Python Course

Session 6b – Regular Expressions

Table of Contents

Regular Expressions

A **regular expression** (**regex**, or **RE**) is a mini-language embedded in Python as the re module. The language syntax is unrelated to Python's syntax, being largely common across a variety of programming languages. It provides an efficient way to define a pattern of characters.

The use of a regular expresssion is in finding occurrences of some designated **pattern** within **strings**. The pattern is described using the "regex" language, coded as a string of characters. A built-in **match engine** performs the matching process in a well-defined way, and methods are provided that allow for returning the matches and modification of the matched portions of the string.

As a programmer, you construct your pattern and direct Python to use the pattern to find the locations of the matches in a given string (e.g. one entered by the user or obtained from some external data source).

A simple pattern is e.g. "ab", which matches once each within the strings: "ab", "drab", "able" and "fabulous". And twice in "abstractable".

Python provides useful opportunities for what to do with the matches found, such as: split the string at the matches; iterate over the matches; replace the matched parts with different strings.

A regex has a sufficiently complex language that it is in fact compiled on the fly and executed via a C bytecode match engine, for efficient repetitive usage. You can obtain that compiled object and re-use it explicitly (to avoid re-compilation), but Python automatically caches up to 512 compiled regex's without you needing to worry about it. Regex's may be employed to parse whole languages, so performance can be very important.

Note that the full power of regular expressions may not be required for simple string matches, where it may be more straightforward to use appropriate string methods.

Some Unix/Linux utilities (such as grep) provide regex capabilities via the -E command line flag.

Regex Pattern Syntax

The characters of a regex pattern are compared one-to-one with the string's characters, except when encountering any of the 14 special characters:

```
. ^ $ * + ? { } [ ] \ | ( )
```

These have special meanings:

`.`	Match any one char.	
`^` `[^chars]`	Match start of string. Match a char not in the set of *`chars`*. (*`chars`* can include hyphenated char ranges e.g. `a-z`)	
`$`	Match end of string.	
`RE``*` *`RE`*`*?`	Match 0 or more *`RE`*'s (greedy) Match 0 or more *`RE`*'s (non-greedy)	
`RE``+` *`RE`*`+?`	Match 1 or more *`RE`*'s (greedy) Match 1 or more *`RE`*'s (non-greedy)	
`RE``?` `??`	Match 0 or 1 *`RE`*'s (greedy) Match 0 or 1 *`RE`*'s (non-greedy)	
`RE``{`*`m`*`}` *`RE`*`{`*`m`*`,`*`n`*`}` *`RE`*`{`*`m`*`,`*`n`*`}?`	Match *`m`* number of *`RE`*'s Match *`m`*..*`n`* *`RE`*'s (greedy) Match *`m`*..*`n`* *`RE`*'s (non-greedy)	
`[`*`chars`*`]`	Match one of a set of *`chars`*	`[a-zA-Z_]` Matches alphabetics and _ `[-a-z$?]` Matches one of: `- a-z $ ?`
`\`*`e`* `\`*`s`*	Escape special char *`e`* and match it Match special sequence *`s`*	`\$\\` Matches: `$\` See Special Sequences
`RE``|`*`RE`*	Match one of the *`RE`*'s (greedy)	`a|de|xyz` Matches one of: `a de xyz`
`(`*`RE`*`)`	Match the *RE* and add the matched char sequence to the list of groups	`A([a-z])B` Adds e.g. `AyB` to the groups
`(?`*`chars`*`)`	Python extensions	

Special Sequences

This table lists special sequences that can be used as part of a pattern:

`\n`	Match the *n*'th group (*n* starts at 1)	`A([a-z])B\1` Matches e.g.: `AwBw`
`\A`	Matches start of string	
`\b`	Matches start or end of a word	
`\B`	The negative of `\b`	
`\d`	Matches any one decimal digit 0-9	`\d+` Matches e.g.: `596`
`\D`	The negative of `\d`	
`\s`	Matches any one whitespace char	
`\S`	The negative of `\s`	
`\w`	Matches any one word char: `a-zA-Z0-9_`	`\w+` Matches e.g.: `Fred_9`
`\W`	The negative of `\w`	`\W` Matches e.g.: `@`
`\Z`	Matches end of string	

Note that since our string is by default a Unicode string, the actual sets of decimal digits, whitespace chars and word chars is more extensive than those listed. That generally makes using these special sequences more appropriate than other regex sequences, particularly when dealing with foreign languages.

