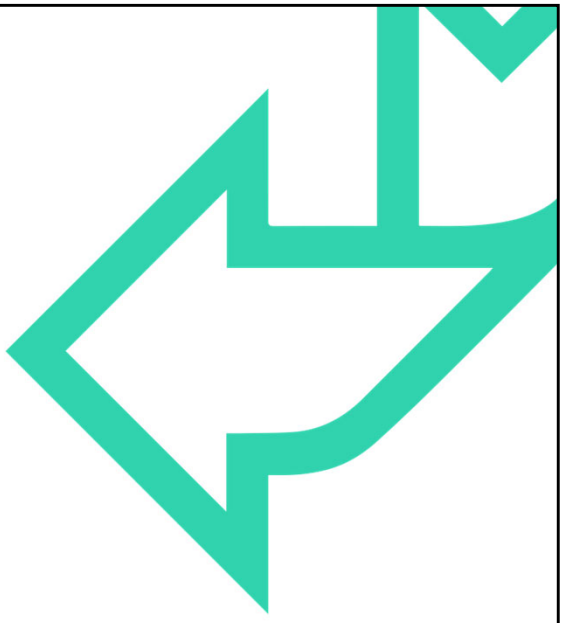
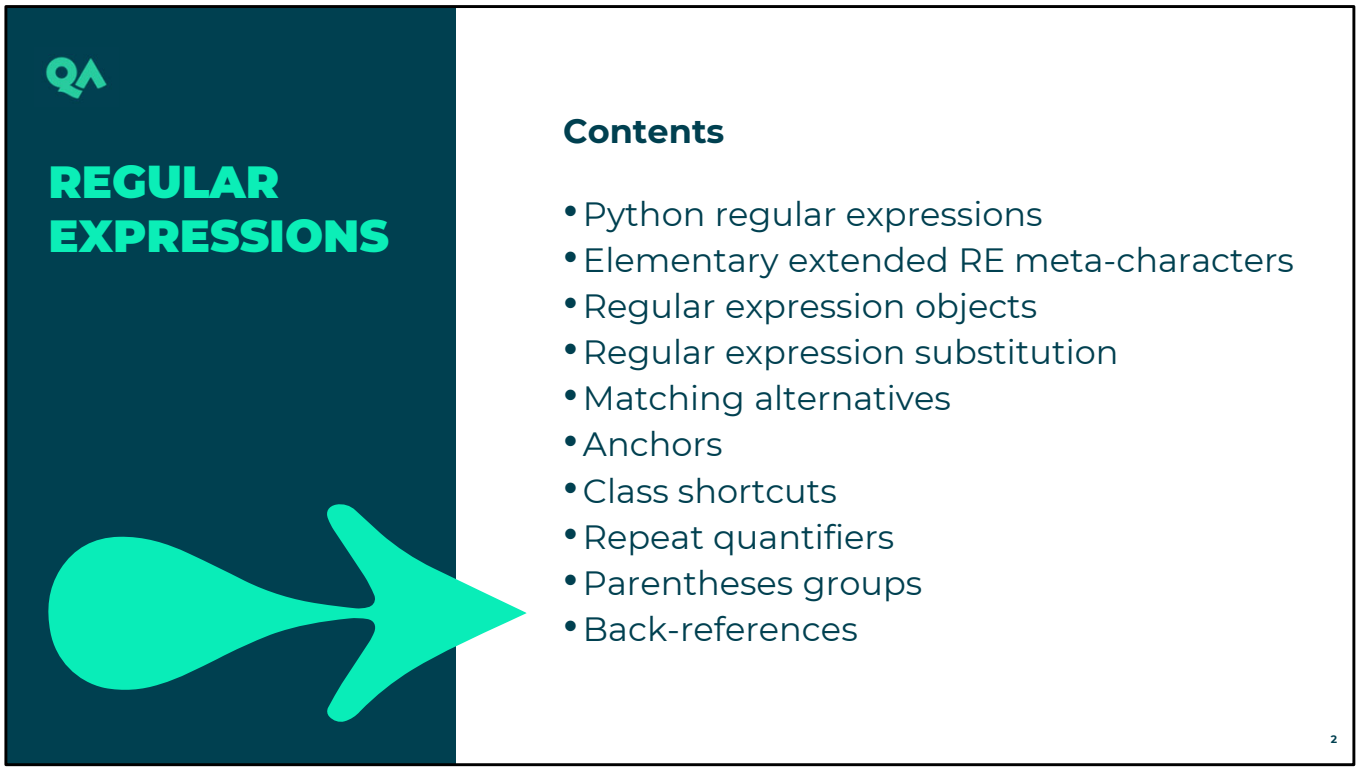


