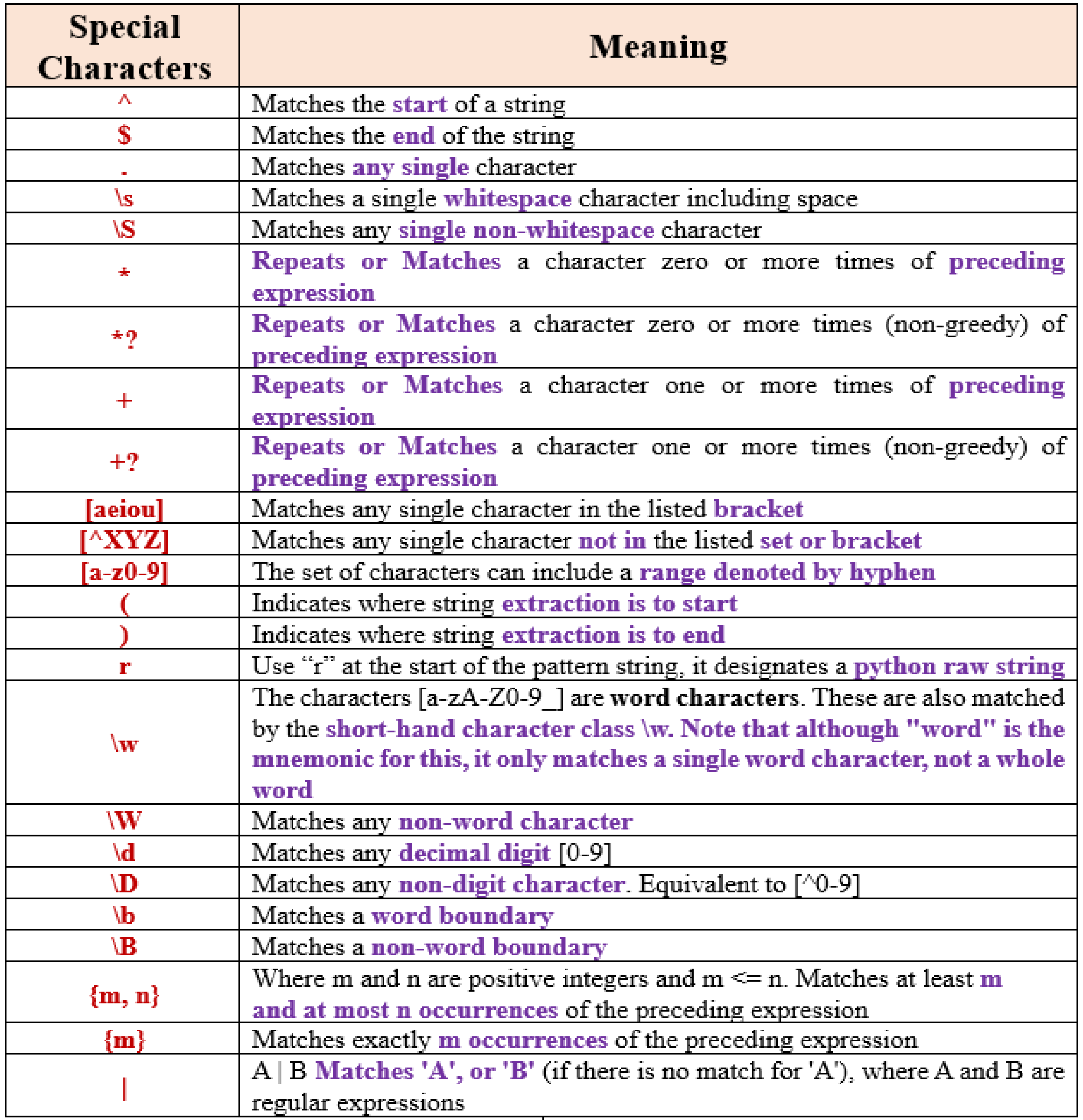
Regular Expression Operations:

- Using Special Characters,
- Regular Expression Methods,
- Named Groups in Python Regular Expressions,
- Regular Expression with glob Module.

