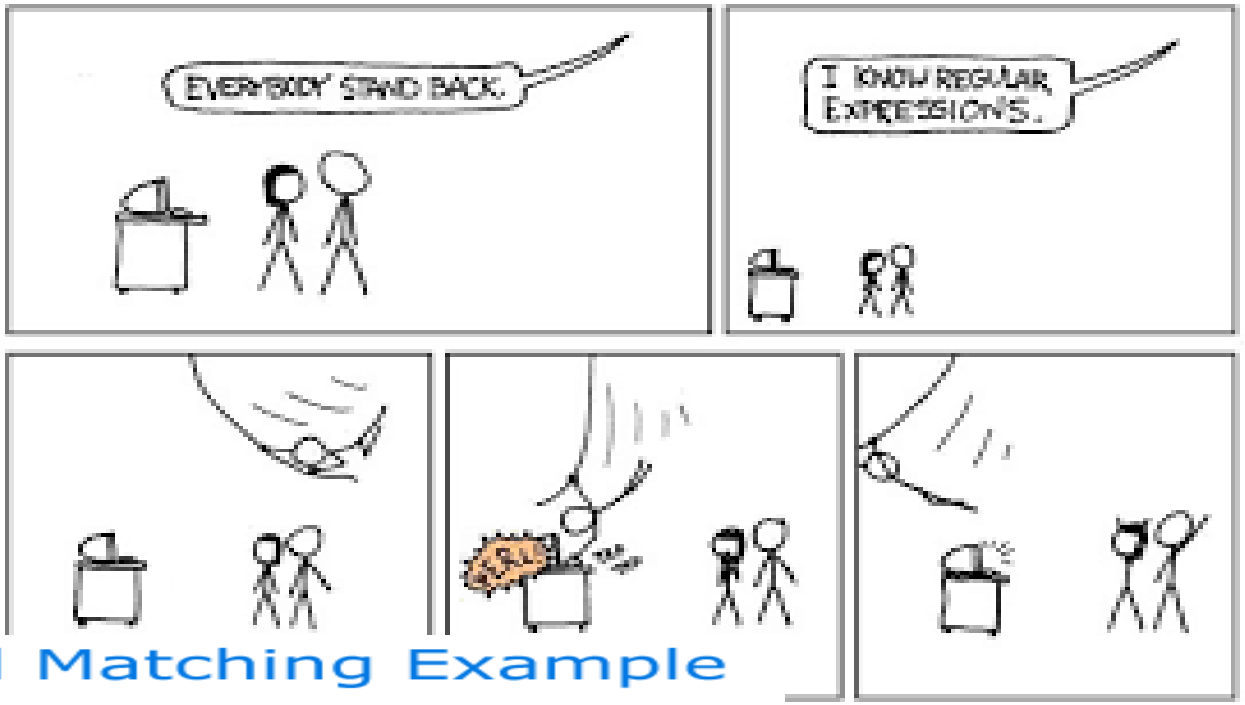
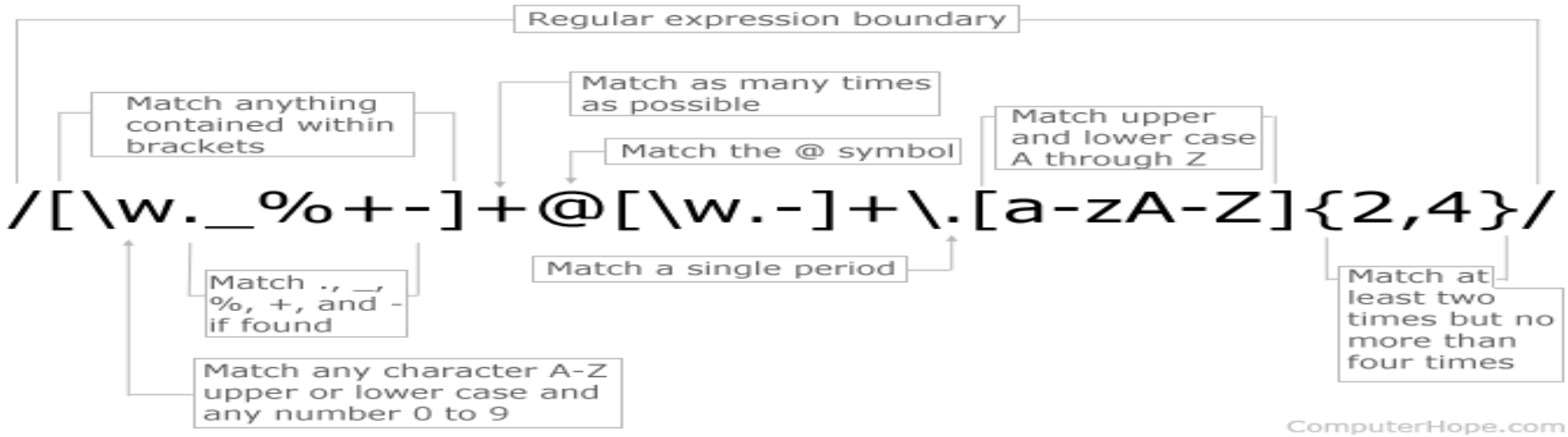
Python

