Regular Expression HOWTO

Release 0.03

A.M. Kuchling

April 17, 2002

akuchlin@mems-exchange.org

Abstract

This document is an introductory tutorial to using regular expressions in Python with the `re` module. It provides a gentler introduction than the corresponding section in the Library Reference.

This document is available in several formats, including PostScript, PDF, HTML and plain ASCII, from the Python HOWTO page at http://www.python.org/doc/howto/.

Contents

1 Introduction

The `re` module was added in Python 1.5, and provides Perl-style regular expressions. Earlier versions of Python provided the `regex` module, which provides Emacs-style expressions. Emacs-style is slightly less readable, and doesn't provide as many features, so there's not much reason to use the `regex` module when writing new code, though you should be aware of it in order to read older code.

Regular expressions (or REs) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as "Does this string match the pattern?", or "Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string, or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes, which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and optimize the RE in order to produce bytecode that runs faster. Optimization isn't really covered in this document, because it requires that you have a good understanding of the matching engine's internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that *can* be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

2 Simple Patterns

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

For a detailed explanation of the computer science underlying regular expressions (deterministic and non-deterministic finite automata), you can refer to almost any textbook on writing compilers.

2.1 Matching Characters

Most letters and characters will simply match themselves. For example, the regular expression `test` will match the string '`test`' exactly. (You can enable a case-insensitive mode that would let this RE match '`Test`' or '`TEST`' as well; more about this later.)

There are exceptions to this rule; some characters are special, and don't match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them. Much of this document is devoted to discussing various metacharacters and what they do.

Here's a complete list of the metacharacters; their meanings will be discussed in the rest of this HOWTO.

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

The first metacharacter we'll look at is '`[`'; it's used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of characters can be indicated by giving two

characters and separating them by a '`-`'. For example, ⌜`[abc]`⌟ will match any of the characters '`a`', '`b`', or '`c`'; this is the same as ⌜`[a-c]`⌟, which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be ⌜`[a-z]`⌟.

Metacharacters are not active inside classes. For example, ⌜`[akm$]`⌟ will match any of the characters '`a`', '`k`', '`m`', or '`$`'; '`$`' is usually a metacharacter, but inside a character class it's stripped of its special nature.

You can match the characters not within a range by *complementing* the set. This is indicated by including a '`^`' as the first character of the class; '`^`' elsewhere will simply match the '`^`' character. For example, `[^5]` will match any character except '`5`'.

Perhaps the most important metacharacter is the backslash, '`\`'. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It's also used to escape all the metacharacters so you can still match them in patterns; for example, if you need to match a '`[`' or '`\`', you can precede them with a backslash to remove their special meaning: ⌜`\[`⌟ or ⌜`\\`⌟.

Some of these special sequences represent predefined sets of characters that are often useful, such as the set of digits, or the set of letters, or the set of anything that isn't whitespace. The following predefined special sequences are available:

`\d` Matches any decimal digit; this is equivalent to the class ⌜`[0-9]`⌟.

`\D` Matches any non-digit character; this is equivalent to the class `[^0-9]`.

`\s` Matches any whitespace character; this is equivalent to the class ⌜`[ \t\n\r\f\v]`⌟.

`\S` Matches any non-whitespace character; this is equivalent to the class `[^ \t\n\r\f\v]`.

`\w` Matches any alphanumeric character; this is equivalent to the class ⌜`[a-zA-Z0-9_]`⌟.

`\W` Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

These sequences can be included inside a character class. For example, ⌜`[\s,.]`⌟ is a character class that will match any whitespace character, or '`,`' or '`.`'.

The final metacharacter in this section is ⌜`.`⌟. It matches anything except a newline character, and there's an alternate mode (`re.DOTALL`) where it will match even a newline. '`.`' is often used where you want to match "any character".

2.2 Repeating Things

Being able to match varying sets of characters is the first thing regular expressions can do that isn't already possible with Python's `string` module. However, if that was the only additional capability of regexes, they wouldn't be much of an advance over `string`. The second capability is that you can specify that portions of the RE must be repeated a certain number of times.

The first metacharacter for repeating things that we'll look at it is ⌜`*`⌟. ⌜`*`⌟ doesn't match the literal character '`*`'; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

For example, ⌜`ca*t`⌟ will match '`ct`' (0 '`a`' characters), '`cat`' (1 '`a`'), '`caaat`' (3 '`a`' characters), and so forth. The RE engine has various internal limitations, stemming from the size of C's `int` type, that will prevent it from matching over 2 billion '`a`' characters; you probably don't have enough memory to construct a string that large, so you shouldn't run into that limit.