Regular Expression Operations

- Regular expressions, also called REs, or regexes, or regex patterns, provide a powerful way to search and manipulate strings.
- Regular expressions are essentially a tiny, highly specialized programming language embedded inside Python and made available through the ***re* module**.
- Regular expressions use a sequence of characters and symbols to define a pattern of text. Such a pattern is used to locate a chunk of text in a string by matching up the pattern against the characters in the string.
- Regular expressions are useful for finding phone numbers, email addresses, dates, and any other data that has a consistent format.
- The module re has to be imported to be able to work with regular expressions.

Using Special Characters



Special Characters	Meaning
^	Matches the **start** of a string
$	Matches the **end** of the string
.	Matches **any single** character
\s	Matches a single **whitespace** character including space
\S	Matches any **single non-whitespace** character
*	**Repeats or Matches** a character zero or more times of **preceding expression**
*?	**Repeats or Matches** a character zero or more times (non-greedy) of **preceding expression**
+	**Repeats or Matches** a character one or more times of **preceding expression**
+?	**Repeats or Matches** a character one or more times (non-greedy) of **preceding expression**
[aeiou]	Matches any single character in the listed **bracket**
[^XYZ]	Matches any single character **not in** the listed **set or bracket**
[a-z0-9]	The set of characters can include a **range denoted by hyphen**
(	Indicates where string **extraction is to start**
)	Indicates where string **extraction is to end**
r	Use "r" at the start of the pattern string, it designates a **python raw string**
\w	The characters [a-zA-Z0-9_] are **word characters**. These are also matched by the **short-hand character class \w. Note that although "word" is the mnemonic for this, it only matches a single word character, not a whole word**
\W	Matches any **non-word character**
\d	Matches any **decimal digit** [0-9]
\D	Matches any **non-digit character**. Equivalent to [^0-9]
\b	Matches a **word boundary**
\B	Matches a **non-word boundary**
{m, n}	Where m and n are positive integers and m <= n. Matches at least **m and at most n occurrences** of the preceding expression
{m}	Matches exactly **m occurrences** of the preceding expression
|	A | B **Matches 'A', or 'B'** (if there is no match for 'A'), where A and B are regular expressions

Using Special Characters

Complete list and description of the special characters that can be used in regular expressions.

Special Character →	[xyz]	- indicate a set of characters.
Special Character →	.	- Matches any single character except newline '\n‘
Special Character →	^	- Matches the start of the string and, in multiline mode, also matches immediately after each newline.
Special Character →	$	- Matches the end of the string or just before the newline at the end of the string.
Special Character →	*	- Matches the preceding expression 0 or more times.
Special Character →	+	- Matches the preceding expression 1 or more times.
Special Character →	*?*	- *Matches the preceding expression 0 or 1 time.*
Special Character →	*\d*	- *Matches any decimal digit [0-9]*
Special Character →	*\D*	- *Matches any non-digit character. Equivalent to [^0-9]*
Special Character →	*\w*	- *Matches a "word" character and it can be a letter or digit or underscore.*
Special Character →	*\W*	- *Matches any non-word character*. Equivalent to [^A-Za-z0-9_]

Using Special Characters

- *Special Character* → \s - Matches a single whitespace character including space, newline, tab, form feed. Equivalent to [\n\t\f]
- *Special Character* → \S - Matches any non-whitespace character. Equivalent to [^ \n\t\f].
- *Special Character* → \b - Matches a word boundary.
- *Special Character* → \B - Matches a non-word boundary.
- *Special Character* → {m, n} - Where m and n are positive integers and m <= n.
- *Special Character* → {m} - Matches exactly m occurrences of the preceding expression.
- *Special Character* → | - A | B Matches 'A', or 'B' (if there is no match for 'A'), where A and B are regular expressions.

Using *r* Prefix for Regular Expressions

- The '*r*' prefix tells Python that the expression is a raw string and are handy in regular expressions. In a raw string, escape sequences are not parsed. For example, '\n' is a single newline character. But, r'\n' would be two characters: a backslash and an 'n'.

Using Parentheses in Regular Expressions

- *Special Character* → (….) - Matches whatever regular expression pattern is inside the parentheses and causes that part of the matched substring to be remembered.