Python Regex

REGEX



Regular Expression E-mail Matching Example



What are Regular EXPRESSIONS

- Regular expressions are basically patterns – patterns of alphanumeric, digits, special characters etc.
- We use them to extract meaningful data like, the calling Number exchange in a CDR, an address , type of Business etc.
- You may be familiar with searching for text by pressing CTRL-F and typing in the words you're looking for. *Regular expressions* go one step further: They allow you to specify a *pattern* of text to search for. You may not know a business's exact phone number, but if you live in the United States or Canada, you know it will be three digits, followed by a hyphen, and then four more digits (and optionally, a three-digit area code at the start). This is how you, as a human, know a phone number when you see it: 415-555-1234 is a phone number, but 4,155,551,234 is not.
- Good, now that we know basic stuff, lets make our hand dirty. ☺
- Consider the following example, suppose there is a file of customer information containing Name, Telephone numbers, Addresses and product preferences,in a haphazard manner. For random Telephonic survey, you will have to copy all the Names and Telephone numbers and put into a fle. Oops, it has already started sounding nasty , mind it!!

Pattern matching

- *You would probably try to read each line entry*
- *From those line entries , you probably would try reading for only alphabet patterns, that would represent a name.(something like using* ***isalpha*** *method)*
- *Next you will again try to locate alpha numeric sub-string for a Mobile number, typically something like +91-9XXXXXXX.*
- *You would probably have to create list objects to hold these details, or maybe a dictionary object if you plan to go for a name/ number Hash pair.*
- *By this time, you already have started feeling something like this*

Pattern matching

- *But if you know regular expressions, or regex for short, it would just take you a few entries like (\w)+ (\d{10}) to find out whatever you are looking for, more of the details later.*
- *So, I would probably do something like this , to find the phone number:*
- *(+91)?-(\d{10}), LO, all phonenumbers would keep coming on.*

CREATING REGEX OBJECTS

- *All the regex functions in Python are in the re module.*
- *Import the module* ***re [import re]***
- *Passing a string value representing your regular expression to* ***re.compile()*** *returns a Regex pattern object (or simply, a Regex object).*
- *To create a Regex object that matches the phone number pattern, enter the following into the interactive shell. (Remember that \d means "a digit character" and (+91)-(\d{10}) is the regular expression for the correct phone number pattern.)*
- ***phoneNumberRegex = re.compile(r'+91-\d{10}')***

Matching regex objects

- *A Regex object's search() method searches the string it is passed for any matches to the regex. The search() method will return None if the regex pattern is not found in the string. If the pattern is found, the search() method returns a Match object. Match objects have a group() method that will return the actual matched text from the searched string. (I'll explain groups shortly.) For example, enter the following into the interactive shell:*

