REGULAR EXPRESSIONS

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 4,15,551,234 is not.

WHAT EXACTLY IS A REGULAR EXPRESSION?

- ***Regular expressions are strings that are used to match and manipulate text***.
- Regular expressions are created using the **regular expression language**, a specialized language designed to do ***find and replace***.
- Like any language, regular expressions have a **special syntax and instructions**.
- The regular expression language is not a full programming language. It is usually not even an actual program or utility that you can install and use. More often than not, regular expressions are mini-languages built in to other languages or products.
- Regular expressions (or regex, for short) are tools, and like all tools, regular expressions are designed to solve a very specific problem.
- Regular expressions are used in searches when the text to be searched for is highly dynamic,
- However, the real power of regex is seen in replace operations.
- Regular expressions are one of the most powerful tools available for text manipulation.
- The regular expressions language is used to construct regular expressions (the actual constructed string is called a regular expression), and regular expressions are used to perform both search and replace operations.

REGULAR EXPRESSIONS

FINDING PATTERNS OF TEXT WITHOUT REGULAR EXPRESSIONS

Say you want to find a phone number in a string.
You know the pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers.
Here's an example: 415-555-4242.

Let's use a function named **isPhoneNumber(x)** to check whether a string **x** matches this pattern, returning either True or False.

Open a new file editor window and enter the following code; then save the file as ***isPhoneNumber.py:***

```
def isPhoneNumber(text):
    if len(text) != 12:
        return False
    for i in range(0, 3):
        if not text[i].isdecimal():
            return False
    if text[3] != '-':
        return False
    for i in range(4, 7):
        if not text[i].isdecimal():
            return False
    if text[7] != '-':
        return False
    for i in range(8, 12):
        if not text[i].isdecimal():
            return False
    return True

print('415-555-4242 is a phone number ??')
print(isPhoneNumber('415-555-4242'))

print('4,15,551,234 is a phone number ??')
print(isPhoneNumber('4,15,551,234'))
```

When this program is run, the output looks like this:

```
415-555-4242 is a phone number ??
True
4,15,551,234 is a phone number ??
False
```

REGULAR EXPRESSIONS

The isPhoneNumber() function has code that does several checks to see whether the string in text is a valid phone number. If any of these checks fail, the function returns False.

- First the code checks that the string is exactly 12 characters.
- Then it checks that the area code (that is, the first three characters in text) consists of only numeric characters.
- The rest of the function checks that the string follows the pattern of a phone number: The number must have the first hyphen after the area code, three more numeric characters, then another hyphen, and finally four more numbers.
- If the program execution manages to get past all the checks, it returns True.
- Calling isPhoneNumber() with the argument '415-555-4242' will return True.
- Calling isPhoneNumber() with 'Moshi moshi' will return False;
- The first test fails because 'Moshi moshi' is not 12 characters long.
- You would have to add even more code to find this pattern of text in a larger string.

Replace the last four print() function calls in isPhoneNumber.py with the following:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
        chunk = message[i:i+12]
                if isPhoneNumber(chunk):
                print('Phone number found: ' + chunk)
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

- On each iteration of the **for** loop, a new chunk of 12 characters from message is assigned to the variable chunk.
- For example, on the first iteration, i is 0, and chunk is assigned message[0:12] (that is, the string 'Call me at 4').
- On the next iteration, i is 1, and chunk is assigned message[1:13] (the string 'all me at 41').
- You pass chunk to isPhoneNumber() to see whether it matches the phone number pattern, and if so, you print the chunk.
- Continue to loop through message, and eventually the 12 characters in chunk will be a phone number.
- The loop goes through the entire string, testing each 12-character piece and printing any chunk it finds that satisfies isPhoneNumber().
- Once we're done going through message, we print Done.

REGULAR EXPRESSIONS

FINDING PATTERNS OF TEXT WITH REGULAR EXPRESSIONS

- The previous phone number–finding program works, but it uses a lot of code to do something limited:
- The isPhoneNumber() function is 17 lines but can find only one pattern of phone numbers.
- What about a phone number formatted like 415.555.4242 or (415) 555-4242?
- What if the phone number had an extension, like 415-555-4242 x99?
- The isPhoneNumber() function would **fail** to validate them.
- You could add yet **more code** for these additional patterns, but there is an easier way.

Regular expressions, called regexes for short, are descriptions for a pattern of text.

For example, a **\d** in a regex stands for a digit character— that is, any single numeral 0 to 9.

The regex \d\d\d-\d\d\d-\d\d\d\d is used by Python to match the same text the previous isPhoneNumber() function did: ***a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers***. Any other string would not match the \d\d\d-\d\d\d-\d\d\d\d regex.

