**Regular Expressions in Python**

Regular expressions are used to identify whether a pattern exists in a given sequence of characters (string) or not. They help in manipulating textual data, which is often a pre-requisite for data science projects that involve text mining. You must have come across some application of regular expressions: they are used at the server side to validate the format of email addresses or password during registration, used for parsing text data files to find, replace or delete certain string, etc.

In Python, regular expressions are supported by the [re](https://docs.python.org/3/library/re.html) module. That means that if you want to start using them in your Python scripts, you have to import this module with the help of import.

“import re”

**Basic Patterns: Ordinary Characters**

You can easily tackle many basic patterns in Python using the ordinary characters. Ordinary characters are the simplest regular expressions. They match themselves exactly and do not have a special meaning in their regular expression syntax.

Examples are 'A', 'a', 'X', '5'.

Ordinary characters can be used to perform simple exact matches:

Example:

import re

pattern = r"Cookie"

sequence = "Cookie"

if re.match(pattern, sequence):

print("Match!")

else:

print("Not a match!")

output:

Match!

The match() function returns a match object if the text matches the pattern. Otherwise it returns None. The re module also contains several other functions.

The “r” at the start of the pattern Cookie is called a “raw string literal”. It changes how the string literal is interpreted. Such literals are stored as they appear.

**Wild Card Characters: Special Characters**

Special characters are characters which do not match themselves as seen but actually have a special meaning when used in a regular expression.

The most widely used special characters are:

1. **.** - A period. Matches any single character except newline character.

re.search(r'Co.k.e', 'Cookie').group()

'Cookie'

The group() function returns the string matched by the re.

1. \w - Lowercase w. Matches any single letter, digit or underscore.

re.search(r'Co\wk\we', 'Cookie').group()

'Cookie'

1. \W - Uppercase w. Matches any character not part of \w (lowercase w).

re.search(r'C\Wke', 'C@ke').group()

'C@ke'

1. \s - Lowercase s. Matches a single whitespace character like: space, newline, tab, return.

re.search(r'Eat\scake', 'Eat cake').group()

'Eat cake'

1. \S - Uppercase s. Matches any character not part of \s (lowercase s).

re.search(r'Cook\Se', 'Cookie').group()

'Cookie'

1. \t - Lowercase t. Matches tab.

re.search(r'Eat\tcake', 'Eat cake').group()

'Eat\tcake'

1. \n - Lowercase n. Matches newline.
2. \r - Lowercase r. Matches return.
3. \d - Lowercase d. Matches decimal digit 0-9.

re.search(r'c\d\dkie', 'c00kie').group()

'c00kie'

1. ^ - Caret. Matches a pattern at the start of the string.

re.search(r'^Eat', 'Eat cake').group()

'Eat'

1. $ - Matches a pattern at the end of string.

re.search(r'cake$', 'Eat cake').group()

'cake'

1. [abc] - Matches a or b or c.
2. [a-zA-Z0-9] - Matches any letter from (a to z) or (A to Z) or (0 to 9). Characters that are not within a range can be matched by complementing the set. If the first character of the set is ^, all the characters that are not in the set will be matched.

re.search(r'Number: [0-6]', 'Number: 5').group()

'Number: 5'

1. \A - Uppercase a. Matches only at the start of the string. Works across multiple lines as well.

re.search(r'\A[A-E]ookie', 'Cookie').group()

'Cookie'

1. \b - Lowercase b. Matches only the beginning or end of the word.

re.search(r'\b[A-E]ookie', 'Cookie').group()

'Cookie'

1. \ - Backslash. If the character following the backslash is a recognized escape character, then the special meaning of the term is taken. For example, \n is considered as newline. However, if the character following the \ is not a recognized escape character, then the \ is treated like any other character and passed through.

Let's look at a couple of examples:

# This checks for '\' in the string instead of '\t' due to the '\' used

re.search(r'Back\\stail', 'Back\stail').group()

'Back\\stail'

# This treats '\s' as an escape character because it lacks '\' at the start of '\s'

re.search(r'Back\stail', 'Back tail').group()

'Back tail'

Repetitions

It becomes quite tedious if you are looking to find long patterns in a sequence. Fortunately, the remodule handles repetitions using the following special characters:

1. + - Checks for one or more characters to its left.

re.search(r'Co+kie', 'Cooookie').group()

'Cooookie'

1. \* - Checks for zero or more characters to its left.

