Regular Expressions

Introduction to regular expressions

A regular expression (or regex in short) is a string that defines a search pattern. Regular expressions can be used to search for specific patterns within a given text. Regular expressions have been used for well over 50 years, with the American pioneer of Computer Science, Kenneth Lane Thompson, creating the foundational notations that many variants of regular expressions still use today.

Regular expression patterns are typically agnostic of programming languages and operating systems. Unix-based systems have made use of the `grep` (**g**lobally search for a **r**egular **e**xpression and **p**rint matches) command line utility function to search for one of more regex patterns in a given set of files and provide the results where these patterns are successfully matched in any of the input files.

Use of regular expressions

Regular expression patterns are available for use in a wide range of tools and languages. For example, Microsoft Word allows us to search using simple wildcards in a word document. Certain patterns can also be used in command line utilities (e.g. PowerShell in Microsoft Windows or Terminal in MacOS) to search and list a set of files matching the wildcard pattern.

For example, in Windows PowerShell,

- `ls *.txt` will list all the files with the extension 'txt' in the current folder (or no output if no matches are found)
- `ls t*` will list all files and folders beginning with the letter 't' in the current folder (or no output if no matches are found)

Wildcards, while strictly not the same as regular expressions provide us a flavour of how they work. More powerful regex patterns can be used in a range of programming languages such as Python, SQL, Java or Perl. For example, Python has a built-in module **re** for handling regular expressions. These regex patterns can be used to search through a body of text for a variety of purposes - for example, when validating a particular field of data or when trying to apply data enrichment using find and replace methods. The core patterns in regular expressions are universal, however, different tools have their own regular expression engines to process regex patterns and these engines can have different behaviours and levels of support for more complex patterns. These variations are known as regex flavours.

Regex patterns

Regex patterns are used to describe the target sequence to be matched. They can be formed by combining a number of different components (as detailed below). Regex patterns are matched from left to right and the leftmost match is always returned.

Literal characters

These include precise matches to the characters. A few examples are given in the table below:

Pattern	Description
t	Matches the letter 't'
toast	Matches the string 'toast'
4	Matches the number 4
123	Matches the number 123

Several characters can be matched at the same time by placing them within square brackets. This also works for a range of characters. Negation can be performed by including a ^ (caret) symbol at the beginning (within the square brackets).

Pattern	Description
[4-8]	Matches any number from 4 to 8
[^aeiou]	Matches any character that is not a lowercase vowel

Character sets

These are special patterns that match a set of characters. A few examples are given in the table below:

Pattern	Description	Notes
`\d`	Matches any character that is a decimal digit (Typically **0-9** but may also include unicode digits from other languages)	Same as `[\d]`
`\D`	Matches any character that is **not** a decimal digit	Same as `[^\d]`
`\w`	Matches any alphanumeric character and underscore (Typically **A-Z, a-z, 0-9,** _ but may also include word characters from other languages)	Same as `[\w]`
`\W`	Matches any character that is not alphanumeric or underscore	Same as `[^\w]`
`\s`	Matches a whitespace character (could also include tabs, newlines etc. depending on the regex flavour)	Same as `[\s]`
`\S`	Matches anything but a whitespace character	Same as `[^\s]`

Several flavours allow restriction matching to ASCII encoding. In this case, `\d` is the same as `[0-9]` and `\w` is the same as `[A-Za-z0-9_]`.

Metacharacters

These are special characters that need to be escaped (preceded) with a backslash in order to be matched literally. Some examples are **+ (plus), ^ (caret), ? (question mark), $ (dollar), * (asterisk), . (dot)**. These metacharacters have a special meaning when used without escaping as given in the table below. These characters can also be used along with multiple characters

(by placing after the character set or after the square brackets above). When placed inside the square brackets, the metacharacters are matched precisely without the need for escaping.

Pattern	Description	Example
`?`	Matches the preceding element zero or one time	`colou?r` matches 'color' or 'colour' `https?` matches 'http' or 'https'
`*`	Matches the preceding element zero or more times	`co*p` matches 'cp','cop','coop',... `12\*3` matches '13','123','1223',...
`+`	Matches the preceding element one or more times	`a+` matches 'a','aa','aaa',... `ab+` matches 'ab','abb','abbb',...
`.`	Matches any character	`l.st` matches 'last','lost','lest','l6st','l%st',... `..t` matches 'cat','bat','cut','ant','net',...
`{min,max}`	Denotes the minimum and maximum matches for the preceding element Note: `max` is optional. `{min, }` matches at least 'min' times and `{min}` matches exactly 'min' times	`1{2,3}` matches '11' or '111' `co{1,2}p` matches 'cop' or 'coop'
`^`	Matches the beginning of a string	`^The` matches the pattern for a string beginning with 'The'
`$`	Matches the end of a string	`$done` matches the pattern for a string ending with 'done'
`|`	Used to separate multiple patterns, any of which can be matched	`a|an` matches 'a' or 'an'
`()`	Used to group patterns together	ab+ matches 'ab', 'abb', 'abbb', ... (ab)+ matches 'ab', 'abab', 'ababab', ...

