

<u>AN INTRODUCTION TO REGULAR EXPRESSIONS.....</u>	<u>2</u>
HISTORY	3
REGULAR EXPRESSIONS AND UNICODE	3
THE POWER OF REGULAR EXPRESSIONS	3
USES FOR REGULAR EXPRESSIONS	4
REGULAR EXPRESSIONS TERMINOLOGY.....	5
DIFFERENT REGULAR EXPRESSION ENGINES	5
REGULAR EXPRESSION TOOLS AND LINKS	5
THE ADVANTAGE OF REGULAR EXPRESSIONS.....	7
REGULAR EXPRESSION SYNTAX.....	7
THE FILE NAME ANALOGY.....	8
THE LANGUAGE ANALOGY.....	8
BUILD A REGULAR EXPRESSION.....	8
ORDER OF PRECEDENCE.....	8
ORDINARY CHARACTERS	9
REGEX BRIEF	9
LITERAL CHARACTERS.....	9
CHARACTER CLASSES OR CHARACTER SETS	10
SHORTHAND CHARACTER CLASSES	10
NON-PRINTABLE CHARACTERS.....	10
THE DOT MATCHES (ALMOST) ANY CHARACTER	11
ANCHORS	11
ALTERNATION	11
REPETITION	11
GROUPING AND BACKREFERENCES	12
REGEXP OBJECT	12
REGEXP OBJECT, PROPERTIES AND METHODS	12
RegExp.Global Property	12
RegExp.IgnoreCase Property	13
RegExp.Pattern Property	13
RegExp.MultiLine Property	14
RegExp.Execute Method.....	14
RegExp.Test Method	15
RegExp.Replace Method.....	15
MATCHES COLLECTION OBJECT.....	16
MATCHES COLLECTION OBJECT PROPERTIES	16
Matches.Count Property.....	16
Matches.Item Property	16
SUBMATCHES COLLECTION OBJECT	17
SUBMATCHES COLLECTION OBJECT PROPERTIES.....	17
SubMatches.Count Property.....	17
SubMatches.Item Property	17
MATCH OBJECT	18
MATCH OBJECT PROPERTIES	18
Match.FirstIndex Property.....	18
Match.Length Property	18
Match.Value Property	19
HOW TO USE THE VBSCRIPT REGEXP OBJECT?.....	19

SIMPLE PATTERNS	19
START AND END OF LINE (ANCHORS)	19
MATCHING ANY ONE OF SEVERAL CHARACTERS (CHAR CLASSES)	20
NEGATED CHARACTER CLASSES	20
MATCHING ANY CHARACTER – DOT	21
MATCHING ANY ONE OF SEVERAL SUBEXPRESSIONS	21
OPTIONAL ITEMS	22
REPETITION	23
LIMITING REPETITION	23
Q&A	23
TIME OF DAY, SUCH AS "9:17 AM" OR "12:30 PM"	23
VBSCRIPT FUNCTION	24
USEFULL EXAMPLES	25
Matching a floating point number	25
Matching a Valid Date	26
Trimming Whitespace	26
U.S. Currency	26
Five Integer US ZIP Code	26
Title Alphanumeric	27
Visa Credit Card	27
ISBN-10	27
IP Address	27
Strong Password	27
APPENDIX 5.A	27
Position Matching	27
Literals	28
Character Classes	28
Repetition	29
Alternation & Grouping	30
Backreferences	30
Interval Settings	30

An Introduction to Regular Expressions.¹

A **regular expression** is a formula for matching strings that follow some pattern. Many people are afraid to use them because they can look confusing and complicated. Unfortunately, nothing in this write up can change that. However, I have found that with a bit of practice, it's pretty easy to write these complicated expressions. Plus, once you get the hang of them, you can reduce hours of laborious and error-prone text editing down to minutes or seconds.

A regular expression (abbreviated as **regexp** or **regex**, with plural forms **regexps**, **regexes**, or **regexen**) is a string that describes or matches a set of strings, according to certain syntax rules. Regular expressions are used by many text editors and utilities to search and manipulate bodies of text based on certain patterns. Many programming languages support regular expressions for string manipulation. For example, **Perl**, and **Tcl** have a powerful regular expression engine built directly into their syntax. The set of utilities (including the

Dani Vainstein

¹ http://en.wikipedia.org/wiki/Regular_expression

editor sed and the filter grep) provided by **Unix** distributions were the first to popularize the concept of regular expressions.

History

The origins of regular expressions lies in automata theory and formal language theory, both of which are part of theoretical computer science. These fields study models of computation (automata) and ways to describe and classify formal languages. In the 1940s, **Warren McCulloch** and **Walter Pitts** described the nervous system by modelling neurons as small simple automata. The mathematician **Stephen Kleene** later (mid 1950s) described these models using his mathematical notation called regular sets. **Ken Thompson** built this notation into the editor **QED**, and then into the **Unix** editor **ed**, which eventually led to **grep**'s use of regular expressions. Ever since that time, regular expressions have been widely used in **Unix** and **Unix-like** utilities such as: **expr**, **awk**, **Emacs**, **vi**, **lex**, and **Perl**.

Perl and **Tcl** regular expressions were derived from **regex** written by **Henry Spencer**. **Philip Hazel** developed **PCRE** (Perl Compatible Regular Expressions) which attempts to closely mimic Perl's regular expression functionality, and is used by many modern tools such as **PHP**, **ColdFusion**, and **Apache**. Part of the effort in the design of the future **Perl6** is to improve Perl's regular expression integration. This is the subject of **Apocalypse 5**.

The use of regular expressions in structured information standards (for document and database modeling) was very important, started in the 1960s, and expanded in the 1980s, when industry standards like **ISO SGML** (precursored by ANSI "GCA 101-1983") consolidated. The kernel of the "structure specification languages" of these standards are regular expressions. The more simple and evident use are in the **DTD** element group syntax.

Regular expressions and Unicode

Regular expressions were originally used with **ASCII** characters. Many regular expression engines can now handle **Unicode**. In most respects it makes no difference what the character set is, but certain issues do arise in the extension of regular expressions to **Unicode**.

One issue is which **Unicode** format is supported. All command-line regular expression engines expect **UTF-8**, but regular expression libraries vary. Some expect **UTF-8**, while others expect other encodings of **Unicode** (**UTF-16**, obsolete **UCS-2** or **UTF-32**).

A second issue is whether the full Unicode range is supported. Many regular expression engines support only the Basic Multilingual Plane, that is, the characters encodable in only 16 bits. Only a few regular expression engines can at present handle the full 21 bit Unicode range.

The Power of Regular Expressions

There's a good reason that regular expressions are found in so many diverse applications: they are extremely powerful. At a low level, a regular expression

describes a chunk of text. You might use it to verify a user's input, or perhaps to sift through large amounts of data. On a higher level, regular expressions allow you to master your data. Control it. Put it to work for you. To master regular expressions is to master your data.

