Chapter 12 - Regular Expressions

12 Regular Expressions

Regular expressions (also known as regex) is a special sequence of characters that defines a pattern for a complex string matching algorithm. Regex is widely used in the UNIX environment. Using regex in Python requires us to import the `re` library into our workspaces as follows `import re`. In addition, the strings used to create the regex expressions can sometimes be mixed up with normal Python strings, to prevent confusion when using known escape characters like `\b`, prefix the pattern string with a `r` character to change it to a **raw string**.

In simple terms, regular expressions tries to match a patterned string with some input strings. This pattern is defined based on rules or business logic. There is no one pattern that 100% fits every type of input string that can be encountered therefore it is advisable to construct a pattern based on a set of well defined conditions and any other outliers have to be checked further by the program logic.

12.1 Syntax

Before we look at how to use the `re` library, we need to know what is the regex syntax. We will break up the syntax into different parts so that it is easier to understand their uses. The following is based off the Python 3's documentation on regex.

12.1.1 Special Characters

Character	Description
`^` (caret symbol)	Matches the expression to its right at the start of a string. If it is a multiline string, it matches every instance of the expression immediately after the newline (`\n`) character of the previous string.
`$` (dollar symbol)	Matches the expression to its left at the end of a string. If it is a multiline string, it matches every instance of the expression just before the newline (`\n`) character of the current string.
`.` (period symbol)	Matches any character except line terminators like the newline (`\n`) character.
`\` (backslash)	Escapes special characters (like `'*'` or `'?'`) so that they can be used literary.
`A|B` (pipe symbol)	Matches expression `A` or `B` where *A* and *B* can be any valid regex. If `A` is matched first, `B` is left untested.
`+` (plus symbol)	Greedily matches the expression to its left 1 or more times.
`*` (multiply symbol)	Greedily matches the expression to its left 0 or more times.
`?` (question mark)	Greedily matches the expression to its left 0 or 1 times.
`{m}`	Matches the expression to its left exactly `m` or more times, not less.
`{m,n}`	Matches the expression to its left between the range defined by `m` to `n` times repetitions, not less.
`{m,n}?`	Matches the expression to its left `m` times repeatedly and ignores `n`.

12.1.2 Character Classes (Special Sequences)

Character	Description
`\w`	Matches alphanumeric characters, which means `a-z`, `A-Z` and `0-9`. It also matches the underscore `_`.
`\W`	Matches non alphanumeric characters.
`\d`	Matches digits, which means `0-9`.
`\D`	Matches any non-digits.
`\s`	Matches whitespace characters, which include the `\t`, `\n`, `\r` and space characters.
`\S`	Matches non-whitespace characters.
`\b`	Matches the boundary (or empty string) at the start and end of a word, that is, between `\w` and `\W`.
`\B`	Matches where `\b` does not, that is, the boundary of `\w` characters.
`\A`	Matches the expression to its right at the absolute start of a string whether in single or multi-line mode.
`\Z`	Matches the expression to its left at the absolute end of a string whether in single or multi-line mode.

12.1.3 Sets

Character	Description
`[ ]`	Contains a set of characters to match.
`[amk]`	Matches either the placeholder `a`, `m` or `k` individually. It does not match `amk` as a whole.
`[a-z]`	Matches any alphabet from `a` to `z`.
`[a\-z]`	Matches `a`, `-` or `z`. It matches `-` because the backslash escape character `\` changed `-` to a literal.
`[a-]` or `[-a]`	Matches `a` or `-`, because `-` is not being used to indicate a series of characters.
`[a-z0-9]`	Matches characters from `a` to `z` and also from `0` to `9`.
`[(+*)]`	Special characters become literal inside a set, so this matches `(`, `+`, `*` and `)`.
`[^ab5]`	Adding `^` excludes any character in the set. Here, it matches characters that are not `a`, `b` or `5`.

