UNIT-3 Regular Expression

Instructor: Prof.Snehal Sathwara, Prof.Sweta Patel

Class: 4CE/IT

Regular expressions are a powerful tool for various kinds of string manipulation.

They are a domain specific language (DSL) that is present as a library in most modern programming languages, not just Python.

They are useful for two main tasks:

- verifying that strings match a pattern (for instance, that a string has the format of an email address),
- performing substitutions in a string (such as changing all American spellings to British ones).

Domain specific languages are highly specialized mini programming languages.

Regular expressions are a popular example, and SQL (for database manipulation) is another.

Private domain-specific languages are often used for specific industrial purposes.

In [ ]:

```
```

QUESTION

Which of the following tasks CANNOT be performed using regular expressions?

A) Checking whether an email address is of the correct format

B) Changing the URL part of an email address

C) Checking whether an email address is real

In [ ]:

```
```

Regular expressions in Python can be accessed using the re module, which is part of the standard library.

After you've defined a regular expression, the re.match function can be used to determine whether it matches at the beginning of a string.

If it does, match returns an object representing the match, if not, it returns None.

To avoid any confusion while working with regular expressions, we would use raw strings as r"expression".

Raw strings don't escape anything, which makes use of regular expressions easier.

In [ ]:

```
```

In [ ]:

```
#Example

import re # Importing Regular Expression Module

pattern = r"SPAM"

if re.match(pattern,"SPAMspamspamspamspamspam"):
  print('Match')
else:
  print("No Match")
```

In [ ]:

```
```

The above example checks if the pattern "spam" matches the string and prints "Match" if it does.

QUESTION

Which of these patterns would not re.match the string "spamspamspam"?

A) spamspam

B) pamspam

C) sp

In [ ]:

```
```

Other functions to match patterns are re.search and re.findall.

The function re.search finds a match of a pattern anywhere in the string.

The function re.findall returns a list of all substrings that match a pattern.

In [ ]:

```
```

In [ ]:

```
# Example
import re

pattern = r"Sneh"

if re.match(pattern, "HelloIamSnehalSathwara"):
  print('Match by re.match')
else:
  print('No Match by re.match')

if re.search(pattern,"HelloIamSnehalSathwara"):
  print('Match by re.search')
else:
  print('No Match by re.search')

print(re.findall(pattern,"HelloIamSnehalSathwaraSnehalisahandsomeboy"))
```

In the example above, the match function did not match the pattern, as it looks at the beginning of the string.

The search function found a match in the string.

The function re.finditer does the same thing as re.findall, except it returns an iterator, rather than a list.

In [ ]:

```

```

QUESTION

Which of these is not a function in the re module?

A) findlist

B) search

C) findall

In [ ]:

```

```

The regex search returns an object with several methods that give details about it.

These methods include group which returns the string matched, start and end which return the start and ending positions of the first match, and span which returns the start and end positions of the first match as a tuple.

In [ ]:

```
#Example
import re

pattern = r"sau"

match=re.search(pattern,"eggspamsausage")

if match:
  print(match.group())
  print(match.start())
  print(match.end())
  print(match.span())
```

In [ ]:

```
# Task Example:

#Helloeveryonehowareyou -----> one  ----> group,start,end,span,findall(repeat)

value=input('Enter a value: ')
match_value=input('Enter value which you match: ')
import re

pattern=r'{}'.format(match_value)

if(re.match(pattern,value)):
  print('Match')
else:
  print('Not Match')
```

Search & Replace

One of the most important re methods that use regular expressions is sub.

Syntax:

re.sub(pattern, repl, string, count=0)

In [ ]:

```

```

This method replaces all occurrences of the pattern in string with repl, substituting all occurrences, unless count provided.

This method returns the modified string.

In [ ]:

```
#Example

import re
str="There is Snehal  and Snehal loves to play MUSICAL INTRUMENTS"
pattern=r"Snehal"
newstr = re.sub(pattern,"Shruti",str)
print(newstr)
```

In [ ]:

```

```

Metacharacters

Metacharacters are what make regular expressions more powerful than normal string methods.

They allow you to create regular expressions to represent concepts like "one or more repetitions of a vowel".

The existence of metacharacters poses a problem if you want to create a regular expression (or regex) that matches a literal metacharacter, such as "$".

You can do this by escaping the metacharacters by putting a backslash in front of them.

However, this can cause problems, since backslashes also have an escaping function in normal Python strings.

This can mean putting three or four backslashes in a row to do all the escaping.

To avoid this, you can use a raw string, which is a normal string with an "r" in front of it. We saw usage of raw strings in the previous lesson.

The first metacharacter we will look at is . (dot).

This matches any character, other than a new line.

In [ ]:

```
# Example

import re

pattern=r"E...."

if re.match(pattern,'English'):
  print('Match 1')

if re.match(pattern,'England'):
  print('Match 2')

if re.match(pattern,'Japanese'):
  print('Match 3')
```

In [ ]:

```
```

The next two metacharacters are ^ and $.

