Chapter 22
Regular Expressions in Python

22.1 Introduction

Regular Expression are a very powerful way of processing text while looking for recurring patterns; they are often used with data held in plain text files (such as log files), CSV files as well as Excel files. This chapter introduces regular expressions, discusses the syntax used to define a regular expression pattern and presents the Python `re` module and its use.

22.2 What Are Regular Expressions?

A Regular Expression (also known as a *regex* or even just *re*) is a sequence of characters (letters, numbers and special characters) that form a pattern that can be used to search text to see if that text contains sequences of characters that match the pattern.

For example, you might have a pattern defined as three characters followed by three numbers. This pattern could be used to look for such a pattern in other strings. Thus, the following strings either match (or contain) this pattern or they do not:

Abc123	Matches the pattern
A123A	Does not match the pattern
123AAA	Does not match the pattern

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_22

Regular Expression are very widely used for finding information in files, for example

- finding all lines in a log file associated with a specific user or a specific operation,
- for validating input such as checking that a string is a valid email address or postcode/ZIP code etc.

Support for Regular Expressions is wide spread within programming languages such as Java, C#, PHP and particularly Perl. Python is no exception and has the built-in module `re` (as well as additional third-party modules) that support Regular Expressions.

22.3 Regular Expression Patterns

You can define a regular expression pattern using any ASCII character or number. Thus, the string 'John' can be used to define a regex pattern that can be used to match any other string that contains the characters 'J', 'o', 'h', 'n'. Thus each of the following strings will match this pattern:

- 'John Hunt'
- 'John Jones'
- 'Andrew John Smith'
- 'Mary Helen John'
- 'John John John'
- 'I am going to visit the John'
- 'I once saw a film by John Wayne'

But the following strings would not match the pattern:

- 'Jon Davies' in this case because the spelling of John is different.
- 'john williams' in this case because the capital J does not match the lowercase j.
- 'David James' in this case because the string does not contain the string John!

Regular expressions (*regexs*) use special characters to allow more complex patterns to be described. For example, we can use the special characters ' [ ] ' to define a set of characters that can match. For example, if we want to indicate that the J may be a capital or a lower-case letter then we can write '[Jj]'—this indicates that either 'J' or 'j' can match the first.

- [Jj]ohn—this states that the pattern starts with either a capital J or a lowercase j followed by 'ohn'.

Now both 'john williams' and 'John Williams' will match this regex pattern.

22.3.1 Pattern Metacharacters

There are several special characters (often referred to as metacharacters) that have a specific meaning within a *regex* pattern, these are listed in the following table:

Character	Description	Example
[]	A set of characters	[a-d] characters in the sequence ‘a’ to ‘d’
\	Indicates a special sequence (can also be used to escape special characters)	‘\d’ indicates the character should be an integer
.	Any character with the exception of the newline character	‘J.hn’ indicates that there can be any character after the ‘J’ and before the ‘h’
^	Indicates a string must start with the following pattern	“^hello” indicates the string must start with ‘hello’
$	Indicates a string must end with the preceding pattern	“world$” indicates the string must end with ‘world’
*	Zero or more occurrences of the preceding pattern	“Python*” indicates we are looking for zero or more times Python is in a string
+	One or more occurrences of preceding pattern	“info+” indicates that we must find info in the string at least once
?	Indicates zero or 1 occurrences of the preceding pattern	“john?” indicates zero or one instances of the ‘John’
{}	Exactly the specified number of occurrences	“John{3}” this indicates we expect to see the ‘John’ in the string three times. “X{1,2}” indicates that there can be one or two Xs next to each other in the string
|	Either or	“True|OK” indicates we are looking for either True or OK
()	Groups together a regular expression; you can then apply another operator to the whole group	“(abc|xyz){2}” indicates that we are looking for the string abc or xyz repeated twice

22.3.2 Special Sequences

A special sequence is a combination of a ‘\’ (backslash) followed by a character combination which then has a special meaning. The following table lists the common special sequences used in Regular Expressions:

Sequence	Description	Example
\A	Returns a match if the following characters are at the beginning of the string	"\AThe" must start with 'The'
\b	Returns a match where the specified characters are at the beginning or at the end of a word	"\bon" or "on\b" indicates a string must start or end with 'on'
\B	Indicates that the following characters must be present in a string but not at the start (or at the end) of a word	r"\Bon" or r"on\B" must not start or end with 'on'
\d	Returns a match where the string contains digits (numbers from 0–9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s ∼ "
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0–9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the following characters are present at the end of the string	"Hunt\Z"

22.3.3 Sets

A set is a sequence of characters inside a pair of square brackets which have specific meanings. The following table provides some examples.

