Regular Expressions:

Definition:

What is "**regular expressions**":
regular expressions (shortened as re or regex or regexp) is a sequence of character(s) mainly used to find and replace patterns in a string or file. They are supported by most of the programming languages like python, Java …etc. . So, learning them helps in multiple ways (later more).

The most common uses of regular expressions are:

Search a string(search and match).

Finding a string(findall).

Break string into a substring.

Replace part of a string(sub).

Regular expression rules:

[ ] is used to indicate a set of characters (express any specific letter within it.)
<u>Special characters lose their special meaning inside sets. For example, [(+*)] will match any of the literal characters '(', '+', '*', or ')'.</u>
If the char **-** is escaped (e.g. [a\-z]) or if it's placed as the first or last character (e.g. [-a] or [a-]), it will match a literal '-'.
To match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set.
For example, both [()[\]{}] and []()[{}] will both match a parenthesis.
So that means:
[a] : search for any **a** in text
[a d] : search for any **a OR d** in text
[a - d] :search for any **a or b or c or d**
[a - d A - D] :search for any **a or A or b or B or c or C or d or D**
[a - d A] : means find any **a or b or c or d or A**
[a A - D] : means find any **a or A or B or C or D**
[a - z A - Z] : means find any **letters both upper and lower case**

[0123456789]: means find **any number**(the same function with **[0-9]**)
[Aa]hmad: find **Ahmad** or **ahmad**
[Aa]hm[ae]d: find **Ahmad** or **ahmad** or **Ahmed** or **ahmed**
[Aa]hm[aeAE]d: find **AhmAd** or **Ahmad** or **AhmEd** or **Ahmed** or **ahmAd** or **ahmad** or **ahmEd** or **ahmed**
----

^ without the brackets ^ matches only at the beginning of the string (the opposite of '$' that matches only at the end of the string). (within [] is used to delete a specific character(negation))

^ has no special meaning if it's not the first character in the set.

that means:

[^ a] :find any thing **except a**.
[^ a d] :find any thing **except a and d**.
[^ a - d] :find any thing **except a and b and c and d**.
[^ a - z] : find any thing **except lower case letter**.
[^ a - z A - Z] : find any thing **except letters**.
[^ S s] : find any thing **except S and s**.
[^ S ^] : find any thing **except S and ^**.
^ s : find s at the beginning. (because ^s is not in [])
a^a: find **a^a**. (because a^a is not in [])
also:
```
print(re.findall("[^s]", "sad"))
```
```
Output:             ['a', 'd']
```
```
print(re.findall(r"^s", "sad"))
```
```
Output:            ['s']
```
```
print(re.findall("[s^s]", "s^sad"))
print(re.findall(r"s^s", "s^sad"))
```
```
Output:            ['s', '^', 's']        ||          []
```

? used to express that the preceding character optional, it may or may not come. That means:

Moham?mad : will find Mohamad and Mohamad (m comes just one time or doesn't come)
helps?: will find help and helps. **o?ps: will find ps and ops.**

(?=...): Matches if ... matches next, but doesn't consume any of the string. This is called a **lookahead** assertion. For example, Isaac (?=Asimov) will match 'Isaac ' only if it's followed by 'Asimov'.

(?!...): Matches if ... doesn't match next. This is a **negative lookahead** assertion. For example, Isaac (?!Asimov) will match 'Isaac ' only if it's not followed by 'Asimov'.

(?<=...): Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a **positive lookbehind** assertion. **(?<=abc)def** will find a match in 'abcdef', the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, that means, that abc or a|b are allowed, but a* and a{3,4} are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function:

```
comment= re.search('abc(?#after Hashtag it is no matter what comes)',
'abcdef')
print('assertion with comment= ',comment)

lookahead_assertion= re.search('abc(?=def)', 'abcdef')
print('lookahead_assertion= ', lookahead_assertion)

negative_lookahead_assertion= re.search('abc(?!xxx)', 'abcdef')
print('negative_lookahead_assertion= ',negative_lookahead_assertion)

positive_lookbehind_assertion= re.search('(?<=abc)def', 'abcdef')
print('positive_lookbehind_assertion= ',positive_lookbehind_assertion)
```

