(.*) Regular Expressions

Regular Expressions

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.

Regular Expressions

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.

Example:

```
import re

pattern = r"spam"

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

Try It Yourself

Result:

```
>>>
Match
>>>
```

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

> Here the pattern is a simple word, but there are various characters, which would have special meaning when they are used in a regular expression.

Regular Expressions

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.

Example:

```
import re

pattern = r"spam"

if re.match(pattern, "eggspamsausagespam"):
  print("Match")
else:
  print("No match")

if re.search(pattern, "eggspamsausagespam"):
  print("Match")
else:
  print("No match")

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

Try It Yourself

Result:

```
>>>
No match
Match
['spam', 'spam']
>>>
```

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.

Regular Expressions

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.
Example:

```
import re

pattern = r"pam"

match = re.search(pattern, "eggspamsausage")
if match:
  print(match.group())
  print(match.start())
  print(match.end())
  print(match.span())
```

Try It Yourself

Result:

```
>>>
pam
4
7
(4, 7)
>>>
```

Tap **Try It Yourself** to play around with the code!

Search & Replace

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

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

This method replaces all occurrences of the **pattern** in string with **repl**, substituting all occurrences, unless **count** provided. This method returns the modified string.
Example:

```
import re

str = "My name is David. Hi David."
pattern = r"David"
newstr = re.sub(pattern, "Amy", str)
print(newstr)
```

Try It Yourself

Result:

```
>>>
My name is Amy. Hi Amy.
>>>
```

Tap **Try It Yourself** to play around with the code!

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.

Metacharacters

The first metacharacter we will look at is . (dot).
This matches **any character**, other than a new line.
Example:

```
import re

pattern = r"gr.y"

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

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

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

Try It Yourself

Result:

```
>>>
Match 1
Match 2
>>>
```

Tap **Try It Yourself** to play around with the code!

Metacharacters

The next two metacharacters are ^ and $.
These match the **start** and **end** of a string, respectively.
Example:

```
import re

pattern = r"^gr.y$"

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

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

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

Try It Yourself

Result:

```
>>>
Match 1
Match 2
>>>
```

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

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 ?.
Example:

```
import re

pattern = r"egg(spam)*"

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

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

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

Try It Yourself

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

```
>>>
Match 1
Match 2
>>>
```

Tap **Try It Yourself** to play around with the code!

Groups

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 **n**th group from the left.
The method **groups()** returns all groups up from 1.
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())
```

Try It Yourself

Result:

```
>>>
abcdefghi
abcdefghi
bc
de
('bc', 'de', 'fgh', 'g')
>>>
```

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

Groups

There are several kinds of special groups.
Two useful ones are **named groups** and **non-capturing groups**.
Named groups have the format **(?P<name>...)**, 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.
Example:

```
import re

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

match = re.match(pattern, "abcdefghi")
if match:
  print(match.group("first"))
  print(match.groups())
```

Try It Yourself

Result:

```
>>>
abc
('abc', 'ghi')
>>>
```

Tap **Try It Yourself** to play around with the code!

Metacharacters

Another important metacharacter is |.
This means "or", so **red|blue** matches either "red" or "blue".
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")
```

Try It Yourself

Result:

```
>>>
Match 1
Match 2
>>>
```

Tap **Try It Yourself** to play around with the code!

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.
Example:

```
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")
```

Try It Yourself

Result:

```
>>>
Match 1
Match 2
>>>
```

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.

Special Sequences

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.
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")
```

Try It Yourself

Result:

```
>>>
Match 1
>>>
```

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

Special Sequences

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.
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.search(pattern, "We scattered.")
if match:
  print ("Match 3")
```

Try It Yourself

Result:

```
>>>
Match 1
Match 2
>>>
```

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

Email Extraction

To demonstrate a sample usage of regular expressions, lets create a program to extract email addresses from a string.
Suppose we have a text that contains an email address:

```
str = "Please contact info@sololearn.com for assistance"
```

Our goal is to extract the substring "info@sololearn.com".
A basic email address consists of a word and may include dots or dashes. This is followed by the @ sign and the domain name (the name, a dot, and the domain name suffix).
This is the basis for building our regular expression.

```
pattern = r"([\w\.-]+)@([\w\.-]+)(\.[\w\.]+)"
```

[\w\.-]+ matches one or more word character, dot or dash.
The regex above says that the string should contain a word (with dots and dashes allowed), followed by the @ sign, then another similar word, then a dot and another word.

Our regex contains three groups:
1 - first part of the email address.
2 - domain name without the suffix.
3 - the domain suffix.

Email Extraction

Putting it all together:

```
import re

pattern = r"([\w\.-]+)@([\w\.-]+)(\.[\w\.]+)"
str = "Please contact info@sololearn.com for assistance"

match = re.search(pattern, str)
if match:
  print(match.group())
```

Try It Yourself

Result:

```
>>>
info@sololearn.com
>>>
```

In case the string contains multiple email addresses, we could use the **re.findall** method instead of **re.search**, to extract all email addresses.

> The regex in this example is for demonstration purposes only.
> A much more complex regex is required to fully validate an email address.

End.