Module No. 3, Unit No. 3.2

Regular Expressions

What are regular expressions?

- Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the ***re*** module.
- Using this little language, you specify the rules for the set of possible strings that you want to match.
- This set of strings might contain English sentences, or e-mail addresses, or TeX commands, etc.
- You can then ask questions such as “Does this string match the pattern?”, or “Is there a match for the pattern anywhere in this string?”.

Simple Patterns : Matching Characters

- Most letters and characters will simply match themselves.
- For example, the regular expression **test** will match the string **test** exactly.
- There are exceptions to this rule; some characters are special **metacharacters**, and don’t match themselves.
- Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning.
- Here’s a complete list of the metacharacters:

. ^ $ * + ? { } [ ] \ | ( )

Square Brackets [ ]

- They’re used for specifying a character class, which is a set of characters that you wish to match.
- Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a hyphen (-)
- For example, [abc] will match any of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters.
- If you wanted to match only lowercase letters, your RE would be [a-z].

Square Brackets [ ] contd.

- Metacharacters (except \) are not active inside classes.
- For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; '
- $' is usually a metacharacter, but inside a character class it’s stripped of its special nature.

Caret ^

- You can match the characters not listed within the class by complementing the set.
- This is indicated by including a '^' as the first character of the class.
- For example, **[^5]** will match any character **except '5'**.
- If the caret appears elsewhere in a character class, it does not have special meaning.
- For example: **[5^]** will match either a **'5'** or a **'^'**.

Backslash \

- Perhaps the most important metacharacter is the backslash, \.
- As in Python string literals, the backslash can be followed by various characters to signal various special sequences.
- It’s also used to escape all the metacharacters so you can still match them in patterns.
- For example, if you need to match a **[** or ****, you can precede them with a backslash to remove their special meaning: **\[** or ****.

List of Special Sequences created with backslash \

Symbol	Function
\b	Specified characters are at the beginning or at the end of a word
\B	This is the opposite of \b, only matching when the current position is not at a word boundary
\d	Any decimal digit (equivalent to [0-9])
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\\n\\r\\f\\v])
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])
\t	The tab character
\n	The newline character
\A	Matches only at the start of the string, even in multiline mode
\Z	Matches only at the end of the string

Asterisk (*)

- * doesn’t match the literal character '*'; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.
- For example, **‘ca*t’** will match **'ct'** (0 'a' characters), **'cat'** (1 'a'), **'caaat'** (3 'a' characters), and so forth.
- Repetitions such as * are greedy; when repeating a RE, the matching engine will try to repeat it as many times as possible.
- If later portions of the pattern don’t match, the matching engine will then back up and try again with fewer repetitions.

Example

- Let’s consider the expression **a[bcd]*b**.
- This matches the letter **'a'**, zero or more letters from the class **[bcd]**, and finally ends with a **'b'**.
- Now we will match this RE against the string **'abcbd'**.

Plus (+)

- Another repeating metacharacter is +, which matches one or more times.
- Pay careful attention to the difference between * and +
- * matches zero or more times, so whatever's being repeated may not be present at all, while + requires at least one occurrence.
- To use a similar example, **ca+t** will match **'cat'**, **'caaat'**, but won't match **'ct'**.

Question mark (?)

- The question mark character, ?, matches either once or zero times.
- You can think of it as marking something as being optional.
- For example, the expression **home-?brew** matches either **'homebrew'** or **'home-brew'**.

{m,n}

- m and n are decimal integers.
- This quantifier means there must be **at least m** repetitions, and **at most n**.
- For example, **a/{1,3}b** will match **'a/b'**, **'a//b'**, and **'a///b'**.
- It won't match **'ab'**, which has no slashes, or **'a////b'**, which has four.
- Omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity.
- {0,} is the same as *, {1,} is equivalent to +, and {0,1} is the same as ?. However it's better to use *, +, or ? when you can, simply because they're shorter and easier to read.

Dot .

