**Introduction**

This new regex implementation is intended eventually to replace Python's current re module implementation.

For testing and comparison with the current 're' module the new implementation is in the form of a module called 'regex'.

Also included are the compiled binary .pyd files for Python 2.5-2.7 and Python 3.1-3.4 on 32-bit Windows.

**Old vs new behaviour**

This module has 2 behaviours:

Version 0 behaviour (old behaviour, compatible with the current re module):

Indicated by the VERSION0 or V0 flag, or (?V0) in the pattern.

.split won't split a string at a zero-width match.

Zero-width matches are handled like in the re module.

Inline flags apply to the entire pattern, and they can't be turned off.

Only simple sets are supported.

Case-insensitive matches in Unicode use simple case-folding by default.

Version 1 behaviour (new behaviour, different from the current re module):

Indicated by the VERSION1 or V1 flag, or (?V1) in the pattern.

.split will split a string at a zero-width match.

Zero-width matches are handled like in Perl and PCRE.

Inline flags apply to the end of the group or pattern, and they can be turned off.

Nested sets and set operations are supported.

Case-insensitive matches in Unicode use full case-folding by default.

If no version is specified, the regex module will default to regex.DEFAULT\_VERSION. In the short term this will be VERSION0, but in the longer term it will be VERSION1.

**Case-insensitive matches in Unicode**

The regex module supports both simple and full case-folding for case-insensitive matches in Unicode. Use of full case-folding can be turned on using the FULLCASE or F flag, or (?f) in the pattern. Please note that this flag affects how the IGNORECASE flag works; the FULLCASE flag itself does not turn on case-insensitive matching.

In the version 0 behaviour, the flag is off by default.

In the version 1 behaviour, the flag is on by default.

**Nested sets and set operations**

It's not possible to support both simple sets, as used in the re module, and nested sets at the same time because of a difference in the meaning of an unescaped "[" in a set.

For example, the pattern [[a-z]--[aeiou]] is treated in the version 0 behaviour (simple sets, compatible with the re module) as:

Set containing "[[" and the letters "a" to "z"

Literal "--"

Set containing letters "a", "e", "i", "o", "u"

but in the version 1 behaviour (nested sets, enhanced behaviour) as:

Set which is:

Set containing the letters "a" to "z"

but excluding:

Set containing the letters "a", "e", "i", "o", "u"

Version 0 behaviour: only simple sets are supported.

Version 1 behaviour: nested sets and set operations are supported.

**Flags**

There are 2 kinds of flag: scoped and global. Scoped flags can apply to only part of a pattern and can be turned on or off; global flags apply to the entire pattern and can only be turned on.

The scoped flags are: FULLCASE, IGNORECASE, MULTILINE, DOTALL, VERBOSE, WORD.

The global flags are: ASCII, BESTMATCH, ENHANCEMATCH, LOCALE, REVERSE, UNICODE, VERSION0, VERSION1.

If neither the ASCII, LOCALE nor UNICODE flag is specified, it will default to UNICODE if the regex pattern is a Unicode string and ASCII if it's a bytestring.

The ENHANCEMATCH flag makes fuzzy matching attempt to improve the fit of the next match that it finds.

The BESTMATCH flag makes fuzzy matching search for the best match instead of the next match.

**Notes on named capture groups**

All capture groups have a group number, starting from 1.

Groups with the same group name will have the same group number, and groups with a different group name will have a different group number.

The same name can be used by more than one group, with later captures 'overwriting' earlier captures. All of the captures of the group will be available from the captures method of the match object.

Group numbers will be reused across different branches of a branch reset, eg. (?|(first)|(second)) has only group 1. If capture groups have different group names then they will, of course, have different group numbers, eg. (?|(?P<foo>first)|(?P<bar>second)) has group 1 ("foo") and group 2 ("bar").

In the regex (\s+)(?|(?P<foo>[A-Z]+)|(\w+) (?<foo>[0-9]+) there are 2 groups:

1. (\s+) is group 1.

2. (?P<foo>[A-Z]+) is group 2, also called "foo".

3. (\w+) is group 2 because of the branch reset.

4. (?<foo>[0-9]+) is group 2 because it's called "foo".

If you want to prevent (\w+) from being group 2, you need to name it (different name, different group number).

**Multithreading**

The regex module releases the GIL during matching on instances of the built-in (immutable) string classes, enabling other Python threads to run concurrently. It is also possible to force the regex module to release the GIL during matching by calling the matching methods with the keyword argument concurrent=True. The behaviour is undefined if the string changes during matching, so use it **only** when it is guaranteed that that won't happen.

**Building for 64-bits**

If the source files are built for a 64-bit target then the string positions will also be 64-bit.

**Unicode**

This module supports Unicode 7.0.

Full Unicode case-folding is supported.

**Additional features**

The issue numbers relate to the Python bug tracker, except where listed as "Hg issue".

**Added partial matches (Hg**[**~~issue 102~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=102)**)**

A partial match is one that matches up to the end of string, but that string has been truncated and you want to know whether a complete match could be possible if the string had not been truncated.

Partial matches are supported by match, search, fullmatch and finditer with the partial keyword argument.

Match objects have a partial attribute, which is True if it's a partial match.

For example, if you wanted a user to enter a 4-digit number and check it character by character as it was being entered:

        >>> pattern = regex.compile(r'\d{4}')  
  
        >>> # Initially, nothing has been entered:  
        >>> print(pattern.fullmatch('', partial=True))  
        <regex.Match object; span=(0, 0), match='', partial=True>  
  
        >>> # An empty string is OK, but it's only a partial match.  
        >>> # The user enters a letter:  
        >>> print(pattern.fullmatch('a', partial=True))  
        None  
        >>> # It'll never match.  
  
        >>> # The user deletes that and enters a digit:  
        >>> print(pattern.fullmatch('1', partial=True))  
        <regex.Match object; span=(0, 1), match='1', partial=True>  
        >>> # It matches this far, but it's only a partial match.  
  
        >>> # The user enters 2 more digits:  
        >>> print(pattern.fullmatch('123', partial=True))  
        <regex.Match object; span=(0, 3), match='123', partial=True>  
        >>> # It matches this far, but it's only a partial match.  
  
        >>> # The user enters another digit:  
        >>> print(pattern.fullmatch('1234', partial=True))  
        <regex.Match object; span=(0, 4), match='1234'>  
        >>> # It's a complete match.  
  
        >>> # If the user enters another digit:  
        >>> print(pattern.fullmatch('12345', partial=True))  
        None  
        >>> # It's no longer a match.  
  
        >>> # This is a partial match:  
        >>> pattern.match('123', partial=True).partial  
        True  
  
        >>> # This is a complete match:  
        >>> pattern.match('1233', partial=True).partial  
        False

**\* operator not working correctly with sub() (Hg**[**~~issue 106~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=106)**)**

Sometimes it's not clear how zero-width matches should be handled. For example, should .\* match 0 characters directly after matching >0 characters?

Most regex implementations follow the lead of Perl (PCRE), but the re module sometimes doesn't. The Perl behaviour appears to be the most common (and the re module is sometimes definitely wrong), so in version 1 the regex module follows the Perl behaviour, whereas in version 0 it follows the legacy re behaviour.

Examples::

        # Version 0 behaviour (like re)  
        >>> regex.sub('(?V0).\*', 'x', 'test')  
        'x'  
        >>> regex.sub('(?V0).\*?', '|', 'test')  
        '|t|e|s|t|'  
  
        # Version 1 behaviour (like Perl)  
        >>> regex.sub('(?V1).\*', 'x', 'test')  
        'xx'  
        >>> regex.sub('(?V1).\*?', '|', 'test')  
        '|||||||||'

**re.group() should never return a bytearray (issue #18468)**

For compatibility with the re module, the regex module returns all matching bytestrings as bytes, starting from Python 3.4.

