Regular Expression HOWTO

Author: A.M. Kuchling <amk@amk.ca>

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.

Introduction

Regular expressions (called REs, or regexes, or regex patterns) 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 write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn’t 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.

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.

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.

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 *metacharacters*, 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 or changing their meaning. 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 metacharacters we'll look at are `[` and `]`. They're 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 (except `\`) 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 listed within the class by *complementing* the set. This is indicated by including a `'^'` as the first character of the class. For example, `[^5]` will match any character except `'5'`. If the caret appears elsewhere in a character class, it does not have special meaning. For example: `[5^]` will match either a `'5'` or a `'^'`.

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 the special sequences beginning with `'\'` represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn't whitespace.

Let's take an example: `\w` matches any alphanumeric character. If the regex pattern is expressed in bytes, this is equivalent to the class `[a-zA-Z0-9_]`. If the regex pattern is a string, `\w` will match all the characters marked as letters in the Unicode database provided by the `unicodedata` module. You can use the more restricted definition of `\w` in a string pattern by supplying the `re.ASCII` flag when compiling the regular expression.

The following list of special sequences isn't complete. For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of Regular Expression Syntax in the Standard Library reference. In general, the Unicode versions match any character that's in the appropriate category in the Unicode database.

`\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".

Repeating Things

Being able to match varying sets of characters is the first thing regular expressions can do that isn't already possible with the methods available on strings. However, if that was the only additional capability of regexes, they wouldn't be much of an advance. Another 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 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.

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 fewer 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 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 has tried zero matches for `[bcd]*`, and if that subsequently fails, the engine 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 `+`; `*` 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 operators or quantifiers. 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 quantifier is `{m,n}`, where *m* and *n* are decimal integers. This quantifier 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.

The simplest case `{m}` matches the preceding item exactly *m* times. For example, `a/{2}b` will only match `'a//b'`.

Readers of a reductionist bent may notice that the three other quantifiers 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 `?` when you can, simply because they're shorter and easier to read.

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.

Compiling Regular Expressions

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

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

`re.compile()` also accepts an optional *flags* argument, used to enable various special features and syntax variations. 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 simply a C extension module included with Python, just like the `socket` or `zlib` modules.

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

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 conflicts 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 and 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. Regular expressions will often be written in Python code using this raw string notation.

In addition, special escape sequences that are valid in regular expressions, but not valid as Python string literals, now result in a `DeprecationWarning` and will eventually become a `SyntaxError`, which means the sequences will be invalid if raw string notation or escaping the backslashes isn't used.

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

Performing Matches

Once you have an object representing a compiled regular expression, what do you do with it? Pattern objects have several methods and attributes. Only the most significant ones will be covered here; consult the `re` docs 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.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator.

`match()` and `search()` return `None` if no match can be found. If they're successful, a match object 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.

This HOWTO uses the standard Python interpreter for its examples. First, run the Python interpreter, import the `re` module, and compile a RE:

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```

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 match object, so you should store the result in a variable for later use.

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Now you can query the match object for information about the matching string. Match object 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) positions 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 patterns 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.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

In actual programs, the most common style is to store the match object 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')
```

Two pattern methods return all of the matches for a pattern. `findall()` returns a list of matching strings:

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

The `r` prefix, making the literal a raw string literal, is needed in this example because escape sequences in a normal "cooked" string literal that are not recognized by Python, as opposed to regular expressions, now result in a `DeprecationWarning` and will eventually become a `SyntaxError`. See The Backslash Plague.

`findall()` has to create the entire list before it can be returned as the result. The `finditer()` method returns a sequence of match object instances as an iterator:

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

Module-Level Functions

You don't have to create a pattern object and call its methods; the `re` module also provides top-level functions called `match()`, `search()`, `findall()`, `sub()`, and so forth. These functions take the same arguments as the corresponding pattern method with the RE string added as the first argument, and still return either `None` or a match object 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.Match object; span=(0, 5), match='From '>
```

Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE won't need to parse the pattern again and again.

Should you use these module-level functions, or should you get the pattern and call its methods yourself? If you're accessing a regex within a loop, pre-compiling it will save a few function calls. Outside of loops, there's not much difference thanks to the internal cache.

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`, for example.) 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
`ASCII`, `A`	Makes several escapes like `\w`, `\b`, `\s` and `\d` match only on ASCII characters with the respective property.
`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` (for 'extended')	Enable verbose REs, which can be organized more cleanly and understandably.

re.I
re.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. Full Unicode matching also works unless the `ASCII` flag is used to disable non-ASCII matches. When the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the `IGNORECASE` flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: 'İ' (U+0130, Latin capital letter I with dot above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). `Spam` will match

`'Spam'`, `'spam'`, `'spAM'`, or `'ſpam'` (the latter is matched only in Unicode mode). This lowercasing doesn't take the current locale into account; it will if you also set the `LOCALE` flag.

`re.L`
`re.LOCALE`

Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale instead of the Unicode database.

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 encoded French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]` in bytes patterns; it won't match bytes corresponding to `é` or `ç`. If your system is configured properly and a French locale is selected, certain C functions will tell the program that the byte corresponding to `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. The use of this flag is discouraged in Python 3 as the locale mechanism is very unreliable, it only handles one "culture" at a time, and it only works with 8-bit locales. Unicode matching is already enabled by default in Python 3 for Unicode (str) patterns, and it is able to handle different locales/languages.

