Chapter 76: Regular Expressions (Regex)

Python makes regular expressions available through the `re` module.

Regular expressions are combinations of characters that are interpreted as rules for matching substrings. For instance, the expression `'amount\D+\d+'` will match any string composed by the word `amount` plus an integral number, separated by one or more non-digits, such as:`amount=100`, `amount is 3`, `amount is equal to: 33`, etc.

Section 76.1: Matching the beginning of a string

The first argument of `re.match()` is the regular expression, the second is the string to match:

```
import re

pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Out: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)

match.group()
# Out: '123'
```

You may notice that the pattern variable is a string prefixed with `r`, which indicates that the string is a *raw string literal*.

A raw string literal has a slightly different syntax than a string literal, namely a backslash `\` in a raw string literal means "just a backslash" and there's no need for doubling up backlashes to escape "escape sequences" such as newlines (`\n`), tabs (`\t`), backspaces (`\`), form-feeds (`\r`), and so on. In normal string literals, each backslash must be doubled up to avoid being taken as the start of an escape sequence.

Hence, `r"\n"` is a string of 2 characters: `\` and `n`. Regex patterns also use backslashes, e.g. `\d` refers to any digit character. We can avoid having to double escape our strings (`"\\d"`) by using raw strings (`r"\d"`).

For instance:

```
string = "\\t123zzb" # here the backslash is escaped, so there's no tab, just '\' and 't'
pattern = "\\t123"   # this will match \t (escaping the backslash) followed by 123
re.match(pattern, string).group()   # no match
re.match(pattern, "\t123zzb").group()  # matches '\t123'

pattern = r"\\t123"
re.match(pattern, string).group()   # matches '\\t123'
```

Matching is done from the start of the string only. If you want to match anywhere use `re.search` instead:

```
match = re.match(r"(123)", "a123zzb")

match is None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
```

```
# Out: '123'
```

Section 76.2: Searching

```
pattern = r"(your base)"
sentence = "All your base are belong to us."

match = re.search(pattern, sentence)
match.group(1)
# Out: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# Out: 'belong to us.'
```

Searching is done anywhere in the string unlike `re.match`. You can also use `re.findall`.

You can also search at the beginning of the string (use `^`),

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

at the end of the string (use $),

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

or both (use both `^` and $):

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

Section 76.3: Precompiled patterns

```
import re

precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("The answer is 41!")
matches.group(1)
# Out: 41

matches = precompiled_pattern.search("Or was it 42?")
matches.group(1)
# Out: 42
```

Compiling a pattern allows it to be reused later on in a program. However, note that Python caches recently-used

expressions (docs, SO answer), so *"programs that use only a few regular expressions at a time needn't worry about compiling regular expressions"*.

```
import re

precompiled_pattern = re.compile(r"(.*\d+)")
matches = precompiled_pattern.match("The answer is 41!")
print(matches.group(1))
# Out: The answer is 41

matches = precompiled_pattern.match("Or was it 42?")
print(matches.group(1))
# Out: Or was it 42
```

It can be used with re.match().

Section 76.4: Flags

For some special cases we need to change the behavior of the Regular Expression, this is done using flags. Flags can be set in two ways, through the `flags` keyword or directly in the expression.

Flags keyword

Below an example for `re.search` but it works for most functions in the `re` module.

```
m = re.search("b", "ABC")
m is None
# Out: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# Out: 'B'

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE)
m is None
# Out: True

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# Out: 'A\nB'
```

Common Flags

Flag	Short Description
`re.IGNORECASE`, `re.I`	Makes the pattern ignore the case
`re.DOTALL`, `re.S`	Makes `.` match everything including newlines
`re.MULTILINE`, `re.M`	Makes `^` match the begin of a line and `$` the end of a line
`re.DEBUG`	Turns on debug information

For the complete list of all available flags check the docs

Inline flags

From the docs:

> `(?iLmsux)` (One or more letters from the set 'i', 'L', 'm', 's', 'u', 'x'.)

> The group matches the empty string; the letters set the corresponding flags: re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode dependent), and re.X (verbose), for the entire regular expression. This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function.
>
> Note that the (?x) flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

Section 76.5: Replacing

Replacements can be made on strings using `re.sub`.

Replacing strings

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# Out: 'my name foo is foo what foo ever foo'
```

Using group references

Replacements with a small number of groups can be made as follows:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

However, if you make a group ID like '10', this doesn't work: `\10` is read as 'ID number 1 followed by 0'. So you have to be more specific and use the `\g<i>` notation:

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Using a replacement function

```
items = ["zero", "one", "two"]
re.sub(r"a\[([0-3])\]", lambda match: items[int(match.group(1))], "Items: a[0], a[1], something,
a[2]")
# Out: 'Items: zero, one, something, two'
```

Section 76.6: Find All Non-Overlapping Matches

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Note that the `r` before `"[0-9]{2,3}"` tells python to interpret the string as-is; as a "raw" string.

You could also use `re.finditer()` which works in the same way as `re.findall()` but returns an iterator with `SRE_Match` objects instead of a list of strings:

```
results = re.finditer(r"([0-9]{2,3})", "some 1 text 12 is 945 here 4445588899")
print(results)
# Out: <callable-iterator object at 0x105245890>
for result in results:
    print(result.group(0))
''' Out:
12
945
444
558
```

```
889
'''
```

Section 76.7: Checking for allowed characters

If you want to check that a string contains only a certain set of characters, in this case a-z, A-Z and 0-9, you can do so like this,

```
import re

def is_allowed(string):
    characherRegex = re.compile(r'[^a-zA-Z0-9.]')
    string = characherRegex.search(string)
    return not bool(string)

print (is_allowed("abyzABYZ0099"))
# Out: 'True'

print (is_allowed("#*@#$%^"))
# Out: 'False'
```

You can also adapt the expression line from `[^a-zA-Z0-9.]` to `[^a-z0-9.]`, to disallow uppercase letters for example.

Partial credit : http://stackoverflow.com/a/1325265/2697955

Section 76.8: Splitting a string using regular expressions

You can also use regular expressions to split a string. For example,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

Section 76.9: Grouping

Grouping is done with parentheses. Calling `group()` returns a string formed of the matching parenthesized subgroups.

```
match.group() # Group without argument returns the entire match found
# Out: '123'
match.group(0) # Specifying 0 gives the same result as specifying no argument
# Out: '123'
```

Arguments can also be provided to `group()` to fetch a particular subgroup.

From the docs:

> If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument.

Calling `groups()` on the other hand, returns a list of tuples containing the subgroups.

```
sentence = "This is a phone number 672-123-456-9910"
pattern = r".*(phone).*?([\d-]+)"

match = re.match(pattern, sentence)

match.groups()   # The entire match as a list of tuples of the paranthesized subgroups
# Out: ('phone', '672-123-456-9910')

m.group()        # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(0)       # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(1)       # The first parenthesized subgroup.
# Out: 'phone'

m.group(2)       # The second parenthesized subgroup.
# Out: '672-123-456-9910'

m.group(1, 2)    # Multiple arguments give us a tuple.
# Out: ('phone', '672-123-456-9910')
```

Named groups

```
match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')
match.group('name')
# Out: 'John Smith'

match.group(1)
# Out: 'John Smith'
```

Creates a capture group that can be referenced by name as well as by index.

Non-capturing groups

Using `(?:)` creates a group, but the group isn't captured. This means you can use it as a group, but it won't pollute your "group space".

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')

re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

This example matches `11+22` or `11`, but not `11+`. This is since the `+` sign and the second term are grouped. On the other hand, the `+` sign isn't captured.

Section 76.10: Escaping Special Characters

Special characters (like the character class brackets `[` and `]` below) are not matched literally:

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

By escaping the special characters, they can be matched literally:

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
```

```
# Out: '[b]'
```

The `re.escape()` function can be used to do this for you:

```
re.escape('a[b]c')
# Out: 'a\\[b\\]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# Out: 'a[b]c'
```

The `re.escape()` function escapes all special characters, so it is useful if you are composing a regular expression based on user input:

```
username = 'A.C.'  # suppose this came from the user
re.findall(r'Hi {}!'.format(username), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!', 'Hi ABCD!']
re.findall(r'Hi {}!'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!']
```

Section 76.11: Match an expression only in specific locations

Often you want to match an expression only in *specific* places (leaving them untouched in others, that is). Consider the following sentence:

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

Here the "apple" occurs twice which can be solved with so called *backtracking control verbs* which are supported by the newer **regex** module. The idea is:

```
forget_this | or this | and this as well | (but keep this)
```

With our apple example, this would be:

```
import regex as re
string = "An apple a day keeps the doctor away (I eat an apple everyday)."
rx = re.compile(r'''
    \([^()]*\) (*SKIP)(*FAIL)  # match anything in parentheses and "throw it away"
    |                          # or
    apple                      # match an apple
    ''', re.VERBOSE)
apples = rx.findall(string)
print(apples)
# only one
```

This matches "apple" only when it can be found outside of the parentheses.

Here's how it works:

- While looking from **left to right**, the regex engine consumes everything to the left, the `(*SKIP)` acts as an "always-true-assertion". Afterwards, it correctly fails on `(*FAIL)` and backtracks.
- Now it gets to the point of `(*SKIP)` **from right to left** (aka while backtracking) where it is forbidden to go any further to the left. Instead, the engine is told to throw away anything to the left and jump to the point where the `(*SKIP)` was invoked.

Section 76.12: Iterating over matches using `re.finditer`

You can use `re.finditer` to iterate over all matches in a string. This gives you (in comparison to `re.findall` extra information, such as information about the match location in the string (indexes):

```
import re
text = 'You can try to find an ant in this string'
pattern = 'an?\w' # find 'an' either with or without a following word character

for match in re.finditer(pattern, text):
    # Start index of match (integer)
    sStart = match.start()

    # Final index of match (integer)
    sEnd = match.end()

    # Complete match (string)
    sGroup = match.group()

    # Print match
    print('Match "{}" found at: [{},{}]'.format(sGroup, sStart,sEnd))
```

Result:

```
Match "an" found at: [5,7]
Match "an" found at: [20,22]
Match "ant" found at: [23,26]
```