Set	Description
[jeh]	Returns a match where one of the specified characters (j, e or h) are present
[a–x]	Returns a match for any lower-case character, alphabetically between a and x
[^zxc]	Returns a match for any character EXCEPT z, x and c
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0–9]	Returns a match for any digit between 0 and 9
[0–9][0–9]	Returns a match for any two-digit numbers from 00 and 99
[a–zA–Z]	Returns a match for any character alphabetically between a and z or A and Z

22.4 The Python re Module

The Python `re` module is the built-in module provided by Python for working with Regular Expressions.

You might also like to examine the third party `regex` module (see https://pypi.org/project/regex) which is backwards compatible with the default `re` module but provides additional functionality.

22.5 Working with Python Regular Expressions

22.5.1 Using Raw Strings

An important point to note about many of the strings used to define the regular expression patterns is that they are preceded by an '`r`' for example `r'/bin/sh$'`.

The '`r`' before the string indicates that the string should be treated as a *raw* string.

A raw string is a Python string in which all characters are treated as exactly that; individual characters. It means that backslash ('`\`') is treated as a literal character rather than as a special character that is used to *escape* the next character.

For example, in a standard string '`\n`' is treated as a special character representing a newline, thus if we wrote the following:

```
s = 'Hello \n world'
print(s)
```

We will get as output:

```
Hello
 World
```

However, if we prefix the string with an '`r`' then we are telling Python to treat it as a raw string. For example:

```
s = r'Hello \n world'
print(s)
```

The output is now

```
Hello \n world
```

This is important for regular expression as characters such as backslash ('`\`') are used within patterns to have a special regular expression meaning and thus we do not want Python to process them in the normal way.

22.5.2 Simple Example

The following simple Python program illustrates the basic use of the `re` module. It is necessary to import the `re` module before you can use it.

```
import re

text1 = 'john williams'
pattern = '[Jj]ohn'
print('looking in', text1, 'for the pattern', pattern)

if re.search(pattern, text1):
    print('Match has been found')
```

When this program is run, we get the following output:

```
looking in john williams for the pattern [Jj]ohn
Match has been found
```

If we look at the code, we can see that the string that we are examining contains 'john williams' and that the pattern used with this string indicates that we are looking for a sequence of 'J' or 'j' followed by 'ohn'. To perform this test we use the `re.search()` function passing the regex pattern, and the text to test, as parameters. This function returns either `None` (which is taken as meaning *False* by the If statement) or a `Match` Object (which always has a Boolean value of *True*). As of course 'john' at the start of `text1` does match the pattern, the `re.search()` function returns a match object and we see the 'Match has been found' message is printed out.

Both the `Match` object and `search()` method will be described in more detail below; however, this short program illustrates the basic operation of a Regular Expression.

22.5.3 The Match Object

Match objects are returned by the `search()` and `match()` functions.

They always have a boolean value of `True`.

The functions `match()` and `search()` return `None` when there is no match and a `Match` object when a match is found. It is therefore possible to use a match object with an `if` statement:

```
import re

match = re.search(pattern, string)
if match:
    process(match)
```

Match objects support a range of methods and attributes including:

- `match.re` The regular expression object whose `match()` or `search()` method produced this match instance.
- `match.string` The string passed to `match()` or `search()`.
- `match.start([group])` / `match.end([group])` Return the indices of the start and end of the substring matched by group.
- `match.group()` returns the part of the string where there was a match.

22.5.4 The search() Function

The `search()` function searches the string for a match, and returns a Match object if there is a match. The signature of the function is:

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

The meaning of the parameters are:

- `pattern` this is the regular expression pattern to be used in the matching process.
- `string` this is the string to be searched.
- `flags` these (optional) flags can be used to modify the operation of the search.

The `re` module defines a set of flags (or indicators) that can be used to indicate any optional behaviours associated with the pattern. These flags include:

Flag	Description
`re.IGNORECASE`	Performs case-insensitive matching
`re.LOCALE`	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B)
`re.MULTILINE`	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string)
`re.DOTALL`	Makes a period (dot) match any character, including a newline
`re.UNICODE`	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B
`re.VERBOSE`	Ignores whitespace within the pattern (except inside a set `[]` or when escaped by a backslash) and treats unescaped # as a comment marker

If there is more than one match, only the first occurrence of the match will be returned:

```
import re


line1 = 'The price is 23.55'
containsIntegers = r'\d+'

if re.search(containsIntegers, line1):
    print('Line 1 contains an integer')
else:
    print('Line 1 does not contain an integer')
```