Repetitions such as ⌜`*`⌟ are *greedy*; when repeating a RE, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don't match, the matching engine will then back up and try again with few repetitions.

A step-by-step example will make this more obvious. Let's consider the expression ⌜`a[bcd]*b`⌟. This matches the letter '`a`', zero or more letters from the class `[bcd]`, and finally ends with a '`b`'. Now imagine matching this RE against the string '`abcbd`'.

Step	Matched	Explanation
1	`a`	The ⌈a⌋ in the RE matches.
2	`abcbd`	The engine matches ⌈`[bcd]*`⌋, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match ⌈b⌋, but the current position is at the end of the string, so it fails.
4	`abcb`	Back up, so that ⌈`[bcd]*`⌋ matches one less character.
5	*Failure*	Try ⌈b⌋ again, but the current position is at the last character, which is a '`d`'.
6	`abc`	Back up again, so that ⌈`[bcd]*`⌋ is only matching '`bc`'.
6	`abcb`	Try ⌈b⌋ again. This time but the character at the current position is '`b`', so it succeeds.

The end of the RE has now been reached, and it has matched '`abcb`'. This demonstrates how the matching engine goes as far as it can at first, and if no match is found, it will then progressively back up and retry the rest of the RE again and again. It will back up until it's tried zero matches for ⌈`[bcd]*`⌋, and if that subsequently fails, it will conclude that the string doesn't match the RE at all.

Another repeating metacharacter is ⌈+⌋, which matches one or more times. Pay careful attention to the difference between ⌈*⌋ and
regexp+; ⌈*⌋ matches *zero* or more times, so whatever's being repeated may not be present at all, while ⌈+⌋ requires at least *one* occurrence. To use a similar example, ⌈`ca+t`⌋ will match '`cat`' (1 '`a`'), '`caaat`' (3 '`a`"s), but won't match '`ct`'.

There are two more repeating qualifiers. The question mark character, ⌈?⌋, matches either once or zero times; you can think of it as marking something as being optional. For example, ⌈`home-?brew`⌋ matches either '`homebrew`' or '`home-brew`'.

The most complicated repeated qualifier is ⌈`{`*m*`,`*n*`}`⌋, where *m* and *n* are decimal numbers. This means there must be at least *m* repetitions, and at most *n*. For example, ⌈`a/{1,3}b`⌋ will match '`a/b`', '`a//b`', and '`a///b`'. It won't match '`ab`', which has no slashes, or '`a////b`', which has four.

You can omit either *m* or *n*; in that case, a reasonable value is assumed for the missing value. Omitting *m* is interpreted as a lower limit of 0, while omitting *n* results in an upper bound of infinity — actually, the 2 billion limit mentioned earlier, but that might as well be infinity.

Readers of a reductionist bent may notice that the 3 other qualifiers can all be expressed using this notation. ⌈`{0,}`⌋ is the same as ⌈*⌋, ⌈`{1,}`⌋ is equivalent to ⌈+⌋, and ⌈`{0,1}`⌋ is the same as ⌈?⌋. It's better to use ⌈*⌋, ⌈+⌋, or ⌈?⌋, simply because they're shorter and easier to read.

3 Using Regular Expressions

Now that we've looked at some simple regular expressions, how do we actually use them in Python? The `re` module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

3.1 Compiling Regular Expressions

Regular expressions are compiled into `RegexObject` instances, which then have methods for various operations such as searching for pattern matches or performing string substitutions.

```
>>> import re
>>> p = re.compile('ab*')
>>> print p
<re.RegexObject instance at 80b4150>
```

`re.compile()` also accepts an optional *flags* argument, used to enable various special features and syntax varia-

tions. We'll go over the available settings later, but for now a single example will do:

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

The RE is passed to `re.compile()` as a string. REs are handled as strings because regular expressions aren't part of the core Python language, and no special syntax was created for expressing them. (There are applications that don't need REs at all, so there's no need to bloat the language specification by including them.) Instead, the `re` module is just a C extension module, just like the `string` module.

Putting REs in strings keeps the Python language simpler, but has one disadvantage which is the topic of the next section.

3.2 The Backslash Plague

As stated earlier, regular expressions use the backslash character ('`\`') to indicate special forms or to allow special characters to be used without invoking their special meaning. This conflictts with Python's usage of the same character for the same purpose in string literals.

