**정규식 RegEx**

특정 문자 검색하기

|  |  |
| --- | --- |
| 정규식 | Ben |
| 문자열 | Hello, my name is Ben. Please Ben visit ben |
| 설명 | 보통 정규식은 처음으로 match일치하는 텍스트를 반환  두개의 일치가 만들어진걸까? 대다수 정규식 표현식 구현에서는 일치하는 목록을 모두 얻을수 있다(배열등)  일치된 결과물을 모두 담을 배열을 반환하고자 할 때 **g(global:전역)** 플래그를 사용한다. |

대소문자 다루기

|  |  |
| --- | --- |
| 정규식 | Ben |
| 문자열 | Hello, my name is Ben. Please Ben visit ben |
| 설명 | 자바스크립트 사용자들은 **i 플래그**를 사용해 대소문자 구별을 무시하고 검색할수있다. |

모든 문자 찾기

팁: .는 줄바꿈문자를 제외한 모든 문자와 일치한다

|  |  |
| --- | --- |
| 정규식 | sales. |
| 문자열 | sales1.xls  sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  na1.xls  na2.xls  sa1.xls |
| 설명 | 정규표현식서는 특별한 문자들(혹은 문자 집합) 을 써 무엇을 검색할지 결정한다.  **마침표(.)** 문자는 아무 문자 하나와 일치한다  **여기서 사용한 sales. 라는 정규식은 sales로 시작하고 sales 바로뒤에 아무 문자가 하나 더붙는 파일명을 모두 찾는다. 파일 9개중에 3개가 이패턴과 일치한다.**  **마침표(.)는 어떠한 문자나 알파벳,숫자,심지어 문장 부호로 쓰인 마친표(.) 자체와도 일치한다** |

|  |  |
| --- | --- |
| 정규식 | .a. |
| 문자열 | sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  na1.xls  na2.xls  sa1.xls |
| 설명 | **마침표(.) 여러 개 동시사용.**  **예를들어 .. 처럼 연속해 사용 하면 어떤 문자든 붙어 있는 문자 두개와 일치한다 아니면 서로 다른위치에 사용할수있다.** |

|  |  |
| --- | --- |
| 정규식 | .a.. |
| 문자열 | sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  na1.xls  na2.xls  sa1.xls |
| 설명 | .a.. 도 .a.와 다르지 않다 우리는 마지막에 마침표가 붙는 글자를 일치시키고 싶었던 것인데 문자가 하나더 붙은 텍스트를 찾았다 . 어떻게해야 마침표를 의미하는 마침표(.)문자만 찾을수있을까??  아래서 확인하자 특수문자 찾기 ↓ |

특수문자 찾기

마침표(.)는 정규 표현식에서 특별한 의미가 있다. 따라서 여러분이 특수문자 마침표(.)를 찾으려한다면,

정규식 표현에게 특별한 의미의 마침표(.) 아니라 진짜 마침표(.) 를 찾고싶다고 알려줘야한다.

앞에 역슬래시(\)문자(메타문자)를 붙이면된다. 이것의 명칭은 secape 이스케이프 라고한다

|  |  |
| --- | --- |
| 정규식 | .a.\.xls |
| 문자열 | sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  na1.xls  na2.xls  sa1.xls |
| 설명 | 역슬래시(\)를 메타문자이 며  (ecape 이스케이프 라고한다)  특수문자 역슬래시(\)를 찾고싶으면 [\\를](file:///\\를) 하면된다 |

문자 집합으로 찾기

|  |  |
| --- | --- |
| 정규식 | [ns]a.\.xls |
| 문자열 | sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  na1.xls  na2.xls  sa1.xls  usa1.xls  ca1.xls |
| 설명 | 정규표현식에는 메타 문자인 대괄호([])를 사용해 문자 집합을 표현한다.  집합에 속한 문자 가운데 하나가 일치한다. 집합에 속한 문자가 모두 일치할필요는 없다. |

|  |  |
| --- | --- |
| 정규식 | [Rr]eg[Ee]x |
| 문자열 | The phrase “regular expression” is often  Abbreviated as RegEx or regex |
| 설명 | 여기서 사용한 패턴에는 문자 집합이 두개있다 [Rr]은 R이나 r 이고 [Ee]는 E나 e이다.  이런식으로 RegEx와 regex둘다 일치시킬수 있다 하지만 REGEX와는 일치하지 않는다 |

문자 집합 범위 사용하기

|  |  |
| --- | --- |
| 정규식 | [ns]a[0123456789]\.xls |
| 문자열 | sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  sam.xls  na1.xls  na2.xls  sa1.xls  ca1.xls |
| 설명 | 문자 집합[ ] 그집합 구성원중에 한문자라도 일치해야 한다(AND 가 아닌 OR이다) |

정규식 표현식을 사용할 때 여러분은 0~9 , A~Z 같이 문자들의 범위 지정하는일이 자주있을것이다.

단순하게 만들 때 정규표현식에서는 특별한 메타 문자인 하이픈(-)을 제공한다.

다음은 위와 같은 예제이지만 패턴에 범위를 지정하였다.

|  |  |
| --- | --- |
| 정규식 | [ns]a[0-9]\.xls |
| 문자열 | sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  sam.xls  na1.xls  na2.xls  sa1.xls  ca1.xls |
| 설명 | 정규표현식에서는 특별한 메타 문자인 하이픈(-)을 제공한다.  다음은 위와 같은 예제이지만 패턴에 범위를 지정하였다.  [0123456789] = [0-9] |

범위 지정 항목

1. A-Z는 A부터Z사이에있는 모든 대문자와 일치
2. a-z는 a부터z사이에있는 모든 소문자와 일치
3. A-F는 A부터F사이에 있는 대문자와 일치
4. A-z는 아스키(ASCII)문자 A와 아스키문자 z 사이에 있는 모든 문자와 일치한다. 아스키 A-z는 [와 ^캐럿 같은 문자도 포함된다.(주의)

|  |  |
| --- | --- |
| 정규식 | #[0-9A-Fa-f] [0-9A-Fa-f] [0-9A-Fa-f] [0-9A-Fa-f] [0-9A-Fa-f] [0-9A-Fa-f] |
| 문자열 | <BODY BGCOLOR=”#336633” TEXT=”#FFFFFF”  MARGINWIDTH=”0” MARGINHEIGHT=”0” TOPMARGIN=”0”  LEFTMARGIN=”0”> |
| 설명 |  |

제외하고 찾기 ^

|  |  |
| --- | --- |
| 정규식 | [ns]a[^0-9]\.xls |
| 문자열 | sales1.xls  orders3.xls  sales2.xls  sales3.xls  apac1.xls  europe2.xls  sam.xls  na1.xls  na2.xls  sa1.xls  ca1.xls |
| 설명 | 모두 나열하기보다는 캐럿(^) 문자를 써 제외할 문자 집합을 지정한다  [^0-9] 는 모든 숫자를 제외하는 것을 찾는다. |

문자클래스

|  |  |
| --- | --- |
| 문자클래스 | 기능 |
| . | 하나의 문자를 지칭한다 |
| [ | 집합의 시작 |
| ] | 집합의 끝 |
| ^ | 뒤에문자 제외 |
| { | 구간 시작 |
| } | 구간 끝 |
| + | 하나이상 |
| \* | 없거나 하나이상 |
| ? | 없거나 특정 문자만 ?앞에 a? |

숫자메타문자

|  |  |
| --- | --- |
| 메타문자 | 기능 |
| \d | 숫자 하나([0-9] 와같다) |
| \D | 숫자를 제외한 문자 하나(^[0-9] 와같다); |
|  |  |
|  |  |
|  |  |
|  |  |

영숫자 메타문자

|  |  |
| --- | --- |
| 메타문자 | 기능 |
| \w | 대소문자와 밑줄을 포함하는 모든 영문자 ( [a-zA-Z0-9\_] 와같다 ) |
| \W | 대소문자와 밑줄을 포함하지않는 모든 영문자 ( [^a-zA-Z0-9\_] 와같다 ) |
|  |  |
|  |  |
|  |  |
|  |  |

공백 메타문자

|  |  |
| --- | --- |
| 메타문자 | 기능 |
| \s | 모든 공백 문자( [\f\n\r\t\v] 와같다) \b는 포함되지않는다 |
| \S | 모든 공백 문자제외문자( [^\f\n\r\t\v] 와같다) \b는 포함되지않는다 |
|  |  |
|  |  |
|  |  |
|  |  |

메타문자

|  |  |
| --- | --- |
| 메타문자 | 기능 |
| [\b] | 역스페이스 |
| \f | 페이지 넘김(form feed) |
| \n | 줄바꿈 |
| \r | 캐리지 리턴 |
| \t | 탭 |
| \v | 수직탭 |

메타문자 사용하기

|  |  |
| --- | --- |
| 정규식 | myArray[0] |
| 문자열 | Var myArray = new Array();  …  If (myArray[0] ==0){  …  } |
| 설명 | 위에서 사용한 [0]은 정규식에서의 집합을 의미한다  [ ] 문자는 정규식에서 집합을 정의하는 문자이기 때문이다.  myArray0 을 찾으면 찾을것이다. |

|  |  |
| --- | --- |
| 정규식 | myArray\[0\] |
| 문자열 | Var myArray = new Array();  …  If (myArray[0] ==0){  …  } |
| 설명 | 이스케이프 처리하여 [문자를 정말 문자로 보았다.  하지만 myArray[10] 은 찾지못한다 당연히 하나의 문자만 숫자로했으니. |

|  |  |
| --- | --- |
| 정규식 | myArray\[\d\] |
| 문자열 | Var myArray = new Array();  …  If (myArray[0] ==0){  …  } |
| 설명 | 이스케이프 처리하여 [문자를 정말 문자로 보았다.  하지만 myArray[10] 은 찾지못한다 당연히 하나의 문자만 숫자로했으니.  숫자메타문자를 썼다 |

|  |  |
| --- | --- |
| 정규식 | \\ |
| 문자열 | \home\ben\sales\ |
| 설명 | 이스케이프 처리 |

공백 및 엔터값처리

|  |  |
| --- | --- |
| 정규식 | \r\n\r\n |
| 문자열 | “101”,”Ben”,”Forta”  “102”,”Jim”,”James”  “103”,”Roberta”,”Robertson”  “104”,”Bob”,”Bobson” |
| 설명 | 엔터 두번 뛰어져있는 부분이 선택된다  \r\n은 줄바꿈과 캐리지 리턴의 조합과 일치한다 따라서 \r\n\r\n을 검색하면 줄끝이 연속해서 두번나오는 부분을 찾게된다 즉 레코드상에 빈공간을 찾는것이다 |

|  |  |
| --- | --- |
| 정규식 | \w\d\w\d\w\d |
| 문자열 | 11213  A1C2E3  48075  48237  M1B4F2  90046  H1H2H2 |
| 설명 | 여기서 \w와 \d를 조합한 패턴사용했는데 캐나다 우편번호만을 찾아냈다 |

16진수나 8진수 표현하기

|  |  |
| --- | --- |
| 16진수값 | 앞에 \x를 붙여 표시한다. 즉 \x0A(아스키문자10)은 줄바꿈 문자가되며 \n과 기능이같다. |
| 8 진수값 | 두자리나 세자리 정수 앞에 \0을 붙여 표시한다 즉 \011(아스키 문자9)은 탭문자이며, \t 와 기능이같다 |

컨트롤 문자

예를들어 \cZ는 Ctrl-Z와 일치한다. 실제로는 거의 안쓰인다.

