Regular Expressions (re) in Python

Regex

Problem

Write a Python code to check if the given mobile number is valid or not. The conditions to be satisfied for a mobile number are:

a) Number of characters must be 10

b) All characters must be digits and must not begin with a ‘0’

Validity of Mobile Number

Input	Processing	Output
A string representing a mobile number	Take character by character and check if it valid	Print valid or invalid

Test Case 1

- Abc8967891

- Invalid
- Alphabets are not allowed

Test Case 2

- 440446845

- Invalid
- Only 9 digits

Test Case 3

- 0440446845

- Invalid
- Should not begin with a zero

Test Case 4

- 8440446845

- Valid
- All conditions satisfied

Python code to check validity of mobile number (Long Code)

```
import sys
number = input()
if len(number)!=10:
    print ('invalid')
    sys.exit()
elif number[0]=='0':
    print ('invalid')
    sys.exit()
else:
    for chr in number:
        if (chr.isdigit()==False):
            print ('invalid')
            sys.exit()
print('Valid')
```

sys.exit()

- The sys.exit() function is used in Python to terminate a program after necessary cleanup (like closing files, releasing resources, or logging)
- You can pass an optional exit status code to indicate success or failure (with 0 indicating success and 1 indicating an error).
- The default exit status for sys.exit() in Python is 0.

- Manipulating text or data is a big thing

- If I were running an e-mail archiving company, and you, as one of my customers, requested all of the e-mail that you sent and received last February, for example, it would be nice if I could set a computer program to collate and forward that information to you, rather than having a human being read through your e-mail and process your request manually.

- So this demands the question of how we can program machines with the ability to look for patterns in text.
- Regular expressions provide such an infrastructure for advanced text **pattern matching**, extraction, and/or search-and-replace functionality.
- **Python supports regexes** through the **standard library re** module -> **import re**

- regexes are **strings containing text and special characters** that describe a **pattern** with which to recognize multiple strings.
- Regexs without special characters

Regex Pattern	String(s) Matched
foo	foo
Python	Python
abc123	abc123

- These are simple expressions that match a single string
- Power of regular expressions comes in when meta characters and special characters are used to define **character sets, subgroup matching, and pattern repetition**

- Metacharacters are special characters in regular expressions that have specific meanings and functionalities beyond their literal representation. They help define patterns for matching strings. Here’s a rundown of some commonly used metacharacters in regex.

. ^ $ * + ?

{n} {n,} {n,m}

[ ] | ()

Quantifier	Matches
*	0 or more
?	0 or 1
+	1 or more
{n}	exactly n times
{n, }	n or more times
{n, m}	n to m times inclusive

Metacharacter	Purpose
\w	[a-zA-Z0-9_] word characters
\s	whitespace characters
\d	[0-9] digit characters
\W	[^a-zA-Z0-9_] non-word characters
\S	non-whitespace characters
\D	[^0-9] non-digit characters

- **Note** Some of the whitespace character are space/tab/new line

The findall() Function

- The findall() function returns a list containing all matches.
- **SYNTAX** re. findall(pattern,source_string)

- Return an empty list if no match was found

```
 import re
txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

```
['ai', 'ai']
```

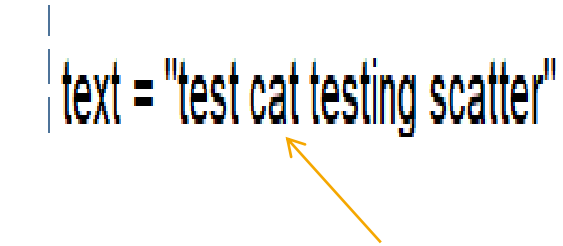
`\b` – Matches at word boundaries (between word and non-word characters).

`\B` – Matches where there is **no** word boundary (between two word characters or two non-word characters).

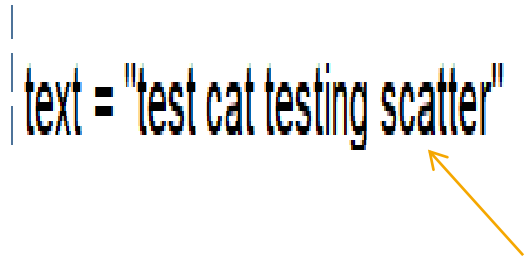
`\w` – Matches word characters (letters, digits, and underscore).

`\W` – Matches non-word characters (everything that is not a word character).