Let's say you want to write a RE that matches the string '`\section`', which might be found in a LaTeX file. To figure out what to write in the program code, start with the desired string to be matched. Next, you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string '`\\section`'. The resulting string that must be passed to `re.compile()` must be `\\section`. However, to express this as a Python string literal, both backslashes must be escaped *again*.

Characters	Stage
`\section`	Text string to be matched
`\\section`	Escaped backslash for `re.compile`
`"\\\\section"`	Escaped backslashes for a string literal

In short, to match a literal backslash, one has to write `'\\\\'` as the RE string, because the regular expression must be '`\\`', and each backslash must be expressed as '`\\`' inside a regular Python string literal. In REs that feature backslashes repeatedly, this leads to lots of repeated backslashes which makes the resulting strings difficult to understand.

The solution is to use Python's raw string notation for regular expressions; backslashes are not handled in any special way in a string literal prefixed with '`r`', so `r"\n"` is a two-character string containing '`\`' and '`n`', while `"\n"` is a one-character string containing a newline. Frequently regular expressions will be expressed in Python code using this raw string notation.

Regular String	Raw string
`"ab*"`	`r"ab*"`
`"\\\\section"`	`r"\\section"`
`"\\w+\\s+\\1"`	`r"\w+\s+\1"`

3.3 Performing Matches

Once you have an object representing a compiled regular expression, what do you do with it? `RegexObject` instances have several methods and attributes. Only the most significant ones will be covered here; consult the Library Reference for a complete listing.

Method/Attribute	Purpose
match	Determine if the RE matches at the beginning of the string.
search	Scan through a string, looking for any location where this RE matches.
split	Split the string into a list, splitting it wherever the RE matches
sub	Find all substrings where the RE matches, and replace them with a different string
subn	Does the same thing as sub(), except you can limit the number of replacements

These methods return None if no match can be found. If they're successful, a MatchObject instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the re module. (If you have Tkinter available, you may also want to look at 'redemo.py', a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. 'redemo.py' can be quite useful when trying to debug a complicated RE.)

First, run the Python interpreter, import the re module, and compile a RE:

```
Python 1.5.1 (#6, Jul 17 1998, 20:38:08)  [GCC 2.7.2.3] on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
<re.RegexObject instance at 80c3c28>
```

Now, you can try matching various strings against the RE [a-z]+. An empty string shouldn't match at all, since + means 'one or more repetitions'. match() should return None in this case, which will cause the interpreter to print no output. You can explicitly print the result of match() to make this clear.

```
>>> p.match( "" )
>>> print p.match( "" )
None
```

Now, let's try it on a string that it should match, such as 'tempo'. In this case, match() will return a MatchObject, so you should store the result in a variable for later use.

```
>>> m = p.match( 'tempo')
>>> print m
<re.MatchObject instance at 80c4f68>
```

Now you can query the MatchObject for information about the matching string. MatchObject instances also have several methods and attributes; the most important ones are:

Method/Attribute	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) of the match

Trying these methods will soon clarify their meaning:

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` returns the substring that was matched by the RE. `start()` and `end()` return the starting and ending index of the match. `span()` returns both start and end indexes in a single tuple. Since the `match` method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search` method of `RegexObject` instances scans through the string, so the match may not start at zero in that case.

```
>>> print p.match('::: message')
None
>>> m = p.search('::: message') ; print m
<re.MatchObject instance at 80c9650>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

In actual programs, the most common style is to store the `MatchObject` in a variable, and then check if it was `None`. This usually looks like:

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'
```

3.4 Module-Level Functions

You don't have to produce a `RegexObject` and call its methods; the `re` module also provides top-level functions called `match()`, `search()`, `sub()`, and so forth. These functions take the same arguments as the corresponding `RegexObject` method, with the RE string added as the first argument, and still return either `None` or a `MatchObject` instance.

```
>>> print re.match(r'From\s+', 'Fromage amk')
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.MatchObject instance at 80c5978>
```

Under the hood, these functions simply produce a `RegexObject` for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls to them using the same RE are faster.

