9. Regular Expressions - Python in a Nutshell, 3rd Edition

Regular Expressions and the re Module

A *regular expression* (RE) is built from a string that represents a pattern. With RE functionality, you can examine any string with the pattern, and see 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.

REs can be difficult to master, and this book does not purport to teach them; we cover only the ways in which you can use REs in Python. For general coverage of REs, we recommend the book *Mastering Regular Expressions*, by Jeffrey Friedl (O'Reilly). Friedl's book offers 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 the online docs. Sites like Pythex and regex101 let you test your REs interactively.

re and bytes Versus Unicode Strings

In v3, by default, REs work in two ways: when applied to `str` instances, an RE matches accordingly (for example, 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 (for example, a byte character `c` is deemed to be "a letter" if `c in string.ascii_letters`). In v2, REs, by default, match in terms of ASCII, whether applied to `str` or `unicode` instances; you can override this behavior by explicitly specifying optional flags such as `re.U`, covered in "Optional Flags". For example, in v3:

```
import reprint(re.findall(r'\w+', 'cittá'))# prints ['cittá']
```

while, in v2:

```
import reprint(re.findall(r'\w+', u'cittá'))# prints ['citt']print(re.findall(r'\w+',
u'cittá', re.U))# prints ['citt\xe1']
```

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 (`\`).
- Punctuation works the other way around: self-matching when escaped, special meaning when unescaped.

- The backslash character is matched by a repeated backslash (i.e., pattern `\\`).

Since RE patterns often contain backslashes, it's best to always specify them using raw-string syntax (covered in "Strings"). Pattern elements (such as `r'\t'`, equivalent to the nonraw string literal `'\\t'`) do match the corresponding special characters (in this case, the tab character `'\t'`, AKA `chr(9)`); so, you can use raw-string syntax even when you need a literal match for such special characters.

Table 9-1 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 9-1. 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 (match as many as possible)
`+`	Matches one or more cases of the previous RE; greedy (match as many as possible)
`?`	Matches zero or one case of the previous RE; greedy (match one if possible)
`*?`, `+?`, `??`	Nongreedy versions of `*`, `+`, and `?`, respectively (match as few as possible)
`{ m , n }`	Matches between `m` and `n` cases of the previous RE (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*
`(?iLmsux)`	Alternate way to set optional flags; no effect on match
`(?:...)`	Like `(...)` but does not indicate a group
`(?P< id >...)`	Like `(...)` but the group also gets the name `id`
`(?P= id )`	Matches whatever was previously matched by group named `id`
`(?#...)`	Content of parentheses is just a comment; no effect on match
`(?=...)`	*Lookahead assertion*: matches if RE `...` matches what comes next, but does not consume any part of the string

Element	Meaning
`(?!...)`	*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 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]`)
`\D`	Matches one nondigit, like the set `[^0-9]` (in Unicode mode, many other Unicode characters also count as “digits” for `\D`, but not for `[^0-9]`)
`\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 `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

Common Regular Expression Idioms

Always use r’...’ syntax for RE pattern literals

Use raw-string syntax for all RE pattern literals, and for them only: this ensures you’ll never forget to escape a backslash (`\`), and improves code readability as 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:

```
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:

```
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:

```
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 want to specify *nongreedy* matching, meaning to 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 non-greedy 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 unescaped, 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 a letter, an apostrophe, or a hyphen, but not a digit (e.g., `"Finnegan-O'Hara"`), use:

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

Escape a hyphen that's part of an RE character set, for readability

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, 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.

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 `|`. To match such an RE, the first subpattern is tried, and if it matches, the others are skipped. If the first subpattern doesn't match, the second subpattern is tried, and so on. `|` is neither greedy nor nongreedy: it just doesn't take the length of the match into account.

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

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

Escaping strings

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 9-3), and you may not want the `\b` word boundary markers on either side. In this case, use the following RE pattern:

```
'|'.join(re.escape(s) for s in L)
```

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. 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 from left to right, 1 to 99; "group 0" means the whole RE.

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:

```
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

A regular expression pattern element with one or more of the letters `iLmsux` between `(?` and `)` lets you set RE options within the pattern, rather than by the `flags` argument to the `compile` function of the `re` module. Options apply to the whole RE, no matter where the options element occurs in the pattern.

