**What are regular expressions?**

The Regex or Regular Expression is a way to define a pattern for searching or manipulating strings. We can use a regular expression to match, search, replace, and manipulate inside textual data.

In simple words, the regex pattern Jessa will match to name Jessa.

Also, you can write a **regex pattern to validate a password** with some predefined constraints, such as the password must contain at least one special character, digit, and one upper case letter. If the pattern matches against the password, we can say that password is correctly constructed.

Also, Regular expressions are instrumental in extracting information from text such as log files, spreadsheets, or even textual documents.

For example, Below are some of the cases where regular expressions can help you to save a lot of time.

* Searching and replacing text in files
* Validating text input, such as password and email address
* Rename a hundred files at a time. For example, You can change the extension of all files using a regex pattern

**The re module**

We will start this tutorial by using the RE module, a built-in Python module that provides all the required functionality needed for handling patterns and regular expressions.

Type import re at the start of your Python file, and you are ready to use the re module's methods and special characters. To get to know the RE module's functionality, methods, and attributes, use the help function.

Just Pass the module's name as an argument to the help function like this print(help(re)) . It will show hundreds of lines simply because this module is vast and comprehensive.

Now let's how to use the re module to perform regex pattern matching in Python.

**Example 1: Write a regular expression to search digit inside a string**

Now, let's see how to use the Python re module to write the regular expression. Let's take a simple example of a regular expression to check if a string contains a number.

For this example, we will use the ( \d ) metacharacter, we will discuss regex metacharacters in detail in the later section of this article.

As of now, keep in mind that a  \d  is a special sequence that matches any digit between 0 to 9.

# import RE module

**import** re

target\_str = "My roll number is 25"

res = re.findall(r"\d", target\_str)

# extract mathing value

**print**(res)

# Output [2, 5]

**Understand this example**

1. We imported the RE module into our program
2. Next, We created a regex pattern \d to match any digit between 0 to 9.
3. After that, we used the re.findall() method to match our pattern.
4. In the end, we got two digits 2 and 5.

**Use raw string to define a regex**

**Note**: I have used a raw string to define a pattern like this r"\d". Always write your regex as a raw string.

As you may already know, the **backslash has a special meaning** in some cases because it may indicate an escape character or escape sequence. To avoid that always use a raw string.

For example, let's say that in Python we are defining a string that is actually a path to an exercise folder like this path = "c:\example\task\new".

Now, let's assume you wanted to search this path inside a target string using a regular expression. let's write code for the same.

**import** re

**print**("without raw string:")

# path\_to\_search = "c:\example\task\new"

target\_string = r"c:\example\task\new\exercises\session1"

# regex pattern

pattern = "^c:\\example\\task\\new"

# \n and \t has a special meaning in Python

# Python will treat them differently

res = re.search(pattern, target\_string)

**print**(res.group())

Notice that inside the pattern we have two escape characters \t and \n. If you execute the above code you will the re.error: bad escape error because \n and \thas a special meaning in Python.

To avoid such issues, always write a regex pattern using a raw string. **The character r denotes the raw string**.

Now replace the existing pattern with pattern = r"^c:\\example\\task\\new" and execute our code again. Now you can get the following output.

with raw string:

Matching path: ['c:\\example\\task\\new']

**Python regex methods**

The Python regex module consists of multiple methods. below is the list of regex methods and their meaning.