12.1.4 Groups

Character	Description
`(...)`	Matches the expression inside the parentheses and groups it.
`(?...)`	The `?` following the `(` acts as an extension notation. Its meaning depends on the character immediately to its right.
`(?P<name>...)`	Matches the expression in the `...` section and it can be accessed via the group name in `<name>`. Each group name must be defined only once within a regex.
`(?aiLmsux)`	Here, `a, i, L, m, s, u` and `x` are flags: • `a` - Matches ASCII only (`re.A` or `re.ASCII`) • `i` - Ignore case (`re.I` or `re.IGNORECASE`) • `L` - Locale dependent (`re.L` or `re.LOCALE`) • `m` - Multi-line (`re.M` or `re.MULTILINE`) • `s` - Matches all (`re.S` or `re.DOTALL`) • `u` - Matches unicode (`re.U` or `re.UNICODE`) • `x` - Verbose (`re.X` or `re.VERBOSE`)
`(?:...)`	Matches the expression as represented by `...` but it cannot be retrieved afterwards.
`(?#...)`	A comment. Contents are for us to read, not for matching.
`A(?=B)`	Lookahead assertion. This matches the expression `A` only if it is followed by `B`.
`A(?!B)`	Negative lookahead assertion. This matches the expression `A` only if it is not followed by `B`.
`(?<=B)A`	Positive lookbehind assertion. This matches the expression `A` only if `B` is immediately to its left. This can only matched fixed length expressions.
`(?<!B)A`	Negative lookbehind assertion. This matches the expression `A` only if `B` is not immediately to its left. This can only matched fixed length expressions.
`(?P=name)`	Matches the expression matched by an earlier group named "name".
`(...)\1`	The number `1` corresponds to the first group to be matched. If we want to match more instances of the same expresion, simply use its number instead of writing out the whole expression again. We can use from `1` up to `99` such groups and their corresponding numbers.

12.1.5 Flags

Abbreviation	Full Name	Description
`re.I`	`re.IGNORECASE`	Makes the regular expression case-insensitive.
`re.L`	`re.LOCALE`	The behaviour of some special sequences like `\w`, `\W`, `\b`,`\s`, `\S` will be made dependent on the current locale, i.e. the user's language, country etc.
`re.M`	`re.MULTILINE`	`^` and `$` will match at the beginning and at the end of each line, seperated by the newline character (`\n`) and not just at the beginning and the end of the string.
`re.S`	`re.DOTALL`	The dot/period (`.`) character will match every character **plus the newline character**.
`re.A`	`re.ASCII`	Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching.
`re.X`	`re.VERBOSE`	Allowing "verbose regular expressions", i.e. whitespace are ignored. This means that regular expressions can be written in separate logical sections of the pattern and comments added. Whitespaces, tabs, carriage returns and `#` are ignored. The backslash escape character have to be used on the ignored characters if they are not defined within the square brackets.

12.2 `re` Library

Now that we have an overview of the regex syntax, we can now look at some of the more popular functions from the `re` library upon which we can apply the regex string patterns. A suggestion is to use this regex tester and debugger to help you format the strings for your regular expression.

To demonstrate the some of the functionalities of the regex functions, we will be using the list of strings shown below

```python
some_emails = ["ankitrai326@gmail.com",
               "From: ben.zulu98@bermudasquare.sg",
               "tester",
               "From: chris-thomas.wick@blueocean.edu",
               "john.wick@hotelblue80.com",
               "fivefive.com",
               "bluesky@.com",
               "example-indeed@strange-example.net"]
```

We assume that valid emails have the following characteristics:

1. a local section (before the `@`) consisting of alphanumeric characters, the period, the hyphen, the underscore, the colon and the whitespace characters.
2. a domain name section (after the `@`) that includes lower case letters, numbers, hyphen characters and a period followed by lower case letters between the length of 2 to 3 characters.

