# **Special characters**

\ escapes special characters

. matches any character (except \n)

^ matches start of the string

$ matches end of the string

R|S matches either regex R or regex S

() creates a capture group, and indicates precedence

\Q..\E escapes a string of chars, matching them as literals

\Q\*\d+\* same as \Q\*\d+\*\E

Quantifier after \E, applied only to last character

11 meta-characters with special meanings: [, \, ^, $, ., |, ?, \*, +, (, ).

**Character classes/sets**

[] character classes/sets

[^x] NOT x

[5b-d] matches any chars '5', 'b', 'c' or 'd'

[^a-c6] matches any char except 'a', 'b', 'c' or '6'

*Within* []*, special chars don’t do anything special, hence they don't need escaping, except for* ']' *and* '-'*, which only need escaping if they are not the first char.*

*e.g.* '[]]' *matches* ']'*.* '^' *also has special meaning, it negates the group if it's the first character in the* [], *and needs to be escaped to match it literally.*

**Quantifiers**

\* 0 or more (append ‘?’ for non-greedy/reluctant/lazy)

+ 1 or more (append ‘?’ for non-greedy/reluctant/lazy)

? makes the preceding token optional

{} (limited) repetition operator - {min,max}

{m} exactly 'm'

{m,n} from m to n. 'm' defaults to 0, 'n' to infinity

{m,n}? from m to n, as few as possible

{0,}  same as \*

{1,}  same as +

**Special sequences**

\A Start of string (file)

\b Matches empty string at word boundary (between \w and \W)

\B Matches empty string not at word boundary

\d Digit

\D Non-digit

\s Whitespace: [ \t\n\r\f\v], more if LOCALE or UNICODE

\S Non-whitespace

\w Alphanumeric: [0-9a-zA-Z\_], or is LOCALE dependent

\W Non-alphanumeric

\Z End of string (file)

# **Character escape sequences**

\a ASCII Bell (BEL)

\f ASCII Formfeed

\n ASCII Linefeed

\r ASCII Carriage return

\t ASCII Tab

\v ASCII Vertical tab

\\ A single backslash

\xHH Two digit hex character

\OOO Three digit octal char

(or use a preceding zero, e.g. \0, \09)

\DD Decimal number 1 to 99, matches previous numbered group

*Special character escapes are much like those already escaped in Python string literals. Hence regex* '\n' *is same as regex* '\\n'

**Module level functions**

.compile(pattern[, flags]) -> RegexObject

.match(pattern, string[, flags]) -> MatchObject

.search(pattner, string[, flags]) -> MatchObject .findall(pattern, string[, flags]) -> list of strings

.finditer(pattern, string[, flags]) -> iter of MatchObjects .split(pattern, string[, maxsplit, flags]) -> list of strings .sub(pattern, repl, string[, count, flags]) -> string .subn(pattern, repl, string[, count, flags]) -> (string, int) .escape(string) -> string

.purge() # the re cache

# **Functions for RegEx objects (returned from compile())**

.match(string[, pos, endpos]) -> MatchObject

.search(string[, pos, endpos]) -> MatchObject

.findall(string[, pos, endpos]) -> list of strings

.finditer(string[, pos, endpos]) -> iter of MatchObjects

.split(string[, maxsplit]) -> list of strings

.sub(repl, string[, count]) -> string

.subn(repl, string[, count]) -> (string, int)

.flags # int passed to compile()

.groups # int number of capturing groups

.groupindex # {} maps group names to ints

.pattern # string passed to compile()

**MatchObjects (returned from match() and search())**

.expand(template) -> string, backslash and group expansion

.group([group1...]) -> string or tuple of strings, 1 per arg

.groups([default]) -> (,) of all groups, non-matching=default

.groupdict([default]) -> {} of named groups, non-matching=default

.start([group]) -> int, start/end of substring matched by group

.end([group]) (group defaults to 0, the whole match)

.span([group]) -> tuple (match.start(group), match.end(group))

.pos # value passed to search() or match()

.endpos # "

.lastindex # int index of last matched capturing group

.lastgroup # string name of last matched capturing group

.re # regex passed to search() or match()

.string # string passed to search() or match()

# **Flags for re.compile(), etc. (combine with '|')**

re.I == re.IGNORECASE Ignore case

re.L == re.LOCALE Make \w, \b, and \s locale dependent

re.M == re.MULTILINE Multiline

re.S == re.DOTALL Dot matches all (including newline)

re.U == re.UNICODE Make \w, \b, \d, and \s unicode dependent re.X == re.VERBOSE Verbose (unescaped whitespace in pattern

is ignored, and '#' marks comment lines)

# **Miscellaneous**

# (.+?) A lazy plus (+?) follows the dot. hence, repeat the dot as few times as possible (minimum is one)

# +,\*,{} Plus, star and repetition using curly braces are greedy.

# (xyz)\* Apply a regex operator, to the entire group.

(?:...) Non-capturing version of regular parentheses called non- capturing parentheses (i.e., indicates no back-reference)

# For e.g., x(?:y) will not create a back-reference; you can insert them into a regular expression without changing the numbers assigned to the back-references

# Parentheses and backreferences cannot be used inside character classes

**Extensions**

(?P<name>...) Creates a named capturing group

(?P=<name>) Matches whatever matched previously named group (?#...) A comment; ignored.

*These do not cause grouping, except for* (?P<name>...)

# **Examples**

# gr[ae]y match gray or grey

# colou?r matches both colour and color

# \b[1-9][0-9]{3}\b match a number between 1000 and 9999

# \b[1-9][0-9]{2,4}\b match a number between 100 and 99999

# \b[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b -> e-mail address

# <[A-Za-z][A-Za-z0-9]\*> match an HTML tag

# <([A-Z][A-Z0-9]\*)\b[^>]\*>.\*?</\1> -> closing tag with a backref

# \b(\w+)\s+\1\b checking for double words (the the)

# ^[ \t]+ delete leading whitespace

# [ \t]+$ trim trailing whitespace