**Click on each method name to study it in detail**.

| **Method** | **Description** |
| --- | --- |
| re.compile('pattern') | Compile a regular expression pattern provided as a string into a re.Pattern object. |
| re.search(pattern, str) | Search for occurrences of the regex pattern inside the target string and return only the first match. |
| re.match(pattern, str) | Try to match the regex pattern at the start of the string. It returns a match only if the pattern is located at the beginning of the string. |
| re.fullmatch(pattern, str) | Match the regular expression pattern to the entire string from the first to the last character. |
| re.findall(pattern, str) | Scans the regex pattern through the entire string and returns all matches. |
| re.finditer(pattern, str) | Scans the regex pattern through the entire string and returns an iterator yielding match objects. |
| re.split(pattern, str) | It breaks a string into a list of matches as per the given regular expression pattern. |
| re.sub(pattern, replacement, str) | Replace one or more occurrences of a pattern in the string with a replacement. |
| re.subn(pattern, replacement, str) | Same as re.sub(). The difference is it will return a tuple of two elements. First, a new string after all replacement, and second the number of replacements it has made. |

Python regex methods

**Example 2: How to use regular expression in Python**

Let's see how to use all regex methods.

# import the RE module

**import** re

target\_string = "Jessa salary is 8000$"

# compile regex pattern

# pattern to match any character

str\_pattern = r"\w"

pattern = re.**compile**(str\_pattern)

# match regex pattern at start of the string

res = pattern.match(target\_string)

# match character

**print**(res.group())

# Output 'J'

# search regex pattern anywhere inside string

# pattern to search any digit

res = re.search(r"\d", target\_string)

**print**(res.group())

# Output 8

# pattern to find all digits

res = re.findall(r"\d", target\_string)

**print**(res)

# Output ['8', '0', '0', '0']

# regex to split string on whitespaces

res = re.split(r"\s", target\_string)

**print**("All tokens:", res)

# Output ['Jessa', 'salary', 'is', '8000$']

# regex for replacement

# replace space with hyphen

res = re.sub(r"\s", "-", target\_string)

# string after replacement:

**print**(res)

# Output Jessa-salary-is-8000$

**The Match object methods**

Also, whenever we found a match to the regex pattern, Python returns us the Match object. Later we can use the following methods of a re.Match object to extract the matched values and positions.

| **Method** | **Meaning** |
| --- | --- |
| group() | Return the string matched by the regex pattern. See capturing groups. |
| groups() | Returns a tuple containing the strings for all matched subgroups. |
| start() | Return the start position of the match. |
| end() | Return the end position of the match. |
| span() | Return a tuple containing the (start, end) positions of the match. |

Python regex Match object methods

**Regex Metacharacters**

We can use both the special and ordinary characters inside a regular expression. For example, Most ordinary characters, like 'A', 'p', are the simplest regular expressions; they match themselves. You can concatenate ordinary characters, so the PYnative pattern matches the string 'PYnative'.

Apart from fo this we also have special characters. For example, characters like '|', '+', or '\*', are special. Special metacharacters don’t match themselves. Instead, they indicate that some rules. Special characters affect how the regular expressions around them are interpreted.

Read more on **Regex Metacharacters Guide**

**Click on see the example to study it in detail**.

| **Metacharacter** | **Description** |
| --- | --- |
| . (DOT) | Matches any character except a newline. See example |
| ^ (Caret) | Matches pattern only at the start of the string. See example |
| $ (Dollar) | Matches pattern at the end of the string. See example |
| \* (asterisk) | Matches 0 or more repetitions of the regex. See example |
| + (Plus) | Match 1 or more repetitions of the regex. See example |
| ? (Question mark) | Match 0 or 1 repetition of the regex. See example |
| [] (Square brackets) | Used to indicate a set of characters. Matches any single character in brackets. For example, [abc] will match either a, or, b, or c character. See example |
| | (Pipe) | used to specify multiple patterns. For example, P1|P2, where P1 and P2 are two different regexes. |
| \ (backslash) | Use to escape special characters or signals a special sequence. For example, If you are searching for one of the special characters you can use a \ to escape them. See example |
| [^...] | Matches any single character not in brackets. |
| (...) | Matches whatever regular expression is inside the parentheses. For example, (abc) will match to substring 'abc' |

Python regex metacharacters

**Regex special sequences (a.k.a. Character Classes)**

The special sequences consist of '\' and a character from the list below. Each special sequence has a unique meaning.