Think about how you search for files on your hard disk. You most likely use the ? and * characters to help find the files you're looking for. The ? character matches a single character in a file name, while the * matches zero or more characters. A pattern such as 'data?.dat' would find the following files:

- data1.dat
- data2.dat
- datax.dat
- dataN.dat

Using the * character instead of the ? character expands the number of files found. 'data*.dat' matches all of the following:

- data.dat
- data1.dat
- data2.dat
- data12.dat
- datax.dat
- dataXYZ.dat

While this method of searching for files can certainly be useful, it is also very limited. The limited ability of the ? and * wildcard characters give you an idea of what regular expressions can do, but regular expressions are much more powerful and flexible.

Uses For Regular Expressions

In a typical search and replace operation, you must provide the exact text you are looking for. That technique may be adequate for simple search and replace tasks in static text, but it lacks flexibility and makes searching dynamic text difficult, if not impossible. With regular expressions, you can:

- Test for a pattern within a string. For example, you can test an input string to see if a telephone number pattern or a credit card number pattern occurs within the string. This called data validation.
- Replace text. You can use a regular expression to identify specific text in a document and either remove it completely or replace it with other text.
- Extract a substring from a string based upon a pattern match. You can find specific text within a document or input field

In **QuickTest** Regular Expressions can be used in 3 places:

- Object Repository
- CheckPoints
- **RegExp** object.

Regular Expressions Terminology

Basically, a regular expression is a **pattern** describing a certain amount of text. Their name comes from the mathematical theory on which they are based. As described. Since most people are lazy to type, you will usually find the name abbreviated to **regex** or **regexp**.

This first example is actually a perfectly valid **regex**. It is the most basic pattern, simply matching the literal text **regex**. A "match" is the piece of text, or sequence of bytes or characters that pattern was found to correspond to by the **regex** processing software.

The pattern : `\b[A-Z0-9._%~]+@[A-Z0-9._%~]+\.[A-Z]{2,4}\b` is a more complex pattern. It describes a series of letters, digits, dots, percentage signs and underscores, followed by an at sign, followed by another series of letters, digits, dots, percentage signs and underscores, finally followed by a single dot and between two and four letters. In other words: this pattern describes an email address.

With the above regular expression pattern, you can search through a text file to find email addresses, or verify if a given string looks like an email address.

Different Regular Expression Engines

A regular expression "engine" is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do not access the engine directly. Rather, the application will invoke it for you when needed, making sure the right regular expression is applied to the right file or data.

As usual in the software world, different regular expression engines are not fully compatible with each other. It is not possible to describe every kind of engine and regular expression syntax.

Many more recent **regex** engines are very similar, but not identical, to the one of **Perl 5**. Examples are the **open source PCRE** engine (used in many tools and languages like **PHP**), the **.NET regular expression library**, and the regular expression package included with version 1.4 and later of the **Java JDK**.

VBScript regular expression engine has been implemented as a **COM** object.

Regular Expression Tools and Links

In this chapter, I will use the regular expression examples by using the **RegexBuddy** Application.

It's a very powerfull tool, to analyze and test, complex and simple regular expressions. The tool can be downloaded at www.regexbuddy.com

Another Tool is **The RegEx-Coach** application can be downloaded from <http://weitz.de/regex-coach/>

The Advantage of Regular Expressions

An obvious advantage of regular expression we'll see in the following example.

The pattern : **reg(ular expressions?)|ex(p|es)?** with **IgnoreCase** is a pattern to search in a text file, the regular expression fits the following expressions:

- Regular Expression
- regular expressions
- regex
- RegExp
- Regexes

If we only had plain text search, we would have needed 5 searches. With **RegExp**, we need just one search.

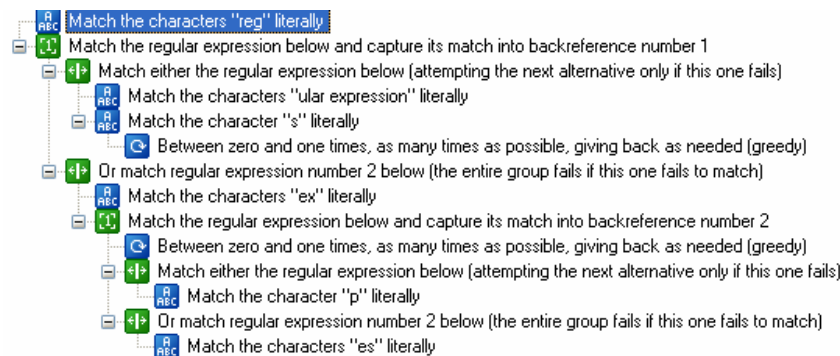


Figure 2 - Searching a pattern

Regular Expression Syntax

A regular expression is a pattern of text that consists of ordinary characters (for example, letters a through z) and special characters, known as metacharacters. The pattern describes one or more strings to match when searching a body of text. The regular expression serves as a template for matching a character pattern to the string being searched.

Here are some examples of regular expression you might encounter:

Regular Expr.	Matches
"^[\ \t]*\$"	Match a blank line.
"\d{2}-\d{5}"	Validate an ID number consisting of 2 digits, a hyphen, and another 5 digits.
"<(.*?)>.*<\1>"	Match an HTML tag.

The File Name Analogy

You know that report.txt is a specific filename, but if you have had any experience with Unix or DOS/Windows, you also know that the pattern "*.txt" can be used to select multiple files. With such filename patterns like this there are a few characters* that have special meanings. The star means "match anything," and a question mark (?) means "match any one character." With "*.txt", we start with a match-anything * and end with the literal .txt , so we end up with a pattern that means "select the files whose names start with anything and end with .txt".

if a particular need is specific enough, such as "selecting files," you can develop a specialized scheme or tool. However, over the years, a generalized pattern language has developed which is powerful and expressive for a wide variety of uses. Each program implements and uses them differently, but in general, this powerful pattern language and the patterns themselves are called regular expressions.

The Language Analogy

Full regular expressions are composed of two types of characters. The special characters (like the * from the filename analogy) are called **metacharacters**, while everything else are called **literal**, or **normal text characters**. What sets regular expressions apart from filename patterns is the scope of power their **metacharacters** provide. Filename patterns provide limited metacharacters for limited needs, but a regular expression "language" provides rich and expressive **metacharacters** for advanced uses.

Build a Regular Expression

Regular expressions are constructed in the same way that arithmetic expressions are created. That is, small expressions are combined using a variety of metacharacters and operators to create larger expressions.

You construct a regular expression by putting the various components of the expression pattern between a pair of delimiters. a pair of quotation marks (") delimit regular expressions. For example:

```
"expression"
```

the regular expression pattern (expression) is stored in the **Pattern** property of the **RegExp** object.

The components of a regular expression can be individual characters, sets of characters, ranges of characters, choices between characters, or any combination of all of these components.

Order of Precedence

Once you have constructed a regular expression, it is evaluated much like an arithmetic expression, that is, it is evaluated from left to right and follows an order of precedence.

Operator	Description
\	Escape
(), (?:), (?=), []	Parentheses and Brackets
*, +, ?, {n}, {n,}, {n,m}	Quantifiers
^, \$, \anymetacharacter	Anchors and Sequences
	Alternation

Table 1 – Order Preference Characters

Characters have higher precedence than the alternation operator, which allows 'm|food' to match "m" or "food". To match "mood" or "food", use parentheses to create a subexpression, which results in '(m|f)ood'.

