In [1]:
```
import re
```

What is Group in Regex?

A group is a part of a regex pattern enclosed in parentheses () metacharacter. We create a group by placing the regex pattern inside the set of parentheses (and) . For example, the regular expression (cat) creates a single group containing the letters ‘c’, ‘a’, and ‘t’.

For example, in a real-world case, you want to capture emails and phone numbers, So you should write two groups, the first will search email, and the second will search phone numbers.

Also, capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses (,).

For example, In the expression, ((\w)(\s\d)), there are three such groups

- ((\w)(\s\d))
- (\w)
- (\s\d)

We can specify as many groups as we wish. Each sub-pattern inside a pair of parentheses will be captured as a group. Capturing groups are numbered by counting their opening parentheses from left to right.

Capturing groups are a handy feature of regular expression matching that allows us to query the Match object to find out the part of the string that matched against a particular part of the regular expression.

Anything you have in parentheses () will be a capture group. using the group(group_number) method of the regex Match object we can extract the matching value of each group.

We will see how to capture single as well as multiple groups.

Example to Capture Multiple Groups Let’s assume you have the following string:

target_string = "The price of PINEAPPLE ice cream is 20"

And, you wanted to match the following two regex groups inside a string

1. To match an UPPERCASE word

2. To match a number
3. To extract the uppercase word and number from the target string we must first write two regular expression patterns.

> 1. Pattern to match the uppercase word (PINEAPPLE)
> 2. Pattern to match the number (20).
> 3. The first group pattern to search for an uppercase word: [A-Z]+

[A-Z] is the character class. It means match any letter from the capital A to capital Z in uppercase exclusively. Then the + metacharacter indicates 1 or more occurrence of an uppercase letter Second group pattern to search for the price: \d+

> > - The \d means match any digit from 0 to 9 in a target string
> > - Then the + metacharacter indicates number can contain a minimum of 1 or maximum any number of digits.
> > - Extract matched group values

In the end, we can use the groups() and group() method of match object to get the matched values.

In [2]:
```
import re

target_string = "The price of PINEAPPLE ice cream is 20"

# two groups enclosed in separate ( and ) bracket
result = re.search(r"(\b[A-Z]+\b).+(\b\d+)", target_string)

# Extract matching values of all groups
print(result.groups())

# Extract match value of group 1
print(result.group(1))

# Extract match value of group 2
print(result.group(2))

```

```
('PINEAPPLE', '20')
PINEAPPLE
20
```

Let's understand the above example

1. First of all, I used a raw string to specify the regular expression pattern. 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 we must use raw string.
2. Now let's take a closer look at the regular expression syntax to define and isolate the two patterns we are looking to match. We need two things.
3. First, we need to enclose each of the two patterns inside a pair of parentheses. So (\b[A-Z]+\b) is the first group, and (\b\d+) is the second group in between parentheses. Therefore each pair of parentheses is a group.

Note:

- The parentheses are not part of the pattern. It indicates a group.
- The \b indicates a word boundary.
- Secondly, we need to consider the larger context in which these groups reside. This means that we also care about the location of each of these groups inside the entire target string and that's why we need to provide context or borders for each group.

- Next, I have added .+ at the start of each group. the dot represents any character except a new line and the plus sign means that the preceding pattern is repeating one or more times. This syntax means that before the group, we have a bunch of characters that we can ignore, only take uppercase words followed by the word boundary (whitespace). it will match to PINEAPPLE.
- I have also added .+ at the start of the second pattern, it means before the second group, we have a bunch of characters that we can ignore, only take numbers followed by a boundary. it will match to 20.

5. Next, we passed both the patterns to the re.search() method to find the match.

The groups() method

- At last, using the groups() method of a Match object, we can extract all the group matches at once. It provides all matches in the tuple format.

Access Each Group Result Separately

We can use the group() method to extract each group result separately by specifying a group index in between parentheses. Capturing groups are numbered by counting their opening parentheses from left to right. In our case, we used two groups.

