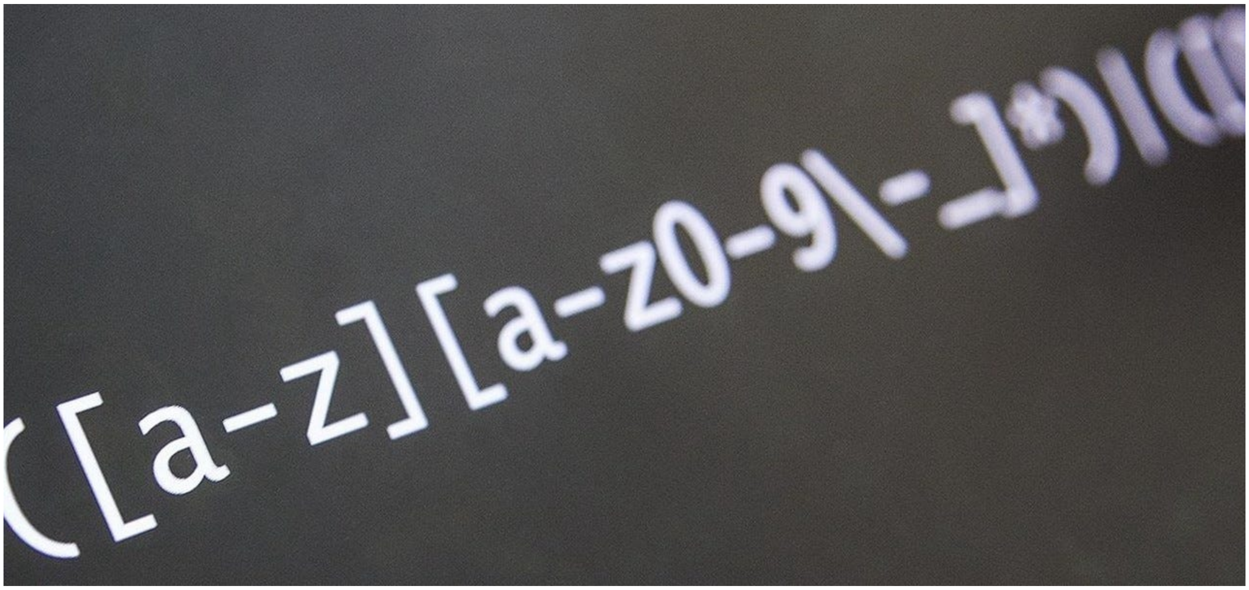
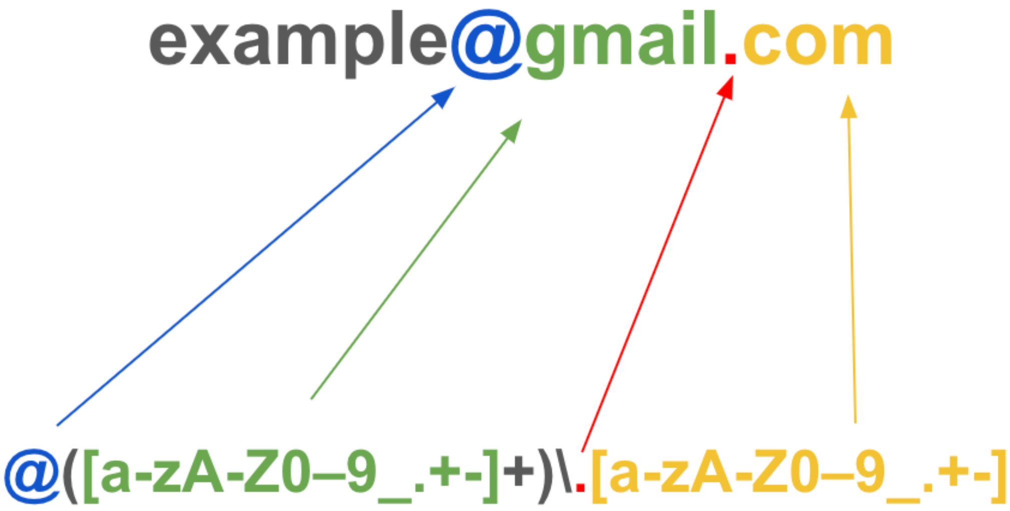
Regular Expression

A regular expression, often referred to as regex or regexp, is a sequence of characters that defines a search pattern. It is primarily used for pattern matching and manipulating strings.



- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.
- Regular expressions are supported by many programming languages and text editors, providing a powerful tool for working with text data.



Certainly! Here are some examples of regular expressions and their applications:

1. **Matching Email Addresses:**

   - Pattern: `\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b`

- This pattern matches a valid email address, ensuring it has the correct format with a username, domain, and top-level domain (TLD).

2. **Matching URLs:**

   - Pattern: `(https?|ftp)://[^\s/$.?#].[^\s]*`
   - This pattern matches a URL, including both HTTP and FTP protocols, allowing for various characters in the path and query parameters.

3. **Matching Dates in YYYY-MM-DD format:**

   - Pattern: `\b\d{4}-\d{2}-\d{2}\b`
   - This pattern matches dates in the format YYYY-MM-DD, such as "2023-06-07".

4. **Matching IP Addresses:**

   - Pattern: `\b(?:\d{1,3}\.){3}\d{1,3}\b`
   - This pattern matches IPv4 addresses, ensuring each segment is between 0 and 255.

5. **Extracting Phone Numbers:**

   - Pattern: `\b\d{3}-\d{3}-\d{4}\b`
   - This pattern matches phone numbers in the format XXX-XXX-XXXX, allowing for extracting phone numbers from text.

6. **Finding Words with Double Letters:**

   - Pattern: `\b\w*(\w)\1\w*\b`
   - This pattern matches words that contain consecutive identical letters, such as "hello" or "bookkeeper".

7. **Removing HTML Tags:**

   - Pattern: `<[^>]+>`
   - This pattern matches HTML tags and can be used for removing or replacing them in a text.

These examples showcase a range of regular expression applications, including validating input, extracting specific patterns from text, and manipulating strings based on matching criteria. Keep in mind that these

Here's a detailed explanation of regular expressions:

1. **Literal Characters:** Regular expressions can include literal characters, which match themselves exactly. For example, the regular expression "hello" matches the string "hello" in a text.
2. **Metacharacters:** Regular expressions also include metacharacters that have special meanings and functions. Some common metacharacters are:

   - . (dot): Matches any character except newline.
   - ^ (caret): Matches the start of a line.
   - $ (dollar): Matches the end of a line.
   - ' * ' (asterisk): Matches zero or more occurrences of the preceding character or group.
   - ' + ' (plus): Matches one or more occurrences of the preceding character or group.
   - ? (question mark): Matches zero or one occurrence of the preceding character or group.
   - \ (backslash): Escapes a metacharacter, treating it as a literal character.

3. **Character Classes:** Character classes allow you to define a set of characters that can match at a certain position in a string. Some common character classes include:

   - [abc]: Matches any character a, b, or c.
   - [a-z]: Matches any lowercase letter.
   - [A-Z]: Matches any uppercase letter.

- [0-9]: Matches any digit.
- [^abc]: Matches any character except a, b, or c.

4. **Quantifiers:** Quantifiers specify how many times a character or group can occur. Some commonly used quantifiers are:

   - {n}: Matches exactly n occurrences of the preceding character or group.
   - {n,}: Matches n or more occurrences of the preceding character or group.
   - {n,m}: Matches between n and m occurrences of the preceding character or group.
   - ?: Equivalent to {0,1}, matches zero or one occurrence.
5. **Grouping and Capturing** Regular expressions allow you to group characters together and capture parts of the matched string. This is done using parentheses. For example, the regular expression "(ab)+" matches one or more occurrences of the sequence "ab" in a text and captures each occurrence.
6. **Anchors:** Anchors are used to specify the position of a match within a string. The most commonly used anchors are:

   - ^ (caret): Matches the start of a line or string.
   - $ (dollar): Matches the end of a line or string.
   - \b: Matches a word boundary.