Ordinary Characters

Ordinary characters consist of all printable and non-printable characters that are not explicitly designated as metacharacters. This includes all uppercase and lowercase alphabetic characters, all digits, all punctuation marks, and some symbols.

The simplest form of a regular expression is a single, ordinary character that matches itself in a searched string. For example, the single-character pattern 'A' matches the letter 'A' wherever it appears in the searched string. Here are some examples of single-character regular expression patterns:

Expression	Equivalent
/a/	"a"
/7/	"7"
/M/	"M"
/a7M/	"a7M"

Table 2 - Ordinary Characters

Notice that there is no concatenation operator. All that is required is that you just put one character after another.

Regex Brief

Literal Characters

The most basic regular expression consists of a single literal character, e.g.: a. It will match the first occurrence of that character in the string. If the string is *Jack* is a boy, it will match the **a** after the **j**.

This regex can match the second **a** too. It will only do so when you tell the regex engine to start searching through the string after the first match. In a text editor, you can do so by using its "Find Next" or "Search Forward" function. In a programming language, there is usually a separate function that you can call to continue searching through the string after the previous match.

Eleven characters with special meanings: the opening square bracket `[`, the backslash `\`, the caret `^`, the dollar sign `$`, the period or dot `.`, the vertical bar or pipe symbol `|`, the question mark `?`, the asterisk or star `*`, the plus sign `+`, the opening round bracket `(` and the closing round bracket `)`. These special characters are often called "metacharacters".

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match `1+1=2`, the correct regex is `1\+1=2`. Otherwise, the plus sign will have a special meaning.

Character Classes or Character Sets

A "character class" matches only one out of several characters. To match an `a` or an `e`, use `[ae]`. You could use this in `gr[ae]y` to match either `gray` or `grey`. A character class matches only a single character. `gr[ae]y` will not match `graay`, `graey` or any such thing. The order of the characters inside a character class does not matter.

You can use a hyphen inside a character class to specify a range of characters. `[0-9]` matches a single digit between 0 and 9. You can use more than one range. `[0-9a-fA-F]` matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. `[0-9a-fxA-FX]` matches a hexadecimal digit or the letter `X`.

Typing a caret after the opening square bracket will negate the character class. The result is that the character class will match any character that is not in the character class. `q[^x]` matches `qu` in `question`. It does not match `Iraq` since there is no character after the `q` for the negated character class to match.

Shorthand Character Classes

`\d` matches a single character that is a digit, `\w` matches a "word character" (alphanumeric characters plus underscore), and `\s` matches a whitespace character (includes tabs and line breaks). The actual characters matched by the shorthands depends on the software you're using. Usually, non-English letters and numbers are included.

Non-Printable Characters

You can use special character sequences to put non-printable characters in your regular expression. Use `\t` to match a tab character (ASCII 0x09), `\r` for carriage return (0x0D) and `\n` for line feed (0x0A). More exotic non-printables are `\a` (bell, 0x07), `\e` (escape, 0x1B), `\f` (form feed, 0x0C) and `\v` (vertical tab, 0x0B). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

Use `\xFF` to match a specify character by its hexadecimal index in the character set. E.g. `\xA9` matches the copyright symbol in the Latin-1 character set.

If your regular expression engine supports Unicode, use `\uFFFF` to insert a Unicode character. E.g. `\u20A0` matches the euro currency sign.

All non-printable characters can be used directly in the regular expression, or as part of a character class.

The Dot Matches (Almost) Any Character

The dot matches a single character, except line break characters. It is short for `[^\n]` (UNIX regex flavors) or `[^\r\n]` (Windows regex flavors). Most regex engines have a "dot matches all" or "single line" mode that makes the dot match any single character, including line breaks.

`gr.y` matches `gray`, `grey`, `gr%oy`, etc. Use the dot sparingly. Often, a character class or negated character class is faster and more precise.

Anchors

Anchors do not match any characters. They match a position. `^` matches at the start of the string, and `$` matches at the end of the string. Most regex engines have a "multi-line" mode that makes `^` match after any line break, and `$` before any line break. E.g. `^b` matches only the first `b` in `bob`.

`\b` matches at a word boundary. A word boundary is a position between a character that can be matched by `\w` and a character that cannot be matched by `\w`. `\b` also matches at the start and/or end of the string if the first and/or last characters in the string are word characters. `\B` matches at every position where `\b` cannot match

Alternation

Alternation is the regular expression equivalent of "or". `cat|dog` will match `cat` in `About cats and dogs`. If the regex is applied again, it will match `dog`. You can add as many alternatives as you want, e.g.: `cat|dog|mouse|fish`.

Alternation is also good to test multilingual applications i.e.

We have a button **Yes** in English, **Si** in Spanish, **Oui** in French, **Ja** in Danish or German, we can use the next regular expression `Yes|Si|Oui|Da|Ja`

Repetition

The question mark makes the preceding token in the regular expression optional. E.g.: `colou?r` matches `colour` or `color`

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. `<[A-Za-z][A-Za-z0-9]*>` matches an HTML tag without any attributes. `<[A-Za-z0-9]+>` is easier to write but matches invalid tags such as `<1>`.

Use curly braces to specify a specific amount of repetition. Use `[1-9][0-9]{3}` to match a number between 1000 and 9999. `[1-9][0-9]{2,4}` matches a number between 100 and 99999.

Grouping and Backreferences

Place round brackets around multiple tokens to group them together. You can then apply a quantifier to the group. E.g. `Set(Value)?` matches **Set** or **SetValue**

Round brackets create a capturing group. The above example has one group. After the match, group number one will contain nothing if **Set** was matched or `Value` if **SetValue** was matched. How to access the group's contents depends on the software or programming language you're using. Group zero always contains the entire regex match.

Use the special syntax `Set(?:Value)?` to group tokens without creating a capturing group. This is more efficient if you don't plan to use the group's contents. Do not confuse the question mark in the non-capturing group syntax with the quantifier.

Regexp Object

The **RegExp** object provides support for regular expression matching; for the ability to search strings for substrings matching general or specific patterns.

In order to conduct a pattern search, you must first instantiate the regular expression object, with code like the following:

```
Dim oRegExp           ' Instance of RegExp object
Set oRegExp = New RegExp
```

To conduct a search using the **RegExp** object, do the following:

- Determine whether the search should be case-sensitive.
- Determine whether all instances or just the first instance of the substring should be returned.
- Supply the pattern string that you want to find.
- Provide a string that the **RegExp** object is to search.

The **RegExp** object allows you to search for a substring that matches your pattern string in any of three ways:

- You can determine whether a pattern match is found in the string.
- You can return one or all of the occurrences of the matching substrings. In this case, results are returned in **Match** objects within the **Matches** collection.
- You can replace all substrings matching the pattern string with another string.

RegExp object, Properties and Methods

RegExp.Global Property

Description

The **Global** property determines whether the search for a pattern string should match all occurrences in the search string or just the first one.

