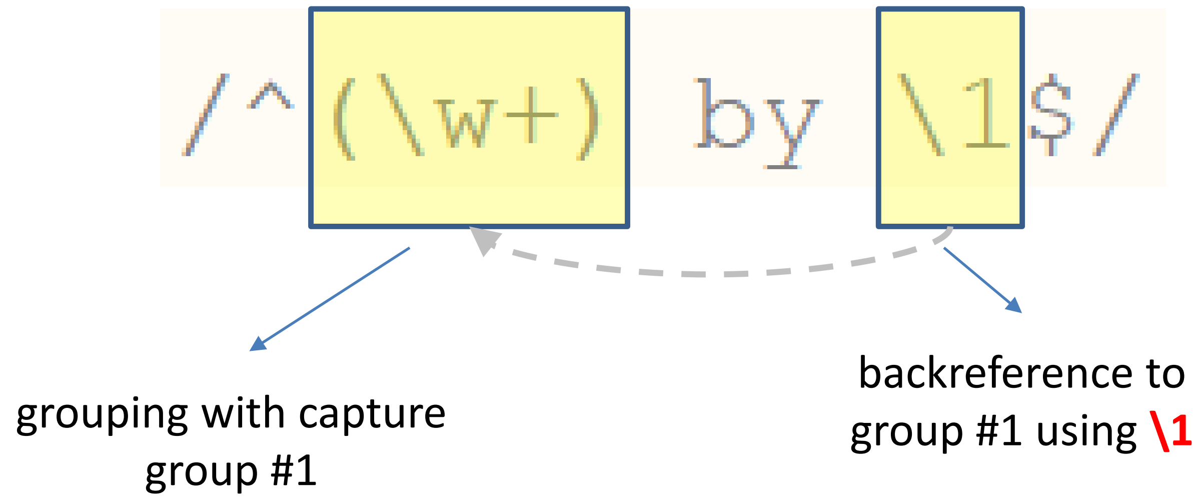
```
/^#([0-9a-z]{3}|[0-9a-f]{6})/i
```

```
/^#([0-9a-z]{3}|[0-9a-f]{6})/i
```

```
preg_match('/^#([0-9a-z]{3}|[0-9a-f]{6})/i' , '#F567E4' ) ;// Match
preg_match('/^#([0-9a-z]{3}|[0-9a-f]{6})/i' , '#F9a' ) ;   // Match
preg_match('/^#([0-9a-z]{3}|[0-9a-f]{6})/i' , '#6AF8' ) ;  // No Match
```

Backreference

Backreferences provide a convenient way to identify a repeated character or substring within a string. For example, if the input string contains multiple occurrences of an arbitrary substring, you can match the first occurrence with a capturing group with parentheses, and then use a backreference to match subsequent occurrences of the substring.

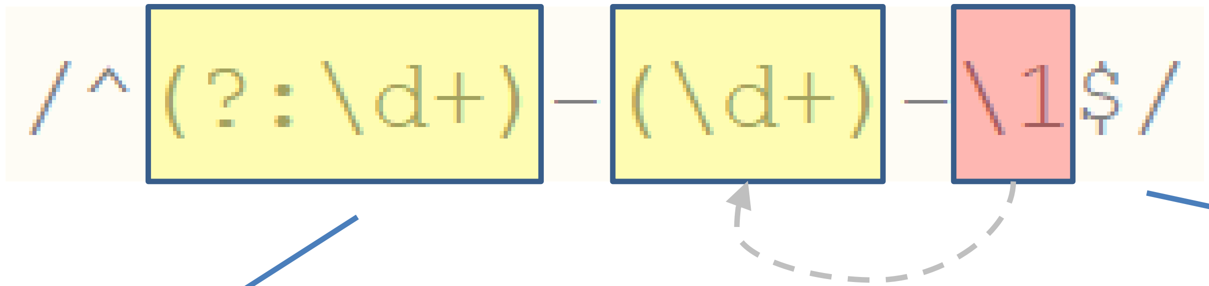

/^(\w+) by \1$/
grouping with capture group #1
backreference to group #1 using \1

Samples:

```
'step by step'   Match
'one by one'     Match
'drop by drop'   Match
'one by two'     No Match
```

Backreference

(   ) : grouping with capture
(?:) : grouping without capture (doesn't save the group content)