포직스 문자 클래스 사용하기

메타문자와 여러문자 집합을 줄여 쓰는 방법에 대한 설명에는 포직스 POSIX

일부분 지원을안하는 엔진도있다. 자바스크립트는 지원안함.

포직스 문자 클래스

|  |  |
| --- | --- |
| 메타문자 | 기능 |
| [:alnum:] | 모든영숫자([a-zA-Z0-9] 와같다) |
| [:alpha:] | 모든 영문자 [a-zA-Z]와 같다. |
| [:blank:] | 빈칸(space)이나 탭문자 ([\t ]와같다) |
| [:cntrl:] | 아스키 제어문자(아스키 0번부터 31, 127번) |
| [:digit:] | 모든 한자리 숫자[0-9] 와같다 |
| [:graph:] | [:print:] 와 동일하나 빈칸(space)은 제외 |
| [:lower:] | 모든 소문자([a-z]와같다); |
| [:print:] | 출력 가능한 모든문자 |
| [:punct:] | [:alnum:]이나[:cntrl:] 가 포함 되지않는 모든문자 |
| [:space:] | 빈칸을 포함한 모든 공백 문자 [\f\n\r\t\v] 와같다 |
| [:upper:] | 모든 대문자[A-Z]와같다 |
| [:xdigit:] | 모든 16진수 숫자 [a-fa-F0-9]와같다 |

|  |  |
| --- | --- |
| 정규식 | #[[:xdigit:]] [[:xdigit:]] [[:xdigit:]] [[:xdigit:]] [[:xdigit:]] [[:xdigit:]] |
| 문자열 | <BODY BGCOLOR=”#336633” TEXT=”#FFFFFF”  MARGINWIDTH=”0” MARGINHEIGHT=”0” TOPMARGIN=”0”  LEFTMARGIN=”0”> |
| 설명 | [0-9A-Fa-f] 를 여섯 번 반복. |

반복찾기

|  |  |
| --- | --- |
| 정규식 | \w@\w\.\w |
| 문자열 | text@w.text |
| 설명 | \w 문자는 모든 영숫자 문자와 일치, 더불어 밑줄도 함께 찾는데.. 이메일을 찾을수 있다  하지만!!!!!!!!!! 앞뒤로 한글자씩밖에 들어는것만 가능하다 그러무로 쓸모없는.. |

문자 하나이상 찾기

|  |  |
| --- | --- |
| + | 문자 뒤에 더하기 (+) 문자를 붙이면된다  하나이상일 때 일치한다(최소한 하나와 일치하고, 없을 때는 일치하지 않는다. |
|  |  |
|  |  |
|  |  |
| [0-9+] 가 정확한게 아니라 [0-9]+ 가 정확한거다 | |

|  |  |
| --- | --- |
| 정규식 | \w+@\w+\.\w+ |
| 문자열 | Send personal email to [ben@forta.com](mailto:ben@forta.com). For questions  About a book use [support@forta.com](mailto:support@forta.com). Feel free to send  Unsolicited email to [spam@forta.com](mailto:spam@forta.com) (wouldn’t it be  Nice if it were that simple, huh?). |
| 설명 | 더하기(+) 메타문자 : 문자 그대로 더하기를 찾으려면 \+로 찾으면된다 |

|  |  |
| --- | --- |
| 정규식 | [\w.]+@ [\w.]+\.\w+ |
| 문자열 | Send personal email to [ben@forta.com](mailto:ben@forta.com). For questions  About a book use good.[support@forta.com](mailto:support@forta.com). Feel free to send  Unsolicited email to [spam@forta.com](mailto:spam@forta.com) (wouldn’t it be  Nice if it were that simple, huh?). |
| 설명 |  |

문자가 없는 경우나 하나 이상 연속하는 문자찾기

문자 없을수도 있고 있을수도있는.. 찾기

|  |  |
| --- | --- |
| \* | 메타 문자인 별표(\*)를 사용하면된다  집합이 없는경우 또는 하나 이상 연속하는 경우 일치한다. |

|  |  |
| --- | --- |
| 정규식 | [\w.]+@ [\w.]+\.\w+ |
| 문자열 | Hello [.ben@forta.com](mailto:.ben@forta.com) is my email address |
| 설명 | 제대로 검색은되지만 보통은 이메일 맨처음에 . 를 쓰지를 않는다는점이다. |

|  |  |
| --- | --- |
| 정규식 | \w+[\w.]\*@ [\w.]+\.\w+ |
| 문자열 | Hello [.ben@forta.com](mailto:.ben@forta.com) is my email address |
| 설명 | 이것이 찾고자 하는 텍스트다. |

문자가 없거나 하나인 문자 찾기

|  |  |
| --- | --- |
| ? | 물음표(?) 는 별표(\*)처럼 물음표(?)는 문자가 있는 경우 일치하고 문자가 없어도 일치하지만 별표(\*)와 달리 문자나 집합이 없거나 하나만 있는 경우 일치하며 하나이상은 일치하지 않는다. 즉, 물음표(?)는 문자 묶음 안에서 있는지 없는지 확실하지 않는 특정한 문자를 하나만 찾을 때 유용하다  **없거나 존재한다면 ? 앞에 있는 문자가 나와야한다**  https?:// 는 ? 는 자기문자 앞에 문자가 없거나 그문자가 하나만 있는 경우 일치한다.!! 여기서는 s다 |

|  |  |
| --- | --- |
| 정규식 | http://[\w./]+ |
| 문자열 | The URL is <http://www.forta.com/> , to connect  Securely use <https://www.forta.com/> instead. |
| 설명 |  |

|  |  |
| --- | --- |
| 정규식 | https?://[\w./]+ |
| 문자열 | The URL is <http://www.forta.com/> , to connect  Securely use <https://www.forta.com/> instead. |
| 설명 | https?:// 는 ? 는 자기문자 앞에 문자가 없거나 그문자가 하나만 있는 경우 일치한다.!! 여기서는 s다  https?://는 <http://나> <https://와> 일치하지만 그외에는 일치하지 않는다. |

공백 및 엔터값처리

|  |  |
| --- | --- |
| 정규식 | [\r]?\n[\r]?\n |
| 문자열 | “101”,”Ben”,”Forta”  “102”,”Jim”,”James”  “103”,”Roberta”,”Robertson”  “104”,”Bob”,”Bobson” |
| 설명 | [\r]?\n은 \r 이 있을경우에는 \r과 일치하고 \n과 반드시 일치한다. |

구간 지정하기 {}

기존 반복찾기로 찾기 힘든것들

1. 더하기(+)와 별표(\*)는 일치하는 문자 수에 제한이 없다. 문자가 최대 몇 개까지 일치하는지 정할수 없다.
2. 더하기(+),별표(\*),물음표(?)가 일치하는 문자 수의 최소값은 0이나 1이다 일치하는 문자수의 최소값을 명시적으로 정의할수 없다
3. 정확히 원하는 만큼만 일치하도록 문자 수를 정의할수 없다.

위문제를 해결하기위해 구간을 사용한다 구간은 중괄호 {}안에 표시한다.

정확한 구간찾기

|  |  |
| --- | --- |
| {숫자} | 구간 설정  {3}은 바로 앞에 있는 문자나 문자 집합이 세번 연속해서 일치하는지 확인한다, 만약 요소가 두개만 잇다면 패턴이 일치하지 않는다. |
| {숫자,숫자} | 구간설정 최소 최대값 정할수있다  {3,10}은 바로 앞에 있는 문자나 문자 집합이 최소3번부터 10번까지 일치시킨다는거다.  구간은 0부터 시작하기도한다. {0,3} 요소가 없는경우나 최대3번 일치함을 의미한다 |
| {숫자,} | 최소 구간찾기  {3,} 은 최소한 요소가 세번 일치함을 의미한다 최대값은 주어지지 않아서 최대는 없다. 무한  더하기(+)는 {1,}와 같다. |

|  |  |
| --- | --- |
| 정규식 | #[[:xdigit:]]{6} |
| 문자열 | <BODY BGCOLOR=”#336633” TEXT=”#FFFFFF”  MARGINWIDTH=”0” MARGINHEIGHT=”0” TOPMARGIN=”0”  LEFTMARGIN=”0”> |
| 설명 | [0-9A-Fa-f] 를 여섯 번 반복.  구간 설정 |

|  |  |
| --- | --- |
| 정규식 | \d{1,2}[-\/]\d{1,2}[-\/]\d{2,4} |
| 문자열 | 4/8/03  10-6-2004  2/2/2  01-01-01 |
| 설명 | 이건 54/67/9999 처럼 값이 틀려도 검사를 통과한다 따라서 사용할수없는것이다. |

|  |  |
| --- | --- |
| 정규식 | \d+: \$\d{3,}\.\d{2} |
| 문자열 | 1001: $496.80  1002: $1496.80  1003: $46.80  1004: $613.80  1005: $6.80  1006: $414.80  1007: $25.80 |
| 설명 | 이건 54/67/9999 처럼 값이 틀려도 검사를 통과한다 따라서 사용할수없는것이다. |

과하게 일치하는 상황 방지하기

범위선택

|  |  |
| --- | --- |
| .\* | 처음매칭된거와 마지막 매칭된거의 사이를 모두 선택한다. |

|  |  |
| --- | --- |
| 정규식 | <[Bb]>.\*</[Bb]> |
| 문자열 | This offer is not available to customers  Living in <B>AK</B> and <B>HI</B> |
| 설명 | .\*는 처음나온 <B> 부터 마지막에 나온 </B> 사이에 있는 모든 텍스트와 일치하기 때문에, AK</B> and <B>HI가 일치한 것이다. 우리가 원하는 텍스트를 포함하긴하지만 찾으려 하지 않는 텍스트도 포함했다. |

바로 별표(\*) 와 더하기(+)같은 메타 문자가 탐욕적(greedy)이기때문인데 이는 가능한 한가장 큰 덩어리를 찾으려 한다는 뜻이다.

이런 메타 문자는 찾으려는 텍스트를 앞에서부터 찾는게 아니라, 텍스트 마지막에서 시작해 거꾸로 찾는다. 의도적으로 수량자(quantifier)를 탐욕적으로 설계했기 때문이다. 하지만 만약 우리가 탐욕적 일치를 원하지 않는다면 어떻게 해야할까?