Note

- The value of the **Global** property is **True** if the search applies to the entire

string, **False** if it does not. Default is **False**.

- A search will attempt to locate only the first occurrence of the pattern string in a search string; that is, the default value of the **Global** property is **False**.
- If you want to search for all occurrences of the pattern string in the search string, you must set the **Global** property to **True**.

Tip

- If you're interested only in determining whether the pattern string exists in the search string, there's no point in overriding the **Global** property's default value of **False**.

RegExp.IgnoreCase Property

Description

The **IgnoreCase** property determines whether the search for a pattern string is case-sensitive.

Note

- The value of the **IgnoreCase** property is **True** if the search applies to the entire string, **False** if it does not. Default is **False**.
- By default, regular expression searches are case-sensitive; that is, the default value of the **IgnoreCase** property is **False**.
- If you don't want the search to be case-sensitive, you must set the **IgnoreCase** property to **True**.

Tip

If your search string does not attempt to match any alphabetic characters (A-Z and a-z), you can safely ignore the setting of **IgnoreCase**, since it won't affect the results of your search.

RegExp.Pattern Property

Description

The **Pattern** property contains a pattern string that defines the substring to be found in a search string.

Note

- By default, regular expression searches are case-sensitive; that is, the default value of the **IgnoreCase** property is **False**.
- If you don't want the search to be case-sensitive, you must set the **IgnoreCase** property to **True**.
- The pattern can be one or a combination of metacharacters and characters
 - [Ordinary characters](#), see in Table 2 on page 9
 - [Position Matching](#), see in Table 3 on page 28
 - [Literals](#), see in Table 4 on page 28
 - [Character Classes](#), see in Table 5 on page 29
 - [Repetition](#), see in Table 6 on page 29

- [Alternation & Grouping](#), see in Table 7 on page 30
- [Backreferences & Grouping](#), Table 7 on page 30

RegExp.MultiLine Property

Description

The **MultiLine** property A read-only Boolean property that reflects whether or not to search strings across multiple lines.

Note

- The value of multiline is true if multiple lines are searched, false if searches must stop at line breaks.
- This property only influences the interpretation of ^ and \$ in a regular expression.
- Because multiline is static, it is not a property of an individual regular expression object. Instead, you always use it as RegExp.multiline.

RegExp.Execute Method

Description

The **Execute** method executes a regular expression search against a specified string.

Syntax

```
object.Execute(string)
```

Arguments

Parameter	Description
<i>string</i>	Required. The text string upon which the regular expression is executed.

Return Value

A **Matches** collection containing one or more **Match** objects.

Note

- The method searches string using the **RegExp** object's **Pattern** property.
- The results are returned in the **Matches** collection, which is a collection of **Match** objects.
- If the search finds no matches, the **Matches** collection is empty.

Tip

- Remember to use the **Set** statement to assign the **Matches** collection returned by the **Execute** method to an object variable.
- You can determine whether the **Matches** collection returned by the **Execute** method is empty by examining its **Count** property. It is empty if the value of **Count** is 0.

RegExp.Test Method

Description

The **Test** method performs a regular expression search against *string* and indicates whether a match was found

Syntax

```
object.Test(string)
```

Arguments

Parameter	Description
<i>string</i>	Required. The text string upon which the regular expression is executed.

Return Value

A **Boolean** indicating whether a match was found.

Note

- Prior to calling the **Test** method, the search string should be defined by setting the **Pattern** property.
- The method searches string using the **RegExp** object's **Pattern** property.
- The method returns **True** if the search succeeds and **False** otherwise.

Tip

- Since a search is successful if one match is found, you do not have to set the **RegExp** object's **Global** property before calling the **Test** method.
- You can use the method to determine whether a match exists before calling either the **Execute** or the **Replace** methods.
- When validating user input, you'll typically want to check if the entire string matches the regular expression. To do so, put a **caret** at the start of the regex, and a **dollar** at the end, to anchor the regex at the start and end of the subject string.

RegExp.Replace Method

Description

The **Replace** method performs a regular expression search against *searchString* and replaces matched substrings with *replaceString*.

Syntax

```
object.Replace(searchString, replaceString)
```

Arguments

Parameter	Description
<i>searchString</i>	Required. The text string in which the text replacement is to occur.
<i>replaceString</i>	Required. The replacement text string.

Return Value

A **String** containing the entire string that results when matched substrings in *searchString* are replaced with *replaceString*.

Note

- The method searches *searchString* using the **RegExp** object's **Pattern** property.
- If no matches are found, the method returns *searchString* unchanged.

Tip

- *replaceString* the replacement string, can contain pattern strings that control how substrings in *searchString* should be replaced.

Matches Collection Object

The **Matches Collection** is a collection of objects that contains the results of a search and match operation that uses a regular expression.

Simply put, a regular expression is a string pattern that you can compare against all or a portion of another string. However, in all fairness, be warned that regular expressions can get very complicated.

The **RegExp** object can be used to search for and match string patterns in another string. A **Match** object is created each time the **RegExp** object finds a match. Since, zero or more matches could be made, the **RegExp** object actually return a collection of **Match** objects which is referred to as a **Matches** collection.

Matches collection object Properties

Matches.Count Property

Description

The **Count** property returns an integer that tells us how many **Match** objects there are in the **Matches** Collection.

Note

- Indicates the number of objects in the collection. A value of zero indicates that the collection is empty. The property is read-only.

Matches.Item Property

Description

The **Item** property allows us to retrieve the value of an item in the collection designated by the specified key argument and also to set that value by using *itemvalue*.

Syntax

```
object.Item(index)
```

Arguments

Parameter	Description
<i>index</i>	Required. The item index

Return Value

A **Match** object

Note

- A read-only value that enables **Match** objects to be randomly accessed from the **Matches** collection object.
- The **Match** objects may also be incrementally accessed from the **Matches** collection object, using a **For-Next** loop.

SubMatches Collection Object

A **SubMatches** collection contains individual submatch strings, and can only be created using the **Execute** method of the **RegExp** object. The **SubMatches** collection's properties are read-only

When a regular expression is executed, zero or more submatches can result when subexpressions are enclosed in capturing parentheses.

Each item in the **SubMatches** collection is the string found and captured by the regular expression.

SubMatches collection object Properties

SubMatches.Count Property

Description

The **Count** property returns an integer that tells us how many **Match** objects there are in the **SubMatches** Collection.

Note

- Indicates the number of objects in the collection. A value of zero indicates that the collection is empty. The property is read-only.

SubMatches.Item Property

Description

The **Item** property allows us to retrieve the value of an item in the collection designated by the specified key argument and also to set that value by using itemvalue.

Syntax

```
object.Item(index)
```

Arguments

Parameter	Description
-----------	-------------

<i>index</i>	Required. takes an index parameter, and returns the text matched by the capturing group. The Item property is the default member.
--------------	---

Return Value

A Variant.

Note

- Each SubMatch, will only hold values if your regular expression has capturing groups.

Match Object

The **Match** object is used to access the three read-only properties associated with the results of a search and match operation that uses a regular expression.