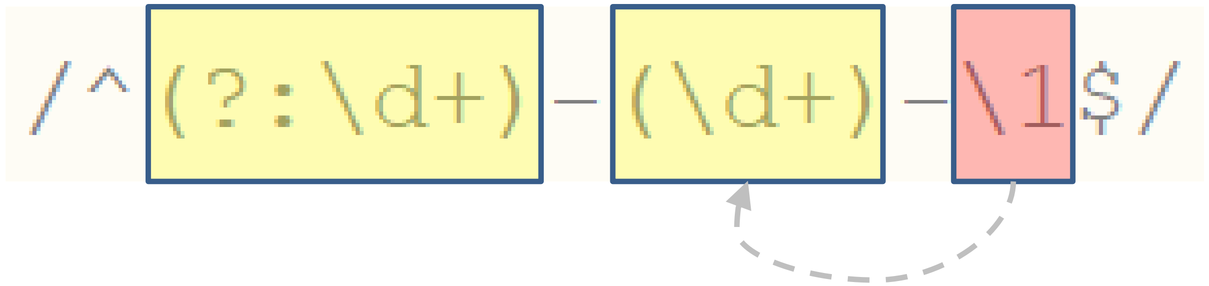
the content of the first group
does not save its content to be used
in backreference. This is the first group.

\1 refers to the first captured group, therefore, the second group.

group with capturing,
its content is saved/captured to be used
in backreference. This is the second group but
the first captured group.

Backreference

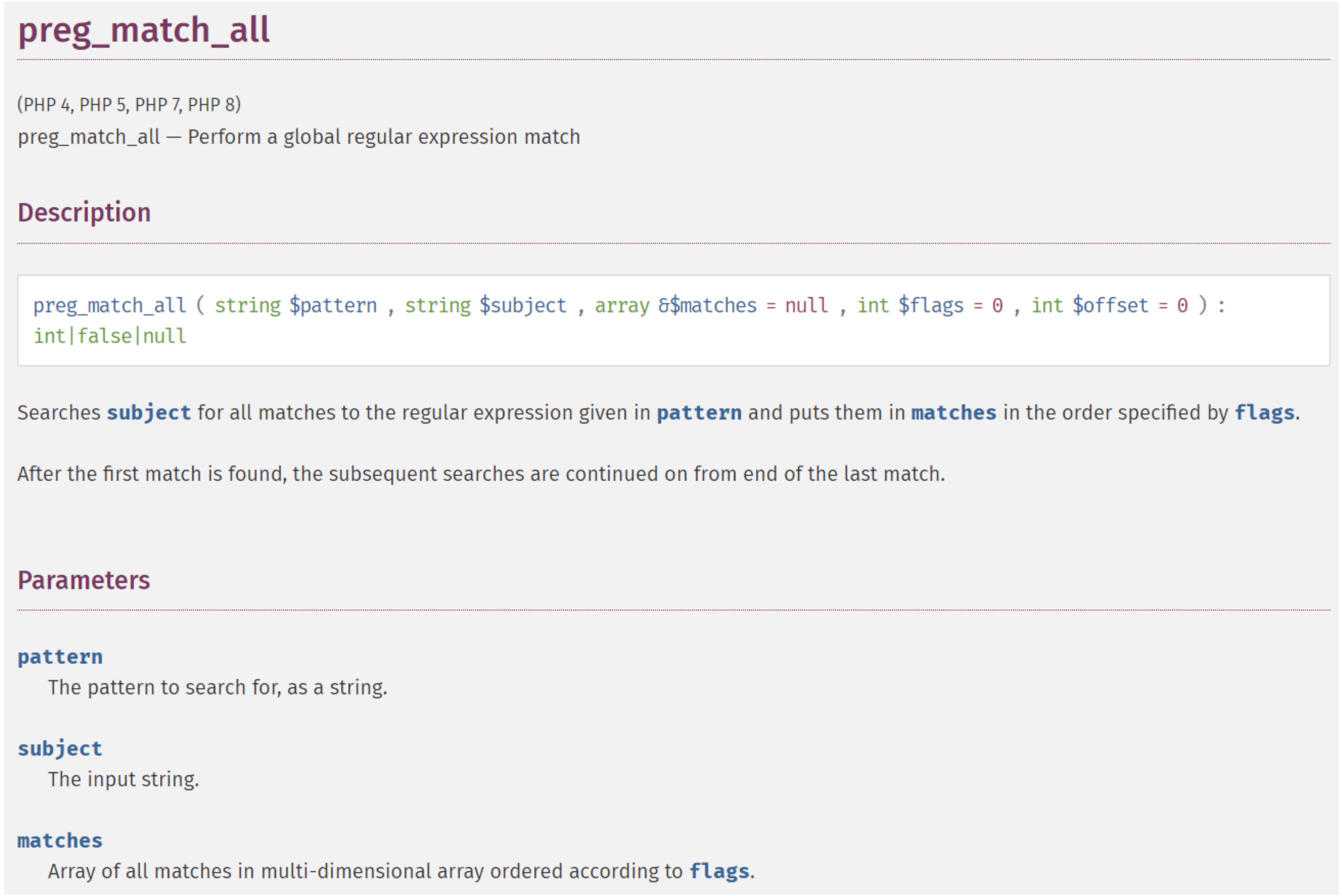
(   ) : grouping with capture
(?:) : grouping without capture (doesn't save the group content)



```
preg_match('/^(?:\d+)-(\d+)-\1$/' , '1234-567-1234' ) ; // No Match

preg_match('/^(?:\d+)-(\d+)-\1$/' , '1234-567-567' ) ;  // Match
```

