Chapter 10. Regular Expressions

Regular expressions (REs, aka regexps) let programmers specify pattern strings and perform searches and substitutions. Regular expressions are not easy to master, but they can be a powerful tool for processing text. Python offers rich regular expression functionality through the built-in `re` module. In this chapter, we thoroughly present all about Python's REs.

Regular Expressions and the re Module

A regular expression is built from a string that represents a pattern. With RE functionality, you can examine any string and check which parts of the string, if any, match the pattern.

The `re` module supplies Python's RE functionality. The `compile` function builds an RE object from a pattern string and optional flags. The methods of an RE object look for matches of the RE in a string or perform substitutions. The `re` module also exposes functions equivalent to an RE object's methods, but with the RE's pattern string as the first argument.

This chapter covers the use of REs in Python; it does not teach every minute detail about how to create RE patterns. For general coverage of REs, we recommend the book *Mastering Regular Expressions*, by Jeffrey Friedl (O'Reilly), offering thorough coverage of REs at both tutorial and advanced levels. Many tutorials and references on REs can also be found online, including an excellent, detailed tutorial in Python's **online docs**. Sites like **Pythex** and **regex101** let you test your REs interactively. Alternatively, you can start IDLE, the Python REPL, or any other interactive interpreter, **`import`** `re`, and experiment directly.

REs and bytes Versus str

REs in Python work in two ways, depending on the type of the object being matched: when applied to `str` instances, an RE matches accordingly (e.g., a Unicode character *c* is deemed to be "a letter" if `'LETTER' in unicodedata.name(`*c*`)`); when applied to `bytes` instances, an RE matches in terms of ASCII (e.g., a byte *c* is deemed to be "a letter" if *c* `in string.ascii_letters`). For example:

```
import re
print(re.findall(r'\w+', 'cittá'))            # prints: ['cittá']
print(re.findall(rb'\w+', 'cittá'.encode()))  # prints: [b'citt']
```

Pattern String Syntax

The pattern string representing a regular expression follows a specific syntax:

- Alphabetic and numeric characters stand for themselves. An RE whose pattern is a string of letters and digits matches the same string.
- Many alphanumeric characters acquire special meaning in a pattern when they are preceded by a backslash (`\`), or *escaped*.
- Punctuation characters work the other way around: they stand for themselves when escaped but have special meaning when unescaped.
- The backslash character is matched by a repeated backslash (`\\`).

An RE pattern is a string concatenating one or more pattern elements; each element in turn is itself an RE pattern. For example, `r'a'` is a one-element RE pattern that matches the letter `a`, and `r'ax'` is a two-element RE pattern that matches an `a` immediately followed by an `x`.

Since RE patterns often contain backslashes, it's best to always specify RE patterns in raw string literal form (covered in "Strings"). Pattern elements (such as `r'\t'`, equivalent to the string literal `'\\t'`) do match the corresponding special characters (in this case, the tab character `\t`), so you can use a raw string literal even when you need a literal match for such special characters.

Table 10-1 lists the special elements in RE pattern syntax. The exact meanings of some pattern elements change when you use optional flags, together with the pattern string, to build the RE object. The optional flags are covered in "Optional Flags".

Table 10-1. RE pattern syntax

Element	Meaning
`.`	Matches any single character except `\n` (if `DOTALL`, also matches `\n`)
`^`	Matches start of string (if `MULTILINE`, also matches right after `\n`)
`$`	Matches end of string (if `MULTILINE`, also matches right before `\n`)
`*`	Matches zero or more cases of the previous RE; greedy (matches as many as possible)
`+`	Matches one or more cases of the previous RE; greedy (matches as many as possible)
`?`	Matches zero or one cases of the previous RE; greedy (matches one if possible)
`*?`, `+?`, `??`	Nongreedy versions of `*`, `+`, and `?`, respectively (match as few as possible)
`{`*m*`}`	Matches *m* cases of the previous RE