Simply put, a regular expression is a string pattern that you can compare against all or a portion of another string. However, in all fairness, be warned that regular expressions can get very complicated.

The **RegExp** object can be used to search for and match string patterns in another string. A **Match** object is created each time the **RegExp** object finds a match. Since, zero or more matches could be made, the **RegExp** object actually return a collection of **Match** objects which is referred to as a **Matches** collection.

Match object Properties

Figure 3 – Match object

Match.FirstIndex Property

Description

The **FirstIndex** property returns the position, counted from the left with the first position being numbered zero, in a string where a match was made.

Note

- Returns a Long data type.
- Indicates the position in the original search string where the regular expression match occurred. The first character in the search string is at position 1.

Match.Length Property

Description

The **Length** property returns the length of the matched text found in a search string.

Note

- Returns a Long data type.
- Indicates the number of characters in the match found in the search string. This is also the number of characters in the **Match** object's Value property.

Match.Value Property

Description

The **Value** property returns the actual text that was matched during the search.

Note

- Returns a **Long** data type.
- The text of the match found in the search string.

How to Use the VBScript RegExp Object?

The advantage of the **RegExp** object's bare-bones nature is that it's very easy to use. Create one, put in a regex, and let it match or replace. Only four properties and three methods are available.

Set oRegExp = **New** RegExp

- After creating the object, assign the regular expression you want to search for to the **Pattern** property.
- If you want to use a literal regular expression rather than a user-supplied one, simply put the regular expression in a double-quoted string. By default, the regular expression is case sensitive.
- Set the **IgnoreCase** property to **True** to make it case insensitive.
- You can make the caret and dollar match at the start and the end of those lines by setting the **Multiline** property to **True**. **VBScript** does not have an option to make the dot match line break characters.
- Finally, if you want the **RegExp** object to return or replace all matches instead of just the first one, set the **Global** property to **True**.

Simple Patterns

We'll start by learning about the simplest possible regular expressions. Since regular expressions are used to operate on strings, we'll begin with the most common task: matching characters.

Start and End of Line (anchors)

Probably the easiest metacharacters to understand are **^** (caret) and **\$** (dollar), which represent the start and end, respectively, of the line of text as it is being checked.

Anchors are a different breed. They do not match any character at all. Instead, they match a position before, after or between characters. They can be used to "anchor" the regex match at a certain position. The caret **^** matches the position

before the first character in the string. Applying `^a` to `abc` matches `a`. `^b` will not match `abc` at all, because the `b` cannot be matched right after the start of the string, matched by `^`.

The regular expression `cat` finds "cat" anywhere on the line, but `^cat` matches only if the "cat" is at the beginning of the line; the `^` is used to effectively anchor the match (of the rest of the regular expression) to the start of the line. Similarly, `cat$` finds "cat" only at the end of the line, such as a line ending with `scat`.

Get into the habit of interpreting regular expressions in a rather literal way. For example, don't think `^cat` matches a line with `cat` at the beginning but rather: `^cat` matches if you have the beginning of a line, followed immediately by `c`, followed immediately by `a`, followed immediately by `t`.

The caret and dollar are particular in that they match a position in the line rather than any actual text characters themselves. There are, of course, various ways to actually match real text. Besides providing literal characters in your regular expression, you can also use some of the items discussed in the next few sections.

The following represent an empty line

Matching any one of several characters (char classes)

Let's say you want to search for "grey," but also want to find it if it were spelled "gray". The `[...]` construct, usually called a character class, lets you list the characters you want to allow at that point: `gr[ea]y`. This means to find "g", followed by "r", followed by an "e" or an "a", all followed by "y".

As another example, maybe you want to allow capitalization of a word's first letter: `[Ss]mith`. Remember that this still matches lines that contain `smith` (or `Smith`) embedded within another word, such as with `blacksmith`.

You can list as many characters as you like. For example, `[123456]` matches any of the listed digits. This particular class might be useful as part of `<H[123456]>`, which matches `<H1>`, `<H2>`, `<H3>`, etc.

This can be useful when searching for HTML headers.

Within a character class, the **metacharacter** `-` (dash) indicates a range of characters: `<H[1-6]>` is identical to the previous example.

`[0-9]` and `[a-z]` are common short-hands for classes to match digits and lowercase letters, respectively. Multiple ranges are fine, so `[0123456789abcdefABCDEF]` can be written as `[0-9a-fA-F]`

Either of these can be useful when processing hexadecimal numbers `[0-9a-fA-F]`. You can even combine ranges with literal characters: `[0-9A-Z_!.?]` matches a digit, uppercase letter, underscore, exclamation point, period, or a question mark.

Negated character classes

If you use `[^...]` instead of `[...]`, the class matches any character that isn't listed. For example, `[^1-6]` matches a character that's not 1 through 6. More or less, the leading `^` in the class "negates" the list, rather than listing the characters you want to include in the class, you list the characters you don't want to be included.

You might have noticed that the `^` used here is the same as the start-of-line caret. The character is the same, but the meaning is completely different. Remember that a negated character class means "match a character that's not "listed" and not "don't match what is listed." These might seem the same, A convenient way to view a negated class is that it is simply a shorthand for a normal class which includes all possible characters except those that are listed.

is important to remember that a negated character class still must match a character. `q[^u]` does not mean: "a `q` not followed by a `u`". It means: "a `q` followed by a character that is not a `u`". like `iraqi`

Matching Any Character – Dot

The **metacharacter** `.` (usually called dot) is a shorthand for a character class that matches any character. It can be convenient when you want to have an "any character here" place-holder in your expression. For example, if you want to search for a date such as `07/04/76`, `07-04-76`, or even `07.04.76`, you could go to the trouble to construct a regular expression that uses character classes to explicitly allow `'/'`, `'-'`, or `'.'` between each number, such as `07[-./]04[-./]76`. You could also try simply using `07.04.76` .

In `07[-./]04[-./]76` , the dots are not **metacharacters** because they are within a character class. (Remember, the list of metacharacters and their meanings are different inside and outside of character classes.) The dashes are also not metacharacters, although within a character class they normally are. A dash is not special when it is the first character in the class. With `07.04.76` , the dots are metacharacters that match any character, including the dash, period, and slash that we are expecting.

`07[-./]04[-./]76` is more precise, but it's more difficult to read and write.

`07.04.76` is easy to understand, but vague. Which should we use? It all depends upon what you know about the data you are searching, and just how specific you feel you need to be.

Matching any one of several subexpressions

A very convenient **metacharacter** is `|` , which means "or" It allows you to combine multiple expressions into a single expression which matches any of the individual expressions used to make it up. For example, Bob and Robert are two separate expressions, while `Bob|Robert` is one expression that matches either. When combined this way, the **subexpressions** are called alternatives.

In the `gr[ea]y` example, it is interesting `grey|gray` , and even `gr(a|e)y` . The latter case (For the record, parentheses are metacharacters too.) Without the parentheses, `gr|ey` means " `gr` or `ey` ", which is not what we want here. Alternation reaches far, but not beyond parentheses. Another example is `(First|1st) [Ss]treet` . Actually, since both `First` and `1st` end with `st` , they can be shortened to `(Fir|1)st [Ss]treet` , but that's not necessarily quite as easy to read. Still, be sure to understand that they do mean the same thing.