7. **Escape Sequences:** Regular expressions support various escape sequences that represent special characters or character classes. Some commonly used escape sequences are:

   - \d: Matches any digit (equivalent to [0-9]).
   - \w: Matches any word character (equivalent to [a-zA-Z0-9_]).
   - \s: Matches any whitespace character.
   - \t: Matches a tab character.
   - \n: Matches a newline character.

These are some of the fundamental concepts and constructs of regular expressions. They provide a powerful and flexible way to search, match, and manipulate text data based on specific patterns. However, regular expressions can become quite complex for advanced usage, involving additional features like lookaheads, lookbehinds, and backreferences.

In [1]:

```python
import re
```

In [2]:

```python
# Try find: re.findall(regexStr, inStr) -> matchedSubstringsList

# r'...' denotes raw strings which ignore escape code, i.e., r'\n' is '\'+'n'


re.findall(r'[0-9]+', 'abc123xyz') # Return a list of matched substrings
```

Out[2]:

```
['123']
```

In [3]:

```
s = "jack email is jack@somehost.com"
match = re.search(r'[\w.-]+@[\w.-]+', s)

# the above regular expression will match a email address

if match:
    print(match.group())
else:
    print("match not found")
```

```
jack@somehost.com
```

Group capturing

In [4]:

```
match = re.search(r'([\w.-]+)@([\w.-]+)', s)
```

In [5]:

```
s = "jack email is jack@somehost.com"

match = re.search('([\w.-]+)@([\w.-]+)', s)

if match:
    print(match.group()) ## tim@somehost.com (the whole match)
    print(match.group(1)) ## tim (the username, group 1)
    print(match.group(2)) ## somehost (the host, group 2)
```

```
jack@somehost.com
jack
somehost.com
```

findall() Function

it is commonly used to extract multiple occurrences of a pattern from a string

Syntax: findall(pattern, string, flags=0[optional])

In [6]:

```
s = "Jack's phone numbers are 12345-41521 and 78963-85214"
match = re.findall(r'\d{5}', s)

if match:
    print(match)
```

```
['12345', '41521', '78963', '85214']
```

In [105]:

```
pattern = r'(ab)+'
text = 'ababababab c'
matches = re.findall(pattern, text)
print(matches)
```

```
['ab']
```

In [7]:
```python
s = " Jack's phone numbers are 12345-41521 and 78963-85214"
match = re.findall(r'(\d{5})-(\d{5})', s)
print(match)

for i in match:
    print()
    print(i)
    print("First group", i[0])
    print("Second group", i[1])
```

```
[('12345', '41521'), ('78963', '85214')]

('12345', '41521')
First group 12345
Second group 41521

('78963', '85214')
First group 78963
Second group 85214
```

In [8]:
```python
txt = "The rain in Hyderabad"
x = re.findall("Hy", txt)
print(x)
```

```
['Hy']
```

In [9]:
```python
txt = "The rain in Hyderabad"
x = re.findall("Hy", txt)
print(x)
```

```
['Hy']
```

In [10]:
```python
txt = "The rain in Hyderabad  "
x = re.findall("Portugal", txt)
print(x) # Gives nothiing because it did not have similar data
```

```
[]
```

In [11]:
```python
string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'

result = re.findall(pattern, string)
print(result)

# Output: ['12', '89', '34']
```

```
['12', '89', '34']
```

Using re.match()

function is used to determine if a pattern matches at the beginning of a string. It checks for a match starting from the first character of the string.

In [12]:

```
s = "python timepass"
match = re.match(r'py', s)
if match:
    print(match.group())
```

```
py
```

In [13]:

```
# You can accomplish the same thing by applying ^ to a pattern with re.search()

s = "python tuts"
match = re.search(r'^py', s)
if match:
    print(match.group())
```

```
py
```

In [14]:

```
pattern = r"Cookie"
sequence = "Cookie"
if re.match(pattern, sequence):
    print("Match!")
else: print("Not a match!")
```

```
Match!
```

In [15]:

```
# [] - Square Brackets represent a character class consisting of a set of characters th
# [a-c] is same as [abc]

string = "The quick brown fox jumps over the lazy dog"
pattern = "[a-m]"
result = re.findall(pattern, string)

print(result)
```

```
['h', 'e', 'i', 'c', 'k', 'b', 'f', 'j', 'm', 'e', 'h', 'e', 'l', 'a', 'd', 'g']
```