Steps- to build and use regular expressions

In order to build and use regular expressions, perform the following steps:

- *Step 1*: Import *re* regular expression module.
- *Step 2*: Compile regular expression pattern using re.*compile()* method. This method returns the regular expression pattern as an object.
- *Step 3*: Invoke an appropriate method supported by the compiled regular expression object which returns a matched object instance containing information about matched strings.
- *Step 4*: Call methods (*group()* method is appropriate for most cases) associated with the matched object to display the results.

Compiling Regular Expressions Using compile() Method of re Module

- Regular expressions can be compiled into a pattern object, which has methods for various operations such as searching for pattern matches, finding all pattern matches or performing string substitutions. This can be accomplished through the use of the re.compile() method: ***re.compile(pattern[,flags])***

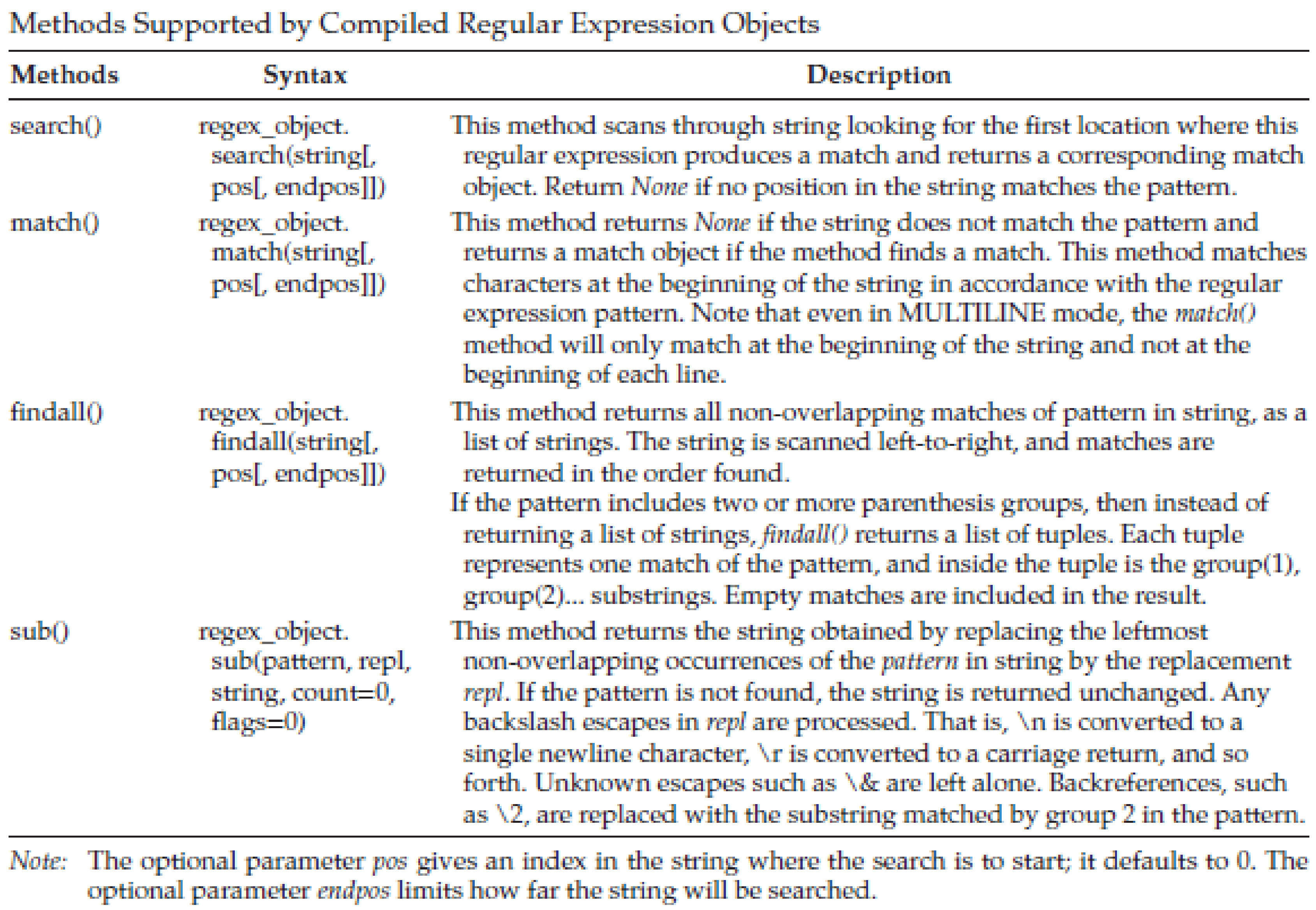
  where pattern is the regular expression and the optional flags argument is used to enable various special features and syntax variations.