Please note that unlike string indexing, which always starts at 0, group numbering always starts at 1.

The group with the number 0 is always the target string. If you call The group() method with no arguments at all or with 0 as an argument you will get the entire target string.

To get access to the text matched by each regex group, pass the group's number to the group(group_number) method.

So the first group will be a group of 1. The second group will be a group of 2 and so on.

In [3]:
```
# Extract first group
print(result.group(1))

# Extract second group
print(result.group(2))

# Target string
print(result.group(0))
```

```
PINEAPPLE
20
PINEAPPLE ice cream is 20
```

So this is the simple way to access each of the groups as long as the patterns were matched.

Regex Capture Group Multiple Times In earlier examples, we used the search method. It will return only the first match for each group. But what if a string contains the multiple occurrences of a regex group and you want to extract all matches.

To capture all matches to a regex group we need to use the finditer() method.

The finditer() method finds all matches and returns an iterator yielding match objects matching the regex pattern. Next, we can iterate each Match object and extract its value.

Note: Don't use the findall() method because it returns a list, the group() method cannot be applied. If you try to apply it to the findall method, you will get AttributeError: 'list' object has no attribute 'groups.'

So always use finditer if you wanted to capture all matches to the group.

Example

In [4]:
```python
import re

target_string = "The price of ice-creams PINEAPPLE 20 MANGO 30 CHOCOLATE 40"

# two groups enclosed in separate ( and ) bracket
# group 1: find all uppercase letter
# group 2: find all numbers
# you can compile a pattern or directly pass to the finditer() method
pattern = re.compile(r"(\b[A-Z]+\b).(\b\d+\b)")

# find all matches to groups
for match in pattern.finditer(target_string):
    # extract words
    print(match.group(1))
    # extract numbers
    print(match.group(2))
```

```
PINEAPPLE
20
MANGO
30
CHOCOLATE
40
```

Extract Range of Groups Matches

One more thing that you can do with the group() method is to have the matches returned as a tuple by specifying the associated group numbers in between the group() method's parentheses. This is useful when we want to extract the range of groups.

For example, get the first 5 group matches only by executing the group(1, 5).

In [5]:
```
import re

target_string = "The price of PINEAPPLE ice cream is 20"
# two pattern enclosed in separate ( and ) bracket
result = re.search(r".+(\b[A-Z]+\b).+(\b\d+)", target_string)

print(result.group(1, 2))
# Output ('PINEAPPLE', '20')
```

```
('PINEAPPLE', '20')
```

Method	Meaning
group()	Return the string matched by the regex pattern.
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.

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 'PYThon' pattern matches the string 'PYThon'.

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.

Metacharacter	Description
. (DOT)	Matches any character except a newline.
^ (Caret)	Matches pattern only at the start of the string.
$ (Dollar)	Matches pattern at the end of the string.
* (asterisk)	Matches 0 or more repetitions of the regex.

Metacharacter	Description
+ (Plus)	Match 1 or more repetitions of the regex.
? (Question mark)	Match 0 or 1 repetition of the regex.
[] (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.
(Pipe)	used to specify multiple patterns. For example, P1
(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.
[^...]	Matches any single character not in brackets.
(...)	Matches whatever regular expression is inside the parentheses. For example, (abc) will match to substring 'abc'

Regex . dot metacharacter

Inside the regular expression, a dot operators represents any character except the newline character, which is \n. Any character means letters uppercase or lowercase, digits 0 through 9, and symbols such as the dollar ($) sign or the pound (#) symbol, punctuation mark (!) such as the question mark (?) commas (,) or colons (:) as well as whitespaces.