The following special sequences have a pre-defined meaning and make specific common patterns more comfortable to use. For example, you can use \d as a simplified definition for [0..9] or \w as a simpler version of [a-zA-z].

**Click on each special sequence**to study it in detail.

| **Special Sequence** | **Meaning** |
| --- | --- |
| \A | Matches pattern only at the start of the string. See example |
| \Z | Matches pattern only at the end of the string. |
| \d | Matches to any digit. Short for character classes [0-9]. See example |
| \D | Matches to any non-digit. short for [^0-9]. |
| \s | Matches any whitespace character. short for character class [ \t\n\x0b\r\f]. See example |
| \S | Matches any non-whitespace character. Short for [^ \t\n\x0b\r\f]. |
| \w | Matches any alphanumeric character. Short for character class [a-zA-Z\_0-9]. See example |
| \W | Matches any non-alphanumeric character. Short for [^a-zA-Z\_0-9] |
| \b | Matches the empty string, but only at the beginning or end of a word. Matches a word boundary where a word character is [a-zA-Z0-9\_]. For example, '\bJessa\b' matches 'Jessa', 'Jessa.', '(Jessa)', 'Jessa Emma Kelly' but not 'JessaKelly' or 'Jessa5'. See example |
| \B | Opposite of a \b. Matches the empty string, but only when it is not at the beginning or end of a word |

Python regex Special Sequences

**Regex Quantifiers**

We use quantifiers to define quantities. A quantifier is a metacharacter that determines how often a preceding regex can occur. you can use it to specify how many times a regex can repeat/occur.

For example, We use metacharacter \*, +, ? and {} to define quantifiers.

Let's see the list of quantifiers and their meaning.

| **Quantifier** | **Meaning** |
| --- | --- |
| \* | Match 0 or more repetitions of the preceding regex. For example, a\* matches any string that contains zero or more occurrences of 'a'. |
| + | Match 1 or more repetitions of the preceding regex. For example, a+ matches any string that contains at least one a, i.e., a, aa, aaa, or any number of a's. |
| ? | Match 0 or 1 repetition of the preceding regex. For example, a? matches any string that contains zero or one occurrence of a. |
| {2} | Matches only 2 copies of the preceding regex. For example, p{3} matches exactly three 'p' characters, but not four. |
| {2, 4} | Match 2 to 4 repetitions of the preceding regex. For example, a{2,4} matches any string that contains 3 to 5 'a' characters. |
| {3,} | Matches minimum 3 copies of the preceding regex. It will try to match as many repetitions as possible. For example, p{3,} matches a minimum of three 'p' characters. |

Python regex quantifiers

**Regex flags**

All RE module methods accept an optional flags argument used to enable various unique features and syntax variations.

For example, you want to search a word inside a string using regex. You can enhance this regex's capability by adding the **RE.I** flag as an argument to the search method to enable case-insensitive searching.

**Click on each flag**to study it in detail.

| **Flag** | **long syntax** | **Meaning** |
| --- | --- | --- |
| re.A | re.ASCII | Perform ASCII-only matching instead of full Unicode matching. |
| re.I | re.IGNORECASE | Perform case-insensitive matching. |
| re.M | re.MULTILINE | This flag is used with metacharacter ^ (caret) and $ (dollar). When this flag is specified, the metacharacter ^ matches the pattern at beginning of the string and each newline’s beginning (\n). And the metacharacter $ matches pattern at the end of the string and the end of each new line (\n) |
| re.S | re.DOTALL | Make the DOT (.) special character match any character at all, including a newline. Without this flag, DOT(.) will match anything except a newline. |
| re.X | re.VERBOSE | Allow comment in the regex. This flag is useful to make regex more readable by allowing comments in the regex. |
| re.L | re.LOCALE | Perform case-insensitive matching dependent on the current locale. Use only with bytes patterns. |

Python regex flags

To specify more than one flag, use the | operator to connect them.

For example:

re.findall(pattern, string, flags=re.I|re.M|re.X)