In this case the output is

```
Line 1 contains an integer
```

Another example of using the `search()` function is given below. In this case the pattern to look for defines three alternative strings (that is the string must contain either Beatles, Adele or Gorillaz):

```
import re

# Alternative words
music = r'Beatles|Adele|Gorillaz'
request = 'Play some Adele'

if re.search(music, request):
    print('Set Fire to the Rain')
else:
    print('No Adele Available')
```

In this case we generate the output:

```
Set Fire to the Rain
```

22.5.5 The match() Function

This function attempts to match a regular expression pattern at the beginning of a string. The signature of this function is given below:

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

The parameters are:

- `pattern` this is the regular expression to be matched.
- `string` this is the string to be searched.
- `flags` modifier flags that can be used.

The `re.match()` function returns a `Match` object on success, `None` on failure.

22.5.6 The Difference Between Matching and Searching

Python offers two different primitive operations based on regular expressions:

- `match()` checks for a match only at the beginning of the string,
- `search()` checks for a match anywhere in the string.

22.5.7 The findall() Function

The `findall()` function returns a list containing all matches. The signature of this function is:

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

This function returns all non-overlapping matches of `pattern` in `string`, as a list of strings.

The `string` is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, then a list of groups is returned; this will be a list of tuples if the pattern has more than one group. If no matches are found, an empty list is returned.

An example of using the `findall()` function is given below. This example looks for a substring starting with two letters and followed by 'ai' and a single character. It is applied to a sentence and returns only the sub string 'Spain' and 'plain'.

```
import re

str = 'The rain in Spain stays mainly on the plain'
results = re.findall('[a-zA-Z]{2}ai.', str)
print(results)
for s in results:
    print(s)
```

The output from this program is

```
['Spain', 'plain']
Spain
plain
```

22.5.8 The finditer() Function

This function returns an iterator yielding matched objects for the regular expression `pattern` in the `string` supplied. The signature for this function is:

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

The `string` is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result. Flags can be used to modify the matches.

22.5.9 The split() Function

The `split()` function returns a list where the string has been split at each match. The syntax of the `split()` function is

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

The result is to split a *string* by the occurrences of *pattern*. If capturing parentheses are used in the regular expression `pattern`, then the text of all groups in the `pattern` are also returned as part of the resulting list. If `maxsplit` is nonzero, at most `maxsplit` splits occur, and the remainder of the string is returned as the final element of the list. Flags can again be used to modify the matches.

```
import re

str = 'It was a hot summer night'
x = re.split('\s', str)
print(x)
```

The output is

```
['It', 'was', 'a', 'hot', 'summer', 'night']
```

22.5.10 The sub() Function

The `sub()` function replaces occurrences of the regular expression pattern in the string with the `repl` string.

```
  re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the regular expression pattern in string with `repl`, substituting all occurrences unless `max` is provided. This method returns the modified string.

```
import re

pattern = '(England|Wales|Scotland)'
input = 'England for football, Wales for Rugby and Scotland for
the Highland games'
print(re.sub(pattern, 'England', input ))
```

Which generates:

```
England for football, England for Rugby and England for the
Highland games
```

You can control the number of replacements by specifying the `count` parameter:
The following code replaces the first 2 occurrences:

```
import re

pattern = '(England|Wales|Scotland)'
input = 'England for football, Wales for Rugby and Scotland for
the Highland games'

x = re.sub(pattern, 'Wales', input, 2)
print(x)
```

which produces

```
Wales for football, Wales for Rugby and Scotland for the
Highland games
```

You can also find out how many substitutions were made using the `subn()` function. This function returns the new string and the number of substitutions in a tuple:

```
import re

pattern = '(England|Wales|Scotland)'
input = 'England for football, Wales for Rugby and Scotland for
the Highland games'

print(re.subn(pattern,'Scotland', input ))
```

The output from this is:

```
('Scotland for football, Scotland for Rugby and Scotland for
the Highland games', 3)
```

22.5.11 The compile() Function

Most regular expression operations are available as both module-level functions (as described above) and as methods on a compiled regular expression object.

The module-level functions are typically simplified or standardised ways to use the compiled regular expression. In many cases these functions are sufficient but if finer grained control is required then a compiled regular expression may be used.

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

The `compile()` function compiles a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods as described below.

The expression's behaviour can be modified by specifying a *flags* value. V

The statements:

```
prog = re.compile(pattern)
result = prog.match(string)
```

are equivalent to