Examples::

        >>> # Python 3.4 and later  
        >>> import regex  
        >>> regex.match(b'.', bytearray(b'a')).group()  
        b'a'  
  
        >>> # Python 3.1-3.3  
        >>> import regex  
        >>> regex.match(b'.', bytearray(b'a')).group()  
        bytearray(b'a')

**Added capturesdict (Hg**[**~~issue 86~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=86)**)**

capturesdict is a combination of groupdict and captures:

groupdict returns a dict of the named groups and the last capture of those groups.

captures returns a list of all the captures of a group

capturesdict returns a dict of the named groups and lists of all the captures of those groups.

Examples::

        >>> import regex  
        >>> m = regex.match(r"(?:(?P<word>\w+) (?P<digits>\d+)\n)+", "one 1\ntwo 2\nthree 3\n")  
        >>> m.groupdict()  
        {'word': 'three', 'digits': '3'}  
        >>> m.captures("word")  
        ['one', 'two', 'three']  
        >>> m.captures("digits")  
        ['1', '2', '3']  
        >>> m.capturesdict()  
        {'word': ['one', 'two', 'three'], 'digits': ['1', '2', '3']}

**Allow duplicate names of groups (Hg**[**~~issue 87~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=87)**)**

Group names can now be duplicated.

Examples::

        >>> import regex  
        >>>  
        >>> # With optional groups:  
        >>>  
        >>> # Both groups capture, the second capture 'overwriting' the first.  
        >>> m = regex.match(r"(?P<item>\w+)? or (?P<item>\w+)?", "first or second")  
        >>> m.group("item")  
        'second'  
        >>> m.captures("item")  
        ['first', 'second']  
        >>> # Only the second group captures.  
        >>> m = regex.match(r"(?P<item>\w+)? or (?P<item>\w+)?", " or second")  
        >>> m.group("item")  
        'second'  
        >>> m.captures("item")  
        ['second']  
        >>> # Only the first group captures.  
        >>> m = regex.match(r"(?P<item>\w+)? or (?P<item>\w+)?", "first or ")  
        >>> m.group("item")  
        'first'  
        >>> m.captures("item")  
        ['first']  
        >>>  
        >>> # With mandatory groups:  
        >>>  
        >>> # Both groups capture, the second capture 'overwriting' the first.  
        >>> m = regex.match(r"(?P<item>\w\*) or (?P<item>\w\*)?", "first or second")  
        >>> m.group("item")  
        'second'  
        >>> m.captures("item")  
        ['first', 'second']  
        >>> # Again, both groups capture, the second capture 'overwriting' the first.  
        >>> m = regex.match(r"(?P<item>\w\*) or (?P<item>\w\*)", " or second")  
        >>> m.group("item")  
        'second'  
        >>> m.captures("item")  
        ['', 'second']  
        >>> # And yet again, both groups capture, the second capture 'overwriting' the first.  
        >>> m = regex.match(r"(?P<item>\w\*) or (?P<item>\w\*)", "first or ")  
        >>> m.group("item")  
        ''  
        >>> m.captures("item")  
        ['first', '']

**Added fullmatch (issue #16203)**

fullmatch behaves like match, except that it must match all of the string.

Examples::

        >>> import regex  
        >>> print(regex.fullmatch(r"abc", "abc").span())  
        (0, 3)  
        >>> print(regex.fullmatch(r"abc", "abcx"))  
        None  
        >>> print(regex.fullmatch(r"abc", "abcx", endpos=3).span())  
        (0, 3)  
        >>> print(regex.fullmatch(r"abc", "xabcy", pos=1, endpos=4).span())  
        (1, 4)  
        >>>  
        >>> regex.match(r"a.\*?", "abcd").group(0)  
        'a'  
        >>> regex.fullmatch(r"a.\*?", "abcd").group(0)  
        'abcd'

**Added subf and subfn (Python 2.6 and above)**

subf and subfn are alternatives to sub and subn respectively. When passed a replacement string, they treat it as a format string.

Examples::

        >>> import regex  
        >>> regex.subf(r"(\w+) (\w+)", "{0} => {2} {1}", "foo bar")  
        'foo bar => bar foo'  
        >>> regex.subf(r"(?P<word1>\w+) (?P<word2>\w+)", "{word2} {word1}", "foo bar")  
        'bar foo'

**Added expandf to match object (Python 2.6 and above)**

expandf is an alternative to expand. When passed a replacement string, it treats it as a format string.

Examples::

        >>> import regex  
        >>> m = regex.match(r"(\w+) (\w+)", "foo bar")  
        >>> m.expandf("{0} => {2} {1}")  
        'foo bar => bar foo'  
        >>>  
        >>> m = regex.match(r"(?P<word1>\w+) (?P<word2>\w+)", "foo bar")  
        >>> m.expandf("{word2} {word1}")  
        'bar foo'

**Detach searched string**

A match object contains a reference to the string that was searched, via its string attribute. The match object now has adetach\_string method that will 'detach' that string, making it available for garbage collection (this might save valuable memory if that string is very large).

Example::

        >>> import regex  
        >>> m = regex.search(r"\w+", "Hello world")  
        >>> print(m.group())  
        Hello  
        >>> print(m.string)  
        Hello world  
        >>> m.detach\_string()  
        >>> print(m.group())  
        Hello  
        >>> print(m.string)  
        None

**Characters in a group name (issue #14462)**

A group name can now contain the same characters as an identifier. These are different in Python 2 and Python 3.

**Recursive patterns (Hg**[**~~issue 27~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=27)**)**

Recursive and repeated patterns are supported.

(?R) or (?0) tries to match the entire regex recursively. (?1), (?2), etc, try to match the relevant capture group.

(?&name) tries to match the named capture group.

Examples::

        >>> regex.match(r"(Tarzan|Jane) loves (?1)", "Tarzan loves Jane").groups()  
        ('Tarzan',)  
        >>> regex.match(r"(Tarzan|Jane) loves (?1)", "Jane loves Tarzan").groups()  
        ('Jane',)  
  
        >>> m = regex.search(r"(\w)(?:(?R)|(\w?))\1", "kayak")  
        >>> m.group(0, 1, 2)  
        ('kayak', 'k', None)

The first two examples show how the subpattern within the capture group is reused, but is *not* itself a capture group. In other words, "(Tarzan|Jane) loves (?1)" is equivalent to "(Tarzan|Jane) loves (?:Tarzan|Jane)".

It's possible to backtrack into a recursed or repeated group.

You can't call a group if there is more than one group with that group name or group number ("ambiguous group reference"). For example, (?P<foo>\w+) (?P<foo>\w+) (?&foo)? has 2 groups called "foo" (both group 1) and (?|([A-Z]+)|([0-9]+)) (?1)? has 2 groups with group number 1.

The alternative forms (?P>name) and (?P&name) are also supported.

**repr(regex) doesn't include actual regex (issue #13592)**

The repr of a compiled regex is now in the form of a eval-able string. For example::

        >>> r = regex.compile("foo", regex.I)  
        >>> repr(r)  
        "regex.Regex('foo', flags=regex.I | regex.V0)"  
        >>> r  
        regex.Regex('foo', flags=regex.I | regex.V0)

The regex module has Regex as an alias for the 'compile' function.