- For example, specifying the flag re.A enables ASCII-only matching, The flag re.I enables case-insensitive matching; expressions like [A-Z] will also match lowercase letters. The flag re.M enables “multi-line matching” . When re.M flag is enabled, the meaning of '^' and '$' changes.

- The special character '^' matches at the beginning of the string and also at the beginning of each line (immediately following each newline); and the special character '$' matches at the end of the string and also at the end of each line (immediately preceding each newline).

- Use the *compile()* method in re module to compile regular expression to match objects. Use various methods like *search()*, *match()*, *findall()*, and *sub()* methods to extract substrings matching a pattern.

Regular Expression Methods



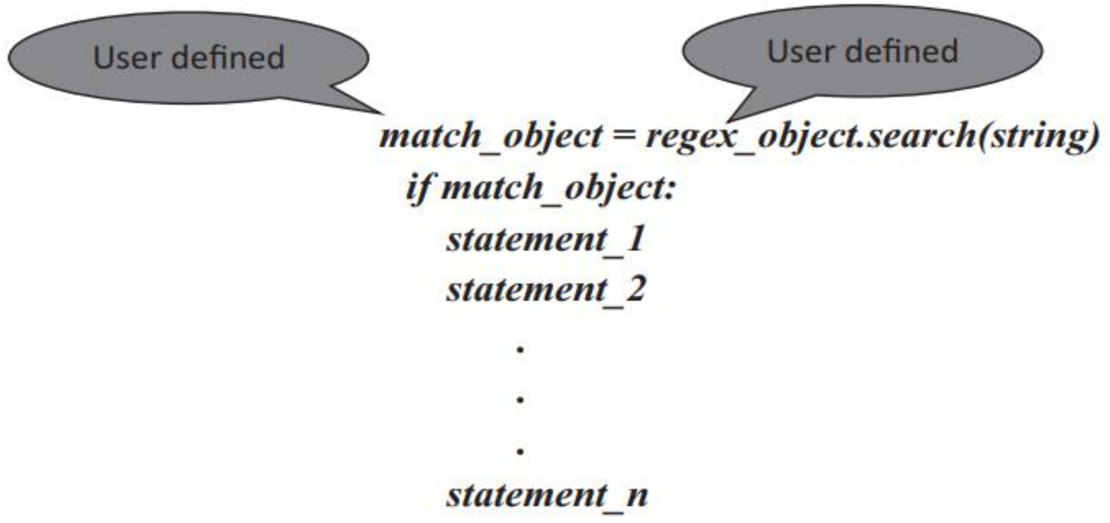
Methods Supported by Compiled Regular Expression Objects

Methods	Syntax	Description
search()	regex_object.search(string[, pos[, endpos]])	This method scans through string looking for the first location where this regular expression produces a match and returns a corresponding match object. Return *None* if no position in the string matches the pattern.
match()	regex_object.match(string[, pos[, endpos]])	This method returns *None* if the string does not match the pattern and returns a match object if the method finds a match. This method matches characters at the beginning of the string in accordance with the regular expression pattern. Note that even in MULTILINE mode, the *match()* method will only match at the beginning of the string and not at the beginning of each line.
findall()	regex_object.findall(string[, pos[, endpos]])	This method 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 the pattern includes two or more parenthesis groups, then instead of returning a list of strings, *findall()* returns a list of tuples. Each tuple represents one match of the pattern, and inside the tuple is the group(1), group(2)... substrings. Empty matches are included in the result.
sub()	regex_object.sub(pattern, repl, string, count=0, flags=0)	This method returns the string obtained by replacing the leftmost non-overlapping occurrences of the *pattern* in string by the replacement *repl*. If the pattern is not found, the string is returned unchanged. Any backslash escapes in *repl* are processed. That is, \n is converted to a single newline character, \r is converted to a carriage return, and so forth. Unknown escapes such as \& are left alone. Backreferences, such as \2, are replaced with the substring matched by group 2 in the pattern.