These match the start and end of a string, respectively.

In [ ]:

```
#Example

import re

pattern=r"^fr.nch$"

if re.match(pattern,'french'):
  print('Match 1')

if re.match(pattern,'franch'):
  print('Match 2')

if re.match(pattern,'fraaaaanch'):
  print('Match 3')
```

In [ ]:

```
#Example -- Test Case-1

import re

pattern=r"^fr[a-zA-Z0-9]{1,}nch$"

if re.match(pattern,'french'):
  print('Match 1')

if re.match(pattern,'frznch'):
  print('Match 2')

if re.match(pattern,'fraaBCa09584651651aanch'):
  print('Match 3')
```

In [ ]:

```
#Example -- Test Case-2

import re

pattern=r"^fr[a-z]{1,}nch$"

if re.match(pattern,'french'):
  print('Match 1')

if re.match(pattern,'frznch'):
  print('Match 2')

if re.match(pattern,'fraaaaanch'):
  print('Match 3')
```

The pattern "^gr.y$" means that the string should start with gr, then follow with any character, except a newline, and end with y.

In [ ]:

```

```

Character Classes

Character classes provide a way to match only one of a specific set of characters.

A character class is created by putting the characters it matches inside square brackets.

In [ ]:

```

```

Character classes can also match ranges of characters.

Some examples:

The class [a-z] matches any lowercase alphabetic character.

The class [G-P] matches any uppercase character from G to P.

The class [0-9] matches any digit.

Multiple ranges can be included in one class. For example, [A-Za-z] matches a letter of any case.

In [ ]:

```
# Example
import re

pattern=r"[A-Za-z0-9][A-Za-z0-9][A-Za-z0-9]"

if re.search(pattern,"Sn0"):
  print('Match 1')

if re.search(pattern,"SN@"):
  print('Match 2')

if re.search(pattern,"sn0"):
  print('Match 3')
```

The pattern in the example above matches strings that contain two alphabetic uppercase letters followed by a digit.

In [ ]:

```

```

Place a ^ at the start of a character class to invert it.

This causes it to match any character other than the ones included.

Other metacharacters such as $ and ., have no meaning within character classes.

The metacharacter ^ has no meaning unless it is the first character in a class.

In [ ]:

```
#Example

import re

pattern=r"[^A-Z]"

if re.search(pattern,"vipul0"):
  print('Match 1')

if re.search(pattern,"viPul"):
  print('Match 2')

if re.search(pattern,"VIPUL0"):
  print('Match 3')
```

In [ ]:

```
#Example

import re

pattern=r"[^0-9]"

if re.search(pattern,"123"):
  print('Match 1')

if re.search(pattern,"hitesh"):
  print('Match 2')

if re.search(pattern,"HITESH"):
  print('Match 3')
```

In [ ]:

```

```

The pattern [^A-Z] excludes uppercase strings.

Note, that the ^ should be inside the brackets to invert the character class.

In [ ]:

```

```

Some more metacharacters are * + ? { and }.

These specify numbers of repetitions.

The metacharacter * means "zero or more repetitions of the previous thing".

It tries to match as many repetitions as possible.

The "previous thing" can be a single character, a class, or a group of characters in parentheses.

In [ ]:

```
# Example
import re

pattern = r"egg(spam)*"

if re.match(pattern,"egg"):
  print('Match 1')

if re.match(pattern,"eggspamspamegg"):
  print('Match 2')

if re.match(pattern,"spam"):
  print('Match 3')
```

In [ ]:

```
# Example  -- Test Case
import re

pattern = r"spam(egg)*"

if re.match(pattern,"egg"):
  print('Match 1')

if re.match(pattern,"spameggspamspamegg"):
  print('Match 2')

if re.match(pattern,"spam"):
  print('Match 3')
```

The example above matches strings that start with "egg" and follow with zero or more "spam"s.

In [ ]:

```

```

The metacharacter + is very similar to *, except it means "one or more repetitions", as opposed to "zero or more repetitions".

In [ ]:

```
#Example
import re

pattern = r"g+"

if re.match(pattern,"g"):
  print('Match 1')

if re.match(pattern,"ggggggggggg"):
  print('Match 2')

if re.match(pattern,"abc"):
  print('Match 3')
```

To summarize:

- matches 0 or more occurrences of the preceding expression.
- matches 1 or more occurrence of the preceding expression.

In [ ]:

```

```

The metacharacter ? means "zero or one repetitions".

In [ ]:

```
#Example
import re

pattern = r"ice(-)?cream"

if re.match(pattern,"ice-cream"):
  print('Match 1')

if re.match(pattern,"icecream"):
  print('Match 2')

if re.match(pattern,"sausages"):
  print('Match 3')

if re.match(pattern,"ice--ice"):
  print('Match 4')
```

Curly braces can be used to represent the number of repetitions between two numbers.