**Improve the repr for regular expression match objects (issue #17087)**

The repr of a match object is now a more useful form. For example::

        >>> regex.search(r"\d+", "abc012def")  
        <regex.Match object; span=(3, 6), match='012'>

**Python lib re cannot handle Unicode properly due to narrow/wide bug (issue #12729)**

The source code of the regex module has been updated to support PEP 393 ("Flexible String Representation"), which is new in Python 3.3.

**Full Unicode case-folding is supported.**

In version 1 behaviour, the regex module uses full case-folding when performing case-insensitive matches in Unicode.

Examples (in Python 3):

        >>> regex.match(r"(?iV1)strasse", "stra\N{LATIN SMALL LETTER SHARP S}e").span()  
        (0, 6)  
        >>> regex.match(r"(?iV1)stra\N{LATIN SMALL LETTER SHARP S}e", "STRASSE").span()  
        (0, 7)

In version 0 behaviour, it uses simple case-folding for backward compatibility with the re module.

**Approximate "fuzzy" matching (Hg**[**~~issue 12~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=12)**, Hg**[**~~issue 41~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=41)**, Hg**[**~~issue 109~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=109)**)**

Regex usually attempts an exact match, but sometimes an approximate, or "fuzzy", match is needed, for those cases where the text being searched may contain errors in the form of inserted, deleted or substituted characters.

A fuzzy regex specifies which types of errors are permitted, and, optionally, either the minimum and maximum or only the maximum permitted number of each type. (You cannot specify only a minimum.)

The 3 types of error are:

* Insertion, indicated by "i"
* Deletion, indicated by "d"
* Substitution, indicated by "s"

In addition, "e" indicates any type of error.

The fuzziness of a regex item is specified between "{" and "}" after the item.

Examples:

    foo match "foo" exactly  
  
    (?:foo){i} match "foo", permitting insertions  
  
    (?:foo){d} match "foo", permitting deletions  
  
    (?:foo){s} match "foo", permitting substitutions  
  
    (?:foo){i,s} match "foo", permitting insertions and substitutions  
  
    (?:foo){e} match "foo", permitting errors

If a certain type of error is specified, then any type not specified will not be permitted.

In the following examples I'll omit the item and write only the fuzziness.

    {i<=3} permit at most 3 insertions, but no other types  
  
    {d<=3} permit at most 3 deletions, but no other types  
  
    {s<=3} permit at most 3 substitutions, but no other types  
  
    {i<=1,s<=2} permit at most 1 insertion and at most 2 substitutions, but no deletions  
  
    {e<=3} permit at most 3 errors  
  
    {1<=e<=3} permit at least 1 and at most 3 errors  
  
    {i<=2,d<=2,e<=3} permit at most 2 insertions, at most 2 deletions, at most 3 errors in total, but no substitutions

It's also possible to state the costs of each type of error and the maximum permitted total cost.

Examples:

    {2i+2d+1s<=4} each insertion costs 2, each deletion costs 2, each substitution costs 1, the total cost must not exceed 4  
  
    {i<=1,d<=1,s<=1,2i+2d+1s<=4} at most 1 insertion, at most 1 deletion, at most 1 substitution; each insertion costs 2, each deletion costs 2, each substitution costs 1, the total cost must not exceed 4

You can also use "<" instead of "<=" if you want an exclusive minimum or maximum:

    {e<=3} permit up to 3 errors  
  
    {e<4} permit fewer than 4 errors  
  
    {0<e<4} permit more than 0 but fewer than 4 errors

By default, fuzzy matching searches for the first match that meets the given constraints. The ENHANCEMATCH flag will cause it to attempt to improve the fit (i.e. reduce the number of errors) of the match that it has found.

The BESTMATCH flag will make it search for the best match instead.

Further examples to note:

    regex.search("(dog){e}", "cat and dog")[1] returns "cat" because that matches "dog" with 3 errors, which is within the limit (an unlimited number of errors is permitted).  
  
    regex.search("(dog){e<=1}", "cat and dog")[1] returns " dog" (with a leading space) because that matches "dog" with 1 error, which is within the limit (1 error is permitted).  
  
    regex.search("(?e)(dog){e<=1}", "cat and dog")[1] returns "dog" (without a leading space) because the fuzzy search matches " dog" with 1 error, which is within the limit (1 error is permitted), and the (?e) then makes it attempt a better fit.

In the first two examples there are perfect matches later in the string, but in neither case is it the first possible match.

The match object has an attribute fuzzy\_counts which gives the total number of substitutions, insertions and deletions.

        >>> # A 'raw' fuzzy match:  
        >>> regex.fullmatch(r"(?:cats|cat){e<=1}", "cat").fuzzy\_counts  
        (0, 0, 1)  
        >>> # 0 substitutions, 0 insertions, 1 deletion.  
  
        >>> # A better match might be possible if the ENHANCEMATCH flag used:  
        >>> regex.fullmatch(r"(?e)(?:cats|cat){e<=1}", "cat").fuzzy\_counts  
        (0, 0, 0)  
        >>> # 0 substitutions, 0 insertions, 0 deletions.

**Named lists (Hg**[**~~issue 11~~**](https://code.google.com/p/mrab-regex-hg/issues/detail?id=11)**)**

\L<name>

There are occasions where you may want to include a list (actually, a set) of options in a regex.

One way is to build the pattern like this::

        p = regex.compile(r"first|second|third|fourth|fifth")

but if the list is large, parsing the resulting regex can take considerable time, and care must also be taken that the strings are properly escaped if they contain any character that has a special meaning in a regex, and that if there is a shorter string that occurs initially in a longer string that the longer string is listed before the shorter one, for example, "cats" before "cat".

The new alternative is to use a named list::

        option\_set = ["first", "second", "third", "fourth", "fifth"]  
        p = regex.compile(r"\L<options>", options=option\_set)

The order of the items is irrelevant, they are treated as a set. The named lists are available as the .named\_lists attribute of the pattern object ::

        >>> print(p.named\_lists)  
        {'options': frozenset({'second', 'fifth', 'fourth', 'third', 'first'})}

**Start and end of word**

\m matches at the start of a word.

\M matches at the end of a word.

Compare with \b, which matches at the start or end of a word.

**Unicode line separators**

Normally the only line separator is \n (\x0A), but if the WORD flag is turned on then the line separators are the pair \x0D\x0A, and \x0A,\x0B, \x0C and \x0D, plus \x85, \u2028 and \u2029 when working with Unicode.

This affects the regex dot ".", which, with the DOTALL flag turned off, matches any character except a line separator. It also affects the line anchors ^ and $ (in multiline mode).

**Set operators**

* Version 1 behaviour only

Set operators have been added, and a set [...] can include nested sets.

The operators, in order of increasing precedence, are:

|| for union ("x||y" means "x or y")

~~ (double tilde) for symmetric difference ("x~~y" means "x or y, but not both")

&& for intersection ("x&&y" means "x and y")

-- (double dash) for difference ("x--y" means "x but not y")

Implicit union, ie, simple juxtaposition like in [ab], has the highest precedence. Thus, [ab&&cd] is the same as [[a||b]&&[c||d]].

Examples:

        [ab] # Set containing 'a' and 'b'  
  
        [a-z] # Set containing 'a' .. 'z'  
  
        [[a-z]--[qw]] # Set containing 'a' .. 'z', but not 'q' or 'w'  
  
        [a-z--qw] # Same as above  
  
        [\p{L}--QW] # Set containing all letters except 'Q' and 'W'  
  
        [\p{N}--[0-9]] # Set containing all numbers except '0' .. '9'  
  
        [\p{ASCII}&&\p{Letter}] # Set containing all characters which are ASCII and letter

**regex.escape (issue #2650)**

regex.escape has an additional keyword parameter special\_only. When True, only 'special' regex characters, such as '?', are escaped.