```
text = "test cat testing scatter"
# \b Matches 'cat' at word boundaries
pattern_b = r"\bcat\b"
print(re.findall(pattern_b, text))
# ['cat']
```



```
# \B Matches 'cat' where it is not at a word boundary
pattern_B = r"\Bcat\B"
print(re.findall(pattern_B, text))
# ['cat'] (from "scatter")
```



The search() Function

- The search() function searches the string for a match, and returns a Match object if there is a match.
- If there is more than one match, only the first occurrence of the match will be returned.
- If no matches are found, the value None is returned.
- **SYNTAX** re.search(pattern,source_string)

Code to search for the first white-space character in the string:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)

print(type(x)
print("The first white-space character is located in position:", x.start())
```

```
<class 're.Match'>

The first white-space character is located in position: 3
```

The match() Function

- **re.match(pattern, string)**: This function checks for a match only at the **beginning** of the string. If the pattern matches the start of the string, it returns a match object; otherwise, it returns None.

  SYNTAX re.match(pattern,source_string)

```
txt ="The rain in Spain"
x = re.match("The",txt)
print(type(x))
print(x.start())


<class 're.Match'>
0 #since The is present in beginning of the string


import re
txt ="The rain in Spain"
x = re.match("aThe",txt)
print(x)
if x is not None:
    print(x.start())
#
```

re.match vs re.search

- Both return the first match of a substring found in the string.
- **re.match()** searches only from the beginning of the string and return match object if found. But if a match of substring is found somewhere in the middle of the string, it returns none.
- **re.search()** searches for the whole string even if the string contains multi-lines and tries to find a match of the substring in all the lines of string.

The split() Function

- The split() function returns a list where the string has been split at each match
- **SYNTAX** re.split(pattern,source_string,maxsplit)

maxsplit is optional argument.

- Split at each white-space character:
- ```
  import re
  txt = "The rain in Spain"
  x = re.split("\s", txt)
  print(x)
  ```
- ['The', 'rain', 'in', 'Spain']
  ```

The split() Function

- You can control the number of occurrences by specifying the maxsplit parameter:
- Split the string only at the first occurrence:
- ```
  import re
  txt = "The rain in Spain"
  x = re.split("\s", txt, 1)
  print(x)
  ```
- ['The', 'rain in Spain']
  ```

The sub() Function for substitution

- The sub() function replaces the matches with the text of your choice:
- **SYNTAX** re.sub(old_pattern, new_pattern, source_string, no. of replacements)

  no. of replacements is optional

- Replace every white-space character with the number 9:
- ```
  import re
  txt = "The rain in Spain"
  x = re.sub("\s", "9", txt)
  print(x)
  ```
- The9rain9in9Spain
  ```

- You can control the number of replacements by specifying the count parameter:
- Replace the first 2 occurrences:
- ```
  import re
  txt = "The rain in Spain"
  x = re.sub("\s", "9", txt, 2)

    #2 is no. of occurences to be replaced
  print(x)
  ```
- The9rain9in Spain
  ```

Basic features of the re module

- **Matching Patterns**: You can check if a string matches a pattern using re.match() or re.search().
- **Finding Patterns**: Use re.findall() to find all occurrences of a pattern in a string.
- **Substituting Patterns**: Use re.sub() to replace occurrences of a pattern with another string.

- In Python's re module, when you find a match using methods like re.search() or re.match(), you get a match object. This match object provides several methods and attributes that allow you to get more information about the match.

