# Finding patterns with regular expressions

We are going to use regular expressions to find a mobile number in a string. (This can be found in Automate the boring stuff 1st edition Part II : Chapter 7)

Regular expressions are called regex for short. These are descriptions for a pattern of text . For example a ‘/d’ in a in a regex stands for a digit character (this can be any numeral from 0 to 9).

The regex \d\d\d-\d\d\d-\d\d\d\d is used by python to find a string of three numbers followed by a hyphen. This regex basically translates to 111-222-3333.

Regular expressions can be more sophisticated, for example, adding a 3 in curly brackets ({3}) after a patters is like saying “match this pattern 3 times”

\d{3}-\d{3}-\d{4}

## Creating REGEX objects.

First we need to import the regex module by simply:

*import* re

Parsing a string value representing your regular expression to re.compile() returns a Regex pattern (or simply a regex object)

To create a a Regex object that matches the phone number pattern, enter a the following code into the shell.

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

In python escape characters are use backslash ‘\n’ – this \n represent a new line for each character. We add in ‘r’ before our string so we don’t have to add two backslashes in front of each character.

‘\d\d\d’ instead of ‘\\d\\d\\d’

**The ‘r’ before the string in python means raw input.**

## Matching REGEX objects

The 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 then it will return the match object.

phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')  
mo = phoneNumRegex.search('My number is 415-555-5125.')  
print('Number found: ' + mo.group())

mo variable stands for matched objects. The group() method will return the actual matched text (we will explain groups shortly).

## Quick review of steps

1. Import the regex module (import re)
2. Create a Regex objects with re.compile() – Remember to use raw string by using ‘r’
3. Pass the string into a search() method 🡪 this returns a Match object.
4. Call the match objects group() method to return a string of actual text

Here is another example using a shortcut with a different type of number

phone\_regex = re.compile(r'\d{10}')  
mo = phone\_regex.search('Numarul meu de telefon este: 0762241497.')  
print("Number found: " + mo.group())

re.compile(r'\d{10}') # this does /d \* 10

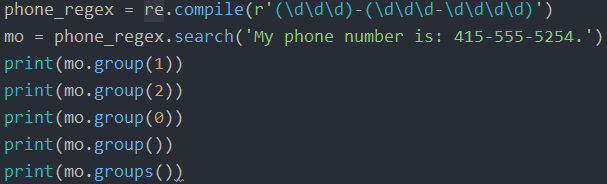
## More pattern matching with Regular Expressions.

Here we are going to try some more powerful pattern-matching capabilities of the re module.

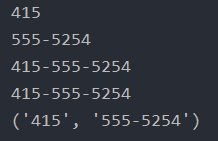
We will create group in the regex by adding parantheses.

(\d\d\d)-(\d\d\d-\d\d\d\d)

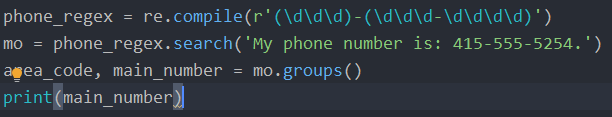
Then we can use the group() to grab the matching text from just one group, the first set of parantheses will resemble our first group.



This will print the following things.



We can also assign variables to this group by



Since re.compile() will return a tuple we will assign to variables to mo.groups() . The first one being area\_code and the second one being the main\_number.

Printing main number will get us the second group (555-5254)

If we want to use a different number format like:

('My phone number is: (415) 555-5254.')

We could do it this way

phone\_regex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')  
mo = phone\_regex.search('My phone number is: (415) 555-5254.')

## 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.

If you want to match an actual pipe character just type “\|”

For example, the regular expression r’Batman|Tina Fey’ will match either “Batman” or “Tina Fey”.

>>> **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'

We can also add several more patterns to the search part of a regex for example:

bat\_regex = re.compile(r'Bat(man|mobile|copter|bat)')  
bat\_search = bat\_regex.search('Batmobile lost a wheel')  
print(bat\_search.group()) # returns Batmobile

print(bat\_search.group(1)) # returns mobile

This expression will match one of the expression in the list, the one that appears first if you type

'Batmobile lost a wheel but batcopter is still going'

It will not match batcopter

## Optional matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally.

The ? character flags the group that precedes it as an optional part of the pattern.

For example:

bat\_regex = re.compile(r'Bat(wo)?man')  
mo1 = bat\_regex.search('The Adventures of Batman')  
print(mo1.group())  
  
mo2 = bat\_regex.search('The Adventures of Batwoman')  
print(mo2.group())

This will be the output:



The (wo)? Part of the regular expression marks that the pattern wo is optional

## Marking Zero or More with the Star

The \* 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.

bat\_regex = re.compile(r'Bat(cunt)\*man')  
mo1 = bat\_regex.search('The Adventures of Batcuntcuntcuntcuntman')  
print(mo1.group())

The ‘cunt’ word is still optional so if you pass in just Batman it will work just fine.

bat\_regex = re.compile(r'Bat(cunt)\*man')  
mo1 = bat\_regex.search('The Adventures of Batman')  
print(mo1.group())

If you want to match an actual start you just have to type ‘\\*’

Like this :

mo1 = bat\_regex.search('The Adventures of Bat\*\*\*\*\*\*\*man')

## Matching one or more with the + sign

While \* means MATCH ZERO OR MORE , the + sign means Match one or more.