Examples:

        >>> regex.escape("foo!?")  
        'foo\\!\\?'  
        >>> regex.escape("foo!?", special\_only=True)  
        'foo!\\?'

**Repeated captures (issue #7132)**

A match object has additional methods which return information on all the successful matches of a repeated capture group. These methods are:

matchobject.captures([group1, ...])

Returns a list of the strings matched in a group or groups. Compare with matchobject.group([group1, ...]).

matchobject.starts([group])

Returns a list of the start positions. Compare with matchobject.start([group]).

matchobject.ends([group])

Returns a list of the end positions. Compare with matchobject.end([group]).

matchobject.spans([group])

Returns a list of the spans. Compare with matchobject.span([group]).

Examples:

        >>> m = regex.search(r"(\w{3})+", "123456789")  
        >>> m.group(1)  
        '789'  
        >>> m.captures(1)  
        ['123', '456', '789']  
        >>> m.start(1)  
        6  
        >>> m.starts(1)  
        [0, 3, 6]  
        >>> m.end(1)  
        9  
        >>> m.ends(1)  
        [3, 6, 9]  
        >>> m.span(1)  
        (6, 9)  
        >>> m.spans(1)  
        [(0, 3), (3, 6), (6, 9)]

**Atomic grouping (issue #433030)**

(?>...)

If the following pattern subsequently fails, then the subpattern as a whole will fail.

**Possessive quantifiers.**

(?:...)?+ ; (?:...)\*+ ; (?:...)++ ; (?:...){min,max}+

The subpattern is matched up to 'max' times. If the following pattern subsequently fails, then all of the repeated subpatterns will fail as a whole. For example, (?:...)++ is equivalent to (?>(?:...)+).

**Scoped flags (issue #433028)**

(?flags-flags:...)

The flags will apply only to the subpattern. Flags can be turned on or off.

**Inline flags (issue #433024, issue #433027)**

(?flags-flags)

Version 0 behaviour: the flags apply to the entire pattern, and they can't be turned off.

Version 1 behaviour: the flags apply to the end of the group or pattern, and they can be turned on or off.

**Repeated repeats (issue #2537)**

A regex like ((x|y+)\*)\* will be accepted and will work correctly, but should complete more quickly.

**Definition of 'word' character (issue #1693050)**

The definition of a 'word' character has been expanded for Unicode. It now conforms to the Unicode specification athttp://www.unicode.org/reports/tr29/. This applies to \w, \W, \b and \B.

**Groups in lookahead and lookbehind (issue #814253)**

Groups and group references are permitted in both lookahead and lookbehind.

**Variable-length lookbehind**

A lookbehind can match a variable-length string.

**Correct handling of charset with ignore case flag (issue #3511)**

Ranges within charsets are handled correctly when the ignore-case flag is turned on.

**Unmatched group in replacement (issue #1519638)**

An unmatched group is treated as an empty string in a replacement template.

**'Pathological' patterns (issue #1566086, issue #1662581, issue #1448325, issue #1721518, issue #1297193)**

'Pathological' patterns should complete more quickly.

**Flags argument for regex.split, regex.sub and regex.subn (issue #3482)**

regex.split, regex.sub and regex.subn support a 'flags' argument.

**Pos and endpos arguments for regex.sub and regex.subn**

regex.sub and regex.subn support 'pos' and 'endpos' arguments.

**'Overlapped' argument for regex.findall and regex.finditer**

regex.findall and regex.finditer support an 'overlapped' flag which permits overlapped matches.

**Unicode escapes (issue #3665)**

The Unicode escapes \uxxxx and \Uxxxxxxxx are supported.

**Large patterns (issue #1160)**

Patterns can be much larger.

**Zero-width match with regex.finditer (issue #1647489)**

regex.finditer behaves correctly when it splits at a zero-width match.

**Zero-width split with regex.split (issue #3262)**

Version 0 behaviour: a string won't be split at a zero-width match.

Version 1 behaviour: a string will be split at a zero-width match.

**Splititer**

regex.splititer has been added. It's a generator equivalent of regex.split.

**Subscripting for groups**

A match object accepts access to the captured groups via subscripting and slicing:

    >>> m = regex.search(r"(?P<before>.\*?)(?P<num>\d+)(?P<after>.\*)", "pqr123stu")  
    >>> print m["before"]  
    pqr  
    >>> print m["num"]  
    123  
    >>> print m["after"]  
    stu  
    >>> print len(m)  
    4  
    >>> print m[:]  
    ('pqr123stu', 'pqr', '123', 'stu')

**Named groups**

Groups can be named with (?<name>...) as well as the current (?P<name>...).

**Group references**

Groups can be referenced within a pattern with \g<name>. This also allows there to be more than 99 groups.

**Named characters**

\N{name}

Named characters are supported. (Note: only those known by Python's Unicode database are supported.)

**Unicode codepoint properties, including scripts and blocks**

\p{property=value}; \P{property=value}; \p{value} ; \P{value}

Many Unicode properties are supported, including blocks and scripts. \p{property=value} or \p{property:value} matches a character whose property property has value value. The inverse of \p{property=value} is \P{property=value} or\p{^property=value}.

If the short form \p{value} is used, the properties are checked in the order: General\_Category, Script, Block, binary property:

1. Latin, the 'Latin' script (Script=Latin).

2. Cyrillic, the 'Cyrillic' script (Script=Cyrillic).

3. BasicLatin, the 'BasicLatin' block (Block=BasicLatin).

4. Alphabetic, the 'Alphabetic' binary property (Alphabetic=Yes).

A short form starting with Is indicates a script or binary property:

1. IsLatin, the 'Latin' script (Script=Latin).

2. IsCyrillic, the 'Cyrillic' script (Script=Cyrillic).

3. IsAlphabetic, the 'Alphabetic' binary property (Alphabetic=Yes).

A short form starting with In indicates a block property:

1. InBasicLatin, the 'BasicLatin' block (Block=BasicLatin).

2. InCyrillic, the 'Cyrillic' block (Block=Cyrillic).

**POSIX character classes**

[[:alpha:]]; [[:^alpha:]]

POSIX character classes are supported. This is actually treated as an alternative form of \p{...}.

**Search anchor**

\G

A search anchor has been added. It matches at the position where each search started/continued and can be used for contiguous matches or in negative variable-length lookbehinds to limit how far back the lookbehind goes:

    >>> regex.findall(r"\w{2}", "abcd ef")  
    ['ab', 'cd', 'ef']  
    >>> regex.findall(r"\G\w{2}", "abcd ef")  
    ['ab', 'cd']

1. The search starts at position 0 and matches 2 letters 'ab'.

2. The search continues at position 2 and matches 2 letters 'cd'.

3. The search continues at position 4 and fails to match any letters.

4. The anchor stops the search start position from being advanced, so there are no more results.

**Reverse searching**

Searches can now work backwards:

    >>> regex.findall(r".", "abc")  
    ['a', 'b', 'c']  
    >>> regex.findall(r"(?r).", "abc")  
    ['c', 'b', 'a']

Note: the result of a reverse search is not necessarily the reverse of a forward search:

    >>> regex.findall(r"..", "abcde")  
    ['ab', 'cd']  
    >>> regex.findall(r"(?r)..", "abcde")  
    ['de', 'bc']

**Matching a single grapheme**

\X

The grapheme matcher is supported. It now conforms to the Unicode specification at http://www.unicode.org/reports/tr29/.