Functions in match object

Method/Attribute	Description
`start()`	Returns the starting index of the match.
`end()`	Returns the index of the character after the match.
`span()`	Returns a tuple (start, end) of the match.
`group()`	Returns the matched substring.
`groups()`	Returns a tuple of all captured groups.
`groupdict()`	Returns a dictionary of named capturing groups.

groups()

- A *groups()* expression returns a tuple containing all the subgroups of the match.
  import re

```
m = re.match(r'(\w+)@(\w+)\.(\w+)','username@hackerrank.com')

print(m.groups())

#Output ('username', 'hackerrank', 'com')
```

groupdict()

?P<name> is a syntax used to define **named capturing groups**.

You can replace name with any valid identifier.

```
import re

m = re.match('(?P<user>\w+)@(?P<website>\w+)\.(?P<extension>\w+)','myname@hackerrank.com')

print(m.groupdict())
```

```
{'user': 'myname', 'website': 'hackerrank', 'extension': 'com'}
```

groupdict()

```
if m:
    print("Full name:", m.group('user'))
    print("website:", m.group('website'))
#Ouput
Full name: myname
website: hackerrank
```

(.)

- **dot or period (.) symbol** (letter, number, whitespace (not including “\n”), printable, non-printable, or a symbol) **matches any single character except for \n**
- To specify a dot character explicitly, you must escape its functionality with a **backslash**, as in **“\.”**

- The caret symbol ^ in regular expressions serves two primary functions, depending on its context.

1. Start of a String

When placed at the beginning of a regex pattern, the caret ^ asserts that the following pattern must match at the **start** of the string.

2. Negation in Character Classes

When used inside square brackets [...], the caret ^ negates the character class, meaning it matches any character **not** included in the brackets.

Matching from the Beginning or End of Strings or Word Boundaries (^, $)

^ - **Match beginning** of string

$ - **Match End** of string

Regex Pattern	Strings Matched
^From	Any string that starts with From
/bin/tcsh$	Any string that ends with /bin/tcsh
^Subject: hi$	Any string consisting solely of the string Subject: hi

if you wanted to match any string that **ends with a dollar sign**, one possible regex solution would be the pattern **.*\$$**

fullmatch

#re.fullmatch(pattern, string) checks if **the entire string** matches the given regex pattern.

Write a Python program that takes three space-separated inputs: name, age, and income.
•Validate that income is a number containing 5 to 8 digits.
•If the validation passes, print the three values on separate lines; otherwise, print "Invalid input".

```
import re

name, age, income = input("Enter name age income: ").split()

# Validate income: 5 to 8 digits
if re.fullmatch(r"\d{5,8}", income):
    print(name)
    print(age)
    print(income)
else:
    print("Invalid input")
```

Write a Python program to compare two strings **only if** they contain **only lowercase English letters (a–z)** and are **non-empty**.
If both strings satisfy the condition, compare them in **lexicographic order**.

Print "First" if the first string is lexicographically greater than the second.

Otherwise, print "Second".

If either string contains uppercase letters, digits, special characters, or is empty, print:

“One or both strings contain non-letter characters or upper case characters or are empty”

```
import re

str1 = 'python'
str2 = 'cplusplus'

pattern = r'[a-z]+'

if re.fullmatch(pattern, str1) and re.fullmatch(pattern, str2):
    print("First" if str1 > str2 else "Second")
else:
    print("One or both strings contain invalid characters")
```

```
import re

text = "123\n"  # Note: newline at the end

# Using fullmatch
print("fullmatch:", re.fullmatch(r"\d+", text))

# Using ^...$ with match
print("^...$ match:", re.match(r"^\d+$", text))
'''
```

re.fullmatch(r"\d+", text) → Fails because the newline at the end is part of the string, so \d+ doesn’t cover the whole string.

re.match(r"^\d+$", text) → Passes because in regex, $ matches the end of the string OR right before a newline. So "123\n" still matches.