Output:

```
assertion with comment=  <re.Match object; span=(0, 3), match='abc'>
lookahead_assertion=  <re.Match object; span=(0, 3), match='abc'>
negative_lookahead_assertion=  <re.Match object; span=(0, 3), match='abc'>
positive_lookbehind_assertion=  <re.Match object; span=(3, 6), match='def'>
```

***** to match 0 or more repetitions of the preceding character that means: **o*ps: will find ps and ops and oops and ooops and ooooops ....**

+ to match at least one or more repetitions of the preceding character (**occurs one or more times** different from * that might not come but + means that the letter occurs **at least one time**) **o*ps: will find ops and oops and ooops and ooooops ... (but it won't find ps like in *)**

{n} is used to express that the preceding letter occurs exactly n-times.As example: \D{3} will find three consecutively non digits like abc or sdf "$% …

{n,m} is used to express that the preceding letter occurs between n-times and m-times .As example: \D{3,6} will find three four five or six consecutively non digits like abc, sedf, "$%(), asdfgh …

{m,n}?: Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the **non-greedy** version of the previous quantifier. For example, on the 6-character string 'aaaaaa', a{3,5} will match 5 'a' characters, while a{3,5}? will only match 3 characters.

```
print(re.findall(r"a{3,5}", "aaaaa"))
print(re.findall(r"a{3,5}?", "aaaaa"))
```

Output: ['aaaaa'] || ['aaa']

{n, } is used to express that the preceding letter occurs more than n-times .As example: \D{3,} will find more than three consecutively non digits like abc or sedf “$%() asdfgh asdghjk asdfghjk etc…

. : (Dot.) this matches any character except a newline (except the space for the new line) as example **beg.n** : will find **begin begun begon begqn beg4n beg@n …**

\. : Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning(with their ordinary meaning). **beg\.n** : will just find **beg.n**

There is second way to handle special characters as ordinary characters is with the so called **Python’s raw string notation**. Backslashes are not handled in any special way in a string literal prefixed with 'r'. So r"\n" is a two-character string containing '\' and 'n', while "\n" is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

$ is used to express the end of the word as example: **\ . $** : the point . means any char but slash\ before the point means the char point . (not any char) $ means that char that is before $ must be at the end of text. So it will find . in Hello. Because . is at end of the text.

| : A|B will match either A or B(where A and B can be arbitrary REs).

(….group…): Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group. As example: re.search(r'ab', 'ab') will give you just one match(ab) but with re.search(r'(a)(b)', 'ab') will give you three matches(ab, a and b).

\d: find a **digit** as example: \d\d-\d will find digits like 22-3, 32-3, 43-2… etc.

\D: find a **non digit** as example: \D\D-\D will find non digits like Ab-c, @b-c, aa-a… etc.

\w: Matches Unicode word characters, this includes characters, numbers and the underscore [a-zA-Z0-9_]. (**Alphanumeric**). As example: \w -\w\w\w will find alphanumeric like A-B_C , 1-2b_ , @-@a2 … etc.

\W: Matches the opposite of \w [^a-zA-Z0-9_] characters which are neither alphanumeric nor the underscore. (**Non-Alphanumeric**).As example: \W-\W\W\W will find alphanumeric like @-“$% , &-)(/ , @-(/& … etc.

\b: Matches the empty string, but only at the beginning or end of a word. <u>\b is the boundary between \w and \W character (or vice versa (\W and \w))</u> or between \w and the beginning/end of the string. This means that r'\bfoo\b' matches 'foo', 'foo.', '(foo)', 'bar foo baz' but not 'foobar' or 'foo3'.

```
print(re.findall(r'\b[aeh]\w+','abc efg hij klm nop qrs')) # search for the first letter in words
print(re.findall(r'\w+[mps]\b','abc efg hij klm nop qrs')) # search for the last letter in words
```

Output: ['abc', 'efg', 'hij'] || ['klm', 'nop', 'qrs']

\B: Matches the empty string, but only when it is not at the beginning or end of a word. This means that r'py\B' matches 'python', 'py3', 'py2', but not 'py', 'py.', or 'py!'.