But regular expressions can be much more sophisticated.
For example, adding a 3 in curly brackets ({3}) after a pattern is like saying, "Match this pattern three times."
So the slightly shorter regex \d{3}-\d{3}-\d{4} also matches the correct phone number format.

CREATING REGEX OBJECTS

All the regex functions in Python are in the **re** module.

```
>>> 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 \d\d\d-\d\d\d-\d\d\d\d is the regular expression for the correct phone number pattern.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Now the **phoneNumRegex** variable contains a **Regex object**.

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***.

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
```

The **mo** variable name is just a generic name to use for **Match objects**. This example might seem complicated at first, but it is much shorter than the earlier isPhoneNumber.py program and does the same thing.

- 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.

Review of Regular Expression Matching

While there are several steps to using regular expressions in Python, each step is fairly simple.
1. **Import** the regex module with import **re**.
2. **Create** a Regex object with the re.compile() function. (Remember to use a raw string.)
3. **Pass** the string you want to search into the Regex object's search() method. This returns a Match object.
4. **Call the Match** object's group() method to return a string of the actual matched text.

MORE PATTERN MATCHING WITH REGULAR EXPRESSIONS

Grouping with Parentheses

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.

```
>>> 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'
```

If you would like to retrieve all the groups at once, use the **groups() method**—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

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

REGULAR EXPRESSIONS

Regular Expressions are used in programming languages to filter texts or text strings. It's possible to check, if a text or a string matches a regular expression. Python provide us regular expression library which we must import before we could use regular expression. The regular expression library is "**re**", so we have to **import re** in the beginning of the program. The power of regular expressions is that they can specify patterns, not just fixed characters.

1) a, X, 9, < -- ***ordinary characters just match themselves exactly***.

2) The metacharacters which do not match themselves because ***they have special meanings*** are: . ^ $ * + ? { [ ] \ | ( )

3) **.** (a period) -- matches ***any single character*** except newline '\n'

4) \w -- (lowercase w) ***matches a "word" character***: ***a letter or digit or underbar [azA- Z0-9_]***. Note that although "word" is the mnemonic for this, it only matches a ***single word char***, not a whole word.

5) \W (upper case W) matches ***any non-word character***.

6) \b -- ***boundary between word and non-word***

7) \s -- (lowercase s) ***matches a single whitespace character*** -- space, newline, return, tab, form [\n\r\t\f].

8) \S (upper case S) ***matches any non-whitespace character***.

9) ^ = start, $ = end -- ***match the start or end of the string***

10) \t, \n, \r -- ***tab, newline, return***

11) \d -- ***decimal digit [0-9]*** (some older regex utilities do not support but \d, but they all support \w and \s)

12) \ -- ***inhibit the "specialness" of a character***. So, for example, use **\.** to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, \@, to make sure it is treated just as a character.

REGULAR EXPRESSIONS

A string always matches itself.

Therefore, the pattern ‘ **xxx** ’ will always match itself in ‘ abc**xxx**abc ’ . Everything else is just additional; the core of what we’re doing is just finding strings in other strings.

You can add special characters to make the patterns match more interesting things. The most commonly used one is the general wildcard ‘ **.** ’ (a period or dot).

The dot matches any one character in a string;
so, for instance, ‘ x.x ’ will match the strings ‘ xxx ’ or ‘ xyx ’ or even ‘ x.x ’ .

What if you really only want to find something with a dot in it, like ‘ x.x ’ ? Actually, specifying ‘ x.x ’ as a pattern won’t work; it will also match ‘ x!x ’ and ‘ xqx ’.

Instead, regular expressions enable you to escape special characters by adding a backslash in front of them. Therefore***, to match ‘ x.x ’ and only ‘ x.x ’ , you would use the pattern ‘ x\.x ’*** , which takes away the special meaning of the period as with an escaped character.

Python also uses the backslash for escape sequences, because ‘ \n ’ specifies a carriage return and ‘ \t ’ is a tab character. So, regular expressions are usually specified as raw strings , which means that you add an ‘ r ’ onto the front of the string constant, and then Python treats them specially.

So, how do you really match ‘ x.x ’ ? *Simple: You specify the pattern r “ x\.x ”*.