In [8]:
```
import re

target_string = "Deepali loves \n Python"
# dot(.) metacharacter to match any character
result = re.search(r'.', target_string)
print(result.group())


# .+ to match any string except newline
result = re.search(r'.+', target_string)
print(result.group())

```

```
D
Deepali loves
```

Explanation

So here, I used the search() method to search for the pattern specified in the first argument. Notice that I used the dot (.) and then the plus (+) sign over here. The plus sign is the repetition operator in regular expressions, and it means that the preceding character or pattern should repeat one or more times.

This means that we are looking to match a sequence of at least one character except for the new line.

Next, we used the group() method to see the result. As you can notice, the substring till the newline (\n) is returned because the DOT character matches any character except the new line.

DOT to match a newline character

If you want the DOT to match the newline character as well, use the re.DOTALL or re.S flag as an argument inside the search() method

In [9]:
```
import re

str1 = "Neelam is a Python developer \n She also knows ML and AI"

# dot(.) characters to match newline
result = re.search(r".+", str1, re.S)
print(result.group())
```

```
Neelam is a Python developer
 She also knows ML and AI
```

Regex ^ caret metacharacter

target_string = "Neel is a Python developer and her salary is 5000$ \n Emma also knows ML and AI" In Python, the caret operator or sign is used to match a pattern only at the beginning of the line. For example, considering our target string, we found two things.

We have a new line inside the string. Secondly, the string starts with the word Emma which is a four-letter word. So assuming we wanted to match any four-letter word at the beginning of the string, we would use the caret (^) metacharacter.

In [11]:
```
import re

target_string = "Neel is a Python developer \n Emma also knows ML and AI"

# caret (^) matches at the beginning of a string
result = re.search(r"^\w{4}", target_string)
print(result.group())

```

```
Neel
```

Explanation

So in this line of code, we are using the search() method, and inside the regular expression pattern, we are using the carrot first.

To match a four-letter word at the beginning of the string, I used the \w special sequence, which matches any alphanumeric characters such as letters both lowercase and uppercase, numbers, and the underscore character.

The 4 inside curly braces say that the alphanumeric character must occur precisely four times in a row. i.e. Neel

caret (^) to match a pattern at the beginning of each new line

Normally the carat sign is used to match the pattern only at the beginning of the string as long as it is not a multiline string meaning the string does not contain any newlines.

However, if you want to match the pattern at the beginning of each new line, then use the re.M flag. The re.M flag is used for multiline matching.

As you know, our string contains a newline in the middle.

In [12]:
```
import re

str1 = "Neel is a Python developer and her salary is 5000$ \nNeel also knows ML and AI"

# caret (^) matches at the beginning of each new line
# Using re.M flag
result = re.findall(r"^\w{4}", str1, re.M)
print(result)

```

```
['Neel', 'Neel']
```

Regex $ dollar metacharacter

This time we are going to have a look at the dollar sign metacharacter, which does the exact opposite of the caret (^) .

In Python, The dollar operator or sign matches the regular expression pattern at the end of the string. Let's test this by matching word AI which is present at the end of the string, using a dollar ($) metacharacter.

In [13]:
```
import re

str1 = "Neel is a Python developer \nNeel also knows ML and AI"
# dollar sign($) to match at the end of the string
result = re.search(r"\w{2}$", str1)
print(result.group())

```

```
AI
```

Regex * asterisk/star metacharacter

Another very useful and widely used metacharacter in regular expression patterns is the asterisk (*). In Python, The asterisk operator or sign inside a pattern means that the preceding expression or character should repeat 0 or more times with as many repetitions as possible, meaning it is a greedy repetition.

When we say * asterisk is greedy, it means zero or more repetitions of the preceding expression.

Let's see the example to match all the numbers from the following string using an asterisk (*) metacharacter.

target_string = "Numbers are 8,23, 886, 4567, 78453" Patter to match: \d\d*

Let's understand this pattern first.

As you can see, the pattern is made of two consecutive \d. The \d special sequences represent any digit.

The most important thing to keep in mind here is that the asterisk (*) at the end of the pattern means zero or more repetitions of the preceding expression. And in this case, the preceding expression is the last \d, not all two of them.