탐욕적 수량자를 게으른(lazy)수량자로 바꿔 이문제를 해결한다 ‘게으른’ 이라고 부르는 이유는 문자가 최소 일치하기 때문이다 게으른 수량자는 기존 수량자 뒤에 물음표(?)를 붙여서 표현한다.

|  |  |
| --- | --- |
| 탐욕적 수량자 | 게으른 수량자 |
| \* | \*? |
| + | +? |
| {n,} | {n,}? |

|  |  |
| --- | --- |
| 정규식 | <[Bb]>.\*?</[Bb]> |
| 문자열 | This offer is not available to customers  Living in <B>AK</B> and <B>HI</B> |
| 설명 | .\*? 게으른 수량자이다 먼저 AK만 일치시켰고 뒤있어 <B>HI</B>를 찾아 두부분을 따로 일치시켰다 |

정리

정규 표현 패턴이 지닌 진짜 능력은 반복 찾기를 할 때 뚜렸하게 나타난다 이장에서는 더하기(+) 하나이상일치

와 별표(\*) 없거나 하나이상있는경우 일치 , 물음표(?) 없거나 하나인경우 일치 로 반복 찾기를 수행하는 법을 소개했다 검색조건을 더 구체적으로 지정하고자 구간으로 정확한 반복 횟수나 최소값 최대값을 정한다 방금말한 수량자들은 탐욕적이여서 너무 넓은 범위와 일치할수있다 이를 방지하고자 게으른 수량자를 사용한다

위치찾기(position matching)

|  |  |
| --- | --- |
| 정규식 | cat |
| 문자열 | The cat scattered his food all over the room. |
| 설명 | cat 패턴은 cat이 있는 부분과 모두 일치한다.cat을 dog로 취환할려고 하는데 scattered 에있는 cat이 검색이 된다면 말도 안되는 소리다. |

단어 경계 지정하기

|  |  |
| --- | --- |
| \b | 단어경계다 단어 경계라는 이름에서 유추할수있듯 \b는 단어의 시작이나 마지막을 일치시킬 때 사용한다. |

|  |  |
| --- | --- |
| 정규식 | \bcat\b |
| 문자열 | The cat scattered his food all over the room. |
| 설명 | 그럼 정확히 \b는 무엇과 일치하는것일까? 정규표현식 엔진은 영어를 비롯해 어떤언어도 이해하지 못한다 따라서 무엇이 단어인지도 알지못한다 \b 는 일반적으로 단어의 일부로 사용하는 문자(영숫자 문자, 밑줄,\w와 일치하는 문자) |

특정한 문자 형태와 일치시키기

## 정규 표현식 - 전후방 탐색(lookaround)

<http://sweeper.egloos.com/3064808>

**1. 전후방 탐색의 필요성**

전후방 탐색이 필요한 예제부터 살펴보자.

여기서는 웹 페이지에서 제목을 추출해야 한다고 가정한다.

HTML 페이지 제목은 HTML 코드에서 <HEAD> 구역 내의 <TITLE>과 </TITLE> 태그 사이에 있다.

**[예문]**

<HEAD>

<TITLE>SooKkaRak's homepage</TITLE>

</HEAD>

**[정규 표현식]**

\<([Tt][Ii][Tt][Ll][Ee])\>.\*?\<\/\1\>

**[결과]**

<HEAD>

**<TITLE>SooKkaRak's homepage</TITLE>**

</HEAD>

[Tt][Ii][Tt][Ll][Ee]를 두번 쓰기 귀찮아서 하위 표현식으로 묶고, 종료 태그에서는 역참조하였다.

무튼 이러한 방식으로는 딱 정확하게 "SooKkaRak's homepage"만 찾을 수가 없다.

이러한 경우 정확하게 일치하는 지점을 지정할 수 있는 전후방 탐색이 필요하다.

**2. 전방 탐색 - 앞으로 찾기**

**전방 탐색**은 실제로는 **하위 표현식**이며, 하위 표현식과 같은 형식으로 작성한다.

전방 탐색 문법은 **?= 로 시작**하고, **등호 다음에 일치할 텍스트가 오는 하위 표현식**이다.

즉, **(?=**일치할 텍스트**)**의 형식인 것이다.

그리고, **전후방 탐색**은 **일치 영역을 발견해도 그 값을 반환하지 않으며, 그저 위치에 대한 힌트만 제공**한다.

**전방 탐색**은 **일치 영역 앞까지**, **후방 탐색**은 **일치 영역 뒤부터** 일치시키라고 힌트를 준다**.**

본문에는 URL 목록이 있고, 각 URL에서 프로토콜만 정확하게 찾아내는 예제를 살펴보자.

**[예문]**

http://www.sweeper.com

https://www.sweeper.com/secure

ftp://file.sweeper.com

**[정규 표현식]**

(?m)^\s\*[A-Za-z]{3,}(?=:)

**[결과]**

**http**://www.sweeper.com

**https**://www.sweeper.com/secure

**ftp**://file.sweeper.com

위에서 사용했던 정규 표현식을 하나씩 까보면...

* 다중행 지원을 위해 (?m)
* 프로토콜 앞에 있을지 모르는 공백에 대한 고려를 위해 ^\s\*
* 영문자가 적어도 최소 세 글자는 있어야 하기에 [A-Za-z]{3,}
* 그리고, :(콜론) 앞으로 찾기 위해 (?=:)

위 예제에서 (?=:)를 사용하였다.

이는 정규 표현식 엔진에게 **:(콜론)을 찾되 :(콜론) 앞까지만 일치시키라고 지시**한다.

전방 탐색에 대한 이해를 더 깊게 하고자, 위 예제에서 전방탐색 메타 문자를 제거해 보자.

**[정규 표현식]**

(?m)^\s\*[A-Za-z]{3,}(:)

**[결과]**

**http:**//www.sweeper.com

**https:**//www.sweeper.com/secure

**ftp:**//file.sweeper.com

위 두 예제에서 처음 정규 표현식은 :(콜론)의 앞 위치를 찾고자 (?=:)를 사용했고 두 번째 정규표현식은 (:)를 사용했다.

두 정규 표현식 모두 다음에 나오는 :(콜론)을 찾았지만, 일치한 :(콜론)을 검색 결과에 포함시키는지 아닌지가 다르다.

**전방 탐색**을 이용하면 정규 표현 분석기는 **:(콜론)과 일치하는 지점 앞까지 분석을 진행**하지만,

**:(콜론) 자체를 찾은 결과로 처리하지는 않는다**.

모든 **하위 표현식 앞쪽에 단순히 ?= 를 붙이면, 전방 탐색 표현으로 바꿀 수 있다**.

**하나의 정규 표현식 안에서 여러 개의 전방 탐색 표현식을 사용**할 수 있고,

또한 그 전방 탐색 표현식은 **정규 표현식 안에서 어느 위치에서든 사용**할 수 있다.

**3. 후방 탐색 - 뒤로 찾기**

후방 탐색은 전방 탐색과 모든 성질이 동일하다 할 수 있다.

다만, 이름의 차이에서 알 수 있듯이 **전방 탐색**이 **일치하는 앞쪽까지**의 의미였다면,

**후방 탐색**은 **일치한 뒤쪽부터**의 의미를 가진다는 차이가 있다.

**후방 탐색**은 **(?<=**일치할 텍스트**)**의 형식으로 사용한다.

전방 탐색에 비해 ?와 = 사이에 **<** 부등호가 하나 더 붙어 있는 셈이다.

다음 예제는 데이터베이스의 제품 목록에서 가격만 추려내는 경우다.

**[예문]**

ABC01: $23.45

HGG42: $5.31

CFMX1: $899.00

XTC99: $69.96

**[정규 표현식]**

\$[\d\.]+

**[결과]**

ABC01: **$23.45**

HGG42: **$5.31**

CFMX1: **$899.00**

XTC99: **$69.96**

잘 동작하지만, $(달러) 기호를 빼야 한다면, 어떻게 해야 할까?

그냥 \$를 정규 표현식에서 빼면 될까?

**[정규 표현식]**

[\d\.]+

**[결과]**

ABC**01**: **$23.45**

HGG**42**: **$5.31**

CFMX**1**: **$899.00**

XTC**99**: **$69.96**

위 결과와 같이 [숫자, .(마침표)]가 하나 이상인 것들이 모조리 일치해버려, 전혀 엉뚱한 결과가 나와 버렸다.

이럴 때 후방 탐색을 사용하여, $(달러)가 일치한 뒤부터 일치시키게 되면 원하는 결과를 얻을 수 있다.

**[정규 표현식]**

(?<=\$)[\d\.]+

**[결과]**

ABC01: $**23.45**

HGG42: $**5.31**

CFMX1: $**899.00**

XTC99: $**69.96**

제대로 동작한다. (?<=\$)는 **$(달러) 기호와 일치하지만, 이를 결과에 포함시키지 않고, 위치에 대한 힌트만 제공**하였다.

첫번째와 마지막 예제를 비교해 보자.

\$[\d\.]+는 $(달러)와 금액이 함께 일치하였다.

(?<=\$)[\d\.]+는 $(달러)가 일치한 뒤부터인 금액 숫자 부분만 일치하였다.

**4. 전후방 탐색 모두 사용하기**

이제 전후방 탐색을 모두 살펴보았으니,

처음에 살펴보았던 <TITLE>과 </TITLE>사이에 있는 제목만 뽑아내는 예제를 전후방 탐색을 사용하여 완성시켜 보자.

**[예문]**

<HEAD>

<TITLE>SooKkaRak's homepage</TITLE>

</HEAD>

**[정규 표현식]**

(?<=\<([Tt][Ii][Tt][Ll][Ee])\>).\*?(?=\<\/\1\>)

**[결과]**

<HEAD>

<TITLE>**SooKkaRak's homepage**</TITLE>

</HEAD>

<TITLE> 뒤에서부터 제목이 시작하니, 앞에서는 후방 탐색을 사용하였고,

</TITLE> 앞까지 제목이 끝나니, 뒤에서는 전방 탐색을 사용하였다.

**5. 부정형 전후방 탐색**

지금까지 살펴봤듯이 **전후방 탐색**은 반환할 텍스트의 위치

즉, **찾고자 하는 부분의 앞뒤를 특별히 지정하고 싶을 때 주로 사용**한다.

이런 방법들을 긍정형 전후방 탐색이라고 한다.

긍정이라는 말을 쓰는 이유는 실제로 일치하는 텍스트를 찾기 때문이다.

전후방 탐색 중에서 부정형 탐색은 비교적 덜 쓰는 방법이다.

**부정형 전방 탐색**은 **앞쪽에서 지정한 패턴과 일치하지 않는 텍스트를 찾고**,

**부정형 후방 탐색**도 이와 비슷하게, **뒤쪽에서 지정한 패턴과 일치하지 않는 텍스트를 찾는다**.

말 그대로 부정형 전후방 탐색은 (긍정형) 전후방 탐색의 정반대 역할을 한다.

부정형 전후방 탐색은 다음과 같이 사용한다.