Note: The optional parameter *pos* gives an index in the string where the search is to start; it defaults to 0. The optional parameter *endpos* limits how far the string will be searched.

Match Objects

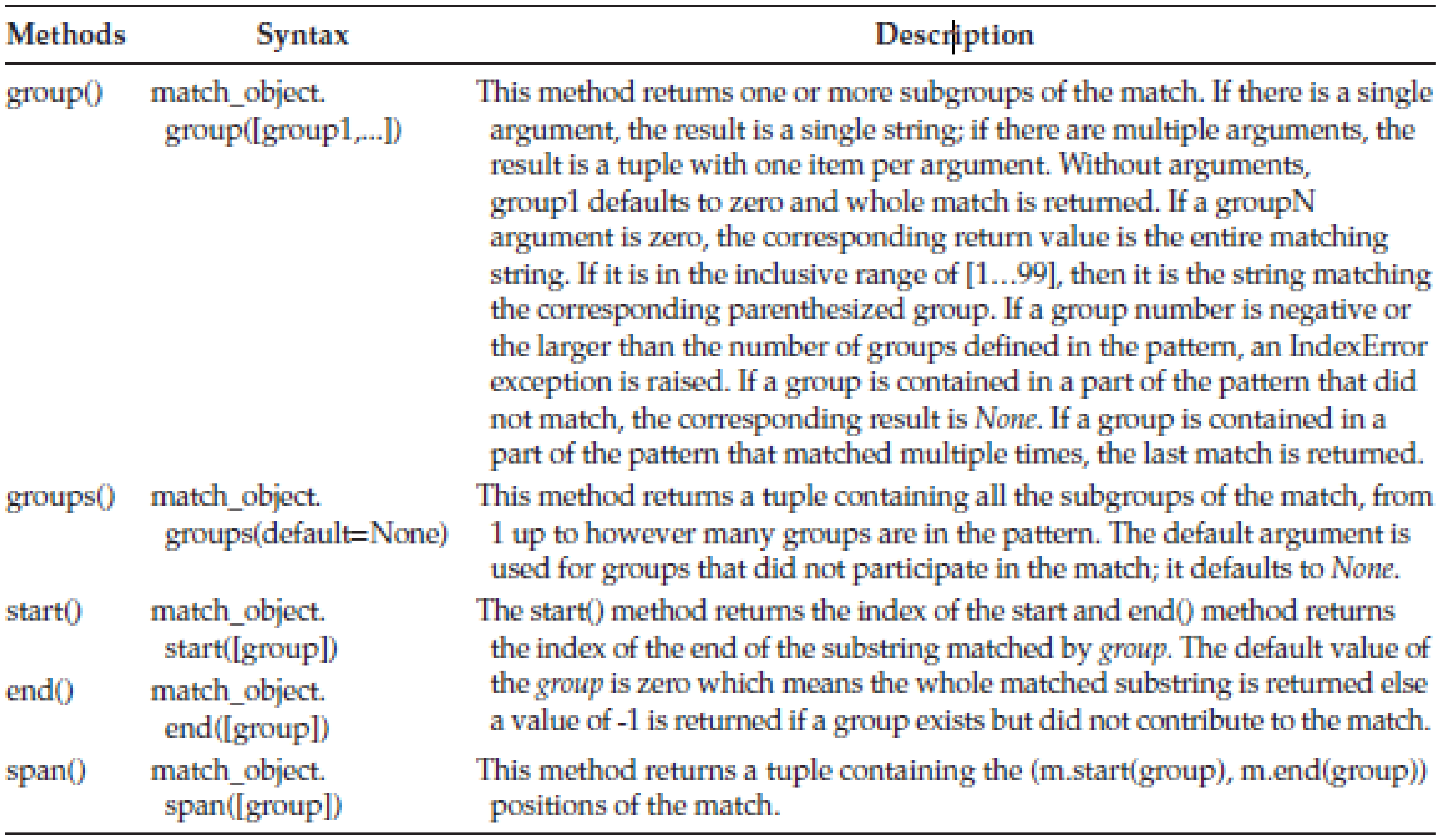
- The match() and search() methods supported by a compiled regular expression object, returns None if no match is found. If they are successful, a match object instance is returned, containing information about the match like the substring it has matched, where the match starts and ends and much more.
- The main difference between search() and match() methods is search() method searches anywhere in the entire string and returns a match object while the match() method matches zero or more characters at the beginning of the string and returns a match object.
- Since match() and search() return None when there is no match, you can test whether there was a match with a simple if statement as shown below.



```
match_object = regex_object.search(string)
 if match_object:
   statement_1
   statement_2
        .
        .
        .
   statement_n
```