This means that we are basically searching for numbers with a minimum of 1 digit and possibly any integer.

We may get the following possible matches

- A single digit, meaning 0 repetitions according to the asterisk Or
- The two-digit number, meaning 1 repetition according to the asterisk Or
- we may have the three-digit number meaning two repetitions of the last \d, or
- The four-digit number as well.

There is no upper limit of repetitions enforced by the * (asterisk) metacharacter. However, the lower limit is zero.

So \d\d* means that the re.findall() method should return all the numbers from the target string.

In [14]:
```
import re

str1 = "Numbers are 8,23, 886, 4567, 78453"
# asterisk sign(*) to match 0 or more repetitions

result = re.findall(r"\d\d*", str1)
print(result)

```

```
['8', '23', '886', '4567', '78453']
```

Regex + Plus metacharacter

Another very useful and widely used metacharacter in regular expression patterns is the plus (+). In Python, The plus operator (+) inside a pattern means that the preceding expression or character should repeat one or more times with as many repetitions as possible, meaning it is a greedy repetition.

When we say plus is greedy, it means 1 or more repetitions of the preceding expression.

Let's see the same example to match two or more digit numbers from a string using a plus (+) metacharacter.

Patter to match: \d\d+

This means that we are basically searching for numbers with a minimum of 2 digits and possibly any integer.

We can get the following possible matches

- We may get the two-digit number, meaning 1 repetition according to the plus (+) Or
- we may have the three-digit number meaning two repetitions of the last \d, or
- we may have the four-digit number as well.
- There is no upper limit of repetitions enforced by the * (asterisk) metacharacter. However, the lower limit is 1.

So \d\d+ means that the re.findall() method should Return all the numbers with a minimum of two digits from the target string.

In [15]:
```
import re

str1 = "Numbers are 8,23, 886, 4567, 78453"
# Plus sign(+) to match 1 or more repetitions
result = re.findall(r"\d\d+", str1)
print(result)

```

```
['23', '886', '4567', '78453']
```

The ? question mark metacharacter

In Python, the question mark operator or sign (?) inside a regex pattern means the preceding character or expression to repeat either zero or one time only. This means that the number of possible repetitions is strictly limited on both ends.

Let's see the example to compare the ? with * and + metacharacters to handle repetitions.

Pattern to match: \d\d\d\d\d?

As you know, the question mark enables the repetition of the preceding character, either zero or one time.

we have five\d, which means that we want to match numbers having at least four digits while the fifth \d may repeat 0 or 1 times, meaning it doesn't exist at all or one time.

In [16]:
```
import re

target_string = "Numbers are 8,23, 886, 4567, 78453"
# Question mark sign(?) to match 0 or 1 repetitions
result = re.findall(r"\d\d\d\d\d?", target_string)
print(result)

```

```
['4567', '78453']
```

We have set a limit of four for the total number of digits in the match. And indeed, the result contains only collections of four-digit and five-digit numbers.

The \ backslash metacharacter In Python, the backslash metacharacter has two primary purposes inside regex patterns.

It can signal a special sequence being used, for example, \d for matching any digits from 0 to 9. If your expression needs to search for one of the special characters, you can use a backslash (\ ) to escape them For example, you want to search for the question mark (?) inside the string. You can use a backslash to escaping such special characters because the question mark has a special meaning inside a regular expression pattern. Let's understand each of these two scenarios, one by one.

To indicate a special sequence \d for any digits \w for any alphanumeric character \s for space Escape special character using a backslash ()

Let' take the DOT metacharacter as you've seen thus far. The DOT has a special meaning when used inside a regular expression. It matches any character except the new line.

However, In the string, the DOT is used to end the sentence. So the question is how to precisely match an actual dot inside a string using regex patterns. But the DOT already has a special meaning when used inside a pattern.