**Branch reset**

(?|...|...)

Capture group numbers will be reused across the alternatives, but groups with different names will have different group numbers.

Examples::

    >>> import regex  
    >>> regex.match(r"(?|(first)|(second))", "first").groups()  
    ('first',)  
    >>> regex.match(r"(?|(first)|(second))", "second").groups()  
    ('second',)

Note that there is only one group.

**Default Unicode word boundary**

The WORD flag changes the definition of a 'word boundary' to that of a default Unicode word boundary. This applies to \b and \B.

**SRE engine do not release the GIL (issue #1366311)**

The regex module can release the GIL during matching (see the above section on multithreading).

Iterators can be safely shared across threads.

# regex 2014.08.28

*Alternative regular expression module, to replace re.*

# Introduction

This new regex implementation is intended eventually to replace Python's current re module implementation.

For testing and comparison with the current 're' module the new implementation is in the form of a module called 'regex'.

Also included are the compiled binary .pyd files for Python 2.5-2.7 and Python 3.1-3.4 on 32-bit Windows.

# Old vs new behaviour

This module has 2 behaviours:

**Version 0** behaviour (old behaviour, compatible with the current re module):

Indicated by the VERSION0 or V0 flag, or (?V0) in the pattern.

.split won't split a string at a zero-width match.

Zero-width matches are handled like in the re module.

Inline flags apply to the entire pattern, and they can't be turned off.

Only simple sets are supported.

Case-insensitive matches in Unicode use simple case-folding by default.

**Version 1** behaviour (new behaviour, different from the current re module):

Indicated by the VERSION1 or V1 flag, or (?V1) in the pattern.

.split will split a string at a zero-width match.

Zero-width matches are handled like in Perl and PCRE.

Inline flags apply to the end of the group or pattern, and they can be turned off.

Nested sets and set operations are supported.

Case-insensitive matches in Unicode use full case-folding by default.

If no version is specified, the regex module will default to regex.DEFAULT\_VERSION. In the short term this will be VERSION0, but in the longer term it will be VERSION1.

# Case-insensitive matches in Unicode

The regex module supports both simple and full case-folding for case-insensitive matches in Unicode. Use of full case-folding can be turned on using theFULLCASE or F flag, or (?f) in the pattern. Please note that this flag affects how the IGNORECASE flag works; the FULLCASE flag itself does not turn on case-insensitive matching.

In the version 0 behaviour, the flag is off by default.

In the version 1 behaviour, the flag is on by default.

# Nested sets and set operations

It's not possible to support both simple sets, as used in the re module, and nested sets at the same time because of a difference in the meaning of an unescaped "[" in a set.

For example, the pattern [[a-z]--[aeiou]] is treated in the version 0 behaviour (simple sets, compatible with the re module) as:

Set containing "[" and the letters "a" to "z"

Literal "--"

Set containing letters "a", "e", "i", "o", "u"

but in the version 1 behaviour (nested sets, enhanced behaviour) as:

Set which is:

Set containing the letters "a" to "z"

but excluding:

Set containing the letters "a", "e", "i", "o", "u"

Version 0 behaviour: only simple sets are supported.

Version 1 behaviour: nested sets and set operations are supported.

# Flags

There are 2 kinds of flag: scoped and global. Scoped flags can apply to only part of a pattern and can be turned on or off; global flags apply to the entire pattern and can only be turned on.

The scoped flags are: FULLCASE, IGNORECASE, MULTILINE, DOTALL, VERBOSE, WORD.

The global flags are: ASCII, BESTMATCH, ENHANCEMATCH, LOCALE, REVERSE, UNICODE, VERSION0, VERSION1.

If neither the ASCII, LOCALE nor UNICODE flag is specified, it will default to UNICODE if the regex pattern is a Unicode string and ASCII if it's a bytestring.

The ENHANCEMATCH flag makes fuzzy matching attempt to improve the fit of the next match that it finds.

The BESTMATCH flag makes fuzzy matching search for the best match instead of the next match.

# Notes on named capture groups

All capture groups have a group number, starting from 1.

Groups with the same group name will have the same group number, and groups with a different group name will have a different group number.

The same name can be used by more than one group, with later captures 'overwriting' earlier captures. All of the captures of the group will be available from the captures method of the match object.

Group numbers will be reused across different branches of a branch reset, eg. (?|(first)|(second)) has only group 1. If capture groups have different group names then they will, of course, have different group numbers, eg. (?|(?P<foo>first)|(?P<bar>second)) has group 1 ("foo") and group 2 ("bar").

In the regex (\s+)(?|(?P<foo>[A-Z]+)|(\w+) (?<foo>[0-9]+) there are 2 groups:

1. (\s+) is group 1.
2. (?P<foo>[A-Z]+) is group 2, also called "foo".
3. (\w+) is group 2 because of the branch reset.
4. (?<foo>[0-9]+) is group 2 because it's called "foo".

If you want to prevent (\w+) from being group 2, you need to name it (different name, different group number).

# Multithreading

The regex module releases the GIL during matching on instances of the built-in (immutable) string classes, enabling other Python threads to run concurrently. It is also possible to force the regex module to release the GIL during matching by calling the matching methods with the keyword argumentconcurrent=True. The behaviour is undefined if the string changes during matching, so use it only when it is guaranteed that that won't happen.

# Building for 64-bits

If the source files are built for a 64-bit target then the string positions will also be 64-bit.

# Unicode

This module supports Unicode 7.0.

Full Unicode case-folding is supported.

# Additional features

The issue numbers relate to the Python bug tracker, except where listed as "Hg issue".

* Added partial matches (Hg issue 102)

A partial match is one that matches up to the end of string, but that string has been truncated and you want to know whether a complete match could be possible if the string had not been truncated.

Partial matches are supported by match, search, fullmatch and finditer with the partial keyword argument.

Match objects have a partial attribute, which is True if it's a partial match.

For example, if you wanted a user to enter a 4-digit number and check it character by character as it was being entered:

>>> pattern = regex.compile(r'\d{4}')

>>> # Initially, nothing has been entered:

>>> print(pattern.fullmatch('', partial=True))

<regex.Match object; span=(0, 0), match='', partial=True>

>>> # An empty string is OK, but it's only a partial match.

>>> # The user enters a letter:

>>> print(pattern.fullmatch('a', partial=True))

None

>>> # It'll never match.

>>> # The user deletes that and enters a digit:

>>> print(pattern.fullmatch('1', partial=True))

<regex.Match object; span=(0, 1), match='1', partial=True>

>>> # It matches this far, but it's only a partial match.

>>> # The user enters 2 more digits:

>>> print(pattern.fullmatch('123', partial=True))

<regex.Match object; span=(0, 3), match='123', partial=True>

>>> # It matches this far, but it's only a partial match.

>>> # The user enters another digit:

>>> print(pattern.fullmatch('1234', partial=True))

<regex.Match object; span=(0, 4), match='1234'>

>>> # It's a complete match.

>>> # If the user enters another digit:

>>> print(pattern.fullmatch('12345', partial=True))

None

>>> # It's no longer a match.

>>> # This is a partial match:

>>> pattern.match('123', partial=True).partial

True

>>> # This is a complete match:

>>> pattern.match('1233', partial=True).partial

False

* \* operator not working correctly with sub() (Hg issue 106)

Sometimes it's not clear how zero-width matches should be handled. For example, should .\* match 0 characters directly after matching >0 characters?