Should you use these module-level functions, or should you get the `RegexObject` and call its methods yourself? That choice depends on how frequently the RE will be used, and on your personal coding style. If a RE is being used at only one point in the code, then the module functions are probably more convenient. If a program contains a lot of regular expressions, or re-uses the same ones in several locations, then it might be worthwhile to collect all the

definitions in the same place, in a section of code that compiles all the REs ahead of time. To take an example from the standard library, here's an extract from 'xmllib.py':

```
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

(I generally prefer to work with the compiled object, even for one-time uses, but few people will be as much of a purist about this as I am.)

3.5 Compilation Flags

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the `re` module under two names, a long name such as `IGNORECASE`, and a short, one-letter form such as `I`. (If you're familiar with Perl's pattern modifiers, the one-letter forms use the same letters; the short form of `re.VERBOSE` is `re.X`.) Multiple flags can be specified by bitwise OR-ing them; `re.I | re.M` sets both the `I` and `M` flags, for example.

Here's a table of the available flags, followed by a more detailed explanation of each one.

Flag	Meaning
`DOTALL`, `S`	Make `.` match any character, including newlines
`IGNORECASE`, `I`	Do case-insensitive matches
`LOCALE`, `L`	Do a locale-aware match
`MULTILINE`, `M`	Multi-line matching, affecting `^` and `$`
`VERBOSE`, `X`	Enable verbose REs, which can be organized more cleanly and understandably.

`I`
`IGNORECASE`

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too, and `Spam` will match '`Spam`', '`spam`', or '`spAM`'.[1] This doesn't take the current locale into account.

`L`
`LOCALE`

Make `\w`, `\W`, `\b`, and `\B`, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]`; it won't match '`é`' or '`ç`'. If your system is configured properly, and a French locale is selected, certain C functions will tell the program that '`é`' should also be considered a letter. Setting the `LOCALE` flag when compiling a regular expression will cause the resulting compiled object to use these C functions for `\w`; this is slower, but also enables `\w+` to match French words as you'd expect.

`M`
`MULTILINE`

Usually `^` matches only at the beginning of the string, and `$` only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, `^` matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the `$` metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

`S`

[1] '`Spam`', '`spam`', '`spam`', '`spam`', '`spam`', ...

DOTALL

Makes the '`.`' special character match any character at all, including a newline; without this flag, '`.`' will match anything *except* a newline.

X
VERBOSE

This flag allows you to write regular expressions that are more readable, by giving you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. It also enables you to put comments within a RE; comments are marked by a '#' that's neither in a character class or preceded by an unescaped backslash. Comments are simply ignored.

For example, here's a RE that uses `re.VERBOSE`; see how much easier it is to read?

```
charref = re.compile(r"""
 &\#      # Start of a numeric entity reference
 (?P<char>
   [0-9]+[^0-9]      # Decimal form
   | 0[0-7]+[^0-7]   # Octal form
   | x[0-9a-fA-F]+[^0-9a-fA-F] # Hexadecimal form
 )
""", re.VERBOSE)
```

Without the verbose setting, the RE would look like this:

```
charref = re.compile("&#(?P<char>[0-9]+[^0-9]"
                     "|0[0-7]+[^0-7]"
                     "|x[0-9a-fA-F]+[^0-9a-fA-F])")
```

In the above example, Python's automatic concatenation of string literals has been used to break up the RE into smaller pieces, but it's still more difficult to understand than the version using `re.VERBOSE`.

4 More Pattern Power

So far we've only covered a part of the features of regular expressions. In this section, we'll cover some new metacharacters, and how to use groups to retrieve portions of the text that was matched.

4.1 More Metacharacters

There are some metacharacters that we haven't covered yet. Most of them will be covered in this section.

Some of the remaining metacharacters to be discussed are *zero-width assertions*. They don't cause the engine to advance through the string at all; instead, they consume no characters at all, and simply succeed or fail. For example, `\b` is an assertion that the current position is located at a word boundary; the position isn't changed by the `\b` at all. This means that zero-width assertions should never be repeated, because if they match once at a given location, they can obviously be matched an infinite number of times.

`|` Alternation, or the "or" operator. If A and B are regular expressions, `A|B` will match any string that matches either '`A`' or '`B`'. `|` has very low precedence, in order to make it work reasonably when you're alternating multi-character strings. `Crow|Servo` will match either '`Crow`' or '`Servo`', not '`Cro`', a '`w`' or an '`S`', and '`ervo`'.

To match a literal '`|`', use `\|`, or enclose it inside a character class, as in `[|]`.

⌈^⌋ Matches at the beginning of lines. Unless the MULTILINE flag has been set, this will only match at the beginning of the string. In MULTILINE mode, this also matches immediately after each newline within the string.

For example, if you wish to match the word 'From' only at the beginning of a line, the RE to use is ^From.