* **부정형 전방 탐색** **(?!)** <-> **(?=)** **긍정형 전방 탐색**
* **부정형 후방 탐색** **(?<!)** <-> **(?<=)** **긍정형 후방 탐색**

즉, 긍정형 전후방 탐색의 = 대신 !를 사용하면 된다.

다음은 긍정형 후방 탐색과 부정형 후방 탐색 사이에 어던 차이가 있는지 알아보는 예제이다.

예제 본문에 가격과 수량을 나타내는 숫자들이 있다.

우선 간단하게 가격만 얻어보자.

**[예문]**

I paid $30 for 100 apples, 50 oranges, and 60 pears.

I saved $5 on this order.

**[정규 표현식]**

(?<=\$)\d+

**[결과]**

I paid $**30** for 100 apples, 50 oranges, and 60 pears.

I saved $**5** on this order.

이번에는 가격이 아니라, 수량을 찾아보자.

**[정규 표현식]**

(?<!\$)\d+

**[결과]**

I paid $3**0** for **100** apples, **50** oranges, and **60** pears.

I saved $5 on this order.

얼레? $30의 0 역시 $ 다음 위치가 아니기에 덩달아 같이 결과에 포함되어 버렸다.

이럴 땐 깔끔하게 **단어 경계**를 붙여 문제를 해결하면 되겠다.

**[정규 표현식]**

\b(?<!\$)\d+\b

**[결과]**

I paid $30 for **100** apples, **50** oranges, and **60** pears.

I saved $5 on this order.

**6. 정리**

전후방 탐색을 사용하면, 무엇을 반환할지 더 구체적으로 명시할 수 있다.

전투방 탐색 명령어는 하위 표현식을 통해 텍스트의 위치를 지정하지만, 소비하지는 않는다.

일치하지만, 검색 결과 텍스트에는 포함되지 않는다는 뜻이다.

전후방 탐색 명령어는 다음과 같은 것들이 있다.

* 긍정형 전방 탐색 : (?=)
* 부정형 전방 탐색 : (?!)
* 긍정형 후방 탐색 : (?<=)
* 부정형 후방 탐색 : (?<!)

# Java - Regular Expressions

# http://www.tutorialspoint.com/java/java\_regular\_expressions.htm

Advertisements

[Previous Page](http://www.tutorialspoint.com/java/java_date_time.htm)

[Next Page](http://www.tutorialspoint.com/java/java_methods.htm)

Java provides the java.util.regex package for pattern matching with regular expressions. Java regular expressions are very similar to the Perl programming language and very easy to learn.

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

The java.util.regex package primarily consists of the following three classes:

* **Pattern Class:** A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static**compile()** methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
* **Matcher Class:** A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the **matcher()** method on a Pattern object.
* **PatternSyntaxException:** A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

## Capturing Groups:

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression (dog) creates a single group containing the letters "d", "o", and "g".

Capturing groups are numbered by counting their opening parentheses from left to right. In the expression ((A)(B(C))), for example, there are four such groups:

* ((A)(B(C)))
* (A)
* (B(C))
* (C)

To find out how many groups are present in the expression, call the groupCount method on a matcher object. The groupCount method returns an int showing the number of capturing groups present in the matcher's pattern.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by groupCount.

## Example:

Following example illustrates how to find a digit string from the given alphanumeric string:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

public static void main( String args[] ){

// String to be scanned to find the pattern.

String line = "This order was placed for QT3000! OK?";

String pattern = "(.\*)(\\d+)(.\*)";

// Create a Pattern object

Pattern r = Pattern.compile(pattern);

// Now create matcher object.

Matcher m = r.matcher(line);

if (m.find( )) {

System.out.println("Found value: " + m.group(0) );

System.out.println("Found value: " + m.group(1) );

System.out.println("Found value: " + m.group(2) );

} else {

System.out.println("NO MATCH");

}

}

}

This would produce the following result:

Found value: This order was placed for QT3000! OK?

Found value: This order was placed for QT300

Found value: 0

## Regular Expression Syntax:

Here is the table listing down all the regular expression metacharacter syntax available in Java:

|  |  |
| --- | --- |
| **Subexpression** | **Matches** |
| ^ | Matches beginning of line. |
| $ | Matches end of line. |
| . | Matches any single character except newline. Using m option allows it to match newline as well. |
| [...] | Matches any single character in brackets. |
| [^...] | Matches any single character not in brackets |
| \A | Beginning of entire string |
| \z | End of entire string |
| \Z | End of entire string except allowable final line terminator. |
| re\* | Matches 0 or more occurrences of preceding expression. |
| re+ | Matches 1 or more of the previous thing |
| re? | Matches 0 or 1 occurrence of preceding expression. |
| re{ n} | Matches exactly n number of occurrences of preceding expression. |
| re{ n,} | Matches n or more occurrences of preceding expression. |
| re{ n, m} | Matches at least n and at most m occurrences of preceding expression. |
| a| b | Matches either a or b. |
| (re) | Groups regular expressions and remembers matched text. |
| (?: re) | Groups regular expressions without remembering matched text. |
| (?> re) | Matches independent pattern without backtracking. |
| \w | Matches word characters. |
| \W | Matches nonword characters. |
| \s | Matches whitespace. Equivalent to [\t\n\r\f]. |
| \S | Matches nonwhitespace. |
| \d | Matches digits. Equivalent to [0-9]. |
| \D | Matches nondigits. |
| \A | Matches beginning of string. |
| \Z | Matches end of string. If a newline exists, it matches just before newline. |
| \z | Matches end of string. |
| \G | Matches point where last match finished. |
| \n | Back-reference to capture group number "n" |
| \b | Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets. |
| \B | Matches nonword boundaries. |
| \n, \t, etc. | Matches newlines, carriage returns, tabs, etc. |
| \Q | Escape (quote) all characters up to \E |
| \E | Ends quoting begun with \Q |

## Methods of the Matcher Class:

Here is a list of useful instance methods:

## Index Methods:

Index methods provide useful index values that show precisely where the match was found in the input string:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public int start()**  Returns the start index of the previous match. |
| 2 | **public int start(int group)**  Returns the start index of the subsequence captured by the given group during the previous match operation. |
| 3 | **public int end()**  Returns the offset after the last character matched. |
| 4 | **public int end(int group)**  Returns the offset after the last character of the subsequence captured by the given group during the previous match operation. |

## Study Methods:

Study methods review the input string and return a Boolean indicating whether or not the pattern is found:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public boolean lookingAt()**  Attempts to match the input sequence, starting at the beginning of the region, against the pattern. |
| 2 | **public boolean find()**  Attempts to find the next subsequence of the input sequence that matches the pattern. |
| 3 | **public boolean find(int start)**  Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index. |
| 4 | **public boolean matches()**  Attempts to match the entire region against the pattern. |

## Replacement Methods:

Replacement methods are useful methods for replacing text in an input string:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public Matcher appendReplacement(StringBuffer sb, String replacement)**  Implements a non-terminal append-and-replace step. |
| 2 | **public StringBuffer appendTail(StringBuffer sb)**  Implements a terminal append-and-replace step. |
| 3 | **public String replaceAll(String replacement)**  Replaces every subsequence of the input sequence that matches the pattern with the given replacement string. |
| 4 | **public String replaceFirst(String replacement)**  Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string. |
| 5 | **public static String quoteReplacement(String s)**  Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement s in the appendReplacement method of the Matcher class. |

## The *start* and *end* Methods:

Following is the example that counts the number of times the word "cat" appears in the input string:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static final String REGEX = "\\bcat\\b";

private static final String INPUT =

"cat cat cat cattie cat";

public static void main( String args[] ){

Pattern p = Pattern.compile(REGEX);

Matcher m = p.matcher(INPUT); // get a matcher object

int count = 0;

while(m.find()) {

count++;

System.out.println("Match number "+count);

System.out.println("start(): "+m.start());

System.out.println("end(): "+m.end());

}

}

}

This would produce the following result:

atch number 1

start(): 0

end(): 3

atch number 2

start(): 4

end(): 7

atch number 3

start(): 8

end(): 11

atch number 4

start(): 19

end(): 22

You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred.

The start method returns the start index of the subsequence captured by the given group during the previous match operation, and end returns the index of the last character matched, plus one.

## The *matches* and *lookingAt* Methods:

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static final String REGEX = "foo";

private static final String INPUT = "fooooooooooooooooo";

private static Pattern pattern;

private static Matcher matcher;

public static void main( String args[] ){

pattern = Pattern.compile(REGEX);

matcher = pattern.matcher(INPUT);

System.out.println("Current REGEX is: "+REGEX);

System.out.println("Current INPUT is: "+INPUT);

System.out.println("lookingAt(): "+matcher.lookingAt());

System.out.println("matches(): "+matcher.matches());

}

}

This would produce the following result:

Current REGEX is: foo

Current INPUT is: fooooooooooooooooo

lookingAt(): true

matches(): false

## The *replaceFirst* and *replaceAll* Methods:

The replaceFirst and replaceAll methods replace text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences.

Here is the example explaining the functionality:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static String REGEX = "dog";

private static String INPUT = "The dog says meow. " +

"All dogs say meow.";

private static String REPLACE = "cat";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// get a matcher object

Matcher m = p.matcher(INPUT);

INPUT = m.replaceAll(REPLACE);

System.out.println(INPUT);

}

}

This would produce the following result:

The cat says meow. All cats say meow.

## The *appendReplacement* and *appendTail* Methods:

The Matcher class also provides appendReplacement and appendTail methods for text replacement.

Here is the example explaining the functionality:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static String REGEX = "a\*b";

private static String INPUT = "aabfooaabfooabfoob";

private static String REPLACE = "-";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// get a matcher object

Matcher m = p.matcher(INPUT);

StringBuffer sb = new StringBuffer();

while(m.find()){

m.appendReplacement(sb,REPLACE);

}

m.appendTail(sb);

System.out.println(sb.toString());

}

}

This would produce the following result:

-foo-foo-foo-

## PatternSyntaxException Class Methods:

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public String getDescription()**  Retrieves the description of the error. |
| 2 | **public int getIndex()**  Retrieves the error index. |
| 3 | **public String getPattern()**  Retrieves the erroneous regular expression pattern. |
| 4 | **public String getMessage()**  Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern. |

PHP 정규표현식 상세 put this document to your twitter  
참조 : <http://kr2.php.net/manual/kr/regexp.reference.php>

\ (backslash)

- 여러가지로 사용하는 일반적인 회피 문자   
이스케이프 특수 문자 - 백슬래시(\) 다음에 나오는 문자입니다. 왼쪽 중괄호({), 캐럿(^) 또는 그 밖의 일부 특수 문자처럼 정규식 구문에 사용되는 문자를 찾는 데 사용할 수 있습니다. 예를 들어 \$를 사용하면 '줄의 끝'을 나타내는 정규식을 구현하는 것이 아니라 달러 기호($) 자체를 검색하게 됩니다. 마찬가지로 정규식 \.를 사용하면 마침표(.)가 정규식으로 사용될 때처럼 임의의 한 문자를 검색하는 것이 아니라 마침표(.) 문자 자체를 검색하게 됩니다.