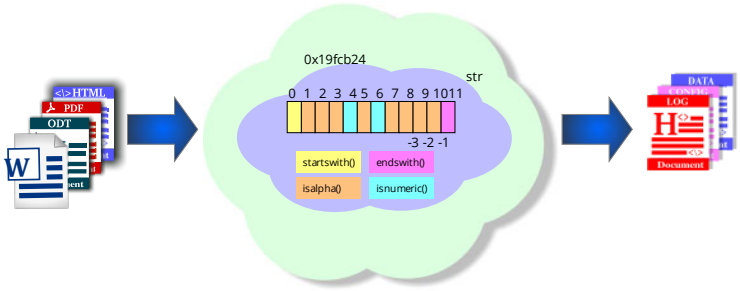
Python 3 Programming

Regular expressions


REGULAR
EXPRESSIONS
Contents
• Python regular expressions
• Elementary extended RE meta-characters
• Regular expression objects
• Regular expression substitution
• Matching alternatives
• Anchors
• Class shortcuts
• Repeat quantifiers
• Parentheses groups
• Back-references
2

Regular expressions

What are regular expressions (REs) ?



Purpose

Search for text or a pattern within a string.
Extract or change interesting parts of a string.

What are they?

Consist of a meta language of special characters.
Two dialects defined in the POSIX standard:

- Basic RE- used by Linux tools.
- Extended RE - used by programming languages.

Python supports ERE – with extensions.

Processing text in this way is called 'Data Munging'.

3

Regular Expressions are commonly used in UNIX command-line tools, and are also present in .Net. Most programming languages support them in one form or another, either built-in or through an external library.
They have a reputation for being difficult to understand, but they are well worth the effort to learn.

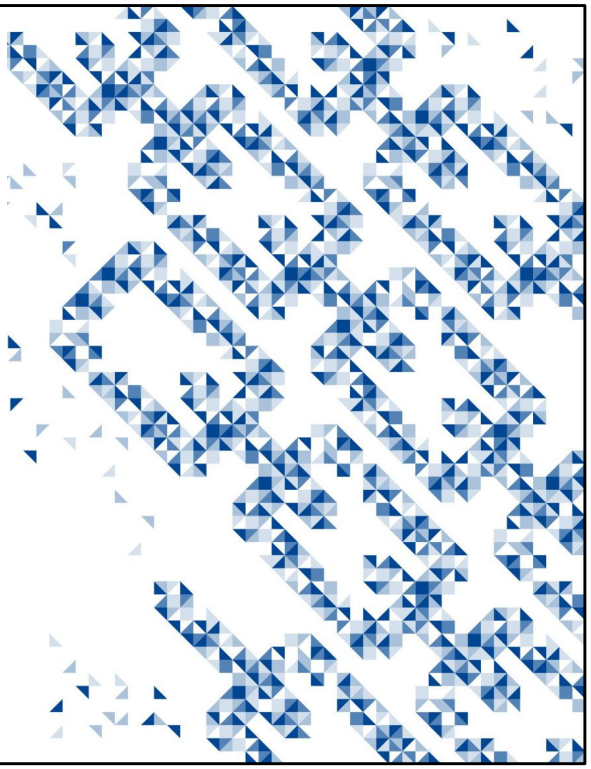
Python regular expressions

Extended regular expressions, with extensions

- Requires the **RE** module (in the standard library).
- Can pre-compile an expression for efficiency.
- Multi-line pattern matches are supported.
- Apply an RE to any number of lines at a time.
- Powerful substitution.
- Replace a pattern using a variable-expression.
- Create self-referencing patterns.
- Match part of a pattern with result of previous sub-pattern.

But, Python string methods are powerful and fast.

- Don't use REs when functions or methods will do.



Python regular expressions are very powerful, yet are easy to learn if you are familiar with tools like grep, vi, sed and awk. Python uses Extended Regular Expressions (ERE), whereas many UNIX tools, like grep, use Basic Regular Expression (BRE) syntax. The most noticeable difference is that BREs need to 'escape' (backslash) parentheses and braces, but thankfully we do not need that in Python.