the same

Match All



preg_match_all

(PHP 4, PHP 5, PHP 7, PHP 8)

preg_match_all — Perform a global regular expression match

Description

```
preg_match_all ( string $pattern , string $subject , array &$matches = null , int $flags = 0 , int $offset = 0 ) :
int|false|null
```

Searches **`subject`** for all matches to the regular expression given in **`pattern`** and puts them in **`matches`** in the order specified by **`flags`**.

After the first match is found, the subsequent searches are continued on from end of the last match.

Parameters

`pattern`
The pattern to search for, as a string.

`subject`
The input string.

`matches`
Array of all matches in multi-dimensional array ordered according to **`flags`**.

Match All

```php
preg_match_all('/\b\d{2}\b/' , 'Grades are 3, 34, 45, 120, 13', $matches) ;
// $matches[0] is an array that contains all matches.
echo "<p>Number of Matches : ", count($matches[0]) , "</p>" ;
// iterate over matched substrings
foreach( $matches[0] as $number ) {
  echo "<p>", $number, "</p>" ;
}
```

```
array (size=1)
  0 =>
    array (size=3)
      0 => string '34' (length=2)
      1 => string '45' (length=2)
      2 => string '13' (length=2)
```

Result:

```
Number of Matches : 3

34

45

13
```

Match All

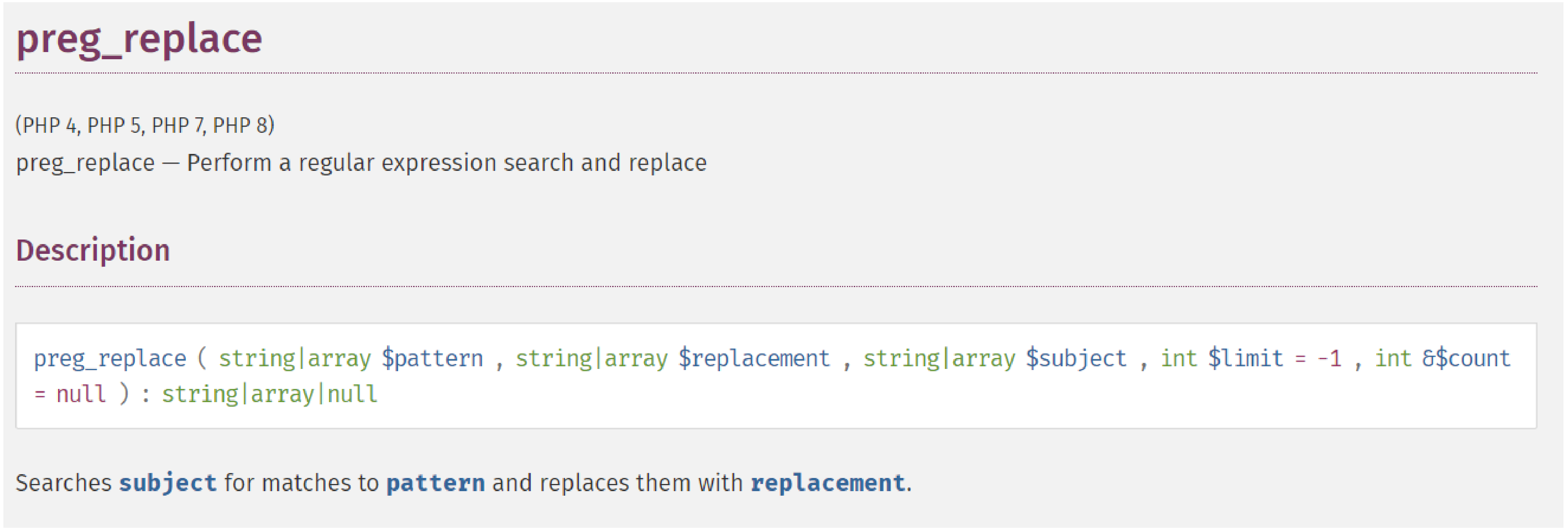
```php
$text= "
  My friends email are sgenc@bilkent.edu.tr, ali@hotmail.com
  and seckin@siemens.com.tr, info@gtech.net
" ;
preg_match_all('/\b(\w+)@(?:\w+\.){1,3}(?:com|tr)\b/i' , $text, $matches) ;

echo "<p>Number of Matches : ", count($matches[0]) , "</p>" ;

// index 0 : Full matches
foreach( $matches[0] as $number ) {
    echo "<p>", $number, "</p>" ;
}
// index 1 : Content of the first captured group
foreach( $matches[1] as $username) {
    echo "<p>", $username, "</p>" ;
}
```

```
array (size=2)
  0 =>
    array (size=3)
      0 => string 'sgenc@bilkent.edu.tr' (length=20)
      1 => string 'ali@hotmail.com' (length=15)