\s: find a **white space** (which includes [\t\n\r\f\v]). As example a\sb\sc will find a b c

---

\S: find a **Non-whitespace**. As example a\Sb\Sc will find a2bxc

---

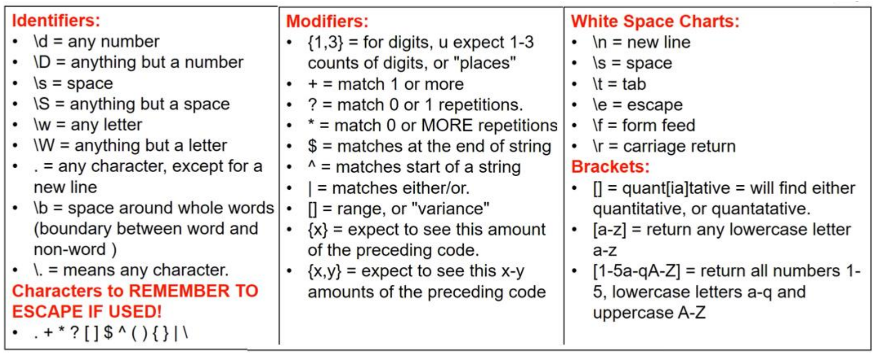
\A: Matches only at the start of the string.

\Z: Matches only at the end of the string.

Verbose: #: allow us to add comments and extra whitespaces within the regular expression. Don’t forget to add **re.VERBOSE or re.X** also as example: bevor verbose (re.compile(r"\d+\.\d*") →after:

```
a = re.compile(r"""\d +  # the integral part
             \.    # the decimal point
             \d *  # some fractional digits""", re.X)
```

Quick view at the re-rules:



Identifiers:

- \d = any number
- \D = anything but a number
- \s = space
- \S = anything but a space
- \w = any letter
- \W = anything but a letter
- . = any character, except for a new line
- \b = space around whole words (boundary between word and non-word)
- \. = means any character.

Characters to REMEMBER TO ESCAPE IF USED!

- . + * ? [ ] $ ^ ( ) { } | \

Modifiers:

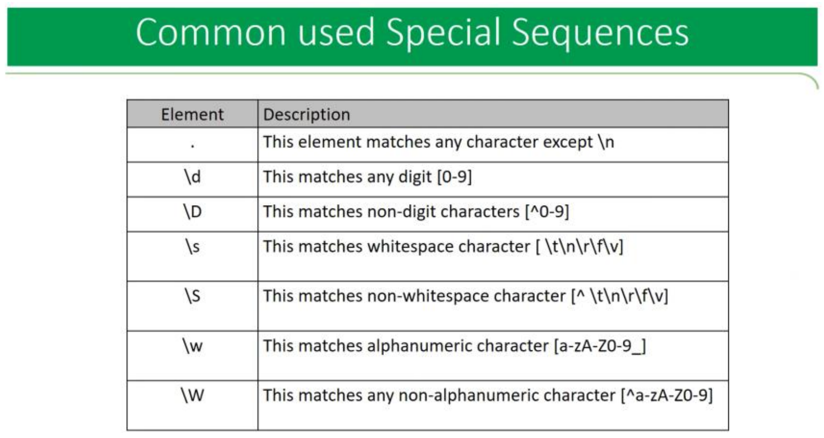
- {1,3} = for digits, u expect 1-3 counts of digits, or "places"
- + = match 1 or more
- ? = match 0 or 1 repetitions.
- * = match 0 or MORE repetitions
- $ = matches at the end of string
- ^ = matches start of a string
- | = matches either/or.
- [] = range, or "variance"
- {x} = expect to see this amount of the preceding code.
- {x,y} = expect to see this x-y amounts of the preceding code

White Space Charts:

- \n = new line
- \s = space
- \t = tab
- \e = escape
- \f = form feed
- \r = carriage return

Brackets:

- [] = quant[ia]tative = will find either quantitative, or quantatative.
- [a-z] = return any lowercase letter a-z
- [1-5a-qA-Z] = return all numbers 1-5, lowercase letters a-q and uppercase A-Z



Common used Special Sequences

Element	Description
.	This element matches any character except \n
\d	This matches any digit [0-9]
\D	This matches non-digit characters [^0-9]
\s	This matches whitespace character [\t\n\r\f\v]
\S	This matches non-whitespace character [^ \t\n\r\f\v]
\w	This matches alphanumeric character [a-zA-Z0-9_]
\W	This matches any non-alphanumeric character [^a-zA-Z0-9]

Notice:

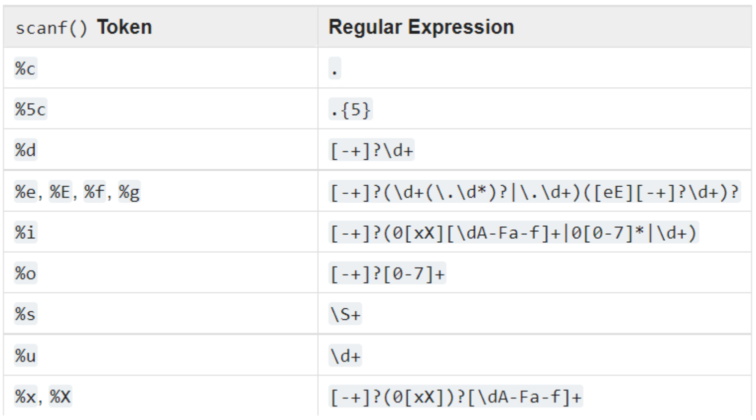
Raw String Notation

Raw string notation (r"text") keeps regular expressions sane. Without it, every backslash ('\') in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
re.match(r"\W(.)\1\W", " ff ")
re.match("\\W(.)\\1\\W", " ff ")
```

Simulating scanf():

Python does not currently have an equivalent to scanf(). Regular expressions are generally more powerful, though also more verbose, than scanf() format strings. The table below offers some more-or-less equivalent mappings between scanf() format tokens and regular expressions.



`scanf()` Token	Regular Expression
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e`, `%E`, `%f`, `%g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`
`%u`	`\d+`
`%x`, `%X`	`[-+]?(0[xX])?[\dA-Fa-f]+`

To extract the filename and numbers from a string like

/usr/sbin/sendmail - 0 errors, 4 warnings

you would use a scanf() format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

(\S+) - (\d+) errors, (\d+) warnings

The most commonly methods in the library re(regular expressions):

1- **re.match(pattern,string):** check for a match **only at the beginning** of the string. If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match. **If you want to locate a match anywhere in *string*, use search() instead**

   example:

```
import re
pattern = r"Awad"
string_1 = "Awad Soleman is my name."
string_2 = "My name is Awad Soleman."
string_3 = "awad soleman is my name."
string_4 = " Awad Soleman is my name."
```

```python
if re.match(pattern,string_1):print("pattern machtes with string_1")
else:
    print("pattern does't macht with string_1")

if re.match(pattern,string_2):print("pattern machtes with string_2")
else:print("pattern does't macht with string_2")

if re.match(pattern,string_3):print("pattern machtes with string_3")
else:print("pattern does't macht with string_3")

if re.match(pattern,string_4):print("pattern machtes with string_4")
else:print("pattern does't macht with string_4")
```

Output:

```
pattern machtes with string_1
pattern does't macht with string_2
pattern does't macht with string_3
pattern does't macht with string_4

Process finished with exit code 0
```

```python
print(re.match(pattern,string_1))
```

Output:

```
<re.Match object; span=(0, 4), match='Awad'>
```

```python
print(re.match(pattern,string_2))
```

Output:

```
None
```

re.match(pattern,string).group(x) give us all x matches results.(in our case we have just one match at the first of the string.