Notes

Of note:

- All patterns in the examples are **raw** strings. It is generally recommended that raw strings e.g. `r"abc"` are used for describing patterns, particularly if you need to use `/` in a pattern. Otherwise Python interprets `/` and then the regex engine interprets `/` too, forcing you to use a `///` escape sequence, which is messy.
- Some special characters have different meanings depending on their syntactic location within the match pattern.
- The match engine has some **flags** that can be applied that affect the interpretation of the special characters. E.g. `re.DEBUG` may be useful for debugging complicated regex's.
- The characters of the pattern (taking into account the special meanings above) and the string are compared sequentially from first to last. The whole pattern has to match or else there is no match.
- The match engine backtracks if necessary. E.g. if the first `|` alternative matched but a subsequent part of the pattern didn't, it backtracks to try the other `|` alternatives.
- A **greedy** construct matches as many characters as possible. A **non-greedy** construct matches as few characters as possible.
- Special characters with the exception of `\` lose their special meaning inside sets of chars.
- The regular expression object returned by `re.compile` has its own set of methods, which are in essence a combination of the re module functions and match objects.

The re Module Functions

compile

```
regex_object = re.compile(pattern, flags=0)
```

Compile and return regex *`pattern`* into a regular expression object.

search

```
match_object = re.search(pattern, string, flags=0)
```

Search *`string`* for a match with *`pattern`* and return a corresponding match object.

match

```
match_object = re.match(pattern, string, flags=0)
```

Match *`pattern`* from start of *`string`* and return a corresponding match object or `None`.

fullmatch

```
match_object = re.fullmatch(pattern, string, flags=0)
```

Match *`pattern`* against the whole *`string`* and return a corresponding match object or `None`.

split

```
split_list = re.split(pattern, string, maxsplit=0, flags=0)
```

Split *`string`* by up to *`maxsplit`* occurrences of *`pattern`* *and return as a list.*

findall

```
match_list = re.findall(pattern, string, flags=0)
```

Return all matches of *`pattern`* in *`string`*, as a list of strings (or tuples for groups).

finditer

```
match_iterator = re.finditer(pattern, string, flags=0)
```

Return an iterator yielding match objects over all matches of *`pattern`* in *`string`*.

sub

```
sub_string = re.sub(pattern, repl, string, count=0, flags=0)
```

Replace up to `count` occurrences of *`pattern`* in *`string`* by the replacement *`repl`*, and return that new string.

subn

```
sub_tuple = re.subn(pattern, repl, string, count=0, flags=0)
```

As for `sub`, but return a tuple `(new_string, number_of_subs_made)`.

Notes

Of note:

- The regular expression object returned by `re.compile` has its own set of methods, which are in essence a combination of the re module functions and match objects.
- If the `findall` pattern has more than one group, the list returned is a list of tuples.
- The Python regex extension `(?P<name>RE)` allows use of named groups.

Match Objects

A match object is returned by some of the re functions and is typically used to obtain the group matches. There are a number of useful methods, particularly `groups` and `group`. E.g.:

```
m = re.match(r"(\w+) (\w+), \w+", “Isaac Newton, physicist”)

m.group(0)          # The full "(\w+) (\w+), \w+" match (whether grouped or not)
'Isaac Newton, physicist'

m.groups()          # Tuple of each group match
('Isaac', 'Newton')

m.group(2)          # The second group
'Newton'

m.group(1, 2, 1, 0)  # Just list what you want to see in the returned tuple
('Isaac', 'Newton', 'Isaac', 'Isaac Newton, physicist')
```

You don't actually need the `".group"` part, E.g.:

```
m[2]
'Newton'
```

Match object with named group:

```
m = re.match(r"(\w+) (?P<surname>\w+), \w+", “Isaac Newton, physicist”)

m["surname"]
'Newton'
```

RE Examples

A simple example of the match process for a pattern "abcdef" against a string "abcxef":
The a's match, the b's match, the c's match, but d doesn't match with x, so overall it didn't match.

A simple program matching a complex regex :-

```
import re

pattern = r"ABC[a-z]{2}DEF\w+ GHI(a|xy|dgp)JKL\1MNO([^rs])PQRx?STUr+VWX"
string = "ABCbdDEFfred GHIxyJKLxyMNOuPQRSTUrrrrVWX"

if re.fullmatch(pattern, string):
    print("Pattern:", pattern)
    print(“Matching string:”, string)
else:
    print("No match")
```

```
Pattern: ABC[a-z]{2}DEF\w+ GHI(a|xy|dgp)JKL\1MNO([^rs])PQRx?STUr+VWX
Matching string: ABCbdDEFfred GHIxyJKLxyMNOuPQRSTUrrrrVWX
```

How that complex pattern matches that string, and how it would match a similar one:

```
r"ABC[a-z]{2}DEF\w+ GHI(a|xy|dgp)JKL\1MNO([^rs])PQRx?STUr+VWX"

          "ABCbdDEFfred GHIxyJKLxyMNOuPQRSTUrrrrVWX"

          "ABCxxDEFjim GHIdgpJKLdgpMNOxPQRxSTUrVWX"
```

Useful Links

Online Python regular Expresssion evaluator and cheat sheet

Python Regex Cheatsheet

PythonSheets regex

shortcutFoo regex