Element	Meaning
`{m, n}`	Matches between *m* and *n* cases of the previous RE; *m* or *n* (or both) may be omitted, defaulting to `m=0` and `n=infinity` (greedy)
`{m, n}?`	Matches between *m* and *n* cases of the previous RE (nongreedy)
`[...]`	Matches any one of a set of characters contained within the brackets
`[^...]`	Matches one character *not* contained within the brackets after the caret `^`
`|`	Matches either the preceding RE or the following RE
`(...)`	Matches the RE within the parentheses and indicates a *group*
`(?aiLmsux)`	Alternate way to set optional flags[a]
`(?:...)`	Like `(...)` but does not capture the matched characters in a group
`(?P<id>...)`	Like `(...)` but the group also gets the name *<id>*
`(?P=<id>)`	Matches whatever was previously matched by the group named *<id>*
`(?#...)`	Content of parentheses is just a comment; no effect on match

Element	**Meaning**
`(?=...)`	*Lookahead assertion*: matches if RE `...` matches what comes next, but does not consume any part of the string
`(?!...)`	*Negative lookahead assertion*: matches if RE `...` does *not* match what comes next, and does not consume any part of the string
`(?<=...)`	*Lookbehind assertion*: matches if there is a match ending at the current position for RE `...` (`...` must match a fixed length)
`(?<!...)`	*Negative lookbehind assertion*: matches if there is no match ending at the current position for RE `...` (`...` must match a fixed length)
`\` *`number`*	Matches whatever was previously matched by the group numbered *`number`* (groups are automatically numbered left to right, from `1` to `99`)
`\A`	Matches an empty string, but only at the start of the whole string
`\b`	Matches an empty string, but only at the start or end of a *word* (a maximal sequence of alphanumeric characters; see also `\w`)
`\B`	Matches an empty string, but not at the start or end of a word
`\d`	Matches one digit, like the set `[0-9]` (in Unicode mode, many other Unicode characters also count as "digits" for `\d`, but not for `[0-9]`)

Element	Meaning
`\D`	Matches one nondigit character, like the set `[^0-9]` (in Unicode mode, many other Unicode characters also count as "digits" for `\D`, but not for `[^0-9]`)
`\N{`*name*`}`	**3.8+** Matches the Unicode character corresponding to *name*
`\s`	Matches a whitespace character, like the set `[\t\n\r\f\v]`
`\S`	Matches a nonwhitespace character, like the set `[^\t\n\r\f\v]`
`\w`	Matches one alphanumeric character; unless in Unicode mode, or if `LOCALE` or `UNICODE` is set, `\w` is like `[a-zA-Z0-9_]`
`\W`	Matches one nonalphanumeric character, the reverse of `\w`
`\Z`	Matches an empty string, but only at the end of the whole string
`\\`	Matches one backslash character

[a] Always place the `(?...)` construct for setting flags, if any, at the start of the pattern, for readability; placing it elsewhere raises `DeprecationWarning`.

Using a \ character followed by an alphabetic character not listed here or in Table 3-4 raises an `re.error` exception.

Common Regular Expression Idioms

ALWAYS USE R’...’ SYNTAX FOR RE PATTERN LITERALS

Use raw string literals for all RE pattern literals, and only for them. This ensures you’ll never forget to escape a backslash (\), and improves code readability since it makes your RE pattern literals stand out.

`.*` as a substring of a regular expression’s pattern string means “any number of repetitions (zero or more) of any character.” In other words, `.*` matches any substring of a target string, including the empty substring. `.+` is similar but matches only a nonempty substring. For example, this:

```
r'pre.*post'
```

matches a string containing a substring `'pre'` followed by a later substring `'post'`, even if the latter is adjacent to the former (e.g., it matches both `'prepost'` and `'pre23post'`). On the other hand, this:

```
r'pre.+post'
```

matches only if `'pre'` and `'post'` are not adjacent (e.g., it matches `'pre23post'` but does not match `'prepost'`). Both patterns also match strings that continue after the `'post'`. To constrain a pattern to match only strings that *end* with `'post'`, end the pattern with `\Z`. For example, this:

```
r'pre.*post\Z'
```

matches `'prepost'` but not `'preposterous'`.

All of these examples are *greedy*, meaning that they match the substring beginning with the first occurrence of `'pre'` all the way to the *last* occurrence of `'post'`. When you care about what part of the string you match, you may often want to specify *nongreedy* matching, which in our example would match the substring beginning with the first occurrence of `'pre'` but only up to the *first* following occurrence of `'post'`.