Most regex implementations follow the lead of Perl (PCRE), but the re module sometimes doesn't. The Perl behaviour appears to be the most common (and the re module is sometimes definitely wrong), so in version 1 the regex module follows the Perl behaviour, whereas in version 0 it follows the legacy re behaviour.

Examples:

# Version 0 behaviour (like re)

>>> regex.sub('(?V0).\*', 'x', 'test')

'x'

>>> regex.sub('(?V0).\*?', '|', 'test')

'|t|e|s|t|'

# Version 1 behaviour (like Perl)

>>> regex.sub('(?V1).\*', 'x', 'test')

'xx'

>>> regex.sub('(?V1).\*?', '|', 'test')

'|||||||||'

* re.group() should never return a bytearray (issue #18468)

For compatibility with the re module, the regex module returns all matching bytestrings as bytes, starting from Python 3.4.

Examples:

>>> # Python 3.4 and later

>>> import regex

>>> regex.match(b'.', bytearray(b'a')).group()

b'a'

>>> # Python 3.1-3.3

>>> import regex

>>> regex.match(b'.', bytearray(b'a')).group()

bytearray(b'a')

* Added capturesdict (Hg issue 86)

capturesdict is a combination of groupdict and captures:

groupdict returns a dict of the named groups and the last capture of those groups.

captures returns a list of all the captures of a group

capturesdict returns a dict of the named groups and lists of all the captures of those groups.

Examples:

>>> import regex

>>> m = regex.match(r"(?:(?P<word>\w+) (?P<digits>\d+)\n)+", "one 1\ntwo 2\nthree 3\n")

>>> m.groupdict()

{'word': 'three', 'digits': '3'}

>>> m.captures("word")

['one', 'two', 'three']

>>> m.captures("digits")

['1', '2', '3']

>>> m.capturesdict()

{'word': ['one', 'two', 'three'], 'digits': ['1', '2', '3']}

* Allow duplicate names of groups (Hg issue 87)

Group names can now be duplicated.

Examples:

>>> import regex

>>>

>>> # With optional groups:

>>>

>>> # Both groups capture, the second capture 'overwriting' the first.

>>> m = regex.match(r"(?P<item>\w+)? or (?P<item>\w+)?", "first or second")

>>> m.group("item")

'second'

>>> m.captures("item")

['first', 'second']

>>> # Only the second group captures.

>>> m = regex.match(r"(?P<item>\w+)? or (?P<item>\w+)?", " or second")

>>> m.group("item")

'second'

>>> m.captures("item")

['second']

>>> # Only the first group captures.

>>> m = regex.match(r"(?P<item>\w+)? or (?P<item>\w+)?", "first or ")

>>> m.group("item")

'first'

>>> m.captures("item")

['first']

>>>

>>> # With mandatory groups:

>>>

>>> # Both groups capture, the second capture 'overwriting' the first.

>>> m = regex.match(r"(?P<item>\w\*) or (?P<item>\w\*)?", "first or second")

>>> m.group("item")

'second'

>>> m.captures("item")

['first', 'second']

>>> # Again, both groups capture, the second capture 'overwriting' the first.

>>> m = regex.match(r"(?P<item>\w\*) or (?P<item>\w\*)", " or second")

>>> m.group("item")

'second'

>>> m.captures("item")

['', 'second']

>>> # And yet again, both groups capture, the second capture 'overwriting' the first.

>>> m = regex.match(r"(?P<item>\w\*) or (?P<item>\w\*)", "first or ")

>>> m.group("item")

''

>>> m.captures("item")

['first', '']

* Added fullmatch (issue #16203)

fullmatch behaves like match, except that it must match all of the string.

Examples:

>>> import regex

>>> print(regex.fullmatch(r"abc", "abc").span())

(0, 3)

>>> print(regex.fullmatch(r"abc", "abcx"))

None

>>> print(regex.fullmatch(r"abc", "abcx", endpos=3).span())

(0, 3)

>>> print(regex.fullmatch(r"abc", "xabcy", pos=1, endpos=4).span())

(1, 4)

>>>

>>> regex.match(r"a.\*?", "abcd").group(0)

'a'

>>> regex.fullmatch(r"a.\*?", "abcd").group(0)

'abcd'

* Added subf and subfn **(Python 2.6 and above)**

subf and subfn are alternatives to sub and subn respectively. When passed a replacement string, they treat it as a format string.

Examples:

>>> import regex

>>> regex.subf(r"(\w+) (\w+)", "{0} => {2} {1}", "foo bar")

'foo bar => bar foo'

>>> regex.subf(r"(?P<word1>\w+) (?P<word2>\w+)", "{word2} {word1}", "foo bar")

'bar foo'

* Added expandf to match object **(Python 2.6 and above)**

expandf is an alternative to expand. When passed a replacement string, it treats it as a format string.

Examples:

>>> import regex

>>> m = regex.match(r"(\w+) (\w+)", "foo bar")

>>> m.expandf("{0} => {2} {1}")

'foo bar => bar foo'

>>>

>>> m = regex.match(r"(?P<word1>\w+) (?P<word2>\w+)", "foo bar")

>>> m.expandf("{word2} {word1}")

'bar foo'

* Detach searched string

A match object contains a reference to the string that was searched, via its string attribute. The match object now has a detach\_string method that will 'detach' that string, making it available for garbage collection (this might save valuable memory if that string is very large).

Example:

>>> import regex

>>> m = regex.search(r"\w+", "Hello world")

>>> print(m.group())

Hello

>>> print(m.string)

Hello world

>>> m.detach\_string()

>>> print(m.group())

Hello

>>> print(m.string)

None

* Characters in a group name (issue #14462)

A group name can now contain the same characters as an identifier. These are different in Python 2 and Python 3.

* Recursive patterns (Hg issue 27)

Recursive and repeated patterns are supported.

(?R) or (?0) tries to match the entire regex recursively. (?1), (?2), etc, try to match the relevant capture group.

(?&name) tries to match the named capture group.

Examples:

>>> regex.match(r"(Tarzan|Jane) loves (?1)", "Tarzan loves Jane").groups()

('Tarzan',)

>>> regex.match(r"(Tarzan|Jane) loves (?1)", "Jane loves Tarzan").groups()

('Jane',)

>>> m = regex.search(r"(\w)(?:(?R)|(\w?))\1", "kayak")

>>> m.group(0, 1, 2)

('kayak', 'k', None)

The first two examples show how the subpattern within the capture group is reused, but is \_not\_ itself a capture group. In other words, "(Tarzan|Jane) loves (?1)" is equivalent to "(Tarzan|Jane) loves (?:Tarzan|Jane)".

It's possible to backtrack into a recursed or repeated group.

You can't call a group if there is more than one group with that group name or group number ("ambiguous group reference"). For example,(?P<foo>\w+) (?P<foo>\w+) (?&foo)? has 2 groups called "foo" (both group 1) and (?|([A-Z]+)|([0-9]+)) (?1)? has 2 groups with group number 1.

The alternative forms (?P>name) and (?P&name) are also supported.

* repr(regex) doesn't include actual regex (issue #13592)

The repr of a compiled regex is now in the form of a eval-able string. For example:

>>> r = regex.compile("foo", regex.I)

>>> repr(r)

"regex.Regex('foo', flags=regex.I | regex.V0)"

>>> r

regex.Regex('foo', flags=regex.I | regex.V0)

The regex module has Regex as an alias for the 'compile' function.

