# Introduction to Regular Expressions
# Version 1.3

Tom Sgouros

June 29, 2001

# Contents

# 1 Beginning Regular Expresions

Regular Expressions are an advanced pattern-matching languages that are a powerful feature of many modern computer programs. Sadly, there are almost as many varieties of regular expressions as there are computer programs that use them. This (very) brief introduction to regular expressions is about the species found in the underbrush of the Perl jungle. Use the real Perl documentation for the definitive guide, and a better introduction. Use this guide if you're lazy, and don't want to look away from your computer screen.

Some of the material here was adapted from Marc Meurrens's tutorial. See http://www.meurrens.org.

A regular expression is simply an expression of characters used to test whether some other set of characters matches some pattern described by the expression. Since patterns can be very complex, regular expressions can also be very complex. The vast majority of uses will be for fairly simple applications, and the sections that follow provide an introduction to those fairly straightforward uses. Do note that even with fairly simple constructs, regular expressions can be constructed to match some very subtle patterns.

## 1.1 The Simple Version

In its simplest form, a regular expression is just a sequence of ordinary characters, such as a word or phrase to search for. For example,

`gauss` would match any character string with the substring `gauss`. Thus, strings with `gauss`, `gaussian` or `degauss` would all be matched, as would a string containing the phrases `de-gauss the monitor` or `gaussian elimination`.

`carbon` Finds any string containing `carbon`, such as `carbonization` or `hydrocarbons` or `carbon-based life forms`.

`hydro` Matches strings `hydro`, `hydrogen` or `hydrodynamics`.

`top ten` Spaces can be part of the regular expression. This would match `top ten`, but also `stop tension`.

There are several characters with special meanings. These can't be used in simple expressions like the ones above without being preceded by a backslash (\). For example, what if you wanted to search for a string containing a period? Suppose we wished to search for references to pi. The following regular expression would not work:

```
3.14
```

This would indeed match `3.14`, but it would also match `3514`, `3f14`, or even `3+14`. In short, any string of the form `3x14`, where `x` is any character, would be matched by the regular expression above.

To get around this, you use the backslash character to "quote" the period, to indicate that it is to be taken literally. Thus, to search for the string `3.14`, we would use:

```
3\.14
```

In general, whenever the backslash is placed before a special character, the searcher treats the special character literally rather than invoking its special meaning.

(Unfortunately, the backslash is used for other things besides quoting. Many "normal" characters take on special meanings when preceded by a backslash. The rule of thumb is that quoting a special character turns it into a normal character, and quoting a normal character *may* turn it into a special character. It's not very satisfying as a rule of thumb, but it's the best available.)

These are the special characters:

```
| ( ) [ ] { } ^ $ * + ? .
```

## 1.2 Difficult Characters

Certain characters that are difficult to type have special abbreviations as in table 1.1

| | |
|---|---|
| `\a` | alarm (beep) |
| `\n` | newline |
| `\r` | carriage return |
| `\t` | tab |
| `\f` | formfeed |
| `\e` | escape |

Table 1.1: Special Character Abbreviations

You can use these abbreviations to match character strings that contain these. So, for example, the pattern `a\nb` would match:

```
a
b
```

Abbreviations like this, as well as any other character that matches something more complicated than itself, are commonly called regular expression *metacharacters* . But don't let that bother you. It's just a name.

There are also a bunch of special abbreviations, or metacharacters, for classes of characters (but also read Chapter 1.5). These are listed in table 1.2.

| | |
|---|---|
| `.` | matches any character except newline |
| `\d` | a digit (anything except the characters 0,1,2,3,4,5,6,7,8 or 9) |
| `\D` | not a digit |
| `\w` | a letter or number (or _) |
| `\W` | not a letter or number (or _) |
| `\s` | white space (space, tab, newline, formfeed, carriage return) |
| `\S` | not white space |

Table 1.2: Special Character Classes

Here are some examples. The regular expression:

```
a.z
```

Will match any three-character sequence that starts with `a` and ends with `z`, including `a.z`, `a_z`, `abz`, `a8z`, and so on. The middle character usually can't be a newline (`\n`), although Perl contains ways to avoid this restriction. Here are more examples:

```
2,\d-Dimethylbutane
```

would match `2,2-Dimethylbutane`, `2,3-Dimethylbutane` and so forth. Similarly,

```
1\.\d\d\d\d\d
```

would match any six-digit floating-point number from `1.00000` to `1.99999` inclusive.