For example, when the string is `'preposterous and post facto'`, the greedy RE pattern `r'pre.*post'` matches the substring `'preposterous and post'`; the nongreedy variant `r'pre.*?post'` matches just the substring `'prepost'`.

Another frequently used element in RE patterns is `\b`, which matches a word boundary. To match the word `'his'` only as a whole word and not its occurrences as a substring in such words as `'this'` and `'history'`, the RE pattern is:

```
r'\bhis\b'
```

with word boundaries both before and after. To match the beginning of any word starting with `'her'`, such as `'her'` itself and `'hermetic'`, but not words that just contain `'her'` elsewhere, such as `'ether'` or `'there'`, use:

```
r'\bher'
```

with a word boundary before, but not after, the relevant string. To match the end of any word ending with `'its'`, such as `'its'` itself and `'fits'`, but not words that contain `'its'` elsewhere, such as `'itsy'` or `'jujitsu'`, use:

```
r'its\b'
```

with a word boundary after, but not before, the relevant string. To match whole words thus constrained, rather than just their beginning or end, add a pattern element `\w*` to match zero or more word characters. To match any full word starting with `'her'`, use:

```
r'\bher\w*'
```

To match just the first three letters of any word starting with `'her'`, but not the word `'her'` itself, use a negative word boundary `\B`:

```
r'\bher\B'
```

To match any full word ending with `'its'`, including `'its'` itself, use:

```
r'\w*its\b'
```

Sets of Characters

You denote sets of characters in a pattern by listing the characters within brackets (`[]`). In addition to listing characters, you can denote a range by giving the first and last characters of the range separated by a hyphen (`-`). The last character of the range is included in the set, differently from other Python ranges. Within a set, special characters stand for themselves, except `\`, `]`, and `-`, which you must escape (by preceding them with a backslash) when their position is such that, if not escaped, they would form part of the set's syntax. You can denote a class of characters within a set by escaped-letter notation, such as `\d` or `\S`. `\b` in a set means a backspace character (`chr(8)`), not a word boundary. If the first character in the set's pattern, right after the `[`, is a caret (`^`), the set is *complemented*: such a set matches any character *except* those that follow `^` in the set pattern notation.

A frequent use of character sets is to match a “word,” using a definition of which characters can make up a word that differs from `\w`’s default (letters and digits). To match a word of one or more characters, each of which can be an ASCII letter, an apostrophe, or a hyphen, but not a digit (e.g., `"Finnegan-O'Hara"`), use:

```
r"[a-zA-Z'\-]+"
```

ALWAYS ESCAPE HYPHENS IN CHARACTER SETS

It’s not strictly necessary to escape the hyphen with a backslash in this case, since its position at the end of the set makes the situation syntactically unambiguous. However, using the backslash is advisable because it makes the pattern more readable, by visually distinguishing the hyphen that you want to have as a character in the set from those used to denote ranges. (When you want to include a backslash in the character set, of course, you denote that by escaping the backslash itself: write it as `\\`.)

Alternatives

A vertical bar (`|`) in a regular expression pattern, used to specify alternatives, has low syntactic precedence. Unless parentheses change the grouping, `|` applies to the whole pattern on either side, up to the start or end of the pattern, or to another `|`. A pattern can be made up of any number of subpatterns joined by `|`. It is important to note that an RE of subpatterns joined by `|` will match the *first* matching subpattern, not the longest. A pattern like `r'ab|abc'` will never match `'abc'` because the `'ab'` match gets evaluated first.

Given a list *`L`* of words, an RE pattern that matches any one of the words is:

```
'|'.join(rf'\b{word}\b' for word in L)
```

ESCAPING STRINGS

If the items of `L` can be more general strings, not just words, you need to *escape* each of them with the function `re.escape` (covered in Table 10-6), and you may not want the `\b` word boundary markers on either side. In this case, you could use the following RE pattern (sorting the list in reverse order by length to avoid accidentally "masking" a longer word by a shorter one):

```
'|'.join(re.escape(s) for s in sorted(
         L, key=len, reverse=True))
```

Groups

A regular expression can contain any number of *groups*, from none to 99 (or even more, but only the first 99 groups are fully supported). Parentheses in a pattern string indicate a group. The element `(?P<id>...)` also indicates a group and gives the group a name, *`id`*, that can be any Python identifier. All groups, named and unnamed, are numbered, left to right, 1 to 99; "group 0" means the string that the whole RE matches.