This \* needs something to appear in the matched string otherwise python will throw you an error, the group preceding group must appear atleast once

bat\_regex = re.compile(r'Bat(wo)+man')  
mo1 = bat\_regex.search('The Adventures of Batwoman')

or

bat\_regex = re.compile(r'Bat(wo)+man')  
mo1 = bat\_regex.search('The Adventures of Batwowowowoman')

## Making specific repetitions using Curly Brackets

If you have a group that you want to repeat a specific number of times. For example the regex (Ha){3} will match the string ‘HaHaHa’ but will not match the string ‘HaHa’

Instead of one number you can specify a range by writing {3,5} where 3 stands for minimum and 5 stands for maximum (don’t forget the comma)

So (Ha){3,5} will match ‘HaHaHa’, ‘HaHaHaHa’, ‘HaHaHaHaHa’.

You can also leave all the minimum and maximum number unbounded like (Ha){,5}

Examples:

bat\_regex = re.compile(r'(HA){3}')  
mo1 = bat\_regex.search('LOLMAXOR HAHAHA')

This will match HAHAHA but it will not match just ‘HA’

bat\_regex = re.compile(r'(HA){1,5}')  
mo1 = bat\_regex.search('LOLMAXOR HA')

And this will return a matched cased for “HA, HAHA, HAHAHA until 5xHA”

## Greedy and Nongreedy matching

Python regular expresions are set to greedy by default which means it will return the longest string of ‘HA’s in our text.

The use the non-greedy version we are going to put a question mark after our compile raw string  
bat\_regex = re.compile(r'(HA){1,5}?')  
mo1 = bat\_regex.search('LOLMAXOR HAHAHA')

Now this will return us the shortest found matched case which in our case is ‘HA’

## The findall() method

In addition to search() we also have findall() . Where search() gives us the first found match , findall() will give us a list of strings with the matched cases.

Example:

phone\_numbers = re.compile(r'\d\d\d\d\d\d\d\d\d\d')  
matched = phone\_numbers.findall('07602414970760241498')

The result should look like this:



We can also group the numbers the following way:

phone\_numbers = re.compile(r'(\d\d\d\d)(\d\d\d\d\d\d)')  
matched = phone\_numbers.findall('07602414970760241498')

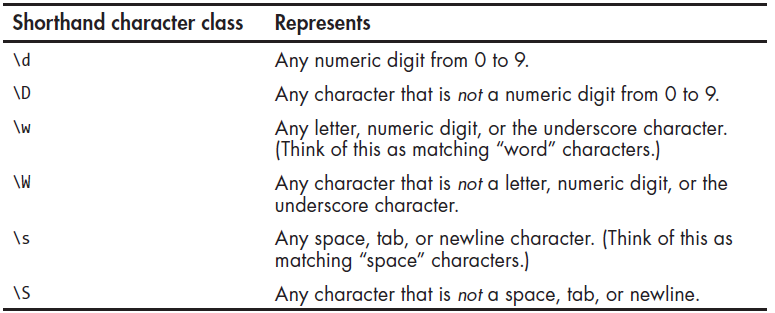
And the result should look like this:



## Character classes

Previously we just used ‘\d’ which is a shorthand for the regular expression: 

There are more shorthands mentioned below.



## Making your own Character classes

You can make your own character classes with the following example:

vowel\_regex = re.compile(r'[aeiouAEIOU]')  
match\_obj = vowel\_regex.findall('What the fuck Jenny? Are you Geh?')  
print(match\_obj)

These lines of code will find all the vowels in the string we put in and it will return:

['a', 'e', 'u', 'e', 'A', 'e', 'o', 'u', 'e']

If we add ^ to the compile it will print us the negative characters. Everything that is not ‘aeiouAEIOU’

vowel\_regex = re.compile(r'[^aeiouAEIOU]')

and it will look like this:

['W', 'h', 't', ' ', 't', 'h', ' ', 'f', 'c', 'k', ' ', 'J', 'n', 'n', 'y', '?', ' ', 'r', ' ', 'y', ' ', 'G', 'h', '?']

## The wildcard character

The . (or dot) in regex is called a wildcard and will match any character except for a new line. Example:

at\_regex = re.compile(r'.at')  
text = at\_regex.findall('The cat in the hat sat on the flat mat.')  
print(text)

This will result in the following list:



## Matching everything with Dot-Star.

Example:

name\_regex = re.compile(r'First Name: (.\*) Last Name: (.\*)')  
matched = name\_regex.findall('First Name: Daniel Last Name: Ghirasim')

and the result will be :



The dot-star uses greedy mode so it will always try to match as much text as possible to use it in a non greedy mode we are going to use the next example