Here is an overview of the most important differences between grep/sed/awk and Python regular expressions:

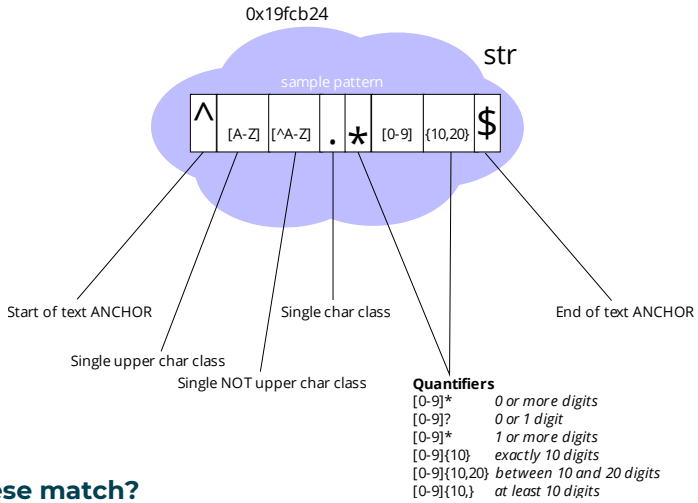
Python REs can be applied across multiple lines, so you can match a group of lines that match specific large patterns, without having to keep track of state yourself.

Python substitution expressions allow you to use variables in the search text, including dictionary and list lookups using parts of the pattern just being matched. You can even replace a pattern by the result of a function or subroutine, being called with selected parts of the pattern being replaced.

Python patterns can be self-referencing: you can build a pattern of multiple parts, and then say ‘between these two

expressions, I want to match whatever I am currently matching at sub-pattern x'.

Elementary ERE meta-characters



What would these match?

"^[A-Z]{1,2}[0-9]{1,2}[A-Z]? [0-9][A-Z]{2}$"

"^[A-CEGHJ-PR-TW-Z][A-CEGHJ-NPR-TW-Z] ?[0-9]{2} ?[0-9]{2} ?[0-9]{2} ?[A-DFMP]$"

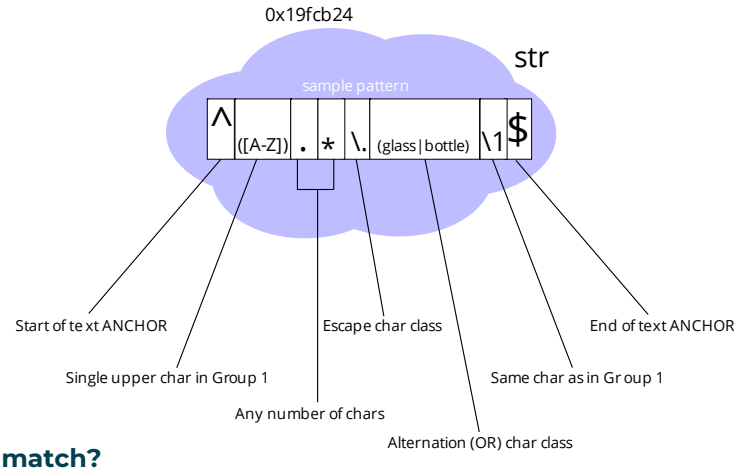
5

The examples listed above cover the most common regular expression meta-characters. If you are new to regular expressions, this is a good table to remember.

The expression Character Class is introduced here. A Character Class may be specified in the 'traditional' way as show, using the square brackets [ ].

Note that meta-characters used inside [...] are different to those used outside. Escaped meta-characters (those prefixed with /) are literals, as are meta-characters inside [].

Elementary ERE meta-characters



What would these match?

"^(.)(.)(.)\3\2\1$"

"^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$"

6

The examples listed above cover the most common regular expression meta-characters. If you are new to regular expressions, this is a good table to remember.
The expression Character Class is introduced here. A Character Class may be specified in the 'traditional' way as show, using the square brackets [ ].
Note that meta-characters used inside [...] are different to those used outside. Escaped meta-characters (those prefixed with /) are literals, as are meta-characters inside [].

Regular expression objects

Must import RE module

- Can pre-compile the RE for efficiency and returns an RE object.

We can search or match:

- `search()` searches for a pattern anywhere in a string.
- `match()` matches from the start of the string.
- `fullmatch` matches from the start to the end of the string (3.4).
- On failure, raises an `re.error` exception.