For any match of the RE with a string, each group matches a substring (possibly an empty one). When the RE uses `|`, some groups may not match any substring, although the RE as a whole does match the string. When a group doesn't match any substring, we say that the group does not *participate* in the match. An empty string (`''`) is used as the matching substring for any group that does not participate in a match, except where otherwise indicated later in this chapter. For example, this:

```
r'(.+)\1+\Z'
```

matches a string made up of two or more repetitions of any nonempty substring. The `(.+)` part of the pattern matches any nonempty substring (any character, one or more times) and defines a group, thanks to the parentheses. The `\1+` part of the pattern matches one or more repetitions of the group, and `\Z` anchors the match to the end of the string.

Optional Flags

The optional `flags` argument to the function `compile` is a coded integer built by bitwise ORing (with Python's bitwise OR operator, `|`) one or more of the following attributes of the module `re`. Each attribute has both a short name (one uppercase letter), for convenience, and a long name (an uppercase multiletter identifier), which is more readable and thus normally preferable:

`A` *or* `ASCII`
: Uses ASCII-only characters for `\w`, `\W`, `\b`, `\B`, `\d`, and `\D`; overrides the default `UNICODE` flag

`I` *or* `IGNORECASE`
: Makes matching case-insensitive

`L` *or* `LOCALE`
: Uses the Python `LOCALE` setting to determine characters for `\w`, `\W`, `\b`, `\B`, `\d`, and `\D` markers; you can only use this option with `bytes` patterns

`M` *or* `MULTILINE`
: Makes the special characters `^` and `$` match at the start and end of each line (i.e., right after/before a newline), as well as at the start and end of the whole string (`\A` and `\Z` always match only the start and end of the whole string)

`S` *or* `DOTALL`
: Causes the special character `.` to match any character, including a newline

`U` *or* `UNICODE`
: Uses full Unicode to determine characters for `\w`, `\W`, `\b`, `\B`, `\d`, and `\D` markers; although retained for backward compatibility, this flag is now the default

`X` *or* `VERBOSE`
: Causes whitespace in the pattern to be ignored, except when escaped or in a character set, and makes a nonescaped `#` character in the pattern begin a comment that lasts until the end of the line

Flags can also be specified by inserting a pattern element with one or more of the letters `aiLmsux` between `(?` and `)`, rather than by the `flags` argument to the `compile` function of the `re` module (the letters correspond to the uppercase flags given in the preceding list). Options should always be placed at the start of the pattern; not doing this produces a deprecation warning. In particular, placement at the start is mandatory if `x` (the inline flag character for verbose RE parsing) is among the options, since `x` changes the way Python parses the pattern. Options apply to the whole RE, except that the `aLu` options can be applied locally within a group.

Using the explicit `flags` argument is more readable than placing an options element within the pattern. For example, here are three ways to define equivalent REs with the `compile` function. Each of these REs matches the word “hello” in any mix of upper- and lowercase letters:

```
import re
r1 = re.compile(r'(?i)hello')
r2 = re.compile(r'hello', re.I)
r3 = re.compile(r'hello', re.IGNORECASE)
```

The third approach is clearly the most readable, and thus the most maintainable, though slightly more verbose. The raw string form is not strictly necessary here, since the patterns do not include backslashes. However, using raw string literals does no harm, and we recommend you always use them for RE patterns to improve clarity and readability.

The option `re.VERBOSE` (or `re.X`) lets you make patterns more readable and understandable through appropriate use of whitespace and comments. Complicated and verbose RE patterns are generally best represented by strings that take up more than one line, and therefore you normally want to use a triple-quoted raw string literal for such pattern strings. For example, to match a string representing an integer that may be in octal, hex, or decimal format, you could use use either of the following:

```
repat_num1 = r'(0o[0-7]*|0x[\da-fA-F]+|[1-9]\d*)\Z'
repat_num2 = r'''(?x)   # (re.VERBOSE) pattern matching int literals
```

```
                 (  0o [0-7]*       # octal: leading 0o, 0+ octal digits
                  | 0x [\da-fA-F]+  # hex: 0x, then 1+ hex digits
                  | [1-9] \d*       # decimal: leading non-0, 0+ digits
                 )\Z                # end of string
                 '''
```

The two patterns defined in this example are equivalent, but the second one is made more readable and understandable by the comments and the free use of whitespace to visually group portions of the pattern in logical ways.