Well, the solution is to use the backslash, and it is called Escaping. You can use the backslash to escape the dot inside the regular expression pattern. And this way, you can match the actual dot inside the target string and remove its special meaning.

In [18]:
```
import re

str1 = "Deep is a Python developer. Emma salary is 5000$. Deep also knows ML and AI."
# escape dot
res = re.findall(r"\.", str1)
print(res)

```

```
['.', '.', '.']
```

The [] square brackets metacharacter

The square brackets are beneficial when used in the regex pattern because they represent sets of characters and character classes.

Let's say we wanted to look for any occurrences of letters N, d, k letters inside our target string. Or, in simple terms, match any of these letters inside the string. We can use the square brackets to represent sets of characters like [Ndk].

In [22]:
```
import re

str1 = "Harsh is a Python developer. Harsh also knows ML and AI."
res = re.findall(r"[edk]", str1)
print(res)

```

```
['d', 'e', 'e', 'e', 'k', 'd']
```

Note: Please note that the operation here is or meaning this is equivalent to saying I am looking for any occurrences of E or d or k. The result is a list containing all the matches that were found inside the target string.

This operation can be beneficial when you want to search for several characters at the same time inside a string without knowing that any or all of them are part of the string.

We can also use the square brackets to specify an interval or a range of characters and use a dash in-between the two ends of the range.

For instance, let's say that we want to match any letter from m to p inside our target string, to do this we can write regex like [m-p] Mean all the occurrences of the letters m, n, o, p.

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].

Special Sequence	Meaning
A	Matches pattern only at the start of the string.
Z	Matches pattern only at the end of the string.
d	Matches to any digit. Short for character classes [0-9].
D	Matches to any non-digit. short for [^0-9].
s	Matches any whitespace character. short for character class [tnx0brf].
S	Matches any non-whitespace character. Short for [^ tnx0brf].
w	Matches any alphanumeric character. Short for character class [a-zA-Z_0-9].
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, 'bJessab' matches 'Jessa', 'Jessa.', '(Jessa)', 'Jessa Emma Kelly' but not 'JessaKelly' or 'Jessa5'.
B	Opposite of a b. Matches the empty string, but only when it is not at the beginning or end of a word

Special Sequence \A and \Z

Backslash A (\A)

The \A sequences only match the beginning of the string. It works the same as the carrot (^) metacharacter.

On the other hand, if we do have a multi-line string, then \A will still match only at the beginning of the string, while the carrot will match at the beginning of each new line of the string.

Backslash Z (\Z) sequences only match the end of the string. It works the same as the dollar ($) metacharacter.

In [24]:
```
import re

target_str = "Deepa is a Python developer, and her salary is 8000"

# \A to match at the start of a string
# match word starts with capital letter
result = re.findall(r"\A([A-Z].*?)\s", target_str)
print("Matching value", result)

# \Z to match at the end of a string
# match number at the end of the string
result = re.findall(r"\d.*?\Z", target_str)
print("Matching value", result)

```

```
Matching value ['Deepa']
Matching value ['8000']
```

Special sequence \d and \D

Backslash d (\d)

The \d matches any digits from 0 to 9 inside the target string. This special sequence is equivalent to character class [0-9] . Use either \d or [0-9].
Backslash capital D (\D)

This sequence is the exact opposite of \d, and it matches any non-digit character. Any character in the target string that is not a digit would be the equivalent of the \D. Also, you can write \D using character class [^0-9] (carrot ^ at the beginning of the character class denotes negation). Example

Now let's do the followings

Use a special sequence \d inside a regex pattern to find a 4-digit number in our target string. Use a special sequence \D inside a regex pattern to find all the non-digit characters.