Returns a MatchObject or None (False)

```
testy = 'The quick brown fox jumps over the lazy dog'

m = re.search(r"(quick|slow).*(fox|camel)", testy)
if m:
    print('Matched', m.groups())
    print('Starting at', m.start())
    print('Ending at', m.end())
```

```
Matched ('quick', 'fox')
Starting at 4
Ending at 19
```

7

Importing the `re` module allows methods on the re class to be called, with search and match being the most common. The `fullmatch` method was introduced at Python 3.4. They all return an object of class `MatchObject`.

The group in parentheses is also known as a *capturing parentheses group*. Text inside parentheses may be referred to later in the RE as a back-reference. A useful method is `groups()`, which returns a tuple containing the matched text, and may be used like back-references. Other methods include `start()` and `end()`, which return the positions of the match, and `group()` which returns the matched string. There are several `MatchObject` attributes, including `re`, which gives the original regular expression, and `string` which gives the original input.

If the search (or match) failed to find the pattern then the empty object None is returned, which is false in Boolean context and a `re.error` exception (enhanced in Python 3.5) is raised.

Python RE syntax is like that used by lower level language libraries, such as the GNU C regex package.

We can compile our REs for efficiency, for example:

```
reobj = re.compile (r"([Ii]).*(\1)")
```

```
    for line in file:
        m = reobj.match(line)
        if m:
            print(m.string[m.start():m.end()])
```

Notice the use of raw strings (r"..."), these mean we do not have to escape (\) special characters like brackets and braces – particularly useful with Regular Expressions.

Regular expression substitution

The sub function returns the modified string:

- `re.sub(`*pattern*`, `*replacement*`, `*string*`[, `*count*`, `*flags*`])`

The subn functions returns a tuple of (modified *string*, *number of changes*):

- `re.subn(`*pattern*`, `*replacement*`, `*string*`[, `*count*`, `*flags*`])`

The optional count argument determines the occurrence to modify:

- Default is to replace *all* occurrences.

```
line = 'Perl for Perl Programmers'
cs, num = re.subn('Perl', 'Python', line)
if num:
    print(cs)

cs, num = re.subn('Perl', 'Python', line, 1)
if num:
    print(cs)
```

```
Python for Python Programmers
```

```
Python for Perl Programmers
```


Substitution is reminiscent of awk, in that we have a couple of method calls. sub is used where we just want the new string, whereas subn returns both the altered string and the number of matches. Both perform global substitutions from the left of the string.
There are other useful re functions, for example split, which is shown on the next slide.
We have also shown a compiled Regular Expression object. Compiling the RE makes for more efficient code when the same pattern is used many times, for example in a loop. Methods like match, fullmatch, findall, search, split, sub, and subn may be called on these objects.

Regular expression split

Similar in functionality to the string split

- Uses a regular expression for the separator instead of a string.
- `re.split(`*pattern*`, `*string*`[,` *max_splits=0*`,` *flags=0*`])`
- Note the default value for *max_splits* is zero, which means no limit.

Only use the RE version if you need alternative separators

- The string version is more efficient

```
import re
line = 'root::0.0:superuser,/root;/bin/sh'
elems = re.split('[:;.,]', line)
print(elems)
```

```
['root', '', '0', '0', 'superuser', '/root', '/bin/sh']
```


The **`re`** module's version of **`split`** enables a regular expression to be used to describe the field delimiter (this is much like `split` in Perl and PHP).
The optional parameter *flags* was added at 3.1 (see later).

Matching alternatives

The | character separates alternative words or patterns.

```
drink = 'A glass of Coors'
if re.search(r'Bud|Miller|Coors', drink):
    print("It's a beer!")
```

Use parentheses to group alternatives.

- Required with text before or following alternatives.

```
pattern = r'A (glass|bottle|barrel) of (Bud|Miller|Coors)'
if re.search(pattern, drink):
    print("This drink is suitable for Americans")
```


The second example above can match nine different possibilities. A second effect of the parenthesis-notation is that they capture the matched text in the groups list, which we see later.

Anchors

The ^ and $ characters indicate start or end of text.