For clarity, always place options at the start of an RE's pattern

In particular, placement at the start is mandatory if `x` is among the options, since `x` changes the way Python parses the pattern. In 3.6, options not at the start of the pattern produce a deprecation warning.

Using the explicit `flags` argument is more readable than placing an options element within the pattern. The `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:

`I` *or* `IGNORECASE`

Makes matching case-insensitive

`L` *or* `LOCALE`

Causes `\w`, `\W`, `\b`, and `\B` matches to depend on what the current locale deems alphanumeric; deprecated in v3

`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`

Makes `\w`, `\W`, `\b`, and `\B` matches depend on what Unicode deems alphanumeric; deprecated in v3

`X` *or* `VERBOSE`

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

For example, here are three ways to define equivalent REs with function `compile`, covered in Table 9-3. 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, even though it is slightly more verbose. The raw-string form is not necessary here, since the patterns do not include backslashes; however, using raw strings is innocuous, and we recommend you always do that for RE patterns, to improve clarity and readability.

Option `re.VERBOSE` (or `re.X`) lets you make patterns more readable and understandable by 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 the triple-quoted raw-string format for such pattern strings. For example, to match old-style Python v2 `int` literals:

```
repat_num1 = r'(0[0-7]*|0x[\da-fA-F]+|[1-9]\d*)L?\Z'
               '''(?x)             # pattern matching int
repat_num2 = rliterals
              (0 [0-7]*         | # octal: leading 0, 0+ octal
digits
               0x [\da-fA-F]+   | # hex: 0x, then 1+ hex
digits
               [1-9] \d*        ) # decimal: leading non-0, 0+
digits
               L?\Z               # optional trailing L, 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're interested in locating possible matches for an RE anywhere in the string, without anchoring (e.g., find the `r'box'` match inside 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; the `match` method deals with matching only from the start. For example:

```
                                                          match
import rer1 = re.compile(r'box')if r1.match('inbox'):    print('succeeds      ')else:
    print('match fails')         # prints: match failsif r1.search('inbox'):    print('
search                          search                          search
succeeds       ')     # prints: succeeds       else:    print('fails       ')
```

Anchoring at String Start and End

The pattern elements ensuring that a regular expression search (or match) is anchored at string start and string end are `\A` and `\Z`, respectively. More traditionally, elements `^` for start and `$` for end are also used in similar roles. `^` is the same as `\A`, and `$` is the same as `\Z`, for RE objects that are not multiline (i.e., that do not contain pattern element `(?m)` and are not compiled with the flag `re.M` or `re.MULTILINE`). For a multiline RE, however, `^` anchors at the start of any line (i.e., either at the start of the whole string or at any position right after a newline character `\n`). Similarly, with a multiline RE, `$` anchors at the end of any line (i.e., either at the end of the whole string or at any position right before a `\n`). On the other hand, `\A` and `\Z` anchor 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()):
              'some lines end with
        print(digits'
)
    else:
              'no line ends with
        print(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

A regular expression object `r` has the following read-only attributes that detail how `r` was built (by the function `compile` of the module `re`, covered in Table 9-3):

`flags`

> The `flags` argument passed to `compile`, or `0` when `flags` is omitted (in v2; in v3, the default when `flags` is omitted is `re.UNICODE`)

`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 get back from a compiled RE object to its pattern string and flags, so you never have to store those separately.

An RE object `r` also supplies methods to locate matches for `r` within a string, as well as to perform substitutions on such matches (Table 9-2). Matches are generally represented by special objects, covered in "Match Objects".

Table 9-2.

findall `r.findall(s)`

When `r` has no groups, `findall` returns a list of strings, each a substring of `s` that is a nonoverlapping match with `r`. For example, 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)
```

When `r` has one group, `findall` also returns a list of strings, but each is the substring of `s` that matches `r`'s group. For example, to print only words that are followed by whitespace (not the ones followed by punctuation), you need to change only one statement in the example:

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

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

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

finditer `r.finditer(s)`

`finditer` is like `findall` except that instead of a list of strings (or tuples), it returns an iterator whose items are match objects. In most cases, `finditer` is therefore more flexible and performs better than `findall`.