      2 => string 'seckin@siemens.com.tr' (length=21)
  1 =>
    array (size=3)
      0 => string 'sgenc' (length=5)
      1 => string 'ali' (length=3)
      2 => string 'seckin' (length=6)
```

Result:

```
Number of Matches : 3

sgenc@bilkent.edu.tr

ali@hotmail.com

seckin@siemens.com.tr

sgenc

ali

seckin
```

Replace



preg_replace

(PHP 4, PHP 5, PHP 7, PHP 8)

preg_replace — Perform a regular expression search and replace

Description

```
preg_replace ( string|array $pattern , string|array $replacement , string|array $subject , int $limit = -1 , int &$count
= null ) : string|array|null
```

Searches **`subject`** for matches to **`pattern`** and replaces them with **`replacement`**.

Replace Samples

1.
```php
$modified = preg_replace('/\bcan\'t\b/i', "can not", "This can't be true") ;
echo "<p>Replaced : $modified</p>" ;
```

can't $\rightarrow$ can not for all matches

Replaced : This can not be true

All whitespaces at the beginning and/or at the end will be replaced by empty string.

2.
```php
// Trimming leading and trailing whitespaces
$orginal = '    Barış Manço    ' ;
$trimmed = preg_replace('/^\s+|\s+$/', '' , $orginal) ;
echo "Orginal : '$orginal'" ;
echo "Trimmed : '$trimmed'" ;
```

empty string

```
/^\s+|\s+$/
```

leading whitespaces

trailing whitespaces

Result:

```
Orginal : '    Barış Manço    '
Trimmed : 'Barış Manço'
```

Replace Samples

«Name Lastname» becomes «Lastname, N.»
«Ali Tarık» becomes «Tarık, A.»
«Özgün Borlu» becomes «Borlu, Ö.»

3.

```
/(\w)\w+\s+(\w+)/u
```

Capture the first letter of the name (\1)

Capture the lastname (\2)

Support all languages inc. Turkish.

```
$orginal = "Özgün Çolak" ;
$transformed = preg_replace('/(\w)\w+\s+(\w+)/u', '\2, \1.', $orginal) ;

echo "<p>Orginal     : '$orginal'</p>" ;
echo "<p>Transformed : '$transformed'</p>" ;
```

"\\2, \\1." if you use double quote

Orginal : 'Özgün Çolak'

Transformed : 'Çolak, Ö.'