* Improve the repr for regular expression match objects (issue #17087)

The repr of a match object is now a more useful form. For example:

>>> regex.search(r"\d+", "abc012def")

<regex.Match object; span=(3, 6), match='012'>

* Python lib re cannot handle Unicode properly due to narrow/wide bug (issue #12729)

The source code of the regex module has been updated to support PEP 393 ("Flexible String Representation"), which is new in Python 3.3.

* Full Unicode case-folding is supported.

In version 1 behaviour, the regex module uses full case-folding when performing case-insensitive matches in Unicode.

Examples (in Python 3):

>>> regex.match(r"(?iV1)strasse", "stra\N{LATIN SMALL LETTER SHARP S}e").span()

(0, 6)

>>> regex.match(r"(?iV1)stra\N{LATIN SMALL LETTER SHARP S}e", "STRASSE").span()

(0, 7)

In version 0 behaviour, it uses simple case-folding for backward compatibility with the re module.

* Approximate "fuzzy" matching (Hg issue 12, Hg issue 41, Hg issue 109)

Regex usually attempts an exact match, but sometimes an approximate, or "fuzzy", match is needed, for those cases where the text being searched may contain errors in the form of inserted, deleted or substituted characters.

A fuzzy regex specifies which types of errors are permitted, and, optionally, either the minimum and maximum or only the maximum permitted number of each type. (You cannot specify only a minimum.)

The 3 types of error are:

* + Insertion, indicated by "i"
  + Deletion, indicated by "d"
  + Substitution, indicated by "s"

In addition, "e" indicates any type of error.

The fuzziness of a regex item is specified between "{" and "}" after the item.

Examples:

foo match "foo" exactly

(?:foo){i} match "foo", permitting insertions

(?:foo){d} match "foo", permitting deletions

(?:foo){s} match "foo", permitting substitutions

(?:foo){i,s} match "foo", permitting insertions and substitutions

(?:foo){e} match "foo", permitting errors

If a certain type of error is specified, then any type not specified will **not** be permitted.

In the following examples I'll omit the item and write only the fuzziness.

{i<=3} permit at most 3 insertions, but no other types

{d<=3} permit at most 3 deletions, but no other types

{s<=3} permit at most 3 substitutions, but no other types

{i<=1,s<=2} permit at most 1 insertion and at most 2 substitutions, but no deletions

{e<=3} permit at most 3 errors

{1<=e<=3} permit at least 1 and at most 3 errors

{i<=2,d<=2,e<=3} permit at most 2 insertions, at most 2 deletions, at most 3 errors in total, but no substitutions

It's also possible to state the costs of each type of error and the maximum permitted total cost.

Examples:

{2i+2d+1s<=4} each insertion costs 2, each deletion costs 2, each substitution costs 1, the total cost must not exceed 4

{i<=1,d<=1,s<=1,2i+2d+1s<=4} at most 1 insertion, at most 1 deletion, at most 1 substitution; each insertion costs 2, each deletion costs 2, each substitution costs 1, the total cost must not exceed 4

You can also use "<" instead of "<=" if you want an exclusive minimum or maximum:

{e<=3} permit up to 3 errors

{e<4} permit fewer than 4 errors

{0<e<4} permit more than 0 but fewer than 4 errors

By default, fuzzy matching searches for the first match that meets the given constraints. The ENHANCEMATCH flag will cause it to attempt to improve the fit (i.e. reduce the number of errors) of the match that it has found.

The BESTMATCH flag will make it search for the best match instead.

Further examples to note:

regex.search("(dog){e}", "cat and dog")[1] returns "cat" because that matches "dog" with 3 errors, which is within the limit (an unlimited number of errors is permitted).

regex.search("(dog){e<=1}", "cat and dog")[1] returns " dog" (with a leading space) because that matches "dog" with 1 error, which is within the limit (1 error is permitted).

regex.search("(?e)(dog){e<=1}", "cat and dog")[1] returns "dog" (without a leading space) because the fuzzy search matches " dog" with 1 error, which is within the limit (1 error is permitted), and the (?e) then makes it attempt a better fit.

In the first two examples there are perfect matches later in the string, but in neither case is it the first possible match.

The match object has an attribute fuzzy\_counts which gives the total number of substitutions, insertions and deletions.

>>> # A 'raw' fuzzy match:

>>> regex.fullmatch(r"(?:cats|cat){e<=1}", "cat").fuzzy\_counts

(0, 0, 1)

>>> # 0 substitutions, 0 insertions, 1 deletion.

>>> # A better match might be possible if the ENHANCEMATCH flag used:

>>> regex.fullmatch(r"(?e)(?:cats|cat){e<=1}", "cat").fuzzy\_counts

(0, 0, 0)

>>> # 0 substitutions, 0 insertions, 0 deletions.

* Named lists (Hg issue 11)

\L<name>

There are occasions where you may want to include a list (actually, a set) of options in a regex.

One way is to build the pattern like this:

p = regex.compile(r"first|second|third|fourth|fifth")

but if the list is large, parsing the resulting regex can take considerable time, and care must also be taken that the strings are properly escaped if they contain any character that has a special meaning in a regex, and that if there is a shorter string that occurs initially in a longer string that the longer string is listed before the shorter one, for example, "cats" before "cat".

The new alternative is to use a named list:

option\_set = ["first", "second", "third", "fourth", "fifth"]

p = regex.compile(r"\L<options>", options=option\_set)

The order of the items is irrelevant, they are treated as a set. The named lists are available as the .named\_lists attribute of the pattern object

>>> print(p.named\_lists)

{'options': frozenset({'second', 'fifth', 'fourth', 'third', 'first'})}

* Start and end of word

\m matches at the start of a word.

\M matches at the end of a word.

Compare with \b, which matches at the start or end of a word.

* Unicode line separators

Normally the only line separator is \n (\x0A), but if the WORD flag is turned on then the line separators are the pair \x0D\x0A, and \x0A, \x0B, \x0Cand \x0D, plus \x85, \u2028 and \u2029 when working with Unicode.

This affects the regex dot ".", which, with the DOTALL flag turned off, matches any character except a line separator. It also affects the line anchors ^ and $ (in multiline mode).

* Set operators

**Version 1 behaviour only**

Set operators have been added, and a set [...] can include nested sets.

The operators, in order of increasing precedence, are:

|| for union ("x||y" means "x or y")

~~ (double tilde) for symmetric difference ("x~~y" means "x or y, but not both")

&& for intersection ("x&&y" means "x and y")

-- (double dash) for difference ("x--y" means "x but not y")

Implicit union, ie, simple juxtaposition like in [ab], has the highest precedence. Thus, [ab&&cd] is the same as [[a||b]&&[c||d]].

Examples:

[ab] # Set containing 'a' and 'b'

[a-z] # Set containing 'a' .. 'z'

[[a-z]--[qw]] # Set containing 'a' .. 'z', but not 'q' or 'w'

[a-z--qw] # Same as above

[\p{L}--QW] # Set containing all letters except 'Q' and 'W'

[\p{N}--[0-9]] # Set containing all numbers except '0' .. '9'

[\p{ASCII}&&\p{Letter}] # Set containing all characters which are ASCII and letter

* regex.escape (issue #2650)

regex.escape has an additional keyword parameter special\_only. When True, only 'special' regex characters, such as '?', are escaped.

Examples:

>>> regex.escape("foo!?")

'foo\\!\\?'

>>> regex.escape("foo!?", special\_only=True)

'foo!\\?'