'''

Denoting Ranges (-) and Negation (^)

- **brackets** [] also support **ranges of characters**
- A **hyphen [a-z] between a pair of symbols** enclosed in brackets is used to indicate a range of characters;
- **For example: A–Z, a–z, or 0–9 or 1-9** for uppercase letters, lowercase letters, and numeric digits, respectively

Regex Pattern	Strings Matched
`z.[0-9]`	"z" followed by any character then followed by a single digit
`[r-u][env-y]` `[us]`	"r," "s," "t," or "u" followed by "e," "n," "v," "w," "x," or "y" followed by "u" or "s"
`[^aeiou]`	A non-vowel character (Exercise: why do we say "non-vowels" rather than "consonants"?)
`[^\t\n]`	Not a TAB or \n
`["-a]`	In an ASCII system, all characters that fall between "" and "a," that is, between ordinals 34 and 97

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

Multiple Occurrence / Repetition

Using Closure Operators (*, +, ?, {})

- **special symbols *, +, and ? ,** all of which can be used to match single, multiple, or no occurrences of string patterns
- **Asterisk or star operator (*)** - **match zero or more occurrences** of the regex immediately to its left
- **Plus operator (+)** - **Match one or more occurrences** of a regex

- **Question mark operator (?) :** match **exactly 0 or 1 occurrences** of a regex.

- There are also **brace operators ({})** with either a **single value or a comma-separated pair of values**. These indicate a **match of exactly N occurrences** (for {N}) or a range of occurrences; **for example**, **{M, N}** will **match from M to N occurrences.**

Regex Pattern	Strings Matched
`[dn]ot?`	"d" or "n," followed by an "o" and, at most, one "t" after that; thus, do, no, dot, not.
`0?[1-9]`	Any numeric digit, possibly prepended with a "0." For example, the set of numeric representations of the months January to September, whether single or double-digits.
`[0-9]{15,16}`	Fifteen or sixteen digits (for example, credit card numbers.

re	Valid	Invalid
[dn]ot?	dot, not, do, no	dnt, dn,dott,dottt
[dn]ot?$	dot, not, do, no	dnt, dn,dott,nott
[dn]ot*$	dott,nott,do,no	dotn
[dn]ot*	Do,no,dot,dottt	
[0-9]{2,5}	12, 123, 1234, 12345, a123456	1,2,4,5,
[0-9]{5}	12345, 123456, a1234567,111111111	12, 123, 1234
aX{2,4}	aXX,aXXX,aXXXX	a ,aX

logical “or”

- The | denotes "OR" operator.
- |. This character separates terms contained within each (...) group.
- example, for instance:

^I like (dogs|penguins), but not (lions|tigers).$

- This expression will match any of the following strings:
- I like dogs, but not lions.
- I like dogs, but not tigers.
- I like penguins, but not lions.
- I like penguins, but not tigers.

metacharacter \d

- You can replace [0-9] by metacharacter \d, but not [1-9].
- Ex.

[1-9][0-9]*|0 **or** **[1-9]\d*|0**

- For "abc123xyz", it matches the substring "123".
- For "abcxyz", it matches nothing.
- For "abc123xyz456_0", it matches substrings: "123", "456" and "0" (three matches).
- For "0012300", it matches substrings: "0", "0" and "12300" (three matches)!!!

^[1-9][0-9]*|0$ or ^[1-9]\d*|0$

- The *position anchors* ^ and $ match the beginning and the ending of the input string, respectively. That is, this regex shall match the entire input string, instead of a part of the input string (substring).
- an *occurrence indicator*, + for one or more, * for zero or more, and ? for zero or one.