Also, take care when using caret or dollar in an expression with alternation. Compare `^From|Subject|Date:` with `^(From|Subject|Date):`: Both appear similar to our earlier email example, but what each matches (and therefore how useful it is) differs greatly. The first is composed of three plain alternatives, so it will match when we have "`^From or Subject or Date: ,`" which is not particularly useful. We want the leading caret and trailing `:` to apply to each alternative.

`^(From|Subject |Date):`

- start-of-line, followed by `F·r·o·m`, followed by `': '`
- or 2) start-of-line, followed by `S·u·b·j·e·c·t`, followed by `': '`
- or 3) start-of-line, followed by `D·a·t·e` , followed by `': '`

Optional Items

Let's look at matching `color` or `colour`. Since they are the same except that one has a `u` and the other doesn't, we can use `colou?r` to match either. The **metacharacter ?** (question mark) means optional. It is placed after the character that is allowed to appear at that point in the expression, but whose existence isn't actually required to still be considered a successful match.

Unlike other metacharacters we have seen so far, the question-mark attaches only to the immediately-**preceding** item. Thus `colou?r` is interpreted as "`c` , then `o` then `l` then `o` then `u?` then `r`

The `u?` part will always be successful: sometimes it will match a `u` in the text, while other times it won't. The whole point of the `?` - optional part is that it's successful either way. This isn't to say that any regular expression that contains `?` will always be successful. For example, against 'semicolon', both `colo` and `u?` are successful (matching `colo` and nothing, respectively). However, the final `r` will fail, and that's what disallows semicolon, in the end, from being matched by `colou?r`

As another example, consider matching a date that represents July fourth, with the July part being either `July` or `Jul`, and the fourth part being fourth, 4th, or simply 4. Of course, we could just use `(July|Jul) (fourth|4th|4)` , but let's explore other ways to express the same thing. First, we can shorten the `(July|Jul)` to `(July?)` . Do you see how they are effectively the same? The removal of the `|` means that the parentheses are no longer really needed. Leaving the parentheses doesn't hurt, but `July?` , with them removed, is a bit less cluttered. This leaves us with `July? (fourth|4th|4)`

Moving now to the second half, we can simplify the `4th|4` to `4(th)?` . As you can see, `?` can attach to a parenthesized expression. Inside the parentheses can be as complex a subexpression as you like, but "from the outside" it is considered a unit. Grouping for `?` is one of the main uses of parentheses. Our expression now looks like `July? (fourth|4(th)?)` . Although there are a fair number of metacharacters, and even nested parentheses, it is not that difficult to decipher and understand.

Repetition

Similar to the question mark are **+** (plus) and *****. The **metacharacter +**, means "one or more of the immediately-preceding item," and ***** means "any number, including none, of the item." Phrased differently **...*** means "try to match it as many times as possible, but it's okay to settle for nothing if need be." The construct with plus, **+**, is similar in that it will also try to match as many times as possible, but different in that it will fail if it can't match at least once. These three metacharacters, question mark, plus, and star, are called quantifiers.

Limiting Repetition

Modern **regex** flavors, have an additional repetition operator that allows you to specify how many times a token can be repeated. The syntax is **{min,max}**, where min is a positive integer number indicating the minimum number of matches, and max is an integer equal to or greater than min indicating the maximum number of matches. If the comma is present but max is omitted, the maximum number of matches is infinite. So **{0,}** is the same as *****, and **{1,}** is the same as **+**. Omitting both the comma and max tells the engine to repeat the token exactly min times.

You could use **\b[1-9][0-9]{3}\b** to match a number between 1000 and 9999. **\b[1-9][0-9]{2,4}\b** matches a number between 100 and 99999.

Q&A

Time of day, such as "9:17 am" or "12:30 pm"

Matching a time can be taken to varying levels of strictness. Something such as **[0-9]?[0-9]:[0-9][0-9] (am|pm)** picks up both **9:17 am** and **12:30 pm**, but also allows **99:99 pm**.

Looking at the hour, we realize that if it is a two-digit number, the first digit must be a one. But **1?[0-9]** still allows an hour of **19** (and also an hour of 0), so maybe it is better to break the hour part into two possibilities: **1[012]** for two-digit hours and **[1-9]** for single-digit hours. The result is **(1[012]|[1-9])**.

The minute part is easier. The first digit should be **[0-5]**. For the second, we can stick with the current **[0-9]**. This gives **(1[012]|[1-9]):[0-5][0-9] (am|pm)** when we put it all together.

Using the same logic, can you extend this to handle 24-hour time with hours from **0** through **23**? As a challenge, allow for a leading zero, at least through to **09:59**.

There are various solutions, but we can use similar logic as before. This time, I'll break the task into three groups: one for the morning (hours 00 through 09, with the leading zero being optional), one for the daytime (hours 10 through 19), and one for the evening (hours 20 through 23)