```
result = re.match(pattern, string)
```

but using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

Compiled regular expression objects support the following methods and attributes:

- `Pattern.search(string, pos, endpos)` Scan through `string` looking for the first location where this regular expression produces a match and return a corresponding Match object. Return `None` if no position in the string matches the pattern. Starting at pos if provided and ending at endpos if this is provided (otherwise process the whole string).

- `Pattern.match(string, pos, endpos)` If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding match object. Return `None` if the string does not match the pattern. The `pos` and `endpos` are optional and specify the start and end positions within which to search.
- `Pattern.split(string, maxsplit = 0)` Identical to the `split()` function, using the compiled pattern.
- `Pattern.findall(string[, pos[, endpos]])` Similar to the `findall()` function, but also accepts optional `pos` and `endpos` parameters that limit the search region like for `search()`.
- `Pattern.finditer(string[, pos[, endpos]])` Similar to the `finditer()` function, but also accepts optional `pos` and `endpos` parameters that limit the search region like for `search()`.
- `Pattern.sub(repl, string, count = 0)` Identical to the `sub()` function, using the compiled pattern.
- `Pattern.subn(repl, string, count = 0)` Identical to the `subn()` function, using the compiled pattern.
- `Pattern.pattern` the pattern string from which the pattern object was compiled.

An example of using the `compile()` function is given below. The pattern to be compiled is defined as containing 1 or more digits (0 to 9):

```
import re

line1 = 'The price is 23.55'
containsIntegers = r'\d+'
rePattern = re.compile(containsIntegers)

matchLine1 = rePattern.search(line1)
if matchLine1:
    print('Line 1 contains a number')
else:
    print('Line 1 does not contain a number')
```

The compiled pattern can then be used to apply methods such as `search()` to a specific string (in this case held in `line1`). The output generated by this is:

```
Line 1 contains a number
```

Of course the compiler pattern object supports a range of methods in addition to `search()` as illustrated by the spilt method:

```
p = re.compile(r'\W+')
s = '20 High Street'
print(p.split(s))
```

The output from this is

```
['20', 'High', 'Street']
```

22.6 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/howto/regex.html Standard Library regular expression how to.
- https://pymotw.com/3/re/index.html the Python Module of the Week page for the `re` module.

Other online resources include

- https://regexone.com An introduction to regular expressions.
- https://www.regular-expressions.info/tutorial.html a regular expressions tutorial.
- https://www.regular-expressions.info/quickstart.html regular expressions quick start.
- https://pypi.org/project/regex A well known third party regular expression module that extends the functionality offered by the builtin re module.

22.7 Exercises

Write a Python function to verify that a given string only contains letters (upper case or lower case) and numbers. Thus spaces and underbars (‘_’) are not allowed. An example of the use of this function might be:

```
print(contains_only_characters_and_numbers('John')) # True
print(contains_only_characters_and_numbers('John_Hunt')) #
False
print(contains_only_characters_and_numbers('42')) # True
print(contains_only_characters_and_numbers('John42')) # True
print(contains_only_characters_and_numbers('John 42')) # False
```

Write a function to verify a UK Postcode format (call it `verify_postcode`). The format of a Postcode is two letters followed by 1 or 2 numbers, followed by a space, followed by one or two numbers and finally two letters. An Example of a postcode is SY23 4ZZ another postcode might be BB1 3PO and finally we might have AA1 56NN (note this is a simplification of the UK Postcode system but is suitable for our purposes).

Using the output from this function you should be able to run the following test code:

```
# True
print("verify_postcode('SY23 3AA'):", verify_postcode('SY23
33AA'))
# True
print("verify_postcode('SY23 4ZZ'):", verify_postcode('SY23
4ZZ'))
# True
print("verify_postcode('BB1 3PO'):", verify_postcode('BB1
3PO'))
# False
print("verify_postcode('AA111 NN56'):", verify_postcode('AA111
NN56'))
# True
print("verify_postcode('AA1 56NN'):", verify_postcode('AA1
56NN'))
# False
print("verify_postcode('AA156NN'):",
verify_postcode('AA156NN'))
# False
print("verify_postcode('AA NN'):", verify_postcode('AA NN'))
```

Write a function that will extract the value held between two strings or characters such as '<' and '>'. The function should take three parameters, the start character, the end character and the string to process. For example, the following code snippet:

```
print(extract_values('<', '>', '<John>'))
print(extract_values('<', '>', '<42>'))
print(extract_values('<', '>', '<John 42>'))
print(extract_values('<', '>', 'The <town> was in the
<valley>'))
```

Should generate output such as:

```
['John']
['42']
['John 42']
['town', 'valley']
```