

## Regular Expressions

### Regular Expressions

A regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or *string matching*. Regular expressions originated in 1956, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called *regular sets*. Python has a library called `re` to help:

```
# PROGRAM MatchingPatterns:
import re
SearchString = "hello world"
pattern      = "hello world"
IsMatch = re.match(pattern, SearchString)
if IsMatch == True:
    print("regex matches")
# ENDIF;
# END.
```

This program compares `SearchString` to `pattern`, and if it matches, it prints out the phrase "regex matches".

### Basic Patterns

**Logical OR:** A vertical bar separates alternatives. For example, `gray|grey` can match "gray" or "grey".

**Grouping:** Parentheses are used to define the scope and precedence of the operators. For example, `gr(a|e)y`

**?:** indicates zero or one occurrences of the preceding element. For example, `colour?r` matches both "color" and "colour".

**\***: indicates zero or more occurrences of the preceding element. For example, `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on.

**+**: indicates one or more occurrences of the preceding element. For example, `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".

**{n}**: The preceding item is matched exactly *n* times.

**{min,}**: The preceding item is matched *min* or more times.

**{min,max}**: The preceding item is matched at least *min* times, but not more than *max* times.

### Basic Pattern Matching

```
'hello world' matches 'hello world'
'hello world' matches 'hello worl'
'hello world' does not matche 'ello world'
```

### Matching Single Characters

```
'hello world' matches 'hel.o world'
'helo world' matches 'hel.o world'
'hel o world' matches 'hel.o world'
'helo world' does not match 'hel.o world'

'hello world' matches 'hel[lp]o world'
'helo world' matches 'hel[lp]o world'
'helPo world' does not match 'hel[lp]o world'

'hello world' does not match 'hello [a-z] world'
'hello b world' matches 'hello [a-z] world'
'hello B world' matches 'hello [a-zA-Z] world'
'hello 2 world' matches 'hello [a-zA-Z0-9] world'
```

### Special Characters

```
'.' matches pattern '\.'
 '[' matches pattern '\['
 ']' matches pattern '\]'
 '(' matches pattern '\('
 ')' matches pattern '\)'
```

### Example Matches

```
'{abc}' matches '\(abc\)'
'1a' matches '\s\d\w'
't5n' does not match '\s\d\w'
'5n' matches '\s\d\w'
```

## Regular Expressions

### Matching Multiple Characters

The asterisk (\*) character says that the previous character can be matched zero or more times.

```
'hello' matches 'hel*o'
'heo' matches 'hel*o'
'hellllo' matches 'hel*o'
```

The pattern `[a-z]*` matches any collection of lowercase words, including the empty string:

```
'A string.' matches '[A-Z][a-z]*[a-z]*\.'
'No .' matches '[A-Z][a-z]*[a-z]*\.'
'' matches '[a-z]*.*'
```

The plus (+) sign in a pattern behaves similarly to an asterisk; it states that the previous character can be repeated one or more times, but, unlike the asterisk is not optional:

```
'0.4' matches '\d+\.\d+'
'1.002' matches '\d+\.\d+'
'1.' does not match '\d+\.\d+'
```

The question mark (?) ensures a character shows up exactly zero or one times, but not more.

```
'1%' matches '\d?\d%'
'99%' matches '\d?\d%'
'999%' does not match '\d?\d%'
```

If we want to check for a repeating sequence of characters, by enclosing any set of characters in parenthesis, we can treat them as a single pattern:

```
'abccc' matches 'abc{3}'
'abccc' does not match '(abc){3}'
'abcabcabc' matches '(abc){3}'
```

### Two Further Patterns

**^:** The start of a string.

**\$:** The end of a string.

### More Complex Patterns

Combining the patterns together allows us to expand our pattern-matching repertoire:

```
'Eat.' matches
'[A-Z][a-z]*([a-z]+)*\.$'
'Eat more good food.' matches
'[A-Z][a-z]*([a-z]+)*\.$'
'A good meal.' matches
'[A-Z][a-z]*([a-z]+)*\.$'
```

### RegEx for a Valid e-mail Format

The regular expression that can be used to represent a valid e-mail is as follows:

```
pattern = "^([a-zA-Z.] + @ ([a-z.] * \. [a-z.] +) $"
```

### More re Methods

In addition to the `match()` function, the `re` module provides a couple other useful functions, `search()`, and `findall()`.

- The `search()` function finds the first instance of a matching pattern, relaxing the restriction that the pattern start at the first letter.
- The `findall()` function behaves similarly to `search()`, except that it finds all non-overlapping instances of the matching pattern, not just the first one.

```
>>> import re

>>> re.findall('a.', 'abacadefagah')
['ab', 'ac', 'ad', 'ag', 'ah']

>>> re.findall('a(.)', 'abacadefagah')
['b', 'c', 'd', 'g', 'h']

>>> re.findall('(a)(.)', 'abacadefagah')
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'g'), ('a', 'h')]

>>> re.findall('((a)(.))', 'abacadefagah')
[('ab', 'a', 'b'), ('ac', 'a', 'c'), ('ad', 'a', 'd'), ('ag', 'a', 'g'), ('ah', 'a', 'h')]
```