- Only when used at start or end of pattern.

```
name, old, new = sys.argv[1:]

new_name = re.sub(fr"\.{old}$", f".{new}", name)
print(f"Renaming {name} to {new_name}")
os.rename(name, new_name)
```

Shortcuts \b matches a word-boundary and \B not a word-boundary.

```
txt = 'Stranger in a strange land'
m = re.search(r'range\b', txt)
print(m.start())

txt = 'Stranger in a strange land'
m = re.search(r'range\B', txt)
print(m.start())
```

```
16

2
```

The $ anchor is special: if you use /00$/, it will match either two zeroes at the end of the search text, or two zeroes followed by a newline at the end of a search text. (it automatically ignores a new-line at the end of a line).
When the ^ character is used anywhere except at the start of a pattern, it indicates a normal ^ character.
When you have a search text that contains multiple lines, the ^ and $ anchors apply to the whole of the text. If you use the m flag (see later), they will be applied to each individual line within the search text. For single-line matches, this is of no importance, but for multi-line matches genuine start of text can be marked with \A, and end of text with \Z.
In addition to the start and end of line anchors, we have the word anchor, \b. It indicates a word boundary (either the beginning or the end of a word), but, like ^ and $, does not take up any space. Exactly what constitutes a word boundary is, however, not always intuitive when it comes to apostrophes. The non-word boundary anchor, \B is used when we explicitly want the text imbedded in another word.
Beware! When used in square brackets (a character class) \b

means a single back-space character!

Class shortcuts

A Character Class describes a set of characters:

- For example: `[a-z] [^A-Z] [aeiou]`

A Class shortcut matches a pre-defined character class:

- Shorthand `\w, \d, \s, \W, \D, \S`

\w	[a-zA-Z0-9_]
\d	[0-9]
\s	[\t\n\r\f]

\W	[^a-zA-Z0-9_]
\D	[^0-9]
\S	[^ \t\n\r\f]

```
m = re.search(r'^ttyp\d$', port)
```

ttyp0…ttyp9

Exact meaning can be changed with flags…

12

The character classes work as in sed and awk, however the typed character classes may also be used, so [\da-fA-F] matches any hexadecimal digit. Note that the character classes may vary with the locale, depending on which flags are set.
There are other \ functions, for example:

\G Continue where previous match left off

Flags

Change the behaviour of the match

Long name	Short	RE	
re.IGNORECASE	re.I	(?i)	Case insensitive match
re.MULTILINE	re.M	(?m)	^ and $ match start and end of *line*
re.DOTALL	re.S	(?s)	. also matches a new-line
re.VERBOSE	re.X	(?x)	Whitespace is ignored, allow comments

- Can be embedded in the RE.
- Can be applied to parts of the string (3.6 – *modifier spans*).
- Can be specified as an optional argument to search, match, split, sub, etc.
- Can be combined with '|' separator.

```
m = re.search(r'(?im)^john', name)
m = re.search(r'^john', name, re.IGNORECASE|re.MULTILINE)
m = re.search(r'^(?i:j)ohn', name)
```

13

It would be tempting to state that the short names are the initial letter of the long name, and that the RE syntax is just the short name in lowercase. You can see that this is not the case, the S and X flags are there for compatibility with other RE engines.
The optional *flags* parameter was added to the `re` methods at Python 3.1.
The first two examples combine the IGNORECASE and MULTILINE flags. They look for 'john' in any case at the start of the text *or* immediately after a new-line character. The third example is a *modifier span* and only applies the `i` (ignore case) to the letter j – so John or john but not JOHN.
When embedded in the RE the single characters can be in any order. When using the re module attribute flags, they are combined with a binary OR |, also in any order. The flags and embedded attributes may be mixed, but that might make the RE even more confusing.
There are two additional flags not shown on the slide. From Python 3.5, they are deprecated, and from 3.6 are only supported for byte

Long name	Short	RE	
re.ASCII	re.A	(?a)	Class shortcuts do not include Unicode
re.LOCALE	re.L	(?L)	Class shortcuts are locale sensitive

Repeat quantifiers

Quantifier characters repeat the preceding pattern

?	optional	0 or 1 times
*	optional	repeat 0 or more times
+	at least once	repeat 1 or more times

```
m = re.search(r'[:;,]?\s*\w+', line)
```

optional :;, followed by zero or more whitespace, followed by at least one alphanumeric

```
m = re.search(r'boink+', sound)
```

boink, boinkk, boinkkk

```
m = re.search(r'(boink)+', sound)
```

boink, boinkboink, boinkboinkboink

The repeat-quantifiers described here are greedy: they eat as much as possible, while not breaking the rest of the pattern.
If we have the text 'The dog eats dog-food', then the pattern

'The.*dog'

will match the text 'The dog eats dog'.
Python also supports minimal repeat-quantifiers, that eat the least amount possible while still making the match work. This is achieved by appending a question mark ? after the repeat quantifier.