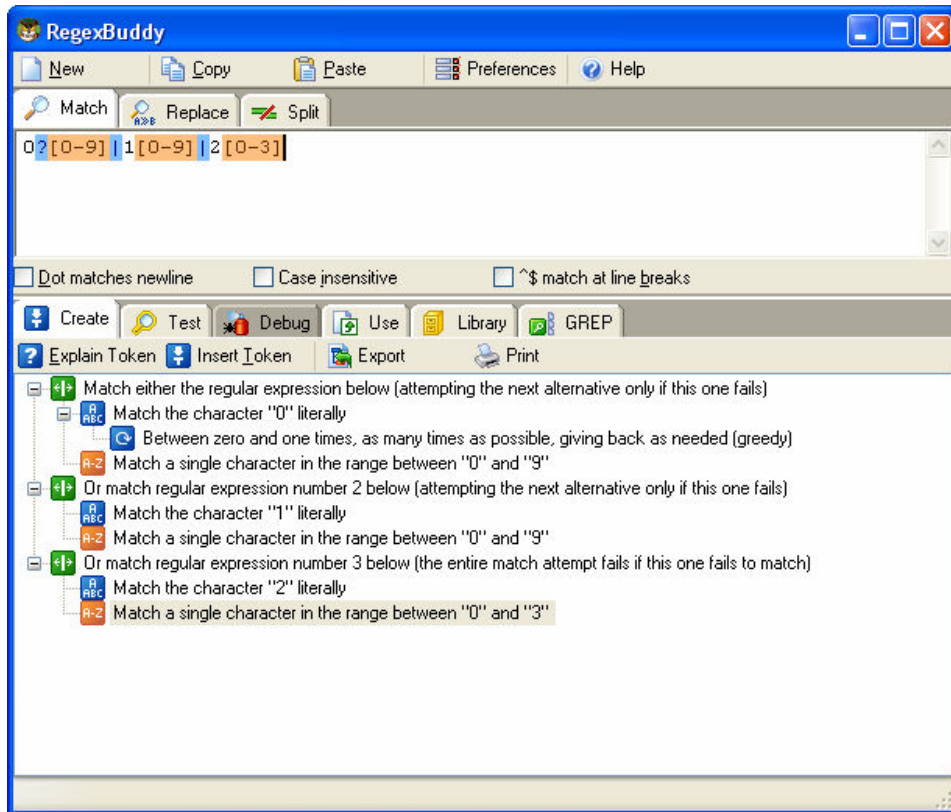


Figure 4 – Time of Day

VBScript Function

The following function taken from the **VBScript** help file with some modifications. The code illustrates how to obtain a Matches collection from a regular expression search, returns by reference the **Matches** object and returns the matches count

```
Function RegExpExecute(ByVal sPatrn, ByVal bIgnoreCase, ByVal sString, byRef oMatches)
    Dim oRegExp                                     ' Create variable.

    Set oRegExp = New RegExp                         ' Create regular expression.
    oRegExp.Pattern = sPatrn                        ' Set pattern.
    oRegExp.IgnoreCase = bIgnoreCase                ' Set case insensitivity.
    oRegExp.Global = True                           ' Set global applicability.
    Set oMatches = oRegExp.Execute(sString)         ' Execute search.
    RegExpExecute = oMatches.Count
End Function
```

Usage

```
nMatches = RegExpExecute(sPattern, True, sExpression, oMatches)
For i = 0 To nMatches - 1
    sTmp = sTmp & i + 1 & ". " & "Position: " & oMatches.Item(i).FirstIndex
    sTmp = sTmp & " Value is: " & oMatches.Item(i).Value & vbNewLine
Next
MsgBox sTmp
```

The Following function taken from the **VBScript** help file with some modifications. The code illustrates how to obtain a **SubMatches** collection from a regular expression search and how to access its individual members:

```
Public Function SubMatchTest( ByVal patrnr, ByVal inpStr)
    Dim oRe, oMatch, oMatches
    Set oRe = New RegExp
    ' Look for an e-mail address (not a perfect RegExp)
    oRe.Pattern = patrnr
    oRe.IgnoreCase = True ' Set case insensitivity.
    oRe.Global = True ' Set global applicability.
    ' Get the Matches collection
    Set oMatches = oRe.Execute(inpStr)
    ' Get the first item in the Matches collection
    Set oMatch = oMatches(0)
    ' Create the results string.
    ' The Match object is the entire match - dragon@xyzzy.com
    retStr = "Email address is: " & oMatch & vbNewline
    ' Get the sub-matched parts of the address.
    retStr = retStr & "Email alias is: " & oMatch.SubMatches(0) ' dragon
    retStr = retStr & vbNewline
    retStr = retStr & "Organization is: " & oMatch.SubMatches(1) ' xyzzy
    SubMatchTest = retStr
End Function
```

Usefull Examples

Matching a floating point number²

In this example, I will show you how you can avoid a common mistake often made by people inexperienced with regular expressions. As an example, we will try to build a regular expression that can match any floating point number. Our regex should also match integers, and floating point numbers where the integer part is not given (i.e. zero). We will not try to match numbers with an exponent, such as 1.5e8 (150 million in scientific notation).

At first thought, the following regex seems to do the trick: `[-+]? [0-9]* \. ? [0-9]*`. This defines a floating point number as an optional sign, followed by an optional series of digits (integer part), followed by an optional dot, followed by another optional series of digits (fraction part).

Spelling out the regex in words makes it obvious: everything in this regular expression is optional. This regular expression will consider a sign by itself or a dot by itself as a valid floating point number. In fact, it will even consider an empty string as a valid floating point number

Not escaping the dot is also a common mistake. A dot that is not escaped will match any character, including a dot. If we had not escaped the dot, 4.4 would be considered a floating point number, and 4X4 too.

When creating a regular expression, it is more important to consider what **it should not match**, than what it should. The above regex will indeed match a proper floating point number, because the regex engine is greedy. But it will also match many things we do not want, which we have to exclude.

Here is a better attempt: `[-+]? ([0-9]* \. [0-9]+ | [0-9]+)`. This regular expression will match an optional sign, that is either followed by zero or more digits followed by a dot and one or more digits (a floating point number with optional integer part), or followed by one or more digits (an integer).

This is a far better definition. Any match will include at least one digit, because there is no way around the `[0-9]+` part. We have successfully excluded the matches we do not want: those without digits.

Dani Vainstein

² <http://www.regular-expressions.info>

We can optimize this regular expression as: `[-+]?[0-9]*\.[0-9]+`

you also want to match numbers with exponents, you can use: `[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?`. Notice how I made the entire exponent part optional by grouping it together, rather than making each element in the exponent optional.

Matching a Valid Date

`(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|1[12][0-9]|3[01])` matches a date in yyyy-mm-dd format from between 1900-01-01 and 2099-12-31, with a choice of four separators. The year is matched by `(19|20)\d\d`. I used alternation to allow the first two digits to be 19 or 20. The round brackets are mandatory. Had I omitted them, the regex engine would go looking for 19 or the remainder of the regular expression, which matches a date between 2000-01-01 and 2099-12-31. Round brackets are the only way to stop the vertical bar from splitting up the entire regular expression into two options.

The month is matched by `0[1-9]|1[012]`, again enclosed by round brackets to keep the two options together. By using character classes, the first option matches a number between 01 and 09, and the second matches 10, 11 or 12.

The last part of the regex consists of three options. The first matches the numbers 01 through 09, the second 10 through 29, and the third matches 30 or 31.

Smart use of alternation allows us to exclude invalid dates such as 2000-00-00 that could not have been excluded without using alternation. To be really perfectionist, you would have to split up the month into various options to take into account the length of the month. The above regex still matches 2003-02-31, which is not a valid date. Making leading zeros optional could be another enhancement.

If you want to require the delimiters to be consistent, you could use a backreference.

`(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|1[12][0-9]|3[01])` will match 1999-01-01 but not 1999/01-01.

Again, how complex you want to make your regular expression depends on the data you are using it on, and how big a problem it is if an unwanted match slips through. If you are validating the user's input of a date in a script, it is probably easier to do certain checks outside of the regex. For example, excluding February 29th when the year is not a leap year is far easier to do in a scripting language. It is far easier to check if a year is divisible by 4 (and not divisible by 100 unless divisible by 400) using simple arithmetic than using regular expressions

Trimming Whitespace

You can easily trim unnecessary whitespace from the start and the end of a string or the lines in a text file by doing a regex search-and-replace. Search for `^[\t]+` and replace with nothing to delete leading whitespace (spaces and tabs). Search for `[\t]+$` to trim trailing whitespace. Do both by combining the regular expressions into `^[\t]+|[\t]+$`. Instead of `[\t]` which matches a space or a tab, you can expand the character class into `[\t\r\n]` if you also want to strip line breaks. Or you can use the shorthand `\s` instead

U.S. Currency

- Pattern: `\$(\d{1,3})(\,\d{3})*|(\d+)(\.\d{2})?`
- Matching Text: \$0.84, \$123458, \$1,234,567.89
- Non-Matching Text: 12345, 1.234\$
- Description This matches US currency format with lead dollar sign. Dollar value must have at least one digit and may or may not be comma separated. Cents value is optional.