^(caret)

- 목표의 처음 (멀티라인 모드에서는 줄의 처음) , [ ] 에서 처음 문자로 올 때, 부정 클래스로 설정   
줄의 시작 - 줄의 시작 위치에 지정된 텍스트가 나올 경우입니다. 예를 들어 정규식 ^When in은 "When in the course of human events"나 "When in town, call me"와 같이 "When in"으로 시작하면서 "When in"이 줄의 시작 위치에 나오는 모든 문자열을 검색합니다. 그러나 이 정규식은 줄의 시작에 나오는 "What and when in the course of human events"와는 일치하지 않습니다.  
예) ^shop (문자열의 처음에 shop을 포함하면 true, else false)

$(dollar)

- 목표의 마지막 (멀티라인 모드에서는 줄의 끝)   
줄의 끝 - 줄의 끝 위치에 지정된 텍스트가 나올 경우입니다. 예를 들어 정규식 professional$는 문자열 "He is a professional"의 끝 부분과 일치하지만 문자열 "They are a group of professionals"와는 일치하지 않습니다.  
예) wiz$(문자열의 끝에 wiz를 포함하면 true, else false)

.(period)

- (기본값으로) 줄바꿈을 제외한 임의의 한 문자   
- 출력 여부에 관계없이 모든 단일 문자와 일치하는 와일드카드 역할을 합니다. 단, 줄 바꿈(\n) 문자는 제외됩니다. 예를 들어 정규식 c.t는 문자열 cat, c t, cot와 일치하지만 cost와는 일치하지 않습니다. 이 예에서 마침표(.)는 단일 문자를 나타내는 와일드카드입니다. 문자 'c'와 't' 사이에 마침표가 있으므로 문자 'c'와 't' 사이에 단일 문자(공백 포함)가 있는 문자열은 모두 이 식과 일치합니다.

[ ](bracket)

- 허용 혹은 비허용 문자 범위를 지정합니다.  
문자 집합 - 대괄호([]) 안에 지정된 임의의 한 문자입니다. [a-z]과 같이 하이픈(-)을 사용하여 문자 범위를 지정할 수 있습니다. 예를 들면 다음과 같습니다.

* 정규식 c[aou]t는 cat, cot 및 cut와 일치하지만 cet나 cit와는 일치하지 않습니다.
* 정규식 [0-9]는 임의의 숫자를 의미합니다. 복수 범위의 문자도 지정할 수 있습니다.
* 정규식 [A-Za-z]는 모든 대소문자와 일치합니다.
* []내에서 "^"이 선행되면 not을 나타냄
* [:문자클래스:]의 형태가 있다(문자클래스 - alpha, blank, cntrl, digit, graph, lower, print, space, uppper, xdigit)

예) [abc] (a, b, c 중 어떤 문자, "[a-c]."과 동일), [Yy] (Y 또는 y), [A-Za-z0-9] (모든 알파벳과 숫자), [-A-Z]. ("-"(hyphen)과 모든 대문자), [^a-z] (소문자 이외의 문자), [^0-9] (숫자 이외의 문자), [[:digit:]] ([0-9]와 동일)

|(bar)

-선택 브랜치 시작, 정규식 패턴의 "또는, OR"을 의미합니다.    
또는 - 두 항목 사이의 선택을 나타냅니다. 즉, OR 기호(|)의 앞이나 뒤에 나오는 식과 일치합니다. 예를 들어 정규식 (him|her)는 "it belongs to him" 또는 "it belongs to her"와 일치하지만 "it belongs to them" 줄과는 일치하지 않습니다.

( )(parenthesis)

- 서브패턴을 지정합니다.   
그룹 식 - 하위 식의 시작과 끝을 표시합니다. 하위 식은 (ha)+와 같이 괄호 ( ) 안에 포함되는 정규식입니다. 이 예에서는 더하기 기호(+)가 괄호 ( ) 그룹 식과 조합되어 하나의 구문이 됩니다. 괄호 ( ) 안에 들어 있는 (ha)가 하위 식입니다. 더하기 기호(+)를 추가하면 반복되는 문자 쌍을 찾을 수 있습니다. 더하기 기호(+)는 'ha'가 1번 이상 있을 수 있음을 의미합니다. 이 식은 'haha' 및 'hahaha'와 일치합니다.

?(asterisk)

- ( 의미 확장, 또는 0회나 1회, 또는 수량어 minimizer 바로 앞 패턴에 대해 있거나/없다 를 의미합니다.

\*(asterisk)

- 0회 이상의 횟수, 바로 앞 패턴에 대해 없거나/여러개를 의미합니다.  
최대(0개 이상) - 가능한 많은 수의 문자와 일치하면서 식 앞의 문자를 0번 이상 반복합니다. 정규식 .\*는 한 문자를 0번 이상 반복합니다. 예를 들어 정규식 b.\*k는 book, back, black, blank 및 buck와 일치합니다. 이 예에서는 마침표(.)가 별표(\*)와 조합되어 하나의 구문이 됩니다. 마침표(.)는 별표(\*) 식 바로 앞에 옵니다. 별표(\*)는 'b'와 'k' 사이에 임의의 문자가 0번 이상 있을 수 있음을 의미합니다. 마침표(.)는 'b'와 'k' 사이의 문자에 대한 와일드카드 역할을 합니다. 이 예에서는 'b'와 'k' 사이에 임의의 문자가 반복될 수 있음을 의미합니다.

+(asterisk)

- 바로 앞 패턴에 대해 1개 이상을 의미합니다.   
최대(1개 이상) - 가능한 많은 수의 문자와 일치하면서 식 앞의 문자를 1번 이상 반복합니다. 정규식 .+는 한 문자를 1번 이상 반복합니다. 예를 들어 정규식 bo+.는 bob, book 및 boot와 일치합니다. 이 예에서는 마침표(.)가 더하기 기호(+)와 조합되어 하나의 구문이 됩니다. 마침표(.)는 더하기 기호(+) 식 바로 뒤에 옵니다. 더하기 기호(+)는 문자 'o'가 1번 이상 있을 수 있음을 의미합니다. 마침표(.)는 각 단어의 마지막 문자(이 예에서는 'b', 'k' 및 't')에 대한 와일드카드 역할을 합니다.

{}(brace)

- 최소/최대 횟수 시작, 바로 앞 패턴(선행문자)의 갯수를 지정합니다.  
태그가 지정된 식 - 괄호 안에 포함된 식과 일치하는 텍스트에 태그를 지정합니다. \N을 사용하면 찾기 식에서 태그가 지정된 텍스트의 다른 항목을 검색하거나 바꾸기 식에서 태그가 지정된 텍스트를 삽입할 수 있습니다. 예를 들어 연속된 두 개의 중복 단어를 검색하려면 {.#} \1 식을 사용합니다. 연속된 단어가 공백 하나로 구분된다고 가정하면 오른쪽 중괄호(})와 백슬래시(\) 사이에 공백을 추가할 수 있습니다. 이 예에서는 # 기호와 마침표(.)가 중괄호({})와 조합되어 하나의 구문이 됩니다. 이 식에서 .#는 임의의 연속 문자를 나타냅니다. 식에서 이 부분은 중괄호({})로 묶여 있으므로 해당 연속 문자에는 태그가 지정되며 이를 \1로 나타낼 수 있습니다. 이 식은 연속 문자 다음에 공백이 오고 그 다음에 정확히 동일한 연속 문자가 또 나오는 경우를 찾습니다.  
예) a{3} ('a'의 3번 반복인 aaa만 해당됨), a{3,} ('a'가 3번 이상 반복인 aaa, aaaa, aaaa, ... 등), a{3,5} (a가 3~5번 반복인 aaa, aaaa, aaaaa 만 해당됨), [0-9]{2} (두 자리 숫자), a[7-9]{2} (7~9인숫자가 2개 나옮), [^Aa]{3} (A와 a를 포함하지 않는 3개의 문자열), .{3,4}com ('com'앞에 3~4개의 문자를 포함하는 문자열)

\d

임의의 10진 숫자

\D

10진 숫자가 아닌 임의의 문자

\h

수평 공백 문자 (PHP 5.2.4부터)

\H

수평 공백 문자를 제외한 모든 문자 (PHP 5.2.4부터)

\s

모든 공백 문자

\S

공백이 아닌 모든 문자

\v

수직 공백 문자 (PHP 5.2.4부터)

\V

수직 공백 문자를 제외한 모든 문자 (PHP 5.2.4부터)

\w

모든 "word" 문자

\W

모든 "non-word" 문자

@

- 정규식 .@는 한 문자를 0번 이상 반복합니다. 예를 들어 정규식 a.@x는 'abxbxb' 내의 'abx'와 'acxcxc' 내의 'acx'와 일치합니다. 이 예에서는 마침표(.)가 @ 기호와 조합되어 하나의 구문이 됩니다. 마침표(.)는 @ 기호 식 바로 앞에 옵니다. @ 기호는 'a'와 'x' 사이에 임의의 문자가 0번 이상 있을 수 있음을 의미합니다. 이 예에서 마침표(.)는 문자 'a'와 'x' 사이의 문자 'b' 및 'c'에 대한 와일드카드 역할을 합니다.

#

-예를 들어 정규식 si.#er는 'sicker' 또는 'silkier'와 일치합니다. 이 예에서는 마침표(.)가 # 기호와 조합되어 하나의 구문이 됩니다. 마침표(.)는 # 기호 식 바로 앞에 옵니다. # 기호는 'si'와 'er' 사이에 임의의 문자가 0번 이상 있을 수 있음을 의미합니다. 마침표(.)는 단어 sicker에 있는 문자 'c' 및 'k'와 단어 silkier에 있는 문자 'l', 'k' 및 'i'에 대한 와일드카드 역할을 합니다.

\N

- N번째 태그가 지정된 식   
- 찾기 식에서 \N은 N번째 태그가 지정된 식과 일치하는 텍스트를 나타냅니다. 여기에서 N은 1부터 9까지의 숫자입니다.   
바꾸기 식에서 \N은 N번째 태그가 지정된 식과 일치하는 텍스트를 삽입합니다. 여기에서 N은 1부터 9까지의 숫자입니다. \0은 전체 찾기 식과 일치하는 텍스트를 삽입합니다.   
예를 들어 연속된 두 개의 중복 단어를 찾아 한 단어로 바꾸려면 {.#} \l 식을 사용합니다. 연속된 단어가 공백 하나로 구분된다고 가정하면 오른쪽 중괄호(})와 백슬래시(\) 사이에 공백을 추가할 수 있습니다. 이 예에서는 # 기호와 마침표(.)가 중괄호({})와 조합되어 하나의 구문이 됩니다. 바꿀 때는 \l 식을 사용합니다. \1은 찾기 문자열의 첫 번째 중괄호 쌍에서 찾은 내용을 나타냅니다. 바꾸기 작업에서 \1을 사용하면 연속된 중복 단어를 해당 단일 단어로 바꿀 수 있습니다.