```
>>> print re.search('^From', 'From Here to Eternity')
<re.MatchObject instance at 80c1520>
>>> print re.search('^From', 'Reciting From Memory')
None
```

To match a literal '^', use ⌈\^⌋, or enclose it inside a character class, as in ⌈[\^]⌋.

⌈$⌋ Matches at the end of lines, which is defined as either the end of the string, or any location followed by a newline character.

```
>>> print re.search('}$', '{block}')
<re.MatchObject instance at 80adfa8>
>>> print re.search('}$', '{block} ')
None
>>> print re.search('}$', '{block}\n')
<re.MatchObject instance at 80adfa8>
```

To match a literal '$', use ⌈\$⌋, or enclose it inside a character class, as in ⌈[$]⌋.

⌈\A⌋ Matches only at the start of the string. When not in MULTILINE mode, ⌈\A⌋ and ⌈^⌋ are effectively the same. In MULTILINE mode, however, they're different; ⌈\A⌋ still matches only at the beginning of the string, but ⌈^⌋ may match at several locations inside the string (anywhere following a newline character).

⌈\Z⌋ Matches only at the end of the string.

⌈\b⌋ Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

The following example matches 'class' only when it's a complete word; it won't match when it's contained inside another word.

```
>>> p = re.compile(r'\bclass\b')
>>> print p.search('no class at all')
<re.MatchObject instance at 80c8f28>
>>> print p.search('the declassified algorithm')
None
>>> print p.search('one subclass is')
None
```

There are two subtleties you should remember when using this special sequence. First, this is the worst collision between Python's string literals and regular expression sequences. In Python's string literals, '\b' is the backspace character, ASCII value 8. If you're not using raw strings, then Python will convert the '\b' to a backspace, and your RE won't match as you expect it to. The following example looks the same as our previous RE, but omits the 'r' in front of the RE string.

```
>>> p = re.compile('\bclass\b')
>>> print p.search('no class at all')
None
>>> print p.search('\b' + 'class' + '\b')
<re.MatchObject instance at 80c3ee0>
```

Second, inside a character class, where there's no use for this assertion, `\b` represents the backspace character, for compatibility with Python's string literals.

`\B` Another zero-width assertion, this is the opposite of `\b`, only matching when the current position is not at a word boundary.

4.2 Grouping

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are also often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a '`:`'. This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value.

Groups are marked by the '`(`', '`)`' metacharacters. '`(`' and '`)`' have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them. For example, you can repeat the contents of a group with a repeating qualifier, such as `*`, `+`, `?`, or `{m,n}`. For example, `(ab)*` will match zero or more repetitions of '`ab`'.

```
>>> p = re.compile('(ab)*')
>>> print p.match('abababababab').span()
(0, 10)
```

Groups indicated with '`(`', '`)`' also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. (Later we'll see how to express groups that don't capture the span of text that they match.) Groups are numbered starting with 0. Group 0 is always present; it's the whole RE. The methods all have group 0 as their default argument.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

`group()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

The `groups()` method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> m.groups()
('abc', 'b')
```

Backreferences allow you to specify that the contents of an earlier capturing group must also be found at the current location in the string. For example, 「`\1`」 will succeed if the exact contents of group 1 can be found at the current position, and fails otherwise. Remember that Python's string literals also use a backslash followed by numbers to allow including arbitrary characters in a string, so be sure to use a raw string when incorporating backreferences in a RE.