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
```

Phone number found: 415-555-4242

- *Here, we pass our desired pattern to re.compile() and store the resulting Regex object in phoneNumRegex. Then we call search() on phoneNumRegex and pass search() the string we want to search for a match.*
- *The result of the search gets stored in the variable mo. In this example, we know that our pattern will be found in the string, so we know that a Match object will be returned.*
- *Knowing that mo contains a Match object and not the null value None, we can call group() on mo to return the match. Writing mo.group() inside our print statement displays the whole match, 415-555-4242.*

Matching regex objects

- *Steps for using Regular Expressions :-*
  - ✓ *Import the regex module with import re.*
  - ✓ *Create a Regex object with the re.compile() function. (Remember to use a raw string.)*
  - ✓ *Pass the string you want to search into the Regex object's search() method. This returns a Match object.*
  - ✓ *Call the Match object's group() method to return a string of the actual matched text.*

GROUPING WITH REGEX

- *Say you want to separate the area code from the rest of the phone number. Adding parentheses will create groups in the regex: (\d\d\d)-(\d\d\d-\d\d\d\d).*
- *Then you can use the group() match object method to grab the matching text from just one group.*
- *The first set of parentheses in a regex string will be group 1. The second set will be group 2. By passing the integer 1 or 2 to the group() match object method, you can grab different parts of the matched text. Passing 0 or nothing to the group() method will return the entire matched text.*
- *Enter the following into the interactive shell:*

GROUPING WITH REGEX

```
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242‘
```

The \(and \) escape characters in the raw string passed to ***re.compile()*** *will match actual parenthesis characters.*

Matching multiple groups with the pipe

- The | character is called a pipe. You can use it anywhere you want to match one of many expressions.

  For example, the regular expression ***r'Batman|Tina Fey'*** *will match either* ***'Batman'*** *or* ***'Tina Fey'****.*

When both Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object. Enter the following into the interactive shell:

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman‘
>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

Note

You can find all matching occurrences with the findall() method that’s discussed in The findall() Method.

Optional Matching with the Question Mark

- Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there.
- The **?** character flags the group that precedes it as an optional part of the pattern.
- For example, enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

Optional Matching with the Question Mark

- Using the earlier phone number example, you can make the regex look for phone numbers that do or do not have an area code. Enter the following into the interactive shell :

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
        '415-555-4242‘
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242‘
```

You can think of the ? as saying, “Match zero or one of the group preceding this question mark.”

If you need to match an actual question mark character, escape it with \?.

Matching zero or more with *

- The * (called the star or asterisk) means "match zero or more"—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again.
- Let's look at the Batman example again :-

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

MATCHING ZERO OR MORE WITH *

- For 'Batman', the (wo)* part of the regex matches zero instances of wo in the string; for 'Batwoman', the (wo)* matches one instance of wo; and for **'Batwowowowoman'**, (wo)* matches four instances of wo.
- If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

MATCHING ONE OR MORE WITH +

- While * means "match zero or more," the + (or plus) means "match one or more." Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear at least once. It is not optional. Enter the following into the interactive shell, and compare it with the star regexes in the previous section:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'
```

MATCHING ONE OR MORE WITH +

```
>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

The regex Bat(wo)+man will not match the string 'The Adventures of Batman' because at least one wo is required by the plus sign.

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: \+.

MATCHING SPECIFIC REPITITIONS WITH CURLY BRACKETS

- If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex (Ha){3} will match the string **'HaHaHa'**, but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.
- Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and **'HaHaHaHaHa'**. You can also leave out the first or second number in the curly brackets to leave the minimum or maximum unbounded.
- For example, (Ha){3,} will match three or more instances of the (Ha) group, while (Ha){,5} will match zero to five instances. Curly brackets can help make your regular expressions shorter. These two regular expressions match identical patterns:

(Ha){3}

(Ha)(Ha)(Ha)

And these two regular expressions also match identical patterns:

(Ha){3,5}

((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))

Enter the following into the interactive shell:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Here, (Ha){3} matches 'HaHaHa' but not 'Ha'. Since it doesn't match 'Ha', search() returns None.

Greedy and non-greedy matches

- Since **(Ha){3,5}** can match three, four, or five instances of Ha in the string **'HaHaHaHaHa'**, you may wonder why the Match object's call to group() in the previous curly bracket example returns **'HaHaHaHaHa'** instead of the shorter possibilities. After all, **'HaHaHa'** and **'HaHaHaHa'** are also valid matches of the regular expression (Ha){3,5}.
- Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark
- Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.
- Enter the following into the interactive shell, and notice the difference between the greedy and nongreedy forms of the curly brackets searching the same string:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'
>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a nongreedy match or flagging an optional group. These meanings are entirely unrelated.