- Write a Python program that:
- Takes input from the user.
- Checks whether the entered value is truthy or falsy as a **string** (non-empty strings are truthy, empty strings are falsy).
- If the input consists only of digits, convert it to an integer and check whether it is truthy or falsy as an **integer** (non-zero integers are truthy, zero is falsy).
- Display appropriate messages for each case.

```
import re

value = input("Enter something: ")

# Check truthiness as a string
print("String is", "truthy" if value else "falsy")

# Check if it's all digits
if re.fullmatch(r"\d+", value):
    num = int(value)
    print("Integer is", "truthy" if num else "falsy")
```

metacharacter \w

- *metacharacter* **\w** for a word character

  [a-zA-Z0-9_].

  Recall that *metacharacter* **\d** can be used for a digit **[0-9]**.

  [a-zA-Z_][0-9a-zA-Z_]* or [a-zA-Z_]\w*

- Begin with one letters or underscore, followed by zero or more digits, letters and underscore.

metacharacter \s

- \s (space) matches any single whitespace like blank, tab, newline
- The uppercase counterpart \S (non-space) matches any single character that doesn't match by \s
- # Sample string
- text = "Hello, this is an example string.\nIt has multiple lines and spaces."

```
# Pattern to find a whitespace character followed by 'example'
pattern = "\s(example)"
print(re.search(pattern, "Hello, this is an example string.\nIt has multiple lines and  spaces."
“))
```

```
<re.Match object; span=(17, 25), match=' example'>
```

[Uppercase]*metacharacter*

- In regex, the uppercase metacharacter denotes the *inverse* of the lowercase counterpart, for example, \w for word character and \W for non-word character; \d for digit and \D or non-digit.

Escape code \

- The \ is known as the escape code, which restore the original literal meaning of the following character. Similarly, *, +, ? (occurrence indicators), ^, $ (position anchors), \. to represent . have special meaning in regex.
- In a character class (ie.,square brackets[]) any characters except ^, -,] or \ is a literal, and do not require escape sequence.
- ie., only these four characters, ^, -,] or \ , require escape sequence inside the bracket list: ^, -,], \
- Ex: **[.-]?** matches an optional character . or -.

Escape code \

- Most of the special regex characters lose their meaning inside bracket list, and can be used as they are; except ^, -,] or \.
  - To include a], place it first in the list, or use escape \].
  - To include a ^, place it anywhere but first, or use escape \^.
  - To include a - place it last, or use escape \-.
  - To include a \, use escape \\.
  - No escape needed for the other characters such as ., +, *, ?, (,), {, }, and etc, inside the bracket list
  - You can also include metacharacters such as \w, \W, \d, \D, \s, \S inside the bracket list.

Escape code \

re.findall('[ab\-c]', '123-456')

Output ['-']

re.findall('[a-c]', 'abdc')

Output ['a', 'b', 'c']

Flags

- re.DOTALL (or re.S)
- re.MULTILINE (or re.M)
- re.IGNORECASE flag (or re.I)

re.DOTALL (or re.S)

- Normally, the dot (.) in regex matches any character *except* a newline. For example, if you want to match text across multiple lines (which includes newline characters), the dot will not work unless you enable the DOTALL flag.
- The DOTALL flag allows the dot (.) in a regular expression to match *any* character, including newline characters (\n).

```
text = "Hello\nWorld"
pattern = ".*"
print(re.match(pattern, text))
# Matches only "Hello“
text = "Hello\nWorld"
pattern = ".*"
print(re.match(pattern, text, re.DOTALL))
# Matches "Hello\nWorld"
```

re.MULTILINE (or re.M)

- The MULTILINE flag changes the behavior of the ^ and $ anchors so that they match at the start and end of each line, not just the start and end of the entire string.
- Without MULTILINE, ^ matches the beginning of the string, and $ matches the end of the string.

```
text = "Hello\nWorld"
pattern = "^World$"
print(re.search(pattern, text))
# No match because "World" is not at the start of the text
text = "Hello\nWorld"
pattern = "^World$"
print(re.search(pattern, text, re.MULTILINE))
# Matches "World" because it's at the start of a new line
```

re.IGNORECASE (or re.I)