The characters `*`, `+`, `?`, `{min,max}` are called repetition qualifiers. `^` and `$` are anchors that match the position at the beginning or the end of the document (Note the operator overloading for ^).

Regular Expressions in Python

Python has a built-in module `re` for handling regular expressions. The following section provides a basic introduction to using this module for compiling and matching regular expressions in Python.

Importing the module

```
import re
```

Basic workflow

The basic workflow for using regex in Python involves two steps:

- Compiling the regex pattern
- Performing matches against a test string

Compiling a regex pattern

Regular expression patterns can be passed as a raw string to `re.compile(...)`. This allows us to escape the special characters inside the string before converting it into a raw string format.

```
pattern = re.compile(r'gr[a|e]y')
```

Performing matches against a test string

The compiled regex pattern can then be used to find matches in a test string in a few different ways.

1. Using `pattern.findall('test string...')` - This returns all matches as a list.
2. Using `pattern.finditer('test string...')` - This returns all matches as an iterator.
3. Using `pattern.search('test string...')` - This searches the string and returns the first match found else `None`.

`pattern.findall(...)`

```
my_str = '''Gray and grey are both valid spellings for the intermediate c
While gray is widely used in the US, grey is more common in the UK.'''
matches = pattern.findall(my_str)
print(matches)

Out: ['grey', 'gray', 'grey']
```

Note: The first word in the string did not match because it starts with an uppercase 'G'.

`pattern.finditer(...)`

```
my_str2 = 'The racing greyhound dogs, however, are usually spelt with gre
matches = pattern.finditer(my_str2)
```

This returns an iterator called `matches`. Each item in this iterator is a match object.

```
m1 = next(matches)
print(m1)

Out: <re.Match object; span=(11, 15), match='grey'>
```

The span is a tuple showing the start and end position of the match. These values can also be obtained using `m1.start()` and `m1.end()` respectively.

pattern.search(...)

```
matches = pattern.search(my_str) # Using the original string again
print(matches)

Out: <re.Match object; span=(9, 13), match='grey'>
```

This matches the third word 'grey' in `my_str`.

For more information on the `re` module in Python, see https://docs.python.org/3/library/re.html

Example: UK Postcode format validation

Let's consider an example where we validate the format of a UK postcode entered by an user in a form (Note that we are only validating the format of the postcode and not verifying if it is a genuine postcode in this example).

Postcodes are alphanumeric (consisting on letters A-Z or a-z and number 0-9) ranging from 5 to 7 characters with an optional space in between the two sections. Some examples of UK postcodes (with the corresponding format in letters(`l`) and numbers(`n`)) are:

Postcode	Format
L4 0TH	ln nll
M16 0RA	lnn nll
SE1 0BE	lln nll
W1J 9HP	lnl nll
SW1A 1AA	llnl nll
BT57 8SU	llnn nll

Let's break down the different variations to construct our pattern.

1. The first part of the postcode can be broken down into three components:
    - One or two mandatory letters `[A-Za-z]{1,2}`
    - One mandatory number `[0-9]`
    - One optional letter or number `[A-Za-z0-9]?`
2. This is followed by an optional space which can be matched using a space character followed a question mark `" ?"` (quotes given to highlight space character)
3. The second part of the postcode follow the same pattern (nll) which can be matched using `[0-9][A-Za-z]{2}`

Putting all these together between `^` and `$` to match the entire string, we get `^[A-Za-z]{1,2}[0-9][A-Za-z0-9]? ?[0-9][A-Za-z]{2}$`

```
postcode = input('Enter a postcode: ')
pattern = re.compile(r'^[A-Za-z]{1,2}[0-9][A-Za-z0-9]? ?[0-9][A-Za-z]{2}$
```

```
match = pattern.search(postcode)
if not match:
    print('Not a valid postcode format!')
else:
    print('Valid postcode format!')
```

Concept Check: UK vehicle registration number (number plate) format validation

The current UK number plate format consists of:

- Two letters (area code)
- Two numbers (age identifier)
- A mandatory space
- Three letters (random sequence)

For example, `BD51 SMR` is a valid UK number plate.

Construct a regex pattern to validate the format of a given UK number plate.

Challenge: Incorporate the following additional rules for validating the number plate.

1. The letters I, Q and Z are not used in the area code (first two letters)
2. The letters I and Q are not used in the random sequence (last three letters)