~x

- 구분 안 함 - 식의 지정된 위치에 x가 나타날 경우 해당 항목을 검색에서 제외합니다. 예를 들어 정규식 real~(ity)는 "realty"와 "really"의 "real"은 구분하지만 "reality"의 "real"은 구분하지 않습니다.

\n

- 바꿈 - 코드 보기의 새 줄 또는 디자인 보기의   
입니다. 이 구문(\n)을 사용하면 모든 줄 바꿈을 간단하게 나타낼 수 있습니다.

\t

- ^\t+ 이 예에서는 캐럿(^)과 더하기 기호(+)가 탭(\t)과 조합되어 하나의 구문이 됩니다. 단일 탭 문자 식 앞에 오는 캐럿(^)은 탭이 지정된 모든 문자가 줄의 시작 위치에 있는 경우만 검색합니다. 더하기 기호(+)는 일치하는 하나 이상의 탭 문자를 나타냅니다.

[^]

- 집합에 없는 한 문자 - 캐럿(^) 뒤의 문자 집합에 없는 한 문자를 나타냅니다. 예를 들어 범위에 없는 문자를 제외한 임의의 문자를 나타내려면 여는 괄호 뒤에 첫 문자로 캐럿(^)을 사용합니다. [^269A-Z] 식은 2, 6, 9 및 모든 대문자를 제외한 모든 문자와 일치합니다.

n

- 반복 식 - 캐럿(^) 앞에 있는 식을 n번 반복합니다. 예를 들어 n이 4인 식 [0-9]^4는 네 자리 수를 모두 검색합니다. 실제로도 이 예와 같이 문자 집합([ ]) 구문과 반복(^n) 구문을 조합하여 많이 사용하고 있습니다.

:a

- [a-zA-Z0-9] 식을 사용하여 하나의 대소문자 또는 숫자 항목을 검색할 수 있습니다. 영숫자 항목이라고도 합니다. [a-zA-Z0-9]의 모든 인스턴스에 줄임 식 :a를 사용할 수 있습니다.

:b

- 공백 - 코드 또는 텍스트의 공백입니다. 예를 들어 줄의 시작 위치에서 단일 공백 문자를 검색하려면 정규식 ^:b를 사용합니다.

:c

- 영문자 - [a-zA-Z] 식과 일치합니다. 이 식을 사용하여 모든 대소문자를 나타낼 수 있습니다. [a-zA-Z]의 모든 인스턴스에 줄임 식 :c를 사용할 수 있습니다.

:d

- 10진수 - [0-9] 식과 일치합니다. 이 식을 사용하여 모든 숫자를 나타낼 수 있습니다. 예를 들어 텍스트 파일에서 주민 등록 번호를 찾으려는 경우 주민 등록 번호 형식인 000000-0000000을 :d^6-:d^7로 나타내거나 [0-9]를 사용하여 동일한 식인 [0-9]^6-[0-9]^7로 나타낼 수 있습니다. [0-9]의 모든 인스턴스에 줄임 식 :d를 사용할 수 있습니다.

:h

- 16진수 - [0-9a-fA-F]+ 식과 일치합니다. 'A'와 'F' 사이의 대소문자와 숫자를 사용한 16진수 조합을 나타내려면 이 식을 사용합니다. 예를 들어 웹 사이트 페이지에 서로 다른 여러 가지 배경색이 있고 해당 페이지의 색을 검정(예: 000000)으로 변경하려 하지만 기존 색의 16진수를 모르는 경우 기존 16진수를 모두 찾으려면 다음 정규식을 사용합니다. \#:h [0-9a-fA-F]를 사용하여 검색할 수도 있지만 이 예에서는 백슬래시(\)와 # 기호가 16진수(:h) 구문과 조합됩니다. \#는 식이 아닌 # 기호를 검색하며 :h는 모든 16진수 문자를 검색합니다. 기존 16진수를 바꾸려면 000000과 같이 원하는 배경색의 16진수를 입력합니다.

:i

- 식별자 - [a-zA-Z\_$][a-zA-Z0-9\_$]\* 식과 일치합니다. 코드 작업을 할 때 모든 프로그램 식별자를 검색하려면 위의 긴 식을 입력하는 대신 줄임 식 :i를 사용할 수 있습니다.

:n

- 유리수 - ([0-9]+\.[0-9]\*)|([0-9]\*\.[0-9]+)|([0-9]+) 식과 일치합니다. 소수점을 포함하는 정수를 모두 검색하려면 위의 긴 식을 입력하는 대신 줄임 식 :n을 사용할 수 있습니다.

:q

- 따옴표 붙은 문자열 - ("[~"]\*")|('[~']\*') 식과 일치합니다. 인용 부호 안에 들어 있는 인용문을 모두 검색하려면 위의 긴 식을 입력하는 대신 줄임 식 :q를 사용할 수 있습니다.

:w

- 영문 문자열 - [a-zA-Z]+ 식과 일치합니다. 이 구문을 사용하면 대소문자에 관계없이 간단하게 영문자를 하나 이상 나타낼 수 있습니다.

:z

- 10진수 정수 - [0-9]+ 식과 일치합니다. 이 구문을 사용하면 0 이상의 숫자를 간단하게 나타낼 수 있습니다.

^^

- 파일의 시작 - 파일의 시작 위치에 지정된 텍스트가 나올 경우입니다. 소스 코드 또는 텍스트 파일에서 텍스트를 검색할 때만 사용할 수 있습니다. 예를 들어 파일의 시작 위치에 나오는 첫 번째 HTML 태그를 검색하려면 정규식 ^^를 사용합니다.

$$

- 파일의 끝 - 파일의 끝 위치에 지정된 텍스트가 나올 경우입니다. 소스 코드 또는 텍스트 파일에서 텍스트를 검색할 때만 사용할 수 있습니다. 예를 들어 파일의 끝에 나오는 마지막 HTML 태그(태그 다음에 공백이 없는 경우)를 검색하려면 정규식 $$를 사용합니다.

Pattern Modifiers(패턴 변경자) :   
http://kr2.php.net/manual/kr/reference.pcre.pattern.modifiers.php

i (PCRE\_CASELESS)

이 변경자를 지정하면, 패턴의 문자는 대문자와 소문자를 구별하지 않습니다.

m (PCRE\_MULTILINE)

기본적으로, PCRE는 주어진 문자열을 하나의 "줄"로 취급합니다. (실제로 몇개의 라인을 가지더라도) "줄 시작" 메타문자(^)는 문자열의 처음만을 인식하며, "줄 끝" 메타문자($)는 문자열의 끝이나 (D 변경자가 지정되지 않는 한) 마지막 뉴라인의 직전만을 인식합니다. 이는 펄과 같습니다. 이 변경자를 지정하면, "줄 시작"과 "줄 끝"은 주어진 문자열의 모든 뉴라인 직후와 직전을 인식합니다. respectively, as well as at the very start and end. 이는 펄의 /m 변경자와 동일합니다. 주어진 문자열에 "\n" 문자가 존재하지 않거나 ^나 $ 패턴이 일어나지 않으면 이 변경자는 아무런 효과가 없습니다.

s (PCRE\_DOTALL)

이 변경자가 지정되면, 패턴의 점 메타문자는 뉴라인을 포함하는 모든 문자를 인식합니다. 지정하지 않으면, 뉴라인은 제외됩니다. 이 변경자는 펄의 /s 변경자와 동일합니다. [^a]와 같은 부정클래스는 이 변경자에 관계 없이 항상 뉴라인 문자를 포함합니다.

x (PCRE\_EXTENDED)

이 변경자가 지정되면, 공백 문자는 이스케이프 되거나 문자 클래스 안에 있을 경우를 제외하고, 완전히 무시합니다. 문자 클래스 밖에서 이스케이프 되지 않은 # 사이와 뉴라인 문자 다음의 문자도 무시합니다. 이는 펄의 /x 변경자와 같고, 복잡한 패턴 안에 코멘트를 사용할 수 있게 합니다. 그러나 이는 데이터 문자에만 해당하는 점에 주의하십시오. 공백 문자는 패턴의 특별한 문자 시퀀스 안에는 존재할 수 없습니다. 예를 들면, 조건 서브 패턴을 나타내는 (?( 시퀀스에는 나와서는 안됩니다.

e (PCRE\_REPLACE\_EVAL)

이 변경자를 지정하면, preg\_replace()는 변경할 문자열을 PHP 코드로 처리하고, 그 결과를 검색된 문자열의 이용하여 일반적인 치환을 합니다. 작은 따옴표, 큰 따옴표, 백슬래시와 NULL 문자는 백슬래시로 이스케이프됩니다. preg\_replace()만 이 변경자를 사용합니다; 다른 PCRE 함수는 무시합니다.

A (PCRE\_ANCHORED)

이 변경자를 지정하면, 패턴을 강제적으로 "고정"합니다. 이는 ("주어진 문자열"에서) 검색된 문자열의 시작에만 매치도록 강제합니다. 패턴 자체에서 특정한 구조를 가지게 하는, 펄에서는 유일한 방법으로 같은 효과를 얻을 수

D (PCRE\_DOLLAR\_ENDONLY)

이 변경자가 설정되면, 패턴의 달러($) 메타문자는 주어진 문자열의 마지막에만 대응합니다. 이 변경자 없이는, 달러는 마지막 문자가 뉴라인일 경우에는 바로 직전의 문자에도 매칭합니다. (마지막이 아닌 뉴라인은 제외합니다) 이 변경자는 m 변경자가 지정되었을때는 무시됩니다. 펄에는 이 변경자가 존재하지 않습니다.

S

패턴이 여러번 이용되면, 매칭에 걸리는 시간을 절약하기 위해서 분석에 더 많은 시간을 들일 가치가 있습니다. 이 변경자를 지정하면, 추가 분석을 행합니다. 현 시점에서, 패턴의 분석은 하나의 고정된 시작 문자를 가지지 않는 비고정 패턴에만 유용합니다.

U (PCRE\_UNGREEDY)

이 변경자는 수량 지시의 "greediness"를 뒤집습니다. 그리하여 기본값으로 not greedy하게 합니다. 하지만 "?"가 붙으면 greedy하게 됩니다. 이는 펄과 호환되지 않습니다. 패턴 안에서 변경자 설정으로 (?U)처럼 지정하거나, 수량지시어 뒤의 물음표로 지정할 수 있습니다. (예. .\*?)