Match object supports several methods and only the most significant ones are covered

Methods Supported by Match Object



Methods	Syntax	Description
group()	match_object. group([group1,...])	This method returns one or more subgroups of the match. 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. Without arguments, group1 defaults to zero and whole match is returned. If a groupN argument is zero, the corresponding return value is the entire matching string. If it is in the inclusive range of [1…99], then it is the string matching the corresponding parenthesized group. If a group number is negative or the larger than the number of groups defined in the pattern, an IndexError exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is *None*. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.
groups()	match_object. groups(default=None)	This method returns a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to *None*.
start()	match_object. start([group])	The start() method returns the index of the start and end() method returns the index of the end of the substring matched by *group*. The default value of the *group* is zero which means the whole matched substring is returned else a value of -1 is returned if a group exists but did not contribute to the match.
end()	match_object. end([group])	
span()	match_object. span([group])	This method returns a tuple containing the (m.start(group), m.end(group)) positions of the match.

Discussions:

- Regular Expressions are developed Based applications by using python module : re
- import re
- Compile() : re module contains compile() function to compile a pattern into RegexObject.

  Example: pattern= re.compile ("python")
- finditer() : Returns an Iterator object which yields Match object for every Match

  Example: matcher=pattern.finditer(" programmin in python")

- On Match object we can call the following methods:
  - ✓ start() -start index of the match
  - ✓ end() - end+1 indes of match
  - ✓ group()-: returns matched stringis

- pattern=re.compile("ab")

  matcher=pattern.finditer("abcababa")

or

- matcher=re.finditer("ab","abcabaaab")

Examples

# program 1

```
import re
pattern=re.compile("ab")
count=0
matcher=pattern.finditer("abcababa")
for i in matcher:
    count+=1
    print("match is available at start index: ", i.start())
print("the no of occurances: ", count)
```

# program 2

```
import re
count=0
pattern=re.compile("ab")
matcher=pattern.finditer("abcababa")
for m in matcher:
    count+=1
    print("start:{},  end:{}, group:{} ".format(m.start(), m.end(), m.group()))
print("the no of occurances: ", count)
```

# program 3

```
import re
count=0
#pattern=re.compile("ab")
#matcher=pattern.finditer("abcababa")
matcher=re.finditer("ab","abcababa")
for m in matcher:
    count+=1
    print("start:{},  end:{}, group:{} ".format(m.start(), m.end(), m.group()))
print("the no of occurances: ", count)
```

Character classes:

character classes can be used to search a group of characters:

1. [abc]	----Either a or b or c
2. [^abc]	----Except a and b and c
3. [a-z]	----Any Lower case alphabet symbol
4. [A-Z]	----Any upper case alphabet symbol
5. [a-zA-Z]	----Any alphabet symbol
6. [0-9]	---- Any digit from 0 to 9
7. [a-zA-Z0-9]	----Any alphanumeric character
8. [^a-zA-Z0-9]	----Except alphanumeric characters(Special Characters)

Example

```
# program 4


import re
matcher=re.finditer("[0-9]","a6B@ k9z")
for i in matcher:
    print(i.start(), '----',i.group())
```

Pre defined Character classes

\s	-- Space character
\S	-- Any character except space character
\d	-- Any digit from 0 to 9
\D	-- Any character except digit
\w	-- Any word character [a-zA-Z0-9]
\W	-- Any character except word character (Special Characters)
.	-- Any character including special characters

Example:

```
import re
matcher=re.finditer("\d","a6B@ k9z")
for i in matcher:
    print(i.start(), '----',i.group())
```

Quantifiers:

- We can use quantifiers to specify the number of occurrences to match.
- a -- Exactly one 'a'
- a+ -- Atleast one 'a'
- a* -- Any number of a's including zero number
- a? -- Atmost one 'a' ie either zero or one number
- a{m} -- Exactly m number of a's
- a{m,n} -- Minimum m number of a's and Maximum n number of a's

- ^a -- It will check whether target string starts with a or not

  We can use ^ symbol to check whether the given target string starts with our provided pattern or not.
- a$ -- It will check whether target string ends with a or not

  We can use $ symbol to check whether the given target string ends with our provided pattern or not

Important functions

finditer()

- Returns the iterator yielding a match object for each match.On each match object we can call start(), end() and group() functions.

match()

- We can use match function to check the given pattern at beginning of target string.If the match is available then we will get Match object, otherwise we will get None.

fullmatch()

- We can use fullmatch() function to match a pattern to all of target string. i.e complete string should be matched according to given pattern. If complete string matched then this function returns Match object otherwise it returns None.

search()

- We can use search() function to search the given pattern in the target string. If the match is available then it returns the Match object which represents first occurrence of the match. If the match is not available then it returns None

Important functions

findall()

- To find all occurrences of the match.This function returns a list object which contains all occurrences.

sub()

- sub means substitution or replacement.
- re.sub(regex,replacement,targetstring)
- In the target string every matched pattern will be replaced with provided replacement.

subn()

- It is exactly same as sub except it can also returns the number of replacements. This function returns a tuple where first element is result string and second element is number of replacements.
- (resultstring, number of replacements)

split()

- If we want to split the given target string according to a particular pattern then we should go for split() function.This function returns list of all tokens.

compile()

- re module contains compile() function to compile a pattern into RegexObject.

Example

```
>>> import re
>>> pattern = re.compile(r'(e)g')
>>> pattern
            re.compile('(e)g')
>>> match_object = pattern.match('egg is nutritional food')
>>> match_object
          <re.Match object; span=(0, 2), match='eg'>
>>> pattern = re.compile(r'(ab)*')
>>> match_object = pattern.match('ababababab')
>>> match_object.span()
          (0, 10)
>>> match_object.start()
          0
>>> match_object.end()
          10
```

```
>>> pattern = re.compile(r'(a(b)c)d')
>>> method_object = pattern.match('abcd')
>>> method_object.group(0)
          'abcd'
>>> method_object.group(1)
          'abc'
>>> method_object.group(2)
          'b'
>>> method_object.group(2,1,2)
          ('b', 'abc', 'b')
>>> method_object.groups()
          ('abc', 'b')
```

Example

```
>>> pattern = re.compile(r'\d+')
>>> match_list = pattern.findall("Everybody think they're famous when they get 100000 followers on Instagram and 5000 on Twitter")
>>> match_list
    ['100000', '5000']
>>> pattern = re.compile(r'([\w\.]+)@([\w\.]+)')
>>> matched_email_tuples = pattern.findall('bill_gates@microsoft.com and steve.jobs@apple.com are visionaries')
>>> print(matched_email_tuples)
  [('bill_gates', 'microsoft.com'), ('steve.jobs', 'apple.com')]
>>> for each_mail in matched_email_tuples:
     print(f"User name is {each_mail[0]}")
     print(f"Domain name is {each_mail[1]}")
    User name is bill_gates
    Domain name is microsoft.com
    User name is steve.jobs
    Domain name is apple.com
    pattern = re.compile(r'(\w+)\s(\w+)')
```

'Program : Given an Input File Which Contains a List of Names and Phone Numbers Separated by Spaces in the Following Format:

Alex 80-23425525

Emily 322-56775342

Grace 20-24564555

Anna 194-49611659

Phone Number Contains a 3- or 2-Digit Area Code and a Hyphen Followed By an 8-Digit Number.

Find All Names Having Phone Numbers with a 3-Digit Area Code Using Regular Expressions.

```
import re
pattern = re.compile(r"(\w+)\s+\d{3}-\d{8}")
with open("person_details.txt", "r") as file_handler:
    print("Names having phone numbers with 3 digit area code")
    for i in file_handler:
        match_object = pattern.search(i)
        if match_object:
            print(match_object.group(1))
```

Program : Write a Python Program to Check the Validity of a Password Given by User.