```python
print(re.match(pattern,string_1).group(0))
```

Output:

```
Awad
```

```python
print(re.match(pattern,string_1).start())
print(re.match(pattern,string_1).end())
```

Output:

```
0
4
```

0 is the position of the first letter of the match and 4 is of the last

2- **re.fullmatch**(pattern, string, flags=0)
If the whole string matches the regular expression pattern, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.

```python
print(re.fullmatch(r'pyhton', 'pyhton is easy'))
print(re.fullmatch(r'pyhton is easy', 'pyhton is easy'))
```

Output:

```
None
<re.Match object; span=(0, 14), match='pyhton is easy'>
```

3- **re.search()**: check for a match **anywhere** in the string:
re.search(pattern, string, flags=0): Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding match object. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

4- **re.findall()**: find all instances of words in a string (Returns a list containing all matches):
re.findall(pattern, string, flags=0):as example:
Return all non-overlapping matches of pattern in string, as a list of strings or tuples. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.
The result depends on the number of capturing groups in the pattern. If there are no groups, return a list of strings matching the whole pattern. If there is exactly one group, return a list of strings matching that group. If multiple groups are present, return a list of tuples of strings matching the groups. Non-capturing groups do not affect the form of the result.

```
print(re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest'))
print(re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10'))
```

Output:
```
['foot', 'fell', 'fastest']
[('width', '20'), ('height', '10')]
```

```
text = "He was carefully disguised but captured quickly by police."
print(re.findall(r"\w+ly\b", text))
```

Output: ['carefully', 'quickly']

5- **finditer()**: if one wants more information about all matches of a pattern than the matched text. Continuing with the previous example, if a writer wanted to find all of the adverbs and their positions in some text:

```
6- text = "He was carefully disguised but captured quickly by police."
   for m in re.finditer(r"\w+ly\b", text):
       print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
```

Output: 07-16: carefully || 40-47: quickly

7- **re.sub()**: re.sub(pattern, repl, string): search a pattern and replace with a new sub string.

```
8- print(re.sub(r'Spain','the world','Barcelona is the strongest
   football club in Spain.'))
```

Output: Barcelona is the strongest football club in the world.

9- **re.compile():** convert a pattern into objects:
re.compile(pattern, flags=0): Compile a regular expression pattern into a regular expression object, which can be used for matching using its match(), search() and other methods.
The expression's behavior can be modified by specifying a flags value. Values can be any of the following variables, combined using bitwise OR (the | operator). The sequence:
prog = re.compile(pattern)
result = prog.match(string)
is equivalent to
result = re.match(pattern, string)
but using re.compile() and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

10- **re.split**(pattern, string, maxsplit=0, flags=0)
Split string by the occurrences of pattern into multiple strings.

```
print(re.split(r's','Jahreszeit, Studiumsplaz, Bildungsort'))
```

Output: ['Jahre', 'zeit, Studium', 'plaz, Bildung', 'ort']

```
print(re.split(r'\s+','This text is splitet by spaces.'))
```

Output: ['This', 'text', 'is', 'splitet', 'by', 'spaces.']

```
print(re.split(r'\W+', 'Words, words, words.'))
```

Output:
```
['Words', 'words', 'words', '']
```

If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list.

```
print(re.split(r'(\W+)', 'Words, words, words.'))
```

Output:
```
['Words', ', ', 'words', ', ', 'words', '.', '']
```

With flag.ignorecase:

```
print(re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE))
```

Output:
```
['0', '3', '9']
```

If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.
If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```
print(re.split(r'\W+', '.Words, words, words.'))
print(re.split(r'(\W+)', '.Words, words, words.'))
```

Output:
```
['', 'Words', 'words', 'words', '']
['', '.', 'Words', ', ', 'words', ', ', 'words', '.', '']
```

```
print(re.split(r'\b', 'Words, words, words.'))
print(re.split(r'\W*', '...words...'))
print(re.split(r'(\W*)', '...words...'))
```