The regex {x,y} means "between x and y repetitions of something".

Hence {0,1} is the same thing as ?.

If the first number is missing, it is taken to be zero. If the second number is missing, it is taken to be infinity.

In [ ]:

```
#Example
import re

pattern = r"9{1,3}$"

if re.match(pattern,"9"):
  print('Match 1')

if re.match(pattern,"999"):
  print('Match 2')

if re.match(pattern,"9999"):
  print('Match 3')
```

"9{1,3}$" matches string that have 1 to 3 nines.

In [ ]:

```

```

Groups

A group can be created by surrounding part of a regular expression with parentheses.

This means that a group can be given as an argument to metacharacters such as * and ?.

In [ ]:

```
# Example

import re

pattern = r"egg(spam)*"

if re.match(pattern,"egg"):
  print('Match 1')

if re.match(pattern,"eggspamspamegg"):
  print('Match 2')

if re.match(pattern,"spam"):
  print('Match 3')
```

(spam) represents a group in the example pattern shown above.

In [ ]:

```

```

The content of groups in a match can be accessed using the group function.

A call of group(0) or group() returns the whole match.

A call of group(n), where n is greater than 0, returns the nth group from the left.

The method groups() returns all groups up from 1.

In [ ]:

```
#Example
import re
pattern = r"a(bc)(de)(f(g)h)i"

match = re.match(pattern, "abcdefghijklmnop")

if match:
  print(match.group())
  print(match.group(0))
  print(match.group(1))
  print(match.group(2))
  print(match.groups())
```

As you can see from the example above, groups can be nested.

In [ ]:

```

```

There are several kinds of special groups.

Two useful ones are named groups and non-capturing groups.

Named groups have the format (?P...), where name is the name of the group, and ... is the content. They behave exactly the same as normal groups, except they can be accessed by group(name) in addition to its number.

Non-capturing groups have the format (?:...).

They are not accessible by the group method, so they can be added to an existing regular expression without breaking the numbering.

In [ ]:

```
# Example

import re

pattern = r"(?P<first>abc)(?:def)(ghi)"

match = re.match(pattern,"abcdefghi")

if match:
  print(match.group('first'))
  print(match.groups())
```

In [ ]:

```

```

Matacharacter

Another important metacharacter is |.

This means "or", so red|blue matches either "red" or "blue".

In [ ]:

```
#Example

import re

pattern = r"gr(a|e)y"

match = re.match(pattern,"gray")
if match:
  print('Match 1')

match = re.match(pattern,"grey")
if match:
  print('Match 2')

match = re.match(pattern,"griy")
if match:
  print('Match 3')
```

Special Sequences

There are various special sequences you can use in regular expressions.

They are written as a backslash followed by another character.

One useful special sequence is a backslash and a number between 1 and 99, e.g., \1 or \17.

This matches the expression of the group of that number.

In [ ]:

```
import re

pattern = r"(.+) \1"

match = re.match(pattern,"word word")
if match:
  print('Match 1')

match = re.match(pattern,"?! ?!")
if match:
  print('Match 2')

match = re.match(pattern,"abc cde")
if match:
  print('Match 3')
```

Note, that "(.+) \1" is not the same as "(.+) (.+)", because \1 refers to the first group's subexpression, which is the matched expression itself, and not the regex pattern.

In [ ]:

```

```

More useful special sequences are \d, \s, and \w.

These match digits, whitespace, and word characters respectively.

In ASCII mode they are equivalent to [0-9], [\t\n\r\f\v], and [a-zA-Z0-9_].

In Unicode mode they match certain other characters, as well. For instance, \w matches letters with accents.

Versions of these special sequences with upper case letters - \D, \S, and \W - mean the opposite to the lower-case versions. For instance, \D matches anything that isn't a digit.

In [ ]:

```
#Example

import re

pattern = r"(\D+\d)"

match = re.match(pattern,"Hi 999!")
if match:
  print('Match 1')

match = re.match(pattern,"1,23,456!")
if match:
  print('Match 2')

match = re.match(pattern,"! $?")
if match:
  print('Match 3')
```

(\D+\d) matches one or more non-digits followed by a digit.

In [ ]:

```

```

Additional special sequences are \A, \Z, and \b.

The sequences \A and \Z match the beginning and end of a string, respectively.

The sequence \b matches the empty string between \w and \W characters, or \w characters and the beginning or end of the string.

Informally, it represents the boundary between words.

The sequence \B matches the empty string anywhere else.

In [ ]:

```
#Example

import re

pattern = r"\b(cat)\b"

match = re.search(pattern,"The cat sat!")
if match:
  print('Match 1')

match = re.search(pattern,"We s>cat<tered?")
if match:
  print('Match 2')

match = re.match(pattern,"We scattered")
if match:
  print('Match 3')
```

\b(cat)\b" basically matches the word "cat" surrounded by word boundaries.

END OF .ipynb