Findall() method

- In addition to the search() method, Regex objects also have a findall() method. While search() will return a Match object of the first matched text in the searched string, the findall() method will return the strings of every match in the searched string. To see how search() returns a Match object only on the first instance of matching text, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d{3}-\d{3}-\d{4}')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

- On the other hand, findall() will not return a Match object but a list of strings—as long as there are no groups in the regular expression. Each string in the list is a piece of the searched text that matched the regular expression.
- Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d{3}-\d{3}-\d{4}')#Has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

Findall() method

- In addition to the search() method, Regex objects also have a findall() method. While search() will return a Match object of the first matched text in the searched string, the findall() method will return the strings of every match in the searched string. To see how search() returns a Match object only on the first instance of matching text, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d{3}-\d{3}-\d{4}')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

- On the other hand, findall() will not return a Match object but a list of strings—as long as there are no groups in the regular expression. Each string in the list is a piece of the searched text that matched the regular expression.
- Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r‘d{3}-\d{3}-\d{4}')#Has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

Findall() method

- If there are groups in the regular expression, then findall() will return a list of tuples. Each tuple represents a found match,and its items are the matched strings for each group in the regex. To see findall() in action, enter the following into the interactive shell (notice that the regular expression being compiled now has groups in parentheses):

```
>>> phoneNumRegex = re.compile(r'(\d{3})-(\d{3})-(\d{4})') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '9999'), ('212', '555', '0000')]
```

- To summarize what the findall() method returns, remember the following:

✓When called on a regex with no groups, such as \d{3}-\d\{3}-\d{4}, the method findall() returns a list of string matches, such as ['415-555-9999', '212-555-0000'].

✓When called on a regex that has groups, such as (\d{3})-(\d{3})-(\d{4}), the method **findall()** returns a list of tuples of strings (one string for each group), such as [('415', '555', '9999'), ('212', '555', '0000')].

CHARACTER CLASSES

- In the earlier phone number regex example, we learned that \d could stand for any numeric digit. That is, \d is shorthand for the regular expression (0|1|2|3|4|5|6|7|8|9). There are many such shorthand character classes, as shown below :-

S.No.	Expression & Matches
1	**a, X, 9, <** ordinary characters just match themselves exactly.
2	**. (a period)** matches any single character except newline '\n'
3	**\w** matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_].
4	**\W** matches any non-word character.
5	**\b** boundary between word and non-word
6	**\s** matches a single whitespace character -- space, newline, return, tab
7	**\S** matches any non-whitespace character.

CHARACTER CLASSES

S.No.	Expression & Matches
8	**\t, \n, \r** tab, newline, return
9	**\d** decimal digit [0-9]
10	**^** matches start of the string
11	**$** match the end of the string
12	**** inhibit the "specialness" of a character.

Character classes are nice for shortening regular expressions. The character class [0-5] will match only the numbers 0 to 5; this is much shorter than typing (0|1|2|3|4|5).

For example, enter the following into the interactive shell:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

The regular expression \d+\s\w+ will match text that has one or more numeric digits (\d+), followed by a whitespace character (\s), followed by one or more letter/digit/underscore characters (\w+). The **findall()** method returns all matching strings of the regex pattern in a list.

YOUR VERY OWN CHARACTER CLASSES

- There are times when we want to match a set of characters but the shorthand character classes (\d, \w, \s, and so on) are too broad. We can then define our own character class using square brackets. For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.
- Enter the following into the interactive shell:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('Robocop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

- You can also include ranges of letters or numbers by using a hyphen. For example, the character class [a-zA-Z0-9] will match all lowercase letters, uppercase letters, and numbers.
- Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the ., *, ?, or () characters with a preceding backslash. For example, the character class [0-5.] will match digits 0 to 5 and a period. You do not need to write it as [0-5\.].
- By placing a caret character (^) just after the character class's opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class. For example, enter the following into the interactive shell:

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')
>>> consonantRegex.findall('Robocop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', '
', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel.