Output:
```
['', 'Words', ', ', 'words', ', ', 'words', '.']
['', '', 'w', 'o', 'r', 'd', 's', '', '']
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

- **search() vs. match()**: **re.match()** checks for a match only at the beginning of the string, while **re.search()** checks for a match anywhere in the string. For example:

re.match("c", "abcdef")   # No match

re.search("c", "abcdef") # Match

Regular expressions beginning with '^' can be used with search() to restrict the match at the beginning of the string:

re.match("c", "abcdef")   # No match

re.search("^c", "abcdef") # No match

re.search("^a", "abcdef") # Match

Note however that in MULTILINE mode match() only matches at the beginning of the string, whereas using search() with a regular expression beginning with '^' will match at the beginning of each line.

re.match('X', 'A\nB\nX', re.MULTILINE) # No match

re.search('^X', 'A\nB\nX', re.MULTILINE) # Match

Match objects always have a boolean value of True. Since match() and search() return None when there is no match, you can test whether there was a match with a simple if statement:

match = re.search(pattern, string)

if match:

    process(match)

and match has following methods and attribute:

1-**match.group([group1, ...])**

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero (the whole match is returned).

```
m= re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
```

If a groupN argument is **zero**, the corresponding return value is the entire matching string;

```
print(m.group(0))       # The entire match
```

Output: Isaac Newton

if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group.

```
print('m.group(1)= ',m.group(1))    # The first parenthesized subgroup.
print('m.group(2)= ',m.group(2))    # The second parenthesized subgroup.
```

Output:→   m.group(1)= Isaac || m.group(2)= Newton

```
print('m.group(1, 2)= ' ,m.group(1, 2))# Multiple arguments give a tuple.
```

Output: m.group(1, 2)= ('Isaac', 'Newton')

If the regular expression uses the (?P<name>...) syntax, the groupN arguments may also be strings identifying groups by their group name. Example:

```
m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
print('m.group(first_name)= ', m.group('first_name'))
print('m.group(last_name)= ', m.group('last_name'))
```

Output: m.group(first_name)= Malcolm || m.group(last_name)= Reynolds

Named groups can also be referred to by their index:

```
print('m.group(0)= ', m.group(0))
print('m.group(1)= ', m.group(1))
print('m.group(2)= ', m.group(2))
```

Output: m.group(0)= Malcolm Reynolds || m.group(1)= Malcolm || m.group(2)= Reynolds

If a group matches multiple times, only the last match is accessible:

```
m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
print('m.group(0)= ',m.group(0))
print('m.group(1)= ',m.group(1))
```

Output: m.group(0)= a1b2c3 || m.group(1)= c3

2- **match.__getitem__(g)**:

This is identical to m.group(g). This allows easier access to an individual group from a match:

```
m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
print('m[0]= ', m[0])        # The entire match
print('m[1]= ', m[1])        # The first parenthesized subgroup.
print('m[2]= ', m[2])        # The second parenthesized subgroup.
```

```
Output:         m[0]=  Isaac Newton    ||        m[1]=  Isaac      ||        m[2]=  Newton
```

Named groups are supported as well:

```
m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Isaac Newton")
print('m[first_name]= ', m['first_name'])
print('m[last_name]= ',m['last_name'])
```

```
Output:         m[first_name]=  Isaac    ||        m[last_name]=  Newton
```

3-**match.groups(default=None)**:

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to None.

```
m = re.match(r"(\d+)\.(\d+)", "24.1632")
print('m.groups()= ', m.groups())
print('m.group(0)= ', m.group(0))
print('m.group(1)= ', m.group(1))
print('m.group(1,2)= ', m.group(1,2))
```

```
Output:                 m.groups()=  ('24', '1632')       ||        m.group(0)=  24.1632              ||
                        m.group(1)=  24           ||        m.group(1,2)=  ('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to None unless the default argument is given:

```
m = re.match(r"(\d+)\.?(\d+)?", "24")
print('m.groups()=' , m.groups())      # Second group defaults to None.
print('m.groups(0)= ', m.groups('0'))    # Now, the second group defaults to
'0'.
```

```
Output:         m.groups()= ('24', None)          ||        m.groups(0)=  ('24', '0')
```

4- **match.groupdict(default=None)**:

Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name. The default argument is used for groups that did not participate in the match; it defaults to None.

```
m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
print('m.groupdict()= ', m.groupdict())
```

```
Output:         m.groupdict()=  {'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

5- **match.start([group]) || match.end([group])**:

Return the indices of the start and end of the substring matched by group:

```
print(m.start())
print(m.end())
print(m.string[m.start():m.end()])
```

```
Output:         0        ||        16        ||        Malcolm Reynolds
```

A good application for this removing a match in a string:

```
email = "tony@tiremove_thisger.net"
m = re.search("remove_this", email)
print(email[:m.start()] + email[m.end():])
```
Output: tony@tiger.net

Different codes:

-Search the string to see if it starts with "The" and ends with "Spain":

```
txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

- writing a poker program where a player's hand is represented as a 5-character string with each character representing a card, "a" for ace, "k" for king, "q" for queen, "j" for jack, "t" for 10, and "2" through "9" representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
valid = re.compile(r"^[a2-9tjqk]{5}$")
print(displaymatch(valid.match("akt5q")))    # Valid
print(displaymatch(valid.match("akt5e")))    # Invalid
print(displaymatch(valid.match("akt")))      # Invalid
print(displaymatch(valid.match("727ak")))    # Valid
```
Output: <Match: 'akt5q', groups=()> ||None||None || <Match: '727ak', groups=()>

- To match this with a regular expression, one could use backreferences as such:

```
pair = re.compile(r".*(.).*\1")
print(displaymatch(pair.match("717ak")))      # Pair of 7s.
print(displaymatch(pair.match("718ak")))      # No pairs.
print(displaymatch(pair.match("354aa")))      # Pair of aces.
```
Output: <Match: '717', groups=('7',)> || None || <Match: '354aa', groups=('a',)>

- To find out what card the pair consists of:

```
print(pair.match("717ak").group(1))
print(pair.match("718ak").group(1))
```
Output: 7 || Error

-Validate a phone number (must be of 10 digits and start with 8 or 9)?

```
def validation(number):
    if re.match(r'[8-9]{1}[0-9]{9}', number):
        print ("This number is a valid phone number.")
    else:
        print ("This number is NOT a valid phone number!")
validation('0123456789')
validation('9876543210')
```

Output: This number is NOT a valid phone number! || This number is a valid phone number.

-How to choose an email-address from a string?

```
string = 'here is the email address: awad-soleman@gmail.com send me a
mail.'
mail = re.search(r'[\w.-]+@[\w.-]+', string).group(0)
print(mail)
```

Output: awad-soleman@gmail.com

-find the number of class in this string 'he is in the fourth class':

```
print(re.search(r'\w+(?=\sclass)', 'he is in the fourth class').group(0))
```

Output: fourth

-give all words in string as a list:

```
print(re.findall(r'\w+','this is a string.')) # \w* returns spaces also.
```

Output: ['this', 'is', 'a', 'string']

-select the first letter of a string:

```
print(re.findall(r'\b\w','this is a string.'))
```

Output: ['t', 'i', 'a', 's']

-select domain name in websites:

```
print(re.findall(r'\w+.\w+.(\w+)','www.google.com www.google.de
www.google.net'))
```

Output: ['com', 'de', 'net']

-select a date of a string:

```
print(re.findall(r'\d{2}-\d{2}-\d{4}','12-12-2022 13:30, 01-12-2021 12:21
,02-02-2002 3:00'))
```

Output: ['12-12-2022', '01-12-2021', '02-02-2002']

-**Phonebook**:

```
text = """Ross McFluff: 834.345.1254 155 Elm Street

Ronald Heathmore: 892.345.3428 436 Finley Avenue
Frank Burger: 925.541.7625 662 South Dogwood Way


Heather Albrecht: 548.326.4584 919 Park Place"""
```

every entry is separated with a new line or more new lines.

```
entries = re.split("\n+", text)
```

we can separate every entry into five: elements first name, last name, number, house number address:

```
y=[re.split(":? ", entry, 4) for entry in entries]
print(y)
```

Output: [['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'], ['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'], ['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'], ['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