# Checks for any occurrence of a or o or both in the given sequence

re.search(r'Ca\*o\*kie', 'Caokie').group()

'Caokie'

1. ? - Checks for exactly zero or one character to its left.

re.search(r'Colou?r', 'Color').group()

'Color'

But what if you want to check for exact number of sequence repetition?

For example, checking the validity of a phone number in an application. re module handles this very gracefully as well using the following regular expressions:

1. {x} - Repeat exactly x number of times.
2. {x,} - Repeat at least x times or more.
3. {x, y} - Repeat at least x times but no more than y times.

re.search(r'\d{9,10}', '0987654321').group()

'0987654321'

The + and \* qualifiers are said to be greedy.

**Groups and Grouping using Regular Expressions**

Suppose that, when you're validating email addresses and want to check the user name and host separately.

This is when the group feature of regular expression comes in handy. It allows you to pick up parts of the matching text.

Parts of a regular expression pattern bounded by parenthesis() are called groups. The parenthesis does not change what the expression matches, but rather forms groups within the matched sequence. You have been using the group() function all along in this tutorial's examples. The plain match.group() without any argument is still the whole matched text as usual.

**Greedy vs Non-Greedy Matching**

When a special character matches as much of the search sequence (string) as possible, it is said to be a "Greedy Match". It is the normal behavior of a regular expression but sometimes this behavior is not desired:

pattern = "cookie"

sequence = "Cake and cookie"

example:

heading = r'<h1>TITLE</h1>'

re.match(r'<.\*>', heading).group()

'<h1>TITLE</h1>'

The pattern <.\*> matched the whole string, right up to the second occurrence of >.

However, if you only wanted to match the first <h1> tag, you could have used the greedy qualifier \*? that matches as little text as possible.

Adding ? after the qualifier makes it perform the match in a non-greedy or minimal fashion; That is, as few characters as possible will be matched. When you run <.\*?>, you will only get a match with <h1>.

heading = r'<h1>TITLE</h1>'

re.match(r'<.\*?>', heading).group()

'<h1>'

**re Python Library**

The re library in Python provides several functions that makes it a skill worth mastering. You have already seen some of them, such as the re.search(), re.match(). Let's check out some useful functions in detail:

**search(pattern, string, flags=0)**

With this function, you scan through the given string/sequence looking for the first location where the regular expression produces a match. It returns a corresponding match object if found, else returns None if no position in the string matches the pattern. Note that None is different from finding a zero-length match at some point in the string.

pattern = "cookie"

sequence = "Cake and cookie"

re.search(pattern, sequence).group()

'cookie'

**match(pattern, string, flags=0)**

Returns a corresponding match object if zero or more characters at the beginning of string match the pattern. Else it returns None, if the string does not match the given pattern.

pattern = "C"

sequence1 = "IceCream"

# No match since "C" is not at the start of "IceCream"

re.match(pattern, sequence1)

sequence2 = "Cake"

re.match(pattern,sequence2).group()

'C'

**search() versus match()**

The match() function checks for a match only at the beginning of the string (by default) whereas the search() function checks for a match anywhere in the string.

**findall(pattern, string, flags=0)**

Finds all the possible matches in the entire sequence and returns them as a list of strings. Each returned string represents one match.

email\_address = "Please contact us at: support@datacamp.com, xyz@datacamp.com"

#'addresses' is a list that stores all the possible match

addresses = re.findall(r'[\w\.-]+@[\w\.-]+', email\_address)

for address in addresses:

print(address)

support@datacamp.com

xyz@datacamp.com

**sub(pattern, repl, string, count=0, flags=0)**

This is the substitute function. It returns the string obtained by replacing or substituting the leftmost non-overlapping occurrences of pattern in string by the replacement repl. If the pattern is not found then the string is returned unchanged.

email\_address = "Please contact us at: xyz@datacamp.com"

new\_email\_address = re.sub(r'([\w\.-]+)@([\w\.-]+)', r'support@datacamp.com', email\_address)

print(new\_email\_address)

Please contact us at: support@datacamp.com

**compile(pattern, flags=0)**

Compiles a regular expression pattern into a [regular expression object](https://docs.python.org/3/library/re.html#re-objects). When you need to use an expression several times in a single program, using the compile() function to save the resulting regular expression object for reuse is more efficient. This is because the compiled versions of the most recent patterns passed to compile() and the module-level matching functions are cached.

pattern = re.compile(r"cookie")

sequence = "Cake and cookie"

pattern.search(sequence).group()

'cookie'

# This is equivalent to:

re.search(pattern, sequence).group()

'cookie'