`re.M`
`re.MULTILINE`

(`^` and `$` haven't been explained yet; they'll be introduced in section More Metacharacters.)

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

`re.S`
`re.DOTALL`

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

`re.A`
`re.ASCII`

Make `\w`, `\W`, `\b`, `\B`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode patterns, and is ignored for byte patterns.

`re.X`
`re.VERBOSE`

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

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
 (
     0[0-7]+         # Octal form
   | [0-9]+          # Decimal form
   | x[0-9a-fA-F]+   # Hexadecimal form
 )
 ;                   # Trailing semicolon
""", re.VERBOSE)
```

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

```
charref = re.compile("&#(0[0-7]+"
                     "|[0-9]+"
                     "|x[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`.

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.

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; 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 posi-

tion 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.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

To match a literal `'^'`, use `\^`.

`$`

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

```
>>> print(re.search('}$', '{block}'))
<re.Match object; span=(6, 7), match='}'>
>>> print(re.search('}$', '{block} '))
None
>>> print(re.search('}$', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
```

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, they're different: `\A` still matches only at the beginning of the string, but `^` may match at any location inside the string that follows 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.Match object; span=(3, 8), match='class'>
>>> 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.Match object; span=(0, 7), match='\x08class\x08'>
```

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.

Grouping

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are 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 `':'`, like this:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

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, and you can repeat the contents of a group with a quantifier, 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()`. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

```
>>> 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 in a pattern 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\b')
>>> 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.

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 is well known for its powerful additions to standard regular expressions. For these new features the Perl developers couldn't choose new single-keystroke metacharacters or new special sequences beginning with `\` without making Perl's regular expressions confusingly different from standard REs. If they 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 by the Perl developers was to use `(?...)` as the extension syntax. `?` immediately after a parenthesis was a syntax error because the `?` would have nothing to repeat, so this didn'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 supports several of Perl's extensions and 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 an extension that's specific to Python.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs.

Sometimes you'll want to use a group to denote 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 replace the `...` with any other regular expression.

```
>>> 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 pattern, 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.

A 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. Named groups behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group's 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'
```

Additionally, you can retrieve named groups as a dictionary with `groupdict()`:

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

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.

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

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

Lookahead 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.

To make this concrete, let's look at 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, so it's inside a character class to only match that specific character. 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].*$` The first attempt above tries to exclude `bat` by requiring that the first character of the extension is not a `b`. This is wrong, because the pattern 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 won't accept a filename with a two-letter extension such as `sendmail.cf`. 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. Worse, if the problem changes and you want to exclude both `bat` and `exe` as extensions, the pattern would get even more complicated and confusing.

A negative lookahead cuts through all this confusion:

`.*[.](?!bat$)[^.]*$` The negative lookahead 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 al-

lowed. The `[^.]*` makes sure that the pattern works when there are multiple dots in the filename.

Excluding another filename extension is now easy; simply add it as an alternative inside the assertion. The following pattern excludes filenames that end in either `bat` or `exe`:

```
.*[.](?!bat$|exe$)[^.]*$
```

Modifying Strings

Up to this point, we've simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways, using the following pattern methods:

Method/Attribute	Purpose
`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()`, but returns the new string and the number of replacements

Splitting Strings

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the `split()` method of strings but provides 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 is 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(r'[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([\W]+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

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.

.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. It replaces colour names with the word `colour`:

```
>>> 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', count=1)
'colour socks and red shoes'
```

The `subn()` method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

Empty matches are replaced only when they're not adjacent to a previous empty 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 `\&` 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.

This example matches the word `section` followed by a string enclosed in `{`, `}`, and changes `section` to `subsection`:

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

There's also a syntax for referring to named groups as defined by the `(?P<name>...)` syntax. `\g<name>` will use the substring matched by the group named `name`, and `\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 control. If *replacement* is a function, the function is called for every non-overlapping occurrence of *pattern*. On each call, the function is passed a match object argument for the match and can use this information to compute the desired replacement string and return it.

In the following example, the replacement function translates decimals into hexadecimal:

```
>>> 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 provided as an object or as a string; if you need to specify regular expression flags, you must either use a pattern object as the first parameter, or use embedded modifiers in the pattern string, e.g. `sub("(?i)b+", "x", "bbbb BBBB")` returns `'x x'`.

Common Problems

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

Use String Methods

Sometimes using the `re` module 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. Strings have several methods for performing operations with fixed strings and they're 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 the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive 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 `replace()`'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 `translate()` is capable of doing both tasks and will be faster than any regular expression operation can be.

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

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 temptation 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, only trying the full match if a `'C'` is found.

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

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 naive 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 quantifiers `*?`, `+?`, `??`, 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 engine advances a character at a time, retrying the `'>'` at every step. This produces just the right result:

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

(Note that parsing HTML or XML with regular expressions is painful. Quick-and-dirty patterns will handle common cases, but HTML and XML have special cases that will break the obvious regular expression; by the time you've written a regular expression that handles all of the possible cases, the patterns will be *very* complicated. Use an HTML or XML parser module for such tasks.)

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*$")
```

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, it exclusively concentrates on Perl and Java's flavours of regular expressions, and doesn't contain any Python material at all, so it won't be useful as a reference for programming in Python. (The first edition covered Python's now-removed `regex` module, which won't help you much.) Consider checking it out from your library.