X (PCRE\_EXTRA

이 변경자는 펄과 호환되지 않는 PCRE의 추가 기능을 사용하게 합니다. 패턴의 문자와 결합된 백슬래시가 특별한 의미를 지니지 않을 경우에 에러를 발생시켜서, 차후에 추가 기능을 위해 예약해둡니다. 기본적으로 펄은, 문자와 결합된 백슬래시가 특별한 의미를 지니지 않을 경우에는 글자로 취급합니다. 이 변경자는 다른 기능을 제어하지 않습니다

J (PCRE\_INFO\_JCHANGED)

내부 옵션 (?J) 설정은 영역의 PCRE\_DUPNAMES 옵션을 변경합니다. 서브패턴에 동일한 이름을 허용합니다.

u (PCRE\_UTF8)

이 변경자는 펄과 호환되지 않는 PCRE의 추가 기능을 사용하게 합니다. 패턴 문자열을 UTF-8으로 취급합니다. 유닉스에서는 PHP 4.1.0부터, win32에서는 PHP 4.2.3부터 사용할 수 있습니다. PHP 4.3.5부터 패턴의 UTF-8 유효성이 검사됩니다.

아주간단한 예를 보면서 기본적인 몇개를 처리해보자

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | $str = "shop-wiz.com";    $result = preg\_match ("/^sh/", $str);//result : 1 (문장의 첫 단어가 sh로 시작  $result = preg\_match ("/^ho/", $str);//result : 0      $result = preg\_match ("/om$/", $str);//result : 1 (문장의 마지막이 om로 끝남  $result = preg\_match ("/co$/", $str);//result : 0    $result = preg\_match ("/w.z/", $str);//result : 1 (문장의 중간에 w로 시작하고 특정한문자 하나있고 z다되는 단어  $result = preg\_match ("/w.i/", $str);//result : 0    $result = preg\_match ("/[^0-9]/", $str);//result : 1  $result = preg\_match ("/[0-9]/", $str);//result : 0    $result = preg\_match ("/w\*z/", $str);//result : 1  $result = preg\_match ("/w\*i/", $str);//result : 1 상기 예인 "."과 차잇점  $result = preg\_match ("/w\*m/", $str);//result : 1 (여러개의 문자도 ok)  $result = preg\_match ("/w\*a/", $str);//result : 0    $result = preg\_match ("/shop|Shop|SHOP/", $str);//result : 1 |

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ## 예제샘플 - 네이버의 포스트 불러와서 대표이미지 가져오기  $pattern = "/http:\/\/(blogfiles|postfiles|cafefiles|cfile).\*\.(jp[e]?g|gif|png).\*[\s | \\\" | \\\']/Ui";  $url = "<http://blog.naver.com/PostView.nhn?blogId=>".$naver\_return\_key[3]."&logNo=".$naver\_return\_key[4];  $contents = file\_get\_contents($url);  preg\_match\_all($pattern, $contents, $match);  foreach($match[0] as $key => $val) {      $val = substr($val, 0, -1);  } |

패턴설명  
/http:\/\/(blogfiles|postfiles|cafefiles|cfile).\*\.(jp[e]?g|gif|png).\*[\s | \\\" | \\\']/Ui  
  
/http : http로 시작한다.  
\/\/ : "//"가 연속되며  
(blogfiles|postfiles|cafefiles|cfile).\* : blogfiles 이나 postfiles 이나 cafefiles 이나 cfile 로 되는 문자열 이후로 모든 문자열이 계속된다.  
\.(jp[e]?g|gif|png) : "."문자이후에 pg "e"가 0또는 1회 존재하며 g인 문자 혹은 gif 혹은 png 의 문자가 나오고  
.\* : 다시 줄바꿈을 제외한 모든 문자가 다수 혹은 안나오면서  
[\s | \\\" | \\\']/: 문자의 마지막"/"은 공백이거나 "이거나 '인 문자를  
Ui : 대소문자 구별없이 자른다. 이부분은 하기 예제와 함께 설명된 패턴 변경자의 특징을 보시기 바란다.

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4  5 | preg\_match\_all("|<[^>]+>(.\*)</[^>]+>|U",  "<b>example: </b><div align=left>this is a test</div>",  $out, PREG\_PATTERN\_ORDER);  echo $out[0][0] . ", " . $out[0][1] . "\n";  echo $out[1][0] . ", " . $out[1][1] . "\n"; |

**example:** ,

this is a test

example: , this is a test

이번에는 또다른 예를 가지고 설명드리도록 하겠습니다.   
아래와 같은 문자열이 존재하고 각각에 대해 원하는 결과를 얻어 보도록 하겠습니다.   
$str = "anycharacter abcd1abcd2abcd3abcd anycharacter";  
  
[문제] 결과 처음 ab를 제외한 모든 글자  
preg\_match("/ab(.\*)/i",$str,$regs);  
  
[결과]  
Array ( [0] => abcd1abcd2abcd3abcd anycharacter [1] => cd1abcd2abcd3abcd anycharacter )   
  
[문제] 결과 처음 ab와 d사이의 모든 글자  
preg\_match("/ab(.\*)d/i",$str,$regs);  
  
[결과]  
Array ( [0] => abcd1abcd2abcd3abcd [1] => cd1abcd2abcd3abc )   
이곳에서 중요하게 집고 넘어가야 할 부분이 있습니다. 아마 처음 정규식을 접하시는 분들은 이 결과값이 "c" 라고 생각하시는 분들이 많을 것 같습니다. 그런데 결과값을 보시면 아시겠지만 "d"는 문장뒤에서 읽어와서 처음으로 만나는 문자입니다. 그사이에 모든 문자를 다 디스플레이 합니다. 그렇다면 다음 문제 "c"를 출력할 수 있는 방법은 무엇일까요?   
  
[문제] 결과 처음 ab와 d사이의 모든 글자  
preg\_match("/ab([^d]\*)/i",$str,$regs);  
  
[결과]  
Array ( [0] => abc [1] => c )   
차잇점이 보이시나요?   
  
[문제] 1일 가져오기  
다양한 경우가 있을 수 있는데 가장 쉬운 예는 문장의 처음에 나타나는 숫자를 구하는 것이다.  
아래 두개는 동일한 명령입니다.  
preg\_match("/([0-9])/i",$str,$regs);  
preg\_match("/([[:digit:]])/i",$str,$regs);  
[결과]  
Array ( [0] => 1 [1] => 1 )   
두번째 경우는 d와 a사이의 문자를 구하는 것입니다.  
preg\_match("/d(.\*)a/i",$str,$regs);  
Array ( [0] => d1abcd2abcd3abcd anychara [1] => 1abcd2abcd3abcd anychar ) 허걱.. 예상과는 다른 결과가 존재합니다.  
이예는 앞선 ab와 d 사이의 "c"를 구하는 것과 유사한 예입니다.  
그러면 d와 a사이의 숫자를 구하는 것으로 압축을 해보죠  
  
preg\_match("/d([0-9])a/i",$str,$regs);  
Array ( [0] => d1a [1] => 1 )   
  
[문제] 2라는 숫자 구하기  
2라는 숫자는 두번째로 나오는 숫자입니다. 그러므로 두번째 나오는 숫자를 찾으면 되겠죠  
preg\_match("/[0-9][a-z]\*([0-9])/i",$str,$regs);  
  
[결과]  
Array ( [0] => 1abcd2 [1] => 2 )   
처음에 숫자가 나오고 임의의 문자가 만복되면서 다시 숫자가 시작되는 것..  
  
  
[문제] 이문장에 포함된 모든 숫자를 가져오기  
preg\_match("/([0-9]).\*([0-9]).\*([0-9])/i",$str,$regs);  
  
[결과]  
Array ( [0] => 1abcd2abcd3 [1] => 1 [2] => 2 [3] => 3 )   
하지만 상기 방식은 우리가 숫자가 3개밖에 없다는 가정하에서 사용한 것입니다. 숫자가 엄청많다면..  
이때 사용하는 것이 preg\_match\_all입니다.  
preg\_match\_all("/([0-9])/i",$str,$regs);  
Array ( [0] => Array ( [0] => 1 [1] => 2 [2] => 3 ) [1] => Array ( [0] => 1 [1] => 2 [2] => 3 ) )   
  
[문제] 3이라는 숫자를 구하기  
3이라는 숫자는 문에서 마지막으로 나오는 숫자입니다.. 상기에서 preg\_match\_all을 사용하여 마지막 인자값을 가져와도 되고 아래와 같이 preg\_match를 사용하여 가져와도 됩니다.  
preg\_match("/([0-9])[^0-9]\*/i",$str,$regs);  
  
[결과]  
Array ( [0] => 1abcd [1] => 1 )   
결과가 우리가 원한 것이 아닙니다. 우리는 문장의 뒤에서 부터 읽을 필요가 있습니다.  
preg\_match("/([0-9])[^0-9]\*$/i",$str,$regs);  
Array ( [0] => 3abcd anycharacter [1] => 3 )   
혹은 어차피 마지막 숫자이므로 아래와 같이 해도 되겠죠  
preg\_match("/.\*([0-9])/i",$str,$regs);  
Array ( [0] => anycharacter abcd1abcd2abcd3 [1] => 3 )

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $subject ="      <body>      PHP 정규식 샘플 페이지 입니다.      <br />      <a href=\"http://www.shop-wiz.com.com\">SHOP-WIZ.COM</a>쇼핑몰솔루션입니다.<br />      <a href=\"mailto:master@shop-wiz.com\">문의사항은 이곳으로</a><br />      좀더 새로운 모습으로 인사드리겠습니다.      <a href=\"http://www.shop-wiz.com\">홈</a>      </body>      "; |

'샘플'과 일치하는 갯수 : 1  
.과 일치하는 갯수 :10  
shop-wiz.com 일치하는 갯수: 3  
shop-wiz.com 일치하는 갯수: 4  
a href=""와 일치하는 갯수   
3  
메치한는 첫 단어출력   
a href="http://www.shop-wiz.com.com"  
메치한는 첫 단어출력   
http://www.shop-wiz.com.com  
메치한는 둘 단어출력   
a href="mailto:master@shop-wiz.com"  
메치한는 둘 단어출력   
mailto:master@shop-wiz.com PHP 정규식 샘플 페이지 입니다.   
[http://SHOP-WIZ.COM](http://www.shop-wiz.com/document/regex)쇼핑몰솔루션입니다.  
[문의사항은 이곳으로](mailto:master@http://shop-wiz.com)  
좀더 새로운 모습으로 인사드리겠습니다. [홈](http://www.shop-wiz.com/document/regex)

서브 패턴()  
서브 패턴의 역활에 대해서 알아보겠습니다.  
서브 패턴은 () 안에 들어가 있는 것인데 상기 예에서는 (.\*) 입니다.   
좀더 상세한 패턴 형식을 구할때 사용되는데 preg\_match\_all($pattern, $string, $matches); 와 같이 사용될때 $matches에는 각각의 결과가 저장됩니다.  
$matches[0] 에는 전체 패턴에 대한 값이 들어가고 이후 $matches[1]... 부터는 서브패턴의 수만큼 결과값이 저장됩니다.  
  