Match Versus Search

So far, we've been using regular expressions to *match* strings. For example, the RE with pattern `r'box'` matches strings such as `'box'` and `'boxes'`, but not `'inbox'`. In other words, an RE *match* is implicitly anchored at the start of the target string, as if the RE's pattern started with `\A`.

Often you'll be interested in locating possible matches for an RE anywhere in the string, without anchoring (e.g., find the `r'box'` match within such strings as `'inbox'`, as well as in `'box'` and `'boxes'`). In this case, the Python term for the operation is a *search*, as opposed to a match. For such searches, use the `search` method of an RE object instead of the `match` method, which matches only from the beginning of the string. For example:

```
import re
r1 = re.compile(r'box')
if r1.match('inbox'):
    print('match succeeds')
else:
    print('match fails')          # prints: match fails

if r1.search('inbox'):
    print('search succeeds')      # prints: search succeeds
```

```
else:
    print('search fails')
```

If you want to check that the *whole* string matches, not just its beginning, you can instead use the method `fullmatch`. All of these methods are covered in **Table 10-3**.

Anchoring at String Start and End

`\A` and `\Z` are the pattern elements ensuring that a regular expression match is *anchored* at the string's start or end. The elements `^` for start and `$` for end are also used in similar roles. For RE objects that are not flagged as `MULTILINE`, `^` is the same as `\A`, and `$` is the same as `\Z`. For a multiline RE, however, `^` can anchor at the start of the string *or* the start of any line (where "lines" are determined based on `\n` separator characters). Similarly, with a multiline RE, `$` can anchor at the end of the string *or* the end of any line. `\A` and `\Z` always anchor exclusively at the start and end of the string, whether the RE object is multiline or not.

For example, here's a way to check whether a file has any lines that end with digits:

```
import re
digatend = re.compile(r'\d$', re.MULTILINE)
with open('afile.txt') as f:
    if digatend.search(f.read()):
        print('some lines end with digits')
    else:
        print('no line ends with digits')
```

A pattern of `r'\d\n'` is almost equivalent, but in that case, the search fails if the very last character of the file is a digit not followed by an end-of-line character. With the preceding example, the search succeeds if a digit is at the very end of the file's contents, as well as in the more usual case where a digit is followed by an end-of-line character.

Regular Expression Objects

Table 10-2 covers the read-only attributes of a regular expression object *r* that detail how *r* was built (by the function `compile` of the module `re`, covered in Table 10-6).

Table 10-2. Attributes of RE objects

`flags`	The `flags` argument passed to `compile`, or `re.UNICODE` when `flags` is omitted; also includes any flags specified in the pattern itself using a leading `(?...)` element
`groupindex`	A dictionary whose keys are group names as defined by elements `(?P<`*`id`*`>...)`; the corresponding values are the named groups' numbers
`pattern`	The pattern string from which *r* is compiled

These attributes make it easy to retrieve from a compiled RE object its original pattern string and flags, so you never have to store those separately.

An RE object *r* also supplies methods to find matches for *r* in a string, as well as to perform substitutions on such matches (see Table 10-3). Matches are represented by special objects, covered in the following section.