Quantifiers

Repeat an indicated number of times:

- {3} repeat exactly three times.
- {3,5} repeat three to five times.
- {5,} repeat at least five times.
- {0,9} repeat up to nine times.

Match a U.S. telephone number:

Start of text or a non-digit character.
Followed by three digits followed by a hyphen.
Followed by between 2 and 4 digits.
Followed by an optional whitespace.
Followed by between 4 and eight digits.
Followed by a non-digit character or end-of text.

```
m = re.search(r'(^|\D)\d{3}-\d{2,4}\s?\d{4,8}(\D|$)',
              phone)
```

15

In their simplest form, quantifiers specify the number of characters, for example, to match a line of at least 80 characters:

.{80,}

Back-references

Python also allows substitution strings to reference groups within the pattern.

- To create self-referencing regular expressions.

Referenced by \n *or* \g<n>, representing the '*n*th' group.

- Don't forget to use a raw string.

Can also be used in the replacement string in sub and subn functions.

```
import re
from datetime import date
year = str(date.today())[:4]

strn = 'copyright 2005-2006'
print(re.sub(r'((19|20)[0-9]{2})-((19|20)[0-9]{2})', r'\1-' + year, strn)
```

Get current year

```
copyright 2005-2016
```


Python supports the use of parentheses to group a number of characters or regular expressions into a single unit and then apply a regular expression quantifier to the entire group. This can be useful when the pattern consists of recurring blocks of text or words.
The group in parentheses is also known as a capturing parentheses group. Text inside parentheses may be referred to later in the RE as a back-reference. This is done by the use of r'\1', r'\2' etc. to refer to the contents of the first and second sets of parentheses. If you nest parentheses, the order of the opening parenthesis is the order in which the back-references are allocated. Make sure you use 'raw' strings for the back-references, since '\1' is itself a special character.
The alternative back-reference syntax using r'\g<1>', r'\g<2>' etc. is a Python extension, and is associated with named captures (see the Advanced Regular Expressions appendix). It has the additional feature of supporting group 0 (zero) in r'\g<0>', which represents the whole of the matched string (r'\0' is not supported).
The use of many back-references in a RE will cause Python to work harder and increase the time taken to find a match. Use sparingly!

Global matches

re.findall

- Returns a list of matches or groups.

```
strn = '/dev/sd3d 135398 69683 52176 57% /home/stuff'
nums = re.findall(r'\b\d+\b', str)
print(nums)
```

```
['135398', '69683', '52176', '57']
```

re.finditer

- Returns an iterator to a match object.

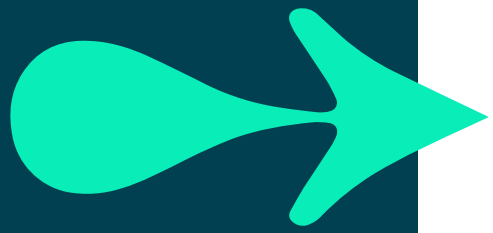
```
strn = '/dev/sd3d 135398 69683 52176 57% /home/stuff'

for m in re.finditer(r'\b(\d+)\b', str):
    print(m.groups())
```

```
('135398',)
('69683',)
('52176',)
('57',)
```


By default, most re methods search for the first (leftmost) occurrence of a pattern. These methods work on all occurrences and return either a list or an iterator to the occurrences. These are particularly useful for repeated patterns over multiple lines.
The `findall` method was added in Python 2.2.

SUMMARY



Regular expressions are used by many programs.

Regular expressions create a MatchObject on match.

Several functions available:

`re.search` - Find a pattern somewhere in the string.
`re.match` - Match from the start of the string.
`re.fullmatch` - Match from start to end of string (3.4).
`re.sub` - Substitute the pattern, returning the new string.
`re.subn` - Substitute the pattern, returning the new string and a count of substitutions.

Support many char classes including:

Class shortcuts	**. \w \d \s \W \D \S**
Alternatives	
characters	[ ]
strings	|
Repeat qualifiers	? * + {*m*,*n*}