The Caret and Dollar Sign Characters

- You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put a dollar sign ($) at the end of the regex to indicate the string must end with this regex pattern.
- And you can use the ^ and $ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.
- For example, the r'^Hello' regular expression string matches strings that begin with 'Hello'. Enter the following into the interactive shell:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

The r'\d$' regular expression string matches strings that end with a numeric character from 0 to 9. Enter the following into the interactive shell:

The Caret and Dollar Sign Characters

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

- The r'^\d+$' regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

MATCHING WITH A .*

- Sometimes you will want to match everything and anything. For example, say you want to match the string 'First Name:', followed by any and all text, followed by 'Last Name:', and then followed by anything again. You can use the dot-star (.*) to stand in for that "anything." Remember that the dot character means "any single character except the newline," and the star character means "zero or more of the preceding character."
- Enter the following into the interactive shell:

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart‘
```

- The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a nongreedy fashion, use the dot, star, and question mark (.*?).
- Like with curly brackets, the question mark tells Python to match in a nongreedy way.

Matching Newlines with the Dot Character

- The dot-star will match everything except a newline. By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match all characters, including the newline character.
- Enter the following into the interactive shell:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.‘
>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
```

'Serve the public trust.\nProtect the innocent.\nUphold the law.‘

CASE INSENSITIVE MATCHING

- Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('Robocop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

- But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile(). Enter the following into the interactive shell:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('Robocop is part man, part machine, all cop.').group()
 'Robocop'
>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'
>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

CASE INSENSITIVE MATCHING

- Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('Robocop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

- But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile(). Enter the following into the interactive shell:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('Robocop is part man, part machine, all cop.').group()
 'Robocop'
>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'
>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

Substituting Strings with the sub() Method

- Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The sub() method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The sub() method returns a string with the substitutions applied.
- For example, enter the following into the interactive shell:

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.‘
```

- Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to sub(), you can type \1, \2, \3, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution."
- For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex Agent (\w)\w* and pass r'\1****' as the first argument to sub(). The \1 in that string will be replaced by whatever text was matched by group 1—that is, the (\w) group of the regular expression.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent
Eve knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.'
```

MANAGING COMPLEX REGEXES

- Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the re.compile() function to ignore whitespace and comments inside the regular expression string. This "verbose mode" can be enabled by passing the variable **re.VERBOSE** as the second argument to re.compile().
- Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}
(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

- You can spread the regular expression over multiple lines with comments like this:

```
phoneRegex = re.compile(r'''(
(\d{3}|\(\d{3}\))?          # area code
(\s|-|\.)?                  # separator
\d{3}                       # first 3 digits
(\s|-|\.)                   # separator
\d{4}                       # last 4 digits
(\s*(ext|x|ext.)\s*\d{2,5})?  # extension
)''', re.VERBOSE)
```

- Note how the previous example uses the triple-quote syntax (''') to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.
- The comment rules inside the regular expression string are the same as regular Python code: The # symbol and everything after it to the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched. This lets you organize the regular expression so it's easier to read.

COMBINING VERBOSE,DOTALL AND IGNORE MODES

- What if you want to use re.VERBOSE to write comments in your regular expression but also want to use re.IGNORECASE to ignore capitalization? Unfortunately, the re.compile() function takes only a single value as its second argument. You can get around this limitation by combining the re.IGNORECASE, re.DOTALL, and re.VERBOSE variables using the pipe character (|), which in this context is known as the bitwise or operator.
- So if you want a regular expression that's case-insensitive and includes newlines to match the dot character, you would form your re.compile() call like this:

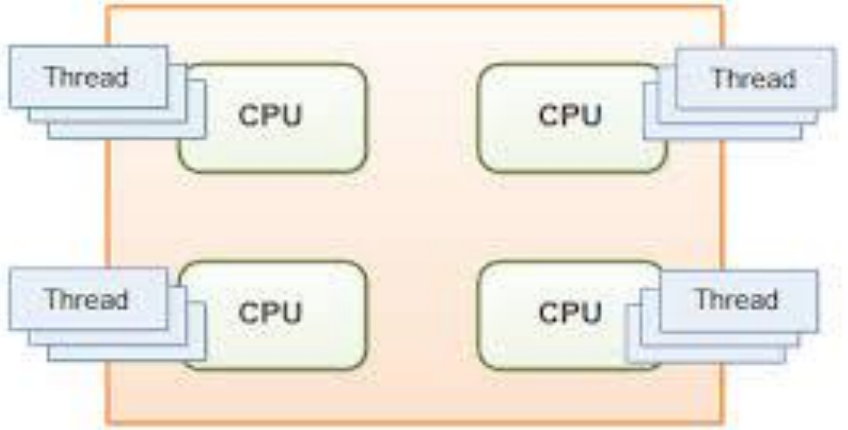
```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

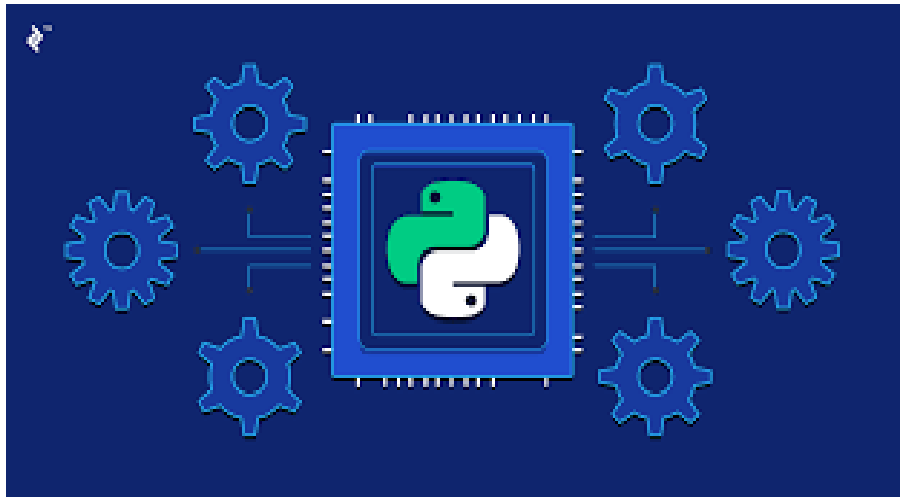
- All three options for the second argument will look like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

- This syntax is a little old-fashioned and originates from early versions of Python.

PYTHON MULTITHREADING





- ✓ One process, running Multiple instances.
- ✓ Helps reduce execution time.
- ✓ Works best with Multicore processors, makes maximum use of multicore processors.
- ✓ A thread has a starting point, an execution sequence, and a result. It has an instruction pointer that holds the current state of the thread and controls what executes next in what order.
- ✓ Python multithreading mechanism is pretty user-friendly which you can learn quickly.

PYTHON MULTITHREADING

PROS	CONS
Multithreading can significantly improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.	On a single processor system, multithreading wouldn't impact the speed of computation. In fact, the system's performance may downgrade due to the overhead of managing threads.
Multithreading allows a program to remain responsive while one thread waits for input and another runs a GUI at the same time. This statement holds true for both multiprocessor or single processor systems	Synchronization is required to avoid mutual exclusion while accessing shared resources of the process. It directly leads to more memory and CPU utilization.
All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables	Multithreading increases the complexity of the program thus also making it difficult to debug.
	It raises the possibility of potential deadlocks.
	It may cause starvation when a thread doesn't get regular access to shared resources. It would then fail to resume its work.

PYTHON MULTITHREADING

- Python implements multithreading by means of the **threading** module.
- Here's a quick look at the steps to create threads in Python :-
  - ✓ Create a thread by means of **threading.Thread()** method.
  - ✓ Start the thread, the **start()** method of the thread object will help you do it.
  - ✓ Join() method to kill one string before starting another.
  - ✓ Use Locks , wherever necessary to ensure that two or more threads don't slug it out for capturing a global resource.
- The *threading* module exposes all the methods of the *thread* module and provides some additional methods −
  - ✓ **threading.activeCount()** − Returns the number of thread objects that are active.
  - ✓ **threading.currentThread()** − Returns the number of thread objects in the caller's thread control.
  - ✓ **threading.enumerate()** − Returns a list of all thread objects that are currently active.
- In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows −
  - ✓ **run()** − The run() method is the entry point for a thread.
  - ✓ **start()** − The start() method starts a thread by calling the run method.
  - ✓ **join([time])** − The join() waits for threads to terminate.
  - ✓ **isAlive()** − The isAlive() method checks whether a thread is still executing.
  - ✓ **getName()** − The getName() method returns the name of a thread.
  - ✓ **setName()** − The setName() method sets the name of a thread.
  - ✓ Ident - Thread identification number