Table 10-3. Methods of RE objects

`findall`	*`r`*`.findall(`*`s`*`)` `When r has no groups,` `findall` returns a list of strings, e substring of *s* that is a nonoverlapping match with *r*. For e to print out all words in a file, one per line: `import re`

```
reword = re.compile(r'\w+')
with open('afile.txt') as f:
    for aword in reword.findall(f.read()):
        print(aword)
```

`findall` *(cont.)*

When *r* has exactly one group, `findall` also returns a list
strings, but each is the substring of *s* that matches *r*’s grou
example, to print only words that are followed by whitesp
words followed by punctuation or the word at end of the s
you need to change only one statement in the preceding e

```
reword = re.compile('(\w+)\s')
```

When *r* has *n* groups (with *n* > 1), `findall` returns a list of
one per nonoverlapping match with *r*. Each tuple has *n* ite
per group of *r*, the substring of *s* matching the group. For
to print the first and last word of each line that has at leas
words:

```
import re
first_last = re.compile(r'^\W*(\w+)\b.*\b(\w+)\
                        re.MULTILINE)
with open('afile.txt') as f:
    for first, last in first_last.findall(f.rea
        print(first, last)
```

`finditer` `r.finditer(s)`

`finditer` is like `findall`, except that, instead of a list of str
tuples, it returns an iterator whose items are match object
(discussed in the following section). In most cases, therefo
`finditer` is more flexible, and usually performs better, tha
`findall`.

`fullmatch`	`r.fullmatch(s, start=0, end=sys.maxsize)` Returns a match object when the complete substring *s*, sta index `start` and ending just short of index `end`, matches *r*. Otherwise, `fullmatch` returns **None**.
`match`	`r.match(s, start=0, end=sys.maxsize)` Returns an appropriate match object when a substring of starting at index `start` and not reaching as far as index en matches *r*. Otherwise, `match` returns **None**. `match` is implici anchored at the starting position `start` in *s*. To search for with *r* at any point in *s* from `start` onward, call `r.search`, `r.match`. For example, here is one way to print all lines in start with digits:

```python
import re
digs = re.compile(r'\d')
with open('afile.txt') as f:
    for line in f:
        if digs.match(line):
            print(line, end='')
```

`search` `r.search(s, start=0, end=sys.maxsize)`

Returns an appropriate match object for the leftmost subs *s*, starting not before index `start` and not reaching as far a `end`, that matches *r*. When no such substring exists, `searc` **None**. For example, to print all lines containing digits, one s approach is as follows:

```
import re
digs = re.compile(r'\d')
with open('afile.txt') as f:
    for line in f:
        if digs.search(line):
            print(line, end='')
```

`split` `r.split(s, maxsplit=0)`

Returns a list *L* of the *splits* of *s* by *r* (i.e., the substrings of separated by nonoverlapping, nonempty matches with *r*). example, here's a way to eliminate all occurrences of `'hel` any mix of lowercase and uppercase) from a string:

```
import re
rehello = re.compile(r'hello', re.IGNORECASE)
astring = ''.join(rehello.split(astring))
```

When *r* has *n* groups, *n* more items are interleaved in *L* be each pair of splits. Each of the *n* extra items is the substrin that matches *r*'s corresponding group in that match, or **No** group did not participate in the match. For example, here' way to remove whitespace only when it occurs between a and a digit:

```
import re
```

```
re_col_ws_dig = re.compile(r'(:)\s+(\d)')
astring = ''.join(re_col_ws_dig.split(astring))
```

If `maxsplit` is greater than `0`, at most `maxsplit` splits are ir
followed by *n* items, while the trailing substring of *s* after r
matches of *r*, if any, is *L*'s last item. For example, to remove
first occurrence of substring `'hello'` rather than *all* of the
change the last statement in the first example here to:

```
astring=''.join(rehello.split(astring, 1))
```

`sub`

`r.sub(repl, s, count=0)`

Returns a copy of *s* where nonoverlapping matches with *r*
replaced by *repl*, which can be either a string or a callable
such as a function. An empty match is replaced only when
adjacent to the previous match. When `count` is greater tha
the first `count` matches of *r* within *s* are replaced. When c
equals `0`, all matches of *r* within *s* are replaced. For examp
another, more natural way to remove only the first occurr
substring `'hello'` in any mix of cases:

```
import re
rehello = re.compile(r'hello', re.IGNORECASE)
astring = rehello.sub('', astring, 1)
```

Without the final `1` (one) argument to `sub`, the example rer
occurrences of `'hello'`.