For example, the following RE detects doubled words in a string.

```
>>> p = re.compile(r'(\b\w+)\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

Backreferences like this aren't often useful for just searching through a string — there are few text formats which repeat data in this way — but you'll soon find out that they're *very* useful when performing string substitutions.

4.3 Non-capturing, and Named Groups

Elaborate REs may use many groups, both to capture substrings of interest, and to group and structure the RE itself. In complex REs, it becomes difficult to keep track of the group numbers. There are two features which help with this problem. Both of them use a common syntax for regular expression extensions, so we'll look at that first.

Perl 5 added several additional features to standard regular expressions, and the Python `re` module supports most of them. It would have been difficult to choose new single-keystroke metacharacters or new special sequences beginning with '\' to represent the new features, without making Perl's regular expressions confusingly different from standard REs. If you chose '`&`' as a new metacharacter, for example, old expressions would be assuming that '`&`' was a regular character and wouldn't have escaped it by writing 「`\&`」 or 「`[&]`」. The solution chosen was to use 「`(?...)`」 as the extension syntax. '`?`' immediately after a parenthesis was a syntax error, because the '`?`' would have nothing to repeat, so this doesn't introduce any compatibility problems. The characters immediately after the '`?`' indicate what extension is being used, so 「`(?=foo)`」 is one thing (a positive lookahead assertion) and 「`(?:foo)`」 is something else (a non-capturing group containing the subexpression 「`foo`」).

Python adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a '`P`', you know that it's a extension that's specific to Python. Currently there are two such extensions: 「`(?P<`*name*`>...)`」 defines a named group, and 「`(?P=`*name*`)`」 is a backreference to a named group. If future versions of Perl 5 add similar features using a different syntax, the `re` module will be changed to support the new syntax, while preserving the Python-specific syntax for compatibility's sake.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs. Since groups are numbered from left to right, and a complex expression may use many groups, it can become difficult to keep track of the correct numbering, and modifying such a complex RE is annoying. Insert a new

group near the beginning, and you change the numbers of everything that follows it.

First, sometimes you'll want to use a group to collect a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a non-capturing group: `(?:...)`, where you can put any other regular expression inside the parentheses.

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

Except for the fact that you can't retrieve the contents of what the group matched, a non-capturing group behaves exactly the same as a capturing group; you can put anything inside it, repeat it with a repetition metacharacter such as '`*`', and nest it within other groups (capturing or non-capturing). `(?:...)` is particularly useful when modifying an existing group, since you can add new groups without changing how all the other groups are numbered. It should be mentioned that there's no performance difference in searching between capturing and non-capturing groups; neither form is any faster than the other.

The second, and more significant, feature, is named groups; instead of referring to them by numbers, groups can be referenced by a name.

The syntax for a named group is one of the Python-specific extensions: `(?P<`*name*`>...)`. *name* is, obviously, the name of the group. Except for associating a name with a group, named groups also behave identically to capturing groups. The `MatchObject` methods that deal with capturing groups all accept either integers, to refer to groups by number, or a string containing the group name. Named groups are still given numbers, so you can retrieve information about a group in two ways:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

Named groups are handy because they let you use easily-remembered names, instead of having to remember numbers. Here's an example RE from the 'imaplib' module:

```
InternalDate = re.compile(r'INTERNALDATE "'
        r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
r'(?P<year>[0-9][0-9][0-9][0-9])'
        r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
        r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'
        r'"')
```

It's obviously much easier to retrieve `m.group('zonem')`, instead of having to remember to retrieve group 9.

Since the syntax for backreferences refers to the number of the group, in an expression like `(...)\1`, there's naturally a variant that uses the group name instead of the number. This is also a Python extension: `(?P=`*name*`)` indicates that the contents of the group called *name* should again be found at the current point. The regular expression for finding doubled words, `(\b\w+)\s+\1` can also be written as `(?P<word>\b\w+)\s+(?P=word)`:

```
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
>>> p.search('Paris in the the spring').group()
'the the'
```

4.4 Other Assertions

Another zero-width assertion is the lookahead assertion. Lookahead assertions are available in both positive and negative form, and look like this:

`(?=...)` Positive lookahead assertion. This succeeds if the contained regular expression, represented here by `...`, successfully matches at the current location, and fails otherwise. But, once the contained expression has been tried, the matching engine doesn't advance at all; the rest of the pattern is tried right where the assertion started.

`(?!...)` Negative lookahead assertion. This is the opposite of the positive assertion; it succeeds if the contained expression *doesn't* match at the current position in the string.

An example will help make this concrete, and will demonstrate a case where a lookahead is useful. Consider a simple pattern to match a filename, and split it apart into a base name and an extension, separated by a '`.`'. For example, in '`news.rc`', '`news`' is the base name, and '`rc`' is the filename's extension.

The pattern to match this is quite simple: `.*[.].*$`. (Notice that the '`.`' needs to be treated specially because it's a metacharacter; I've put it inside a character class. Also notice the trailing `$`; this is added to ensure that all the rest of the string must be included in the extension.) This regular expression matches '`foo.bar`' and '`autoexec.bat`' and '`sendmail.cf`' and '`printers.conf`'.

Now, consider complicating the problem a bit; what if you want to match filenames where the extension is not '`bat`'? Some incorrect attempts:

```
.*[.][^b].*$
```

First attempt: Exclude '`bat`' by requiring that the first character of the extension is not a '`b`'. This is wrong, because it also doesn't match '`foo.bar`'.