12.2.1 Functions

- `search()/match()` - these functions functionality is to scan through a the string looking for the **first occurrence** of the regex pattern. Returns the matched object if found, otherwise return `None`. Difference is that the `match()` function will start the scan from the **beginning** of the string (aka it's like having the `^` caret functionality built-in) and if the pattern is **not found** at the beginning of the string, it is deemed *not found* whereas the `search()` function scans through the whole string *looking* for the first match. In the example we are using the `search()` function to extract strings with the valid email pattern defined above from the list of strings.

  ```python
  import re
  valid_emails = []
  # to check if the strings are valid email address
  for email in some_emails:
      matched_obj = re.search('[\w.: -]+@[a-z0-9-]+\.[a-z]{2,3}', email)

      if matched_obj:
          # extracts the matched string value from the matched_obj object
          valid_emails.append(matched_obj.group())
  ```

  The regular expression conforming to those requirements would be `[\w.: -]+@[a-z0-9-]+\.[a-z]{2,3}` where:

  - `[\w.: -]+` - matches all alphanumeric characters (including underscores), the period, the hyphen , the colon and the whitespace characters as many times as possible
  - `@` - matches `@` symbol

    - `[a-z0-9-]+` - matches lower case letters, number from 0-9 and the hyphen character as many times as possible
    - `\.` - matches the period character
    - `[a-z]{2,3}` - matches a string length between 2 to 3 lower case letters.
- `findall()` - this function finds and **returns a list** of all strings matching the pattern. String are scanned from left to right and returned in the same order. If there are more than 1 groups in the pattern string, it will return a list of tuples in accordance to the groups of the pattern string. The example will be using the `valid_email` list created by the previous example to extract the strings with the `From:` at the start of the string

```
# valid_email contains: ['ankitrai326@gmail.com',
#    'From: ben.zulu98@bermudasquare.sg',
#    'From: chris-thomas.wick@blueocean.edu',
#    'john.wick@hotelblue80.com',
#    'example-indeed@strange-example.net']

ext_from = []
# extract out the ones with the 'From:'
for email in valid_emails:
    obj = re.findall("From:.*", email)

    if obj:
        ext_from.extend(obj)
```

  The regex pattern used here `From:.*` means to find all strings with starting with `From:` to the excluding the newline character.

- `finditer()` - this function returns an iterator that stores all non-overlapping matching objects from the matched pattern. The searched string is scanned left to right. Usage will be demonstrated during the lecture.
- `split()` - this function works similar to the string `split()` function from the string object but it uses a regular expression pattern instead. It returns a list of strings split in accordance to the regex pattern and the **remainder** of the string is always returned as the **last element** of the list. The example will be using the `ext_from` list from the previous example. We will split the strings based on the non alphabet and non numerical characters.

```
# ext_from contains: ['From: ben.zulu98@bermudasquare.sg',
#    'From: chris-thomas.wick@blueocean.edu']

for item in ext_from:
    sp = re.split("[:._@-]", item)
    print(sp)

# output :
# ['From', ' ben', 'zulu98', 'bermudasquare', 'com']
# ['From', ' chris', 'thomas', 'wick', 'blueocean', 'edu']
```

- `sub()` - this function works similar to the string `replace()` function from the string object but it uses a regular expression pattern instead. Returns a string. An example would be to replace all letters from `a` to `h` with `*`.

```
re.sub('[a-h]', '*', some_emails[3])
# output: 'From: **ris-t*om*s.wi*k@*lu*o***n.**u'
```

- `compile()` - this function compiles the regex pattern string into an regex object for use with any regex function. This efficiency of this function is best seen when the same regex pattern string is used repeatedly throughout a program. Calling `re` functions changes to using the regex pattern object instead.

  ```
  pattern_obj = re.compile("[:._@-]")
  result = pattern_obj.spilt(item)
  ```

12.3 References

2. Sturtz, April 2020, Regular Expressions: Regexes in Python (Part 1), https://realpython.com/regex-python/
3. Lubanovic, 2019, 'Chapter 12. Wrangle and Mangle Data' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.
4. re — Regular expression operations, https://docs.python.org/3/library/re.html
5. Klein, B, Regular Expressions, https://www.python-course.eu/python3_re.php