The Capital version, `\D`, is used to match anything except a digit, so:

```
a\Dz
```

would match `abz`, `aTz` or ``a`z``, but would *not* match `a2z`, `a5z` or `a9z`.

There are a few other abbreviations we'll encounter. In general, the upper-case version matches the converse of the lower-case version.

Here are two more examples:

```
a\wz
```

would match `abz`, `aTz`, `a5z`, `a_z`, or any three-character string starting with `a`, ending with `z`, and whose second character was either a letter (upper- or lower-case), a number, or the underscore. Similarly,

```
a\Wz
```

would not match `abz`, `aTz`, `a5z`, or `a_z`. It would match ``a`z``, `a:z`, `a?z` or any three-character string starting with `a` and ending with `z` and whose second character was not a letter, number, or underscore.

## 1.3 Multiple Matches

If you want to match more than one of something, you can use one of the regular expression *quantifiers* (also called *closures* ). These simply modify the preceding characters to say how many of these should be matched. The available forms are listed in table 1.3

| | |
|---|---|
| `*` | Zero or more |
| `+` | One or more |
| `?` | Zero or one |
| `{2}` | Two exactly |
| `{2,}` | Two or more |
| `{2,5}` | At least two but no more than five |

Table 1.3: Multiple Match Quantifiers

Examples:

`m?ethane` would match either `ethane` or `methane`.

`comm?a` would match either `coma` or `comma`.

`ab*c` would match `ac`, `abc`, `abbc`, `abbbc`, `abbbbbbbbc`, and any string that starts with an `a`, is followed by a sequence of `b`'s, and ends with a `c`.

`ab+c` would *not* match `ac`, but it *would* match `abc`, `abbc`, `abbbc`, `abbbbbbbbc` and so on.

`ab{3}c` would match `abbbc`, and nothing else.

`ab{3,}c` would match `abbbc`, `abbbbc`, `abbbbbc`, and so on.

`ab{3,5}c` would match `abbbc`, `abbbbc`, `abbbbbc`, and nothing else.

`cyclo.*ane` would match `cyclodecane`, `cyclohexane` and even `cyclones drive me insane`, and any string that starts with `cyclo`, is followed by an arbitrary string, and ends with `ane`. Note that the null string will be matched by the period-star pair; thus, `cycloane` would be also match.

`cyclo...ane` would match `cyclodecane` and `cyclohexane`, but would not match `cyclones drive me insane.` Only strings eleven characters long which start with `cyclo` and end with `ane` will be matched. (Note that `cyclopentane` would not be matched, however, since cyclopentane has twelve characters, not eleven.)

Usually, these quantifiers match the largest number of multiples they are allowed to. You can change this so that a pattern matches the smallest number possible. If you know enough about pattern matching to want to change this behavior, you have graduated from this tutorial, and need to check out the documentation listed in Chapter 3.

## 1.4 Grouping Strings and Alternate Patterns

If you want to match multiple copies of groups of letters, you can group them with parentheses, so that `(apple)+` means `apple`, and also `appleapple`, and `appleappleapple` and so on.

You can also provide matching alternatives by using the `|` character inside a group. For example, `(TEXT|text)` matches `TEXT` or `text`, but not `Text` or `tExt`, and so on. To take care of all combinations of upper and lower case, you could do this: `(T|t)(E|e)(X|x)(T|t)`. There are a variety of other ways to do this, as you will see if you read on.

## 1.5 Character Classes

In table 1.2, we saw a list of characters that stand for entire classes of characters. If these classes aren't right for you, you can create new classes with the `[]` characters, and the `^` character. Any group of characters contained in square brackets is interpreted as defining a class of characters. So:

```
a[bz]c
```

matches `abc` and `azc`, and nothing else. You can put as many characters as you like in the brackets, and indicate ranges with the hyphen (`-`). (If you want to include a hyphen in the class, list it first.)

`a[bhi:'9]c` matches `abc`, `ahc`, `aic`, `a:c`, `a'c`, and `a9c`.

`a[4-7]c` matches `a4c`, `a5c`, `a6c`, and `a7c`.

`a[-0-9]c` matches `a-c`, `a0c`, `a1c`, and so on to `a9c`.

`a[4-7]*c` matches `ac`, `a4c`, `a45c`, and any sequence that starts with `a`, ends with `c`, and has any combination of `4`, `5`, `6`, or `7` in between.

A character class can appear anywhere in a regular expression that a regular character can appear.