- In Python's re module, the re.IGNORECASE flag (also written as re.I) is used to make the regular expression case-insensitive. When this flag is applied, the regex will match letters regardless of whether they are uppercase or lowercase.

```
text = "Hello World"
pattern = "hello"
match = re.search(pattern, text)
print(match)
# No match, because "hello" does not match "Hello" (case-sensitive)
text = "Hello World"
pattern = "hello"
match = re.search(pattern, text, re.IGNORECASE)
print(match)
# Match found, because "hello" matches "Hello" (case-insensitive)
```

Raw string r

```
valid = re.match('[a-zA-z]{2}\.[a-zA-Z]{2}$', input())
print("valid" if valid else "invalid")
#OUTPUT WARNING
#SyntaxWarning: invalid escape sequence '\.'
#Solution- use r (raw string)
valid = re.match(r'[a-zA-z]{2}\.[a-zA-Z]{2}$', input())
print("valid" if valid else "invalid")
#Alternate solution - use double back slash
valid = re.match(r'[a-zA-z]{2}\\.[a-zA-Z]{2}$', input())
print("valid" if valid else "invalid")
```

Lookaheads

Positive and negative lookaheads in regular expressions (regex) are zero-width assertions used to check for the presence or absence of a pattern after the current matching position without including that pattern in the final match. They are "zero-width" because they don't consume any characters themselves.

Positive Lookahead

- **Purpose:** The positive lookahead (?=...) asserts that the pattern inside the parentheses must follow the current position for a match to occur. However, the characters matched by the lookahead pattern are not included in the overall match result.
- **Syntax:** X(?=Y) where X is the pattern to be matched, and Y is the pattern that must follow X.
- **Example:** To match "apple" only if it is followed by "pie", but only return "apple":

  apple(?=pie)

In the string "applepie", this would match "apple". In "applejuice", it would not match anything.

Negative Lookahead

- **Purpose:**

The negative lookahead (?!...) asserts that the pattern inside the parentheses must not follow the current position for a match to occur. Similar to positive lookahead, the characters checked by the lookahead are not included in the overall match.

- **Syntax:**

X(?!Y) where X is the pattern to be matched, and Y is the pattern that must not follow X.

- **Example:**

To match "cat" only if it is not followed by "erpillar", but only return "cat":

  - cat(?!erpillar)
  - In the string "cat", this would match "cat". In "caterpillar", it would not match anything.

Try out

1.Check whether the given register number of a VIT student is valid or not.

Register number is valid

If it has two digits (must not start with 0)

Followed by three letters

Followed by four digits

Example of valid register number – 17BME1001, 18bme2001, 21Bme1006

Sample Input 1
20bce2023

Sample Output 1

valid

Sample Input 2
02bce2023

Sample Output 2

Try out

2. Check the validity of mobile number.

The mobile number must

*start with a digit except 0

*followed by 9 digits

3. check if the given pan no is valid.

*Length should be 10

*5 alphabets

*followed by 4 digits

*followed by 1 alphabet

Sample Input afzps7865m **Sample Output** valid

Sample Input afzps765m **Sample Output** invalid

Try out

4. Regex pattern for exactly 3 to 5 alphabets

Sample Input AsdDs **Sample Output** valid

Sample Input eeew2 **Sample Output** invalid

5. check if the given email id is valid

- **Sample Input** ss@ee.com **Sample Output** valid
- **Sample Input** ss@e@e.com **Sample Output** invalid

6. Extract url part from the given string and display in a list.

Sample Input

"Check out https://examples.com and http://test.com for more info."

Sample Output

['https://example.com', 'http://test.com']

Note If no match, then display empty list i.e []

Try out

7. Find All Words(with only alphabets) in a String and display in list.

sample input This is my phone number 444-333-3333

sample output ['this', 'is', 'my', 'phone', 'number']