Performing Queries with Regular expression (Regex) in Python

The '**re'** package provides several methods to ***perform queries on an input string***. These are as follows:

a) re.match()
b) re.search()
c) re.findall()

MATCH METHOD

The match() method checks for matches at the beginning, returning None if no match is found. The match() method works such that ***it will only find matches if they occur at the start of the string*** being searched. So for example, calling match() on the string 'orange apple orange mango', looking for the pattern 'orange' will match but calling match() on the string 'apple orange mango orange', looking for the pattern 'orange' will NOT match.

```
import re
string ='apple, orange, mango, orange'
strmatch = re.match(r'orange', string)
print(strmatch)
>>>
None

string ='orange, apple, mango, orange'
strmatch = re.match(r'orange', string)
print(strmatch)
>>>
<_sre.SRE_Match object; span=(0, 6), match='orange'>
```

The match is found in the beginning so it is showing the match. But the correct way to print is as follows:

```
import re
string ='orange, apple, mango, orange'
strmatch = re.match(r'orange', string)
print(strmatch.group(0))
>>>
orange
>>>
```

So, ***match***, will match a text with a string IF that exact text is found at the beginning of the string, OR, if the string is the exact same as the text.

SEARCH METHOD

The **search()** method works like match and ***matches anywhere in the string***. **search()** doesn't restrict us to only finding matches at the beginning of the string, so searching for 'orange' in the below example string ***finds a match and returns the first instance***:

```
import re
string ='apple, orange, mango, orange'
match = re.search(r'orange', string)
print(match.group(0))
>>>
orange
>>>
```

The search() method, stops looking after it finds a match, so search()-ing for 'orange' in our example string only finds the first occurrence i.e., second item after apple.

FINDALL METHOD

The re.findall() method get a list of all matching patterns. The output of the above example is a **list** of all search key word "orange". In our string variable we have two oranges so the output of the list returned two orange:

```
>>> import re
>>>string ='apple, orange, mango, orange'
>>>strmatch = re.findall(r'orange', string)
>>>print(strmatch)
>>>
['orange', 'orange']
>>>
```

The findall() method come very handy to search common matching from a large text string. One common example is to find the email #id's from an email text sting. Let's write a program to find email# in the below example:

```
>>>string = '''Hello from michaeldevine@gmail.com to somebody@gmail.com about the party
@8PM. Hey Somebody, I am reminding you for the party, Please don't forget my favourite
brand! '''
>>>searchstr = re.findall('\S+@\S+', string)
>>>print(searchstr)
>>>
['michaeldevine@gmail.com', 'somebody@gmail.com']
```

The important point to note the expression "searchstr = re.findall('\S+@\S+', string)". We have used a two character sequence that matches a non-whitespace character (\S). The "\S+" matches as many non whitespace characters as possible but it would not match the string "@8PM" because there are no non-blank characters before the at-sign.
The above example returned us a list of the email #ids because of non-blank or non-whitespace character in the beginning of the ids, as mentioned above that it has ignored the @8PM. Let's modify the above program and add some more email #ids with some characters and see what happens:

```
>>>string = '''Hello from michaeldevine@gmail.com to somebody@gmail.com
;dinesh@gmail.com ;kuldeep@gmail.com about the party @8PM. Hey Somebody, I am
reminding you for the party, Please don't forget my favourite brand! '''
>>>searchstr = re.findall('\S+@\S+', string)
>>>print(searchstr)
>>>
['michaeldevine@gmail.com', 'somebody@gmail.com', ';dinesh@gmail.com',
';kuldeep@gmail.com']
```

- In the above output we could see that the other email #ids one extra character ";" was also included.
- But that was not our intension to print.
- If we do not want such junk character which may be present in the search string we have to modify our search parameters.
- We were interested with email starting with letters only because all email #id begins with a letter.
- Let's modify the above search string so that it searches with letters in the below program:

```
>>>string = '''Hello from 111baba@gmail.com michaeldevine@gmail.com to
somebody@gmail.com ; dinesh@gmail.com ;kuldeep@gmail.com about the party @8PM. Hey
Somebody, I am reminding you for the party, Please don't forget my favourite brand! '''
>>>searchstr = re.findall('[a-zA-Z]\S*@\S*[a-zA-Z]', string)
>>>print(searchstr)

['baba@gmail.com', 'michaeldevine@gmail.com', 'somebody@gmail.com',
'dinesh@gmail.com', 'kuldeep@gmail.com']
```

- The junk character was ignored.
- But there was an email #id starting with number "111baba@gmail.com" which we were not expecting.
- We can modify our search string to include numbers. In the below example let's see:

```
>>>string = '''Hello from 111baba@gmail.com michaeldevine@gmail.com to
somebody@gmail.com ; dinesh@gmail.com ;kuldeep@gmail.com about the party @8PM. Hey
Somebody, I am reminding you for the party, Please don't forget my favourite brand! '''
>>>searchstr = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', string)
>>>print(searchstr)

['111baba@gmail.com', 'michaeldevine@gmail.com', 'somebody@gmail.com',
'dinesh@gmail.com', 'kuldeep@gmail.com']
```

