# *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.

# 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.

# Using Special Characters

- *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_]
- *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* → \ -

- *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.

- **Using Parentheses in Regular Expressions**
- *Special Character* → (….)

# Regular Expression Methods

- **Compiling Regular Expressions Using *compile()* Method of *re* Module**
- ***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.

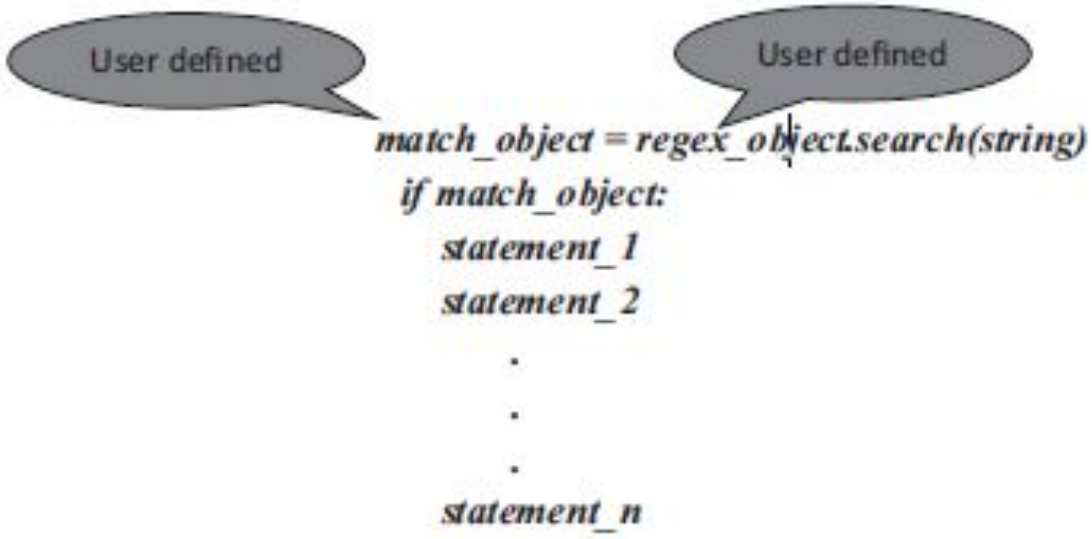
Methods Supported by Compiled Regular Expression Objects

| 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

- **Match Objects**
- The *match()* and *search()* methods supported by a compiled regular expression object, returns *None* if no match is found.



- 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. |

# 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.

# 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('abababababab')
>>> match_object.span()
(0, 10)
>>> match_object.start()
0
>>> match_object.end()
10
```

# Example

```
>>> 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]}")

# Example

```
>>> 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:

1. Contain at least 1 letter between a and z
2. Contain at least 1 number between 0 and 9
3. Contain at least 1 letter between A and Z
4. Contain at least 1 character from $, #, @
5. Minimum length of password: 6
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.

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.

```
>>> 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'
```

# Regular Expression with *glob* Module

- The glob module finds all the file names matching a specified pattern.

The syntax for glob method is,

***glob.glob(pathname, **, recursive=True)***

## 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)
```

# Summary

- The term “regular expressions” is also called regexes or regex patterns.
- The module re has to be imported to be able to work with regular expressions.
- 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.