```
.*[.]([^b]..|.[^a].|..[^t])$
```

The expression gets messier when you try to patch up the first solution by requiring one of the following cases to match: the first character of the extension isn't '`b`'; the second character isn't '`a`'; or the third character isn't '`t`'. This accepts '`foo.bar`' and rejects '`autoexec.bat`', but it requires a three-letter extension, and doesn't accept '`sendmail.cf`'. Another bug, so we'll complicate the pattern again in an effort to fix it.

```
.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$
```

In the third attempt, the second and third letters are all made optional in order to allow matching extensions shorter than three characters, such as '`sendmail.cf`'.

The pattern's getting really complicated now, which makes it hard to read and understand. When you write a regular expression, ask yourself: if you encountered this expression in a program, how hard would it be to figure out what the expression was intended to do? Worse, this solution doesn't scale well; if the problem changes, and you want to exclude both '`bat`' and '`exe`' as extensions, the pattern would get still more complicated and confusing.

A negative lookahead cuts through all this. Go back to the original pattern, and, before the `.*` which matches the extension, insert `(?!bat$)`. This means: if the expression `bat` doesn't match at this point, try the rest of the pattern; if `bat$` does match, the whole pattern will fail. (The trailing `$` is required to ensure that something like '`sample.batch`', where the extension only starts with '`bat`', will be allowed.

After this modification, the whole pattern is `.*[.](?!bat$).*$`. Excluding another filename extension is now easy; simply add it as an alternative inside the assertion. `.*[.](?!bat$|exe$).*$` excludes both '`bat`' and

‘exe’.

5 Modifying Strings

Up to this point, we’ve simply performed searches against a static string. Regular expressions are commonly used to modify a string in various ways.

5.1 Splitting Strings

The `split()` method of a `RegexObject` splits a string apart wherever the RE matches, returning a list of the pieces. It’s similar to `string.split()`, providing much more generality in the delimiters that you can split by; `string.split()` only supports splitting by whitespace, or by a fixed string. As you’d expect, there’s a module-level `re.split()` function, too.

split(*string* [, *maxsplit* = *0*])
: Split *string* by the matches of the regular expression. If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits are performed.

You can limit the number of splits made, by passing a value for *maxsplit*. When *maxsplit* is nonzero, at most *maxsplit* splits will be made, and the remainder of the string is returned as the final element of the list. In the following example, the delimiter will be any sequence of non-alphanumeric characters.

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

Sometimes you’re not only interested in what the text between delimiters is, but also need to know what the delimiter was. If capturing parentheses are used in the RE, then their values are also returned as part of the list. Compare the following calls:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '... ', 'is', ' ', 'a', ' ', 'test', '.', '']
```

The module-level function `re.split()` adds the RE to be used as the first argument, but is otherwise the same.

```
>>> re.split('[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('([\W]+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

5.2 Search and Replace

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed. Python strings are immutable, so this function will return a new string.

sub(*replacement*, *string*[, *count* = *0*])

Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in *string* by the replacement *replacement*. If the pattern isn't found, *string* is returned unchanged.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. The default value of 0 means to replace all occurrences.

Here's a simple example of using the `sub()` method.

```
>>> p = re.compile( '(blue|white|red)')
>>> p.sub( 'colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub( 'colour', 'blue socks and red shoes', 1)
'colour socks and red shoes'
```

Empty matches are replaced only when not they're not adjacent to a previous match.

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b-d-'
```

If *replacement* is a string, any backslash escapes in it are processed. That is, '`\n`' is converted to a single newline character, '`\r`' is converted to a carriage return, and so forth. Unknown escapes such as '`\j`' are left alone. Backreferences, such as '`\6`', are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First} section{second}')
'subsection{First} subsection{second}'
```

In addition to character escapes and backreferences as described above, '`\g<name>`' will use the substring matched by the group named '`name`', as defined by the `(?P<name>...)` syntax. '`\g<`*number*`>`' uses the corresponding group number. '`\g<2>`' is therefore equivalent to '`\2`', but isn't ambiguous in a replacement string such as '`\g<2>0`'. ('`\20`' would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character '`0`'.) The following substitutions are all equivalent, but use all three variations of the replacement string.