Five Integer US ZIP Code

- Pattern `\d{5}`
- Matching Text: 33333, 55555, 23445
- Non-Matching: Text abcd, 1324, as;lkjdf
- Description: Matches 5 numeric digits, such as a zip code.

Title Alphanumeric

- Pattern: `[a-zA-Z0-9]+$`
- Matching Text: 10a, ABC, A3fg
- Non-Matching Text: 45.3, this or that, \$23
- Description: Matches any alphanumeric string (no spaces).

Visa Credit Card

- Pattern: `([4]{1})([0-9]{12,15})`
- Matching Text: 4125632152365, 418563256985214, 4125632569856321
- Non-Matching Text: 3125652365214, 41256321256, 42563985632156322
- Description: Validate against a visa card number. All visa cards start with a 4 and are followed by 12 to 15 more numbers.

ISBN-10

- Pattern: `ISBN\x20(?:\d{1,5}[-]\d{1,7}\d{1,6})\d{1,X}`
- Matching Text: ISBN 0 93028 923 4, ISBN 1-56389-668-0, ISBN 1-56389-016-X
- Non-Matching Text: 123456789X, ISBN 9-87654321-2, ISBN 123 456-789X
- Description: This RE validates the format of an ISBN number

IP Address

- Pattern: `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`
- Matching Text: match any IP address just fine, but will also match 999.999.999.999 as if it were a valid IP address.
- Non-Matching Text:
- Description: Matching an IP address.

Strong Password

- Pattern: `[a-zA-Z]\w{3,14}`
- Matching Text: abcd, aBc45DSD_sdf, password
- Non-Matching Text: afv, 1234, reallylongpassword
- Description: The password's first character must be a letter, it must contain at least 4 characters and no more than 15 characters and no characters other than letters, numbers and the underscore may be used

Appendix 5.A

Position Matching

Position matching involves the use of the `^` and `$` to search for beginning or ending of strings. Setting the pattern property to "**^VBScript**" will only successfully match "**VBScript is cool.**" But it will fail to match "**I like VBScript.**"

Symbol	Function	Samples pattern/match	
<code>^</code>	Only match the beginning of a string.	<code>^abc</code>	abc, abcdefg, abc123, ...

\$	Only match the ending of a string.	abc\$	abc, endsinabc, 123abc, ...
\b	Matches at the position between a word character (anything matched by \w) and a non-word character (anything matched by [^\w] or \W) as well as at the start and/or end of the string if the first and/or last characters in the string are word characters.	Matches a backspace \u0008 if in a []; otherwise matches a word boundary (between \w and \W characters). .\b matches c in abc	
\B	Matches at the position between two word characters (i.e the position between \w\w) as well as at the position between two non-word characters (i.e. \W\W).	\B.\B matches b in abc	

Table 3 – Position Matching

Literals

Literals can be taken to mean alphanumeric characters, ACSII, octal characters, hexadecimal characters, UNICODE, or special escaped characters. Since some characters have special meanings, we must escape them. To match these special characters, we precede them with a "\" in a regular expression.

Symbol	Function
n	Matches a new line
\f	Matches a form feed
\r	Matches carriage return
\t	Matches horizontal tab
\v	Matches vertical tab
\?	Matches ?
*	Matches *
\+	Matches +
\.	Matches .
\\	Matches
\{	Matches {
\}	Matches }
\\	Matches \
\[Matches [
\]	Matches]
\(Matches (
\)	Matches)
\xxx	Matches the ASCII character expressed by the octal number xxx.
\xdd	Matches the ASCII character expressed by the hex number dd.
\uxxxx	Matches the ASCII character expressed by the UNICODE xxxx.

Table 4 – Literals

Character Classes

Character classes enable customized grouping by putting expressions within []

braces. A negated character class may be created by placing ^ as the first character inside the []. Also, a dash can be used to relate a scope of characters. For example, the regular expression "[^a-zA-Z0-9]" matches everything except alphanumeric characters. In addition, some common character sets are bundled as an escape plus a letter.

Symbol	Function	Samples pattern/match	
[xyz]	Match any one character enclosed in the character set.	a[bB]c	abc, aBc
[^xyz]	Match any one character not enclosed in the character set.		
.	Match any character except \n.	a.c	abc, aac, acc, adc, aec, ...
\w	Match any word character. Equivalent to [a-zA-Z_0-9].		
\W	Match any non-word character. Equivalent to [^a-zA-Z_0-9].		
\d	Match any digit. Equivalent to [0-9].		
\D	Match any non-digit. Equivalent to [^0-9].		
\s	Match any space character. Equivalent to [\t\r\n\v\f].	a\s c	a c
\S	Match any non-space character. Equivalent to [^\t\r\n\v\f].		

Table 5 – Character Classes

Repetition

Repetition allows multiple searches on the clause within the regular expression. By using repetition matching, we can specify the number of times an element may be repeated in a regular expression.

Symbol	Function	Samples pattern/match	
?	Match zero or one occurrences. Equivalent to {0,1}. "a?s?b" matches "ab" or "a b".	ab?c	ac, abc
*	Match zero or more occurrences. Equivalent to {0,}.	ab*c	ac, abc, abbc, abbbc, ...
+	Match one or more occurrences. Equivalent to {1,}.	ab+c	abc, abbc, abbbc, ...
{x}	Match exactly x occurrences of a regular expression. "\d{5}" matches 5 digits.	a{3} \d{4} ab{2}c	Aaa From 10 to 99 abbc
{x,} where x>=1	Match x or more occurrences of a regular expression. "\s{2,}" matches at least 2 space characters.		
{x,y} where x>=1 and y>=x	Matches x to y number of occurrences of a regular expression. "\d{2,3}" matches at least 2 but no more than 3 digits.	a{2,4}	aa, aaa, aaaa

Table 6 – Repetition

Alternation & Grouping

Alternation and grouping is used to develop more complex regular expressions. Using alternation and grouping techniques can create intricate clauses within a regular expression, and offer more flexibility and control.

Symbol	Function	Samples pattern/match	
()	Grouping a clause to create a clause. May be nested. "(ab)?(c)" matches "abc" or "c".	(abc){2}	abcabc
	Alternation combines clauses into one regular expression and then matches any of the individual clauses. "(ab) (cd) (ef)" matches "ab" or "cd" or "ef". The pipe has the lowest precedence of all operators. Use grouping to alternate only part of the regular expression.	bill ted	ted, bill
		abc(def xyz)	abcdef, abcxyz

Table 7 – Alternating & Grouping

Backreferences

Backreferences enable the programmer to refer back to a portion of the regular expression. This is done by use of parenthesis and the backslash (\) character followed by a single digit. The first parenthesis clause is referred by \1, the second by \2, etc.

Symbol	Function
()\n	Matches a clause as numbered by the left parenthesis "(\w+)\s+\1" matches any word that occurs twice in a row, such as "hubba hubba."

Table 8 – Backreferences

Interval Settings

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of Year
d	Day
w	Weekday
ww	Week of Year
h	Hour
n	Minute
s	Second

Table 9 – Interval Settings