The Password Should Satisfy the Following Criteria:

- Contain at least 1 letter between a and z
- Contain at least 1 number between 0 and 9
- Contain at least 1 letter between A and Z
- Contain at least 1 character from $, #, @
- Minimum length of password: 6
- Maximum length of password: 12

```
import re
lower_case_pattern = re.compile(r'[a-z]')
upper_case_pattern = re.compile(r'[A-Z]')
number_pattern = re.compile(r'\d')
special_character_pattern = re.compile(r'[$#@]')
password = input("Enter a Password ")
if len(password) < 6 or len(password) > 12:
        print("Invalid Password. Length Not Matching")
elif not lower_case_pattern.search(password):
        print("Invalid Password. No Lower-Case Letters")
elif not upper_case_pattern.search(password):
        print("Invalid Password. No Upper-Case Letters")
elif not number_pattern.search(password):
         print("Invalid Password. No Numbers")
elif not special_character_pattern.search(password):
        print("Invalid Password. No Special Characters")
else:
        print("Valid Password")
```

Program : Write Python Program to Validate U.S.-based Social Security Number

```
import re
pattern = re.compile(r"\b\d{3}-?\d{2}-?\d{4}\b")
match_object = pattern.search("Social Security Number for James is 916-30-2017")
if match_object:
    print(f"Extracted Social Security Number is {match_object.group()}")
else:
    print("No Match")
```

Named Groups in Python Regular Expressions

- Regular expressions use groups to capture strings of interest.
- As the regular expression becomes complex, it gets difficult to keep track of the number of groups in the regular expression.
- In order to overcome this problem Python provides named groups. Instead of referring to the groups by numbers, you can reference them by a name.
- The syntax for a named group is, ***(?P<name>RE)***

  where the first name character is ?, followed by

  letter *P* (uppercase letter) that stands for Python Specific extension,

  name is the name of the group written within angle brackets, and

  RE is the regular expression.
- Named groups behave exactly like capturing groups, and additionally associate a name with a group.
- The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group’s name.

Named Groups in Python Regular Expressions

```
>>> pattern = re.compile(r'(?P<word>\b\w+\b)')
>>> match_object = pattern.search('laugh out loud')
>>> match_object.group('word')
        'laugh'
>>> match_object.group(1)
        'laugh‘
```

- In the above code, the regular expression has a group that matches the pattern of a word boundary followed by one of more alphanumeric characters, that is, a-z, A-Z, 0-9 and _, followed by a word boundary.
- The name given to this group is <word> specified within angle brackets .
- Pass the string from which you want to extract the pattern as an argument to the *search()* method.
- By passing the group name ‘word’ as an argument to the *group()* method, you can extract the matched substring . This named group can still be used to retrieve information by the passed integer numbers instead of group name

Regular Expression with glob Module

- The glob module finds all the file names matching a specified pattern.
- Starting with Python version 3.5, the glob module supports the "**" directive (which is parsed only if you pass recursive flag).
- In earlier Python versions, *glob.glob()* did not list files in subdirectories recursively.
- The syntax for glob method is, ***glob.glob(pathname, **, recursive=True)***
- The *glob()* method of the glob module returns a possible list of file names that match a pathname, which must be a string containing a path specification.
- Here *pathname* can be either absolute (like C:\Anaconda\Python\Python3.6.exe) or relative like..\..\Tools**.gif).
- If recursive is True, the pattern "**" will match any files and zero or more directories and subdirectories.

Program: Write Python Program to Change the File Extension from .csv to .txt of All the Files (Including from Sub Directories) for a Given Path

```
import os
import glob
def rename_files_recursively(directory_path):
    print("File extension changed from .csv to .txt")
    for file_path in glob.glob(directory_path + '\**\*.csv', recursive=True):
        print(f"File with .csv extension {file_path} changed to", end="")
        try:
            pre, ext = os.path.splitext(file_path)
            print(f" File with .txt extension {pre + '.txt'}")
            os.rename(file_path, pre + '.txt')
        except Exception as e:
            print(e)

directory_path = input('Enter the directory path from which you want to convert the files recursively ')
rename_files_recursively(directory_path)
```