* Repeated captures (issue #7132)

A match object has additional methods which return information on all the successful matches of a repeated capture group. These methods are:

matchobject.captures([group1, ...])

Returns a list of the strings matched in a group or groups. Compare with matchobject.group([group1, ...]).

matchobject.starts([group])

Returns a list of the start positions. Compare with matchobject.start([group]).

matchobject.ends([group])

Returns a list of the end positions. Compare with matchobject.end([group]).

matchobject.spans([group])

Returns a list of the spans. Compare with matchobject.span([group]).

Examples:

>>> m = regex.search(r"(\w{3})+", "123456789")

>>> m.group(1)

'789'

>>> m.captures(1)

['123', '456', '789']

>>> m.start(1)

6

>>> m.starts(1)

[0, 3, 6]

>>> m.end(1)

9

>>> m.ends(1)

[3, 6, 9]

>>> m.span(1)

(6, 9)

>>> m.spans(1)

[(0, 3), (3, 6), (6, 9)]

* Atomic grouping (issue #433030)

(?>...)

If the following pattern subsequently fails, then the subpattern as a whole will fail.

* Possessive quantifiers.

(?:...)?+ ; (?:...)\*+ ; (?:...)++ ; (?:...){min,max}+

The subpattern is matched up to 'max' times. If the following pattern subsequently fails, then all of the repeated subpatterns will fail as a whole. For example, (?:...)++ is equivalent to (?>(?:...)+).

* Scoped flags (issue #433028)

(?flags-flags:...)

The flags will apply only to the subpattern. Flags can be turned on or off.

* Inline flags (issue #433024, issue #433027)

(?flags-flags)

Version 0 behaviour: the flags apply to the entire pattern, and they can't be turned off.

Version 1 behaviour: the flags apply to the end of the group or pattern, and they can be turned on or off.

* Repeated repeats (issue #2537)

A regex like ((x|y+)\*)\* will be accepted and will work correctly, but should complete more quickly.

* Definition of 'word' character (issue #1693050)

The definition of a 'word' character has been expanded for Unicode. It now conforms to the Unicode specification athttp://www.unicode.org/reports/tr29/. This applies to \w, \W, \b and \B.

* Groups in lookahead and lookbehind (issue #814253)

Groups and group references are permitted in both lookahead and lookbehind.

* Variable-length lookbehind

A lookbehind can match a variable-length string.

* Correct handling of charset with ignore case flag (issue #3511)

Ranges within charsets are handled correctly when the ignore-case flag is turned on.

* Unmatched group in replacement (issue #1519638)

An unmatched group is treated as an empty string in a replacement template.

* 'Pathological' patterns (issue #1566086, issue #1662581, issue #1448325, issue #1721518, issue #1297193)

'Pathological' patterns should complete more quickly.

* Flags argument for regex.split, regex.sub and regex.subn (issue #3482)

regex.split, regex.sub and regex.subn support a 'flags' argument.

* Pos and endpos arguments for regex.sub and regex.subn

regex.sub and regex.subn support 'pos' and 'endpos' arguments.

* 'Overlapped' argument for regex.findall and regex.finditer

regex.findall and regex.finditer support an 'overlapped' flag which permits overlapped matches.

* Unicode escapes (issue #3665)

The Unicode escapes \uxxxx and \Uxxxxxxxx are supported.

* Large patterns (issue #1160)

Patterns can be much larger.

* Zero-width match with regex.finditer (issue #1647489)

regex.finditer behaves correctly when it splits at a zero-width match.

* Zero-width split with regex.split (issue #3262)

Version 0 behaviour: a string won't be split at a zero-width match.

Version 1 behaviour: a string will be split at a zero-width match.

* Splititer

regex.splititer has been added. It's a generator equivalent of regex.split.

* Subscripting for groups

A match object accepts access to the captured groups via subscripting and slicing:

>>> m = regex.search(r"(?P<before>.\*?)(?P<num>\d+)(?P<after>.\*)", "pqr123stu")

>>> print m["before"]

pqr

>>> print m["num"]

123

>>> print m["after"]

stu

>>> print len(m)

4

>>> print m[:]

('pqr123stu', 'pqr', '123', 'stu')

* Named groups

Groups can be named with (?<name>...) as well as the current (?P<name>...).

* Group references

Groups can be referenced within a pattern with \g<name>. This also allows there to be more than 99 groups.

* Named characters

\N{name}

Named characters are supported. (Note: only those known by Python's Unicode database are supported.)

* Unicode codepoint properties, including scripts and blocks

\p{property=value}; \P{property=value}; \p{value} ; \P{value}

Many Unicode properties are supported, including blocks and scripts. \p{property=value} or \p{property:value} matches a character whose property property has value value. The inverse of \p{property=value} is \P{property=value} or \p{^property=value}.

If the short form \p{value} is used, the properties are checked in the order: General\_Category, Script, Block, binary property:

* + Latin, the 'Latin' script (Script=Latin).
  + Cyrillic, the 'Cyrillic' script (Script=Cyrillic).
  + BasicLatin, the 'BasicLatin' block (Block=BasicLatin).
  + Alphabetic, the 'Alphabetic' binary property (Alphabetic=Yes).

A short form starting with Is indicates a script or binary property:

* + IsLatin, the 'Latin' script (Script=Latin).
  + IsCyrillic, the 'Cyrillic' script (Script=Cyrillic).
  + IsAlphabetic, the 'Alphabetic' binary property (Alphabetic=Yes).

A short form starting with In indicates a block property:

* + InBasicLatin, the 'BasicLatin' block (Block=BasicLatin).
  + InCyrillic, the 'Cyrillic' block (Block=Cyrillic).
* POSIX character classes

[[:alpha:]]; [[:^alpha:]]

POSIX character classes are supported. This is actually treated as an alternative form of \p{...}.

* Search anchor

\G

A search anchor has been added. It matches at the position where each search started/continued and can be used for contiguous matches or in negative variable-length lookbehinds to limit how far back the lookbehind goes:

>>> regex.findall(r"\w{2}", "abcd ef")

['ab', 'cd', 'ef']

>>> regex.findall(r"\G\w{2}", "abcd ef")

['ab', 'cd']

* + The search starts at position 0 and matches 2 letters 'ab'.
  + The search continues at position 2 and matches 2 letters 'cd'.
  + The search continues at position 4 and fails to match any letters.
  + The anchor stops the search start position from being advanced, so there are no more results.
* Reverse searching

Searches can now work backwards:

>>> regex.findall(r".", "abc")

['a', 'b', 'c']

>>> regex.findall(r"(?r).", "abc")

['c', 'b', 'a']

Note: the result of a reverse search is not necessarily the reverse of a forward search:

>>> regex.findall(r"..", "abcde")

['ab', 'cd']

>>> regex.findall(r"(?r)..", "abcde")

['de', 'bc']

* Matching a single grapheme

\X

The grapheme matcher is supported. It now conforms to the Unicode specification at http://www.unicode.org/reports/tr29/.

* Branch reset

(?|...|...)

Capture group numbers will be reused across the alternatives, but groups with different names will have different group numbers.

Examples:: >>> import regex >>> regex.match(r"(?|(first)|(second))", "first").groups() ('first',) >>> regex.match(r"(?|(first)|(second))", "second").groups() ('second',)

Note that there is only one group.

* Default Unicode word boundary

The WORD flag changes the definition of a 'word boundary' to that of a default Unicode word boundary. This applies to \b and \B.

* SRE engine do not release the GIL (issue #1366311)

The regex module can release the GIL during matching (see the above section on multithreading).

Iterators can be safely shared across threads.