You can also negate a class by including the `^` sign at the start of the string. The class `[^0-9]` includes everything except digits, and is equivalent to `\D`.

`[^abc]*` Matches any string of characters that does not include `a`, `b`, or `c`.

`[^\(\)]` Matches any string that doesn't contain parentheses. Note that we had to quote the parentheses with the backslash character.

### 1.5.1 Special Classes

Typically, the story doesn't end there. There are several pre-defined special character classes you can use. Use them in class brackets like this: `[:alpha:]`. You can negate these classes by putting a `^` in front, like this: `[:^alpha:]`.

These classes may be useful to you if you want your patterns to be independent of the locale in which they are used. That is, the definition of a "printable character" is different depending on locales. What is an unprintable control code in one locale may be an extended character set accented character in another.

Table 1.4 lists all the special classes available.

You can use these special classes inside other character classes, like this: `[01[:alpha:]]`, which matches any alphabetic character, as well as zero or one.

| | |
|---|---|
| `[:alpha:]` | alphabetic characters |
| `[:alnum:]` | alphanumeric characters |
| `[:ascii:]` | 7-bit ascii character |
| `[:cntrl:]` | unprintable control character |
| `[:digit:]` | a number (`[0-9]` or `\d`) |
| `[:graph:]` | printable (a combination of `alnum` and `punct`) |
| `[:lower:]` | lower case letter |
| `[:print:]` | printable (a combination of `alnum` and `punct`, and the space character) |
| `[:punct:]` | punctuation (not including the space character) |
| `[:space:]` | whitespace character (space, tab, carriage-return, newline, vertical tab, and form-feed) |
| `[:upper:]` | upper case letter |
| `[:word:]` | a component of a word (`alpha` plus _) |
| `[:xdigit:]` | a hex digit (`[0-9a-fA-f]`) |

Table 1.4: Special Character Classes

## 1.6 Special Characters

The basic matching characters have now been covered. What's left is a few special characters you can use to modify the matches you've come up with. For example, if you want to match a pattern when it appears at the end of a string (or the end of a line), you can append `$` to it.

A list of all the special characters is shown in table 1.5

| | |
|---|---|
| `$` or `\Z` or `\z` | Match the end of the string |
| `^` or `\A` | Match the beginning of the string |
| `\b` | Match only at a word boundary |
| `\B` | Don't match at a word boundary |
| `?` | Change the behavior of the preceding multiple match to match the minimum number of characters possible. |

Table 1.5: Match Modifiers

`.*and$` matches `hatband` and `marching band`, but not `band-aid`.

`^[Gg].*` matches any string beginning with a g (upper or lower case).

`^hello$` matches the string `hello` and nothing else.

`.*\band.*` matches `He's androgynous` and `She's an andiron`, but not `It's a hatband`.

`.*\Band.*` matches `The food is bland` or `This land is your land`, but not `a one and a two`.

The behavior of the `?` modifier is a little subtle. Usually, a multiple match pattern matches as many copies of its pattern as possible. The `?` changes this behavior to match the fewest. For further description and examples of its use, see the references in Chapter 3.

# 2 DODS-Specific Examples

If you are writing a configuration file for a DODS server, the following examples may be of some assistance.

`.*\.(NC|nc|cdf|CDF)$` matches any string that ends in `.NC`, `.nc`, `.cdf`, or `.CDF`. So `data.nc` and `DATA.NC` would be matched, but not `data.nc.gz` or `data.Nc`.

`.*\.(HDF|hdf)(.Z|.gz)*$ hdf` matches any string of the form `data.HDF` or `data.hdf`, that may or may not have a `.Z` or `.gz` appended to it.

`^dods-data\/.*` matches any file name at all in the `dods-data` directory.

`^dods-data\/.*\.(NC|nc)(.Z|.gz)*$` matches any file in the `dods-data` directory, of the form `data.NC` or `data.nc`, and that may or may not have `.Z` or `.gz` appended to it.

# 3

# Backreferences, Extensions, and What-Have-You

The regular expression syntax contains methods for referring to earlier matches in a string, embedding comments in an expression, and extending the semantics to include whatever you please. These topics are not covered here, however.

If you've gotten this far, you understand enough to read the real documentation. If you want to know about regular expression arcana, there are a multitude of places you can look. My favorites are the *Programming Perl* book by Larry Wall (O'Reilly, 2000), or the book *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly, 1997).

Enjoy.