When *repl* is a callable object, *repl* must accept one argur
match object) and return a string (or **None**, which is equiva
returning the empty string `''`) to use as the replacement fo
match. In this case, `sub` calls *repl*, with a suitable match ol
argument, for each match with *r* that `sub` is replacing. For

here's one way to uppercase all occurrences of words start
'h' and ending with 'o' in any mix of cases:

```
import re
h_word = re.compile(r'\bh\w*o\b', re.IGNORECASE)
def up(mo):
    return mo.group(0).upper()
astring = h_word.sub(up, astring)
```

sub *(cont.)*	When *repl* is a string, sub uses *repl* itself as the replacem except that it expands backreferences. A *backreference* is a substring of *repl* of the form \g<*id*>, where *id* is the name group in *r* (established by the syntax (?P<*id*>...) in *r*'s pa string) or \dd, where *dd* is one or two digits taken as a grou number. Each back reference, named or numbered, is rep with the substring of *s* that matches the group of *r* that the reference indicates. For example, here's a way to enclose e word in braces:
	 `import re`
	 `grouped_word = re.compile('(\w+)')`
	 `astring = grouped_word.sub(r'{\1}', astring)`

subn	*r*.subn(*repl*, *s*, count=0)
	 subn is the same as sub, except that subn returns a pair (*new_string*, *n*), where *n* is the number of substitutions tl has performed. For example, here's one way to count the r of occurrences of substring 'hello' in any mix of cases:
	 `import re`
	 `rehello = re.compile(r'hello', re.IGNORECASE)`

```
_, count = rehello.subn('', astring)
print(f'Found {count} occurrences of "hello"')
```

Match Objects

Match objects are created and returned by the methods `fullmatch`, `match`, and `search` of a regular expression object, and are the items of the iterator returned by the method `finditer`. They are also implicitly created by the methods `sub` and `subn` when the argument *`repl`* is callable, since in that case the appropriate match object is passed as the only argument on each call to *`repl`*. A match object *`m`* supplies the following read-only attributes that detail how `search` or `match` created *`m`*, listed in **Table 10-4**.

Table 10-4. Attributes of match objects

`pos`	The *`start`* argument that was passed to `search` or `match` (i.e., the index into *`s`* where the search for a match began)
`endpos`	The *end* argument that was passed to `search` or `match` (i.e., the index into *`s`* before which the matching substring of *`s`* had to end)
`lastgroup`	The name of the last-matched group (**None** if the last-matched group has no name, or if no group participated in the match)
`lastindex`	The integer index (`1` and up) of the last-matched group (**None** if no group participated in the match)
`re`	The RE object *`r`* whose method created *`m`*

`string`	The string *`s`* passed to `finditer`, `fullmatch`, `match`, `search`, `sub`, or `subn`

In addition, match objects supply the methods detailed in **Table 10-5**.

Table 10-5. Methods of match objects

`end`, `span`, `start`	*`m`*`.end(groupid=0)`, *`m`*`.span(groupid=0)`, *`m`*`.start(groupid=0)` These methods return indices within *`m`*`.string` of the substring that matches the group identified by *`groupid`* (a group number or name; `0`, the default value for `groupid`, means "the whole RE"). When the matching substring is *`m`*`.string[`*`i`*`:`*`j`*`]`, *`m`*`.start` returns *`i`*, *`m`*`.end` returns *`j`*, and *`m`*`.span` returns `(`*`i`*`, `*`j`*`)`. If the group did not participate in the match, *`i`* and *`j`* are `-1`.
`expand`	*`m`*`.expand(`*`s`*`)` Returns a copy of *`s`* where escape sequences and backreferences are replaced in the same way as for the method *`r`*`.sub`, covered in **Table 10-3**.
`group`	*`m`*`.group(groupid=0, *`*`groupids`*`)` Called with a single argument `groupid` (a group number or name), *`m`*`.group` returns the substring matching the group identified by `groupid`, or **`None`** when that group did not participate in the match. *`m`*`.group()`—or *`m`*`.group(0)`—returns the whole matched substring (group `0` means the whole RE). Groups can also be accessed using `m[`*`index`*`]` notation, as if called using *`m`*`.group(`*`index`*`)` (in either case, *`index`* may be an `int` or a `str`). When `group` is called with multiple arguments, each argument must be a group number or name. `group` then returns a tuple with one item per argument, the

	substring matching the corresponding group, or **`None`** when that group did not participate in the match.
`groupdict`	`m.groupdict(default=`**`None`**`)` Returns a dictionary whose keys are the names of all named groups in *`r`*. The value for each name is the substring that matches the corresponding group, or `default` if that group did not participate in the match.
`groups`	`m.groups(default=`**`None`**`)` Returns a tuple with one item per group in *`r`*. Each item is the substring matching the corresponding group, or `default` if that group did not participate in the match. The tuple does not include the `0` group representing the full pattern match.