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

Returns an appropriate match object when a substring of `s`, starting at index `start` and not reaching as far as index `end`, matches `r`. Otherwise, `match` returns `None`. Note that `match` is implicitly anchored at the starting position `start` in `s`. To search for a match with `r` at any point in `s` from `start` onward, call `r.search`, not `r.match`. For example, here one way to print all lines in a file that start with digits:

```
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 substring of `s`, starting not before index `start` and not reaching as far as index `end`, that matches `r`. When no such substring exists, `search` returns `None`. For example, to print all lines containing digits, one simple 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 `s` separated by nonoverlapping, nonempty matches with `r`). For example, here's one way to eliminate all occurrences of substring `'hello'` (in 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` between each pair of splits. Each of the `n` extra items is the substring of `s` that matches `r`'s corresponding group in that match, or `None` if that group did not participate in the match. For example, here's one way to remove whitespace only when it occurs between a colon 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 in `L`, each followed by `n` items as above, while the trailing substring of `s` after `maxsplit` matches of `r`, if any, is `L`'s last item. For example, to remove only the first occurrence of substring `'hello'` rather than all of them, change the last statement in the first example above to:

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

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

Returns a copy of `s` where non-overlapping matches with `r` are replaced by `repl`, which can be either a string or a callable object, such as a function. An empty match is replaced only when not adjacent to the previous match. When `count` is greater than `0`, only the first `count` matches of `r` within `s` are replaced. When `count` equals `0`, all matches of `r` within `s` are replaced. For example, here's another, more natural way to remove only the first occurrence of 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` argument to `sub`, the example removes all occurrences of `'hello'`.

When `repl` is a callable object, `repl` must accept one argument (a match object) and return a string (or `None`, which is equivalent to returning the empty string `''`) to use as the replacement for the match. In this case, `sub` calls `repl`, with a suitable match-object argument, for each match with `r` that `sub` is replacing. For example, here's one way to uppercase all occurrences of words starting with `'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)
```

When `repl` is a string, `sub` uses `repl` itself as the replacement, except that it expands back references. A *back reference* is a substring of `repl` of the form `\g<id>`, where `id` is the name of a group in `r` (established by syntax `(?P<id>...)` in `r`'s pattern string) or `\dd`, where `dd` is one or two digits taken as a group number. Each back reference, named or numbered, is replaced with the substring of `s` that matches the group of `r` that the back reference indicates. For example, here's a way to enclose every 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 that `subn` has performed. For example, here's one way to count the number of occurrences of substring `'hello'` in any mix of cases:

```
import re
rehello = re.compile(r'hello', re.IGNORECASE)
_, count = rehello.subn('', astring)
                        'occurrences of
print('Found', count, "hello"'                )
```

Match Objects

Match objects are created and returned by the methods `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 argument on each call to `repl`. A match object `m` supplies the following read-only attributes that detail how `m` was created:

`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`, `match`, `search`, `sub`, or `subn`

A match object `m` also supplies several methods, detailed in Table 9-3.

Table 9-3.

end, span, start	`m.end(groupid=0)` `m.span(groupid=0)` `m.start(groupid=0)` These methods return the limit indices, within `m.string`, of the substring that matches the group identified by `groupid` (a group number or name; "group `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 back references are replaced in the same way as for the method `r.sub`, covered in Table 9-2.
group	`m.group(groupid=0,*groupids)` When called with a single argument `groupid` (a group number or name), `group` returns the substring that matches the group identified by `groupid`, or `None` if that group did not participate in the match. The idiom *`m`*`.group()`, also spelled *`m`*`.group(0)`, returns the whole matched substring, since group number `0` means the whole RE. 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` 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 that matches the corresponding group, or `default` if 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.

Functions of the re Module

The `re` module supplies the attributes listed in "Optional Flags". It also provides one function for each method of a regular expression object (`findall`, `finditer`, `match`, `search`, `split`, `sub`, and `subn`), each with an additional first argument, a pattern string that the function implicitly compiles into an RE object. It's 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 slightly 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'(?i)hello', '', astring)
print('Found', count, "hello"'                  'occurrences of'                )
```

In such cases, RE options (here, case insensitivity) must be encoded as RE pattern elements (here `(?i)`): the functions of the `re` module do not accept a `flags` argument. 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:

compile `compile(pattern,flags=0)`

Creates and returns an RE object, parsing string `pattern` as per the syntax covered in "Pattern-String Syntax", and using integer `flags`, as covered 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.