아래는 다음에 자세한 설명을 드릴 url을 구하는 정규식인데 각각의 결과를 보시면서 서브패턴과 결과가 어떻게 달라지는지 보실 수 있습니다.

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4 | $str = '<a href="<http://www.shop-wiz.com>">숍위즈</a> <a href="\'<http://www.naver.com>\'">네이버</a><a href="<http://www.daum.net>">다음</a>';  //preg\_match\_all("/]\* href=(['|\"]\*)([^\\1\040>]\*)\\1[^>]\*>/is", $str, $matches);  preg\_match\_all("/href=(['|\"]\*)([^\\1>]\*)\\1[^>]\*>/is", $str, $matches);  print\_r($matches); |

Array  
(  
[0] => Array  
(  
[0] => <a href="http://www.shop-wiz.com'>  
[1] => <a href='http://www.naver.com'>  
[2] => <a href=http://www.daum.net>  
)

href 로 시작하고 ' 혹은 " 이 0개이상 포함되고(0도 포함) ">"이 아닌 문장이 계속되다가 마지막은 ">"을만난다

[1] => Array  
(  
[0] => "  
[1] => '  
[2] =>   
)

첫번째 sub pattern (['|\"]\*);

[2] => Array  
(  
[0] => http://www.shop-wiz.com  
[1] => http://www.naver.com  
[2] => http://www.daum.net  
)

)

두번째 sub pattern ([^\\1>]\*);  
참조 : 마지막 라인의 \\1[^>]\*> 은 현재 두번째 서브패턴은 ">" 이전 까지 포함하므로 ' 혹은 "등이 포함 될수 있다.  
따라서 끝이 ">"아닌 특정단어 하나를 빼 줌으로서 " 을 삭제할 수 있다

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3 | $str = "<http://www.shop-wiz.com> 숍위즈 <http://www.naver.com> 네이버 <http://www.daum.net> 다음";  preg\_match\_all("/(http|mms|ftp|telnet):\/\/[-a-z0-9]+(\.[-a-z0-9]+)/is", $str, $matches);  print\_r($matches); |

Array  
(  
[0] => Array  
(  
[0] => http://www.shop-wiz  
[1] => http://www.naver  
[2] => http://www.daum  
)

정규식에 걸러지는 전체 문장 : http 혹은 mms 혹은 ftp 혹은 telnet으로 시작하고 :// 이 오면서 - 과 영소문자 및 숫자가 1개이상 나오면서 마지막은 . 혹은 -과 영소문자 및 숫자가 한개이상 나오는 구문. 즉 http 등으로 시작하고 - 혹은 영문자 숫자외의 문자가 오면 패턴이 끝나는 것이다.

[1] => Array  
(  
[0] => http  
[1] => http  
[2] => http  
)

첫번째 sub pattern (http|mms|ftp|telnet)

[2] => Array  
(  
[0] => .shop-wiz  
[1] => .naver  
[2] => .daum  
)

)

두번째 sub pattern (\.[-a-z0-9]+))  
만약 두번째 패튼에서 .을 제외한 문자만을 얻고 싶을때는 \.([-a-z0-9]+) 라고 처리하면 됩니다.

두개가 결과는 다르지만 비슷한 결과를 출력하는데 matches[0]에는 전체 정규식 결과가 출력되고 현재 서브패턴("()")이 두개가 있으므로 이후 배열 matches[1], matches[2]에는 서브패턴된 각각의 값이 들어가는 것을 보실 수 있습니다.

Pattern Modifiers(패턴 변경자)의 실예   
## u (PCRE\_UTF8)

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3 | $str = '한글 english どをウィ 中國 ＃＆＊§※☆★';  preg\_match\_all('/./u', $str, $match );  echo print\_r($match); |

결과 :   Array ( [0] => Array ( [0] => 한 [1] => 글 [2] => [3] => e [4] => n [5] => g [6] => l [7] => i [8] => s [9] => h [10] => [11] => ど [12] => を [13] => ウ [14] => ィ [15] => [16] => 中 [17] => 國 [18] => [19] => ＃ [20] => ＆ [21] => ＊ [22] => § [23] => ※ [24] => ☆ [25] => ★ ) )

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4  5 | $pattern = "/p.\*p/";  $contetns = "apple, orange, p p, kk poPartist 팝아티스트";  preg\_match\_all($pattern, $contetns, $match);  print\_r($match); |

결과 : Array ( [0] => Array ( [0] => pple, orange, p p, kk p ) )

현재 주어진 문장을 읽어 문장에서 맨처음 나오는 p와 맨나중에 나오는 p사이의 모든 문장을 읽어들인다.

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4 | $pattern = "/p.\*p/";  $contetns = "apple, orange, p p, kk poPartist 팝아티스트\n나의 이름은 product pop 이다.";  preg\_match\_all($pattern, $contetns, $match);  print\_r($match); |

결과 : Array ( [0] => Array ( [0] => pple, orange, p p, kk p [1] => product pop ) )

두번째 줄에 다른 문장이 하나더 들어갔다.  
결과 처럼 문장별로 새로운 배열결과가 출력됨을 확인할수 있다.

## U (PCRE\_UNGREEDY)

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4 | $pattern = "/p.\*p/U";  $contetns = "apple, orange, p p, kk poPartist 팝아티스트 나의 이름은 product pop 이다.";  preg\_match\_all($pattern, $contetns, $match);  print\_r($match); |

결과 : Array ( [0] => Array ( [0] => pp [1] => p p [2] => poPartist 팝아티스트 나의 이름은 p [3] => pop ) )

패턴 변경자인 U를 추가하였다. U를 설명할때 "not greedy"라는 표현을 사용했는데 아마 이샘플을 보시면 그 의미를 아실 수 있을 것입니다. 문장의 라인별(\n)이 아니라 일치하는 구문구문을 모두 자른다는 것을 보실수 있습니다.

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4 | $pattern = "/p.\*p/U";  $contetns = "apple, orange, p p, kk poPartist 팝아티스트\n나의 이름은 product pop 이다.";  preg\_match\_all($pattern, $contetns, $match)  print\_r($match); |

결과 : Array ( [0] => Array ( [0] => pp [1] => p p [2] => product p ) )

바로위 패턴과 비교하여 \n(줄바꿈이 하나더 들어갔다)  
예상을 했나요? 그렇습니다. 정규식에서 \n은 연속이 아니라 단절을 의미합니다. 즉 다음라인에서부터 처음부터 새로이 검색을 하는 거죠.

## i (PCRE\_CASELESS)

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4 | $pattern = "/p.\*p/Ui";  $contetns = "apple, orange, p p, kk poPartist 팝아티스트";  preg\_match\_all($pattern, $contetns, $match);  print\_r($match); |

결과 : Array ( [0] => Array ( [0] => pp [1] => p p [2] => poP ) )

i는 대소문자를 구분하지 않습니다. 현재 결과와 같이 poP의 P대문자도 같이 인식을 하게 됩니다.

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4 | $pattern = "|p.\*p|U";  $contetns = "apple, orange, p p, kk poPartist 팝아티스트\n나의 이름은 product pop 이다.";  preg\_match\_all($pattern, $contetns, $match);  print\_r($match); |

결과 : Array ( [0] => Array ( [0] => pp [1] => p p [2] => product p ) )

또다른 예로 기존에 설명드린예에서 "/" 대신에 "|"을 사용해 보았습니다. 결론은 같지요. 그런데 왜 다르게 사용할까요? 그것은 저도 모름.^^ 어디서 설명서를 봐야한는데. 찾는 대로 다시 업데이트 하겠습니다.  
<http://www.php.net/manual/en/regexp.reference.delimiters.php>   
결국 "/"은Delimiters 에 속하고 | 은 정규식 표현중에 하나죠  
PCRE 함수를 사용할 경우 Delimiters에 감싸줘야 한다고 나와 있네요

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3 | //연속된 두개의 공백을 하나로 바꾸는 스크립트  $str = "this is       my   new face ";  $str = preg\_replace("/\s+/"," ",$str); |

[?](http://www.shop-wiz.com/document/regex)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | //iframe 제거  $STRING = preg\_replace("!<iframe(.\*?)<\/iframe>!is","",$STRING);      //script 제거     $STRING = preg\_replace("!\<script(.\*?)<\/script>!is","",$STRING);         //meta 제거     $STRING = preg\_replace("!<meta(.\*?)>!is","",$STRING);         //style 태그 제거     $STRING = preg\_replace("!<style(.\*?)<\/style>!is","",$STRING);         //연속된 공백 1개로     $STRING = preg\_replace("/\s{2,}/"," ",$STRING);         //태그안에 style= 속성 제거     $STRING = preg\_replace("/ style=([^\"\']+) /"," ",$STRING); // style=border:0... 따옴표가 없을때     $STRING = preg\_replace("/ style=(\"|\')?([^\"\']+)(\"|\')?/","",$STRING); // style="border:0..." 따옴표 있을때         //태그안의 width=, height= 속성 제거     $STRING = preg\_replace("/ width=(\"|\')?\d+(\"|\')?/","",$STRING);     $STRING = preg\_replace("/ height=(\"|\')?\d+(\"|\')?/","",$STRING);       //url 을 링크로 처리     $status\_text = 'Click My URL :  <http://www.shop-wiz.com>';     $m = '|([\w\d]\*)\s?(https?://([\d\w\.-]+\.[\w\.]{2,6})[^\s\]\[\<\>]\*/?)|i';     $r = '$1 <a href="$2">$3</a>';     $status\_text = preg\_replace($m,$r,$status\_text);     echo $status\_text;       //url을 링크처리(간단버전)     $status\_text = 'Click My URL :  <http://www.shop-wiz.com>';     $m = '/(https?:\/\/([\d\w\.-]\*))/i';     //$m = '/(https?:\/\/(\S\*))/i';     $r = '<a href="$1">$2</a>';       $status\_text = preg\_replace($m,$r,$status\_text);     echo $status\_text;       //url to email     $status\_text = 'Click My Email :  wangta69@naver.com';     $m = '/[\s:]?([\w\d.]\*@[\S]\*)/i';     $r = '<a href="mailto:$1">$1</a>';     $status\_text = preg\_replace($m,$r,$status\_text);       // 날짜 분리     $date = "2014-10-10 18:16:30";     $match = preg\_split("/[- :]+/", $date);     print\_r($match);       //img 태그 추출 src 추출       preg\_match("/<img[^>]\*src=[\"']?([^>\"']+)[\"']?[^>]\*>/i",$STRING,$RESULT);     preg\_match\_all("/<img[^>]\*src=[\"']?([^>\"']+)[\"']?[^>]\*>/i",$STRING,$RESULT);           //src뒤에 "http 또는 'http 또는 http로 시작하지 않는 경우에만 $path를 삽입합니다.       $path="<http://www.naver.com>";     $body="<img src=1.jpg>";     $body=preg\_replace("/<img(.\*)[sS][rR][cC]=((?!'http|http|\"http).\*)\s/U", "<img$1src=$path$2 ", $body); |

양식의 맨 위