Dot(.) symbol matches only a single character except for the newline character (\n). For example –

- **a.b** will result in a match for the strings that contains any character at the place of the dot such as **acb, acbd, abbb**, etc
- It will not result in a match in case of the string **ab**

Dollar $

Dollar($) symbol matches the end of the string i.e checks whether the string ends with the given character(s) or not. For example –

- **s$** will match with the strings that end with **s** such as **apples**, **ends**, **s**, etc.
- **es$** will match with the strings that end with **es** such as **apples**,**clothes**, etc.
- **es$** will not match with **dress**

Parentheses ( ) , OR |

- While [] denotes a character class, () denotes a capturing group.
- The pattern **[a-z0-9]** matches **one** character in the string that is in the range of **a-z OR 0-9**
- The pattern **(a-z0-9)** matches the exact substring **a-z0-9** in the string and not the ranges.
- The pattern **(ed$|ing$)** finds the words ending in **‘ed’** OR **‘ing’**

Regular expression operators

Operator	Behavior
.	Wildcard, matches any character
^abc	Matches some pattern abc at the start of a string
abc$	Matches some pattern abc at the end of a string
[abc]	Matches one of a set of characters
[^abc]	Matches any character NOT in the set of characters
[A-Z0-9]	Matches one of a range of characters
ed|ing|s	Matches one of the specified strings (disjunction)

Regular expression operators

Operator	Behavior
*	Zero or more of previous item, e.g. a*, [a-z]*
+	One or more of previous item, e.g. a+, [a-z]+
?	Zero or one of the previous item (i.e. optional), e.g. a?, [a-z]?
{n}	Exactly n repeats where n is an integer
{n,}	At least n repeats
{,n}	No more than n repeats
{m,n}	At least m and no more than n repeats
a(b|c)+	One or more occurrences of b or c immediately after a

Compiling Regular Expressions and Performing Matches

- Regular expressions are **compiled** into **pattern objects**, which have methods for searching for pattern matches

Method	Purpose
match()	Determine if the RE matches at the beginning of the string.
search()	Scan through a string, looking for any location where this RE matches.
findall()	Find all substrings where the RE matches, and returns them as a list.
finditer()	Find all substrings where the RE matches, and returns them as an iterator.

- **The findall() Function**
- The findall() function returns a list containing all matches.

- import re

  ```
  txt = "The rain in Spain"
  x = re.findall("ai", txt)
  print(x)
  ```

- The list contains the matches in the order they are found.

- If no matches are found, an empty list is returned

- **The search() Function**
- The search() function searches the string for a match, and returns a Match object if there is a match.

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

- import re

  ```
  txt = "The rain in Spain"
  x = re.search("Portugal", txt)
  print(x)
  ```

- **The split() Function**
- The split() function returns a list where the string has been split at each match:
- import re
  txt = "The rain in Spain"
  x = re.split("\s", txt)
  print(x)
- You can control the number of occurrences by specifying the maxsplit parameter:
- import re
  txt = "The rain in Spain"
  x = re.split("\s", txt, 1)
  print(x)

- **The sub() Function**
- The sub() function replaces the matches with the text of your choice:
- Replace every white-space character with the number 9:
- import re
  txt = "The rain in Spain"
  x = re.sub("\s", "9", txt)
  print(x)
- You can control the number of replacements by specifying
  the count parameter:
- Example:Replace the first 2 occurrences:
- import re
  txt = "The rain in Spain"
  x = re.sub("\s", "9", txt, 2)
  print(x)

Performing Matches

Now you can query the match object for information about the matching string. Match object instances also have several methods and attributes; the most important ones are:

Method	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

```python
import re

s = 'K J Somaiya College of Engineering'

match = re.search(r'College', s)

print('Start Index:', match.start())
print('End Index:', match.end())
```

```
Start Index: 12
End Index: 19
```

- import re

  txt = "The rain in Spain"
  x = re.search(r"\bS\w+", txt)
  print(**x.group()**)

Raw Strings

- Python strings become raw strings when they are prefixed with r or R, such as r'...' and R'...'.
- Raw strings treat backslashes as literal characters.

```
str = "This is a \n normal string example"
print(str)
raw_str = r"This is a \n raw string example"
print(raw_str)
```

```
This is a
normal string example
This is a \n raw string example
```