Functions of the re Module

In addition to the attributes listed in **"Optional Flags"**, the `re` module provides one function for each method of a regular expression object (`findall`, `finditer`, `fullmatch`, `match`, `search`, `split`, `sub`, and `subn`, described in **Table 10-3**), each with an additional first argument, a pattern string that the function implicitly compiles into an RE object. It is usually better to compile pattern strings into RE objects explicitly and call the RE object's methods, but sometimes, for a one-off use of an RE pattern, calling functions of the module `re` can be handier. For example, to count the number of occurrences of `'hello'` in any mix of cases, one concise, function-based way is:

```
import re
_, count = re.subn(r'hello', '', astring, flags=re.I)
print(f'Found {count} occurrences of "hello"')
```

The `re` module internally caches RE objects it creates from the patterns passed to functions; to purge the cache and reclaim some memory, call `re.purge`.

The `re` module also supplies `error`, the class of exceptions raised upon errors (generally, errors in the syntax of a pattern string), and two more functions, listed in **Table 10-6**.

Table 10-6. Additional `re` functions

`compile`	`compile(`*`pattern`*`, flags=0)` Creates and returns an RE object, parsing the string *`pattern`*, as per the syntax covered in **"Pattern String Syntax"**, and using integer `flags`, as described in **"Optional Flags"**
`escape`	`escape(`*`s`*`)` Returns a copy of string *`s`* with each nonalphanumeric character escaped (i.e., preceded by a backslash, \); useful to match string *`s`* literally as part of an RE pattern string

REs and the := Operator

The introduction of the `:=` operator in Python 3.8 established support for a successive-match idiom in Python similar to the one that's common in Perl. In this idiom, a series of **`if`**/**`elsif`** branches tests a string against different regular expressions. In Perl, the **`if`** `($var =~ /regExpr/)` statement both evaluates the regular expression and saves the successful match in the variable `var`:[1]

```
if    ($statement =~ /I love (\w+)/) {
  print "He loves $1\n";
}
elsif ($statement =~ /Ich liebe (\w+)/) {
  print "Er liebt $1\n";
}
elsif ($statement =~ /Je t\'aime (\w+)/) {
  print "Il aime $1\n";
}
```

Prior to Python 3.8, this evaluate-and-store behavior was not possible in a single **`if`/`elif`** statement; developers had to use a cumbersome cascade of nested **`if`/`else`** statements:

```
m = re.match('I love (\w+)', statement)
if m:
    print(f'He loves {m.group(1)}')
else:
    m = re.match('Ich liebe (\w+)', statement)
    if m:
        print(f'Er liebt {m.group(1)}')
    else:
         m = re.match('J'aime (\w+)', statement)
        if m:
            print(f'Il aime {m.group(1)}')
```

Using the `:=` operator, this code simplifies to:

```
if m := re.match(r'I love (\w+)', statement):
    print(f'He loves {m.group(1)}')

elif m := re.match(r'Ich liebe (\w+)', statement):
    print(f'Er liebt {m.group(1)}')

elif m := re.match(r'J'aime (\w+)', statement):
    print(f'Il aime {m.group(1)}')
```

The Third-Party regex Module

As an alternative to the Python standard library's `re` module, a popular package for regular expressions is the third-party **regex module**, by Matthew Barnett. `regex` has an API that's compatible with the `re` module and adds a number of extended features, including:

- Recursive expressions
- Defining character sets by Unicode property/value
- Overlapping matches
- Fuzzy matching
- Multithreading support (releases GIL during matching)
- Matching timeout
- Unicode case folding in case-insensitive matches
- Nested sets

1 This example is taken from regex; see **"Match groups in Python" on Stack Overflow**.