```
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

replacement can also be a function, which gives you even more powerful control. If *replacement* is a function, the function is called for every non-overlapping occurance of *pattern*, and is passed a `MatchObject` argument. The

function can use that information to compute the desired replacement string and return it. For example:

```
>>> def hexrepl( match ):
...     "Return the hex string for a decimal number"
...     value = int( match.group() )
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

When using the module-level `re.sub()` function, the pattern is passed as the first argument. The pattern may be a string or a `RegexObject`; if you need to specify regular expression flags, you must either use a `RegexObject` as the first parameter, or use embedded modifiers in the pattern, e.g. `sub("(?i)b+", "x", "bbbb BBBB")` returns `'x x'`.

6 Common Problems

Regular expressions are a powerful tool for some applications, but in some ways their behaviour isn't intuitive. At times they don't behave the way you may expect them to. This section will point out some of the most common pitfalls.

6.1 Know About the string Module

Sometimes, using the `re` module at all is a mistake. If you're matching a fixed string, or a single character class, and you're not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. The `string` module has several functions for performing operations with fixed strings and it's usually much faster, because the implementation is a single small C loop that's been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace '`word`' with '`deed`'. `re.sub()` seems like the function to use for this, but consider `string.replace()`. Note that `string.replace()` will also replace '`word`' inside words, turning '`swordfish`' into '`sdeedfish`', but the naïve RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that '`word`' have a word boundary on either side. This takes the job beyond the `string` module's abilities.)

Another common task is deleting every occurrence of a single character from a string, or replacing it with another single character. You might do this with something like `re.sub('\n', ' ', S)`, but `string.translate()` is capable of doing both these tasks, and will be much faster that any regular expression operation can ever be.

In short, before turning to the `re` module, consider whether your problem can be solved with the faster and simpler `string` module.

6.2 match() versus search()

The `match()` function only checks if the RE matches at the beginning of the string, while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn't start at zero, `match()` will *not* report it.

```
>>> print re.match('super', 'superstition').span()
(0, 5)
>>> print re.match('super', 'insuperable')
None
```

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```
>>> print re.search('super', 'superstition').span()
(0, 5)
>>> print re.search('super', 'insuperable').span()
(2, 7)
```

Sometimes you'll be tempted to keep using `re.match()`, and just add `.*` to the front of your RE. Resist this tempation, and use `re.search()` instead. The regular expression compiler does some analysis of REs in order to speed up the process of looking for a match. One such analysis figures out what the first character of a match must be; for example, a pattern starting with `Crow` must match starting with a 'C'. The analysis lets the engine quickly scan through the string looking for the starting character, and only try the full match if one is found.

Adding `.*` defeats this optimization, and requires scanning to the end of the string and then backtracking to find a match for the rest of the RE. Use `re.search()` instead.

6.3 Greedy versus Non-Greedy

When repeating a regular expression, as in `a*`, the resulting action is to consume as much of the pattern as possible. This fact often bites you when you're trying to match a pair of balanced delimiters, such as the angle brackets surrounding an HTML tag. The naïve pattern for matching a single HTML tag doesn't work because of the greedy nature of `.*`.

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print re.match('<.*>', s).span()
(0, 32)
>>> print re.match('<.*>', s).group()
<html><head><title>Title</title>
```

The RE matches the '<' in '`<html>`', and the `.*` consumes the rest of the string. There's still more left in the RE, though, and the `>` can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the `>`. The final match extends from the '<' in '`<html>`' to the '>' in '`</title>`', which isn't what you want.

In this case, the solution is to use the non-greedy qualifiers `*?`, `+?`, `??`, or `{m,n}?`, which match as *little* text as possible. In the above example, the '>' is tried immediately after the first '<' matches, and when it fails, the engines advances a character at a time, retrying the '>' at every step. This produces just the right result:

```
>>> print re.match('<.*?>', s).group()
<html>
```

6.4 Not using re.VERBOSE

By now you've probably noticed that regular expressions are a very compact notation, but they're not terribly readable. REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that *isn't* inside a character class is ignored. This means that an expression such as `dog | cat` is equivalent to the less readable `dog|cat`, but `[a b]` will still match the characters 'a', 'b', or a space. In addition, you can also put comments inside a RE; comments extend from a '#' character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```
pat = re.compile(r"""
 \s*                 # Skip leading whitespace
 (?P<header>[^:]+)   # Header name
 \s* :               # Whitespace, and a colon
 (?P<value>.*?)      # The header's value -- *? used to
                     # lose the following trailing whitespace
 \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

This is far more readable than:

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

7 Feedback

Regular expressions are a complicated topic. Did this document help you understand them? Were there parts that were unclear, or Problems you encountered that weren't covered here? If so, please send suggestions for improvements to the author.

The most complete book on regular expressions is almost certainly Jeffrey Friedl's *Mastering Regular Expressions*, published by O'Reilly. Unfortunately, the Python material in this book dates from before the `re` module — all the examples use the old `regex` module — but it covers writing good regular expressions in great detail.