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

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\d\d)-(\d\d\d)-(\d\d\d\d)')              # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '1122'), ('212', '555', '0000')]
```

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

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

2. When called on a regex that has groups, such as (\d\d\d)-(\d\d\d)-(\d\ d\d\d), the method findall() returns a list of tuples of strings (one string for each group), such as [('415', '555', '1122'), ('212', '555', '0000')].

REGULAR EXPRESSIONS

REGULAR EXPRESSIONS WITH LAMBDA AND FILTER

1. Start the Python interpreter and import the **re** module:

```
>>> python
> > > import re
```

2. Now define a **list** of interesting - looking strings to filter with various regular expressions:

```
>>> t = ('abcxxxabc', 'xyx', 'abc', 'xxx', 'x.x', 'axa', 'axxxxa', 'cccb', 'cc', 'axxya', 'xxxc')
```

3. Do the simplest of all regular expressions first. A ***match***(***matches only if found at the beginning***):

```
>>> a=filter ((lambda t: re.match(r"xxx", t)), t)
>>> print(*a)
xxx xxxc

>>> a=filter ((lambda t: re.match(r"abc", t)), t)
>>> print(*a)
abcxxxabc abc

>>> a=filter ((lambda t: re.match(r"x", t)), t)
>>> print(*a)
xyx xxx x.x xxxc


>>> st='abc acd, xxy, xyz'
>>> a=filter ((lambda st: re.match(r"x", st)), st)
>>> print(*a)

>>> a=filter ((lambda st: re.match(r"a", st)), st)
>>> list(a)
['a', 'a']

>>> a=filter ((lambda st: re.match(r"a", st)), st)
>>> print(*a)
a a
```

```
>>> b=re.match(r"a", st)
>>> b
<_sre.SRE_Match object; span=(0, 1), match='a'>
>>> b.group()
'a'

>>> b=re.match(r"x", st)
>>> b.group()
Traceback (most recent call last):
File "<pyshell#43>", line 1, in <module>
b.group()
AttributeError: 'NoneType' object has no attribute 'group'
```

4. Why didn't that find ' axxxxa ' , too in the previous example ? In Python the re.match function looks for matches only at the start of its input. This string starts with 'a', and 'xxx' is inside the string.

To find strings anywhere in the input, use re.search:

```
> > > b=filter ((lambda t: re.search(r"xxx", t)), t)
> > > print(*b)
xxx, abcxxxabc, axxxxa

>>> a=filter ((lambda t: re.match(r"bc", t)), t)
>>> print(*a)

>>> a=filter ((lambda t: re.search(r"bc", t)), t)
>>> print(*a)
abcxxxabc abc
```

5. OK, look for the period (dot):

```
>>> c=filter ((lambda t: re.search(r"x.x", t)), t)
>>> print(*c)
abcxxxabc xyx xxx x.x axxxxa xxxc
```

6. Here's how you match only the period (by escaping the special character):

```
>>> d=filter ((lambda t: re.search(r"x\.x", t)), t)
>>> print(*d)
x.x
>>>
```

7. You also can search for any number of x's by using the asterisk, which can match a series of whatever character is in front of it:

```
>>> e=filter ((lambda t: re.search(r"x.*x", t)), t)
>>> print(*e)
abcxxxabc xyx xxx x.x axxxxa axxya xxxc
>>>
```

8. How did ' x.*x ' match ' axxya ' if there was nothing between the two x's? The secret is that the asterisk is tricky — ***it matches zero or more occurrences of a character between two x's***. If you really want to make sure something is between the x's, ***use a plus instead***, which matches one or more characters:

```
>>> f=filter ((lambda t: re.search(r"x.+x", t)), t)
>>> print(*f)
abcxxxabc xyx xxx x.x axxxxa xxxc
>>>
```

9. Now you know how to match anything with, say, a ' c ' in it:

```
>>> g=filter ((lambda t: re.search(r"c+", t)), t)
>>> print(*g)
abcxxxabc abc cccb cc xxxc
>>>
```

10. How would you match anything without a ' c ' ? Regular expressions use square brackets to denote special sets of characters to match, and if there's a ***caret at the beginning of the list***, ***it means any string that has a character that isn't the specified character*** in the set, so:

```
>>> h=filter ((lambda t: re.search(r"[^c]*", t)), t)
>>> print(*h)
abcxxxabc xyx abc xxx x.x axa axxxxa cccb cc axxya xxxc
>>>
```

11. To really match any string without a ' c ' in it, you have to use the ^ and $ special characters to refer to the beginning and end of the string and then tell **re** that you want strings composed only of non - c characters from beginning to end:

```
>>> i=filter ((lambda t: re.search(r"^[^c]*$", t)), t)
>>> print(*i)
xyx xxx x.x axa axxxxa axxya
>>>
```