In [25]:
```
import re

target_str = "8000 dollar"

# \d to match all digits
result = re.findall(r"\d", target_str)
print(result)

# \d to match all numbers
result = re.findall(r"\d+", target_str)
print(result)

# \D to match non-digits
result = re.findall(r"\D", target_str)
print(result)

```

```
['8', '0', '0', '0']
['8000']
[' ', 'd', 'o', 'l', 'l', 'a', 'r']
```

Special Sequence \w and \W

Backslash w (\w)

The \w matches any alphanumeric character, also called a word character. This includes lowercase and uppercase letters, the digits 0 to 9, and the underscore character. Equivalent to character class [a-zA-z0-9_]. You can use either \w or [a-zA-z0-9_]. Backslash capital W (\W)

This sequence is the exact opposite of \w, i.e., It matches any NON-alphanumeric character. Any character in the target string that is not alphanumeric would be the equivalent of the \W. You can write \W using character class [^a-zA-z0-9_] . Example

Now let’s do the followings

Use a special sequence \w inside a regex pattern to find all alphanumeric character in the string Use a special sequence \W inside a regex pattern to find all the non-alphanumeric characters.

In [26]:
```
import re

target_str = "Deepa and Shiva!!"

# \w to match all alphanumeric characters
result = re.findall(r"\w", target_str)
print(result)

# \w{5} to 5-letter word
result = re.findall(r"\w{5}", target_str)
print(result)

# \W to match NON-alphanumeric
result = re.findall(r"\W", target_str)
print(result)

```

```
['D', 'e', 'e', 'p', 'a', 'a', 'n', 'd', 'S', 'h', 'i', 'v', 'a']
['Deepa', 'Shiva']
[' ', ' ', '!', '!']
```

Special Sequence \s and \S

Backslash lowercase s (\s)

The \s matches any whitespace character inside the target string. Whitespace characters covered by this sequence are as follows

common space generated by the space key from the keyboard. (" ") Tab character (\t) Newline character (\n) Carriage return (\r) form feed (\f) Vertical tab (\v) Also, this special sequence is equivalent to character class [\t\n\x0b\r\f] . So you can use either \s or [\t\n\x0b\r\f].

Backslash capital S (\S)

This sequence is the exact opposite of \s, and it matches any NON-whitespace characters. Any character in the target string that is not whitespace would be the equivalent of the \S.

Also, you can write \S using character class [^ \t\n\x0b\r\f] .

Example

Now let's do the followings

Use a special sequence \s inside a regex pattern to find all whitespace character in our target string Use a special sequence \S inside a regex pattern to find all the NON-whitespace character

In [27]:
```
import re

target_str = "Deepa \t \n  "

# \s to match any whitespace
result = re.findall(r"\s", target_str)
print(result)

# \S to match non-whitespace
result = re.findall(r"\S", target_str)
print(result)

# split on white-spaces
result = re.split(r"\s+", "Deepa and shiva")
print(result)

# remove all multiple white-spaces with single space
result = re.sub(r"\s+", " ", "Deepa   and   \t \t Shiva  ")
print(result)

```

```
[' ', '\t', ' ', '\n', ' ', ' ']
['D', 'e', 'e', 'p', 'a']
['Deepa', 'and', 'shiva']
Deepa and Shiva
```

Special Sequence \b and \B

Backslash lowercase b (\b)

The \b special sequence matches the empty strings bordering the word. The backslash \b is used in regular expression patterns to signal word boundaries, or in other words, the borders or edges of a word.

Note: A word is a set of alphanumeric characters surrounded by non-alphanumeric characters (such as space).

Example

Let’s try to match all 6-letter word using a special sequence \w and \b

In [28]:
```
import re

target_str = "  Deepa salary is 8000$ She is Python developer"

# \b to word boundary
# \w{6} to match six-letter word
result = re.findall(r"\b\w{6}\b", target_str)
print(result)

# \b need separate word not part of a word
result = re.findall(r"\bthon\b", target_str)
print(result)

```

```
['salary', 'Python']
[]
```

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.

Quantifier	Meaning
{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.

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.

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.

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

In [ ]:
```
1
```