In [16]:

```
## Suppose we have a text with many email addresses

str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

## Here re.findall() returns a list of all the found email strings

emails = re.findall(r'[\w\.-]+@[\w\.-]+', str) ## ['alice@google.com', 'bob@abc.com']

for email in emails:
# do something with each found email string
     print(email)
```

```
alice@google.com
bob@abc.com
```

In [ ]:

```
# findall with files

# Open file

f = open('test.txt', 'r')

# Feed the file text into findall(); it returns a list of all the found strings

strings = re.findall(r'some pattern', f.read())
```

In [17]:

```
# findall groups

str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

tuples = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)

print(tuples)  ## [('alice', 'google.com'), ('bob', 'abc.com')]
```

```
[('alice', 'google.com'), ('bob', 'abc.com')]
```

In [18]:

```
for tuple in tuples:
    print(tuple[0])  ## username
    print(tuple[1])  ## host
```

```
alice
google.com
bob
abc.com
```

In [19]:

```
# ^ - Caret = matches the beginning of the string i.e. checks whether the string starts


# Match strings starting with "The"

regex = r'^The'

strings = ['The quick brown fox', 'The lazy dog', 'A quick brown fox']

for string in strings:
    if re.match(regex, string):
        print(f'Matched: {string}')
    else:
        print(f'Not matched: {string}')
```

```
Matched: The quick brown fox
Matched: The lazy dog
Not matched: A quick brown fox
```

In [20]:
```
 #$ - Dollar($) symbol matches the end of the string i.e checks whether the string ends

string = "Hello World!"
pattern = r"World!$"

match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

```
Match found!
```

In [21]:
```
# Dot(.) symbol matches only a single character except for the newline character (\n).

string = "The quick brown fox jumps over the lazy dog."
pattern = r"brown.fox"

match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

```
Match found!
```

re.complie

Function is used to compile a regular expression pattern into a pattern object. This pattern object can then be used for matching operations.

In [22]:
```
# compile() creates regular expression

# character class [a-e],

# which is equivalent to [abcde].

# class [abcde] will match with string with

# 'a', 'b', 'c', 'd', 'e'.

p = re.compile('[a-e]')

# findall() searches for the Regular Expression

# and return a list upon finding

print(p.findall("Aye,  brown fox jumps over the lazy dog"))
```

```
['e', 'b', 'e', 'e', 'a', 'd']
```

In [23]:
```
# \d is equivalent to [0-9].

p = re.compile('\d')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

# \d+ will match a group on [0-9], group

# of one or greater size

p = re.compile('\d+')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

```
['1', '1', '4', '1', '8', '8', '6']
['11', '4', '1886']
```

In [24]:
```
# \w is equivalent to [a-zA-Z0-9_].

p = re.compile('\w')
print(p.findall("He said * in some_lang."))

# \w+ matches to group of alphanumeric character.

p = re.compile('\w+')
print(p.findall("I went to him at 11 A.M., he \
said *** in some_language."))


# \W matches to non alphanumeric characters.

p = re.compile('\W')
print(p.findall("he said *** in some_language."))
```

```
['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '_', 'l', 'a', 'n', 'g']
['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in', 'some_language']
[' ', ' ', '*', '*', '*', ' ', ' ', '.']
```

In [25]:
```
# '*' replaces the no. of occurrence
# of a character.
p = re.compile('ab*')
print(p.findall("ababbaabbb"))
```

```
['ab', 'abb', 'a', 'abbb']
```

The search() Function

function is used to search for a pattern anywhere within a string. It looks for the first occurrence of the pattern and returns a match object if found.

In [26]:
```
txt = "The rain in  Hyderabad "
x = re.search("\s", txt)

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

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

In [27]:

```
z = 'Regular expressions Helpful in Datascience'
x = re.search("\s",z)

print("The first white-space character is located in position:", x.start()) # 7 -regula
```

```
The first white-space character is located in position: 7
```

In [28]:

```
#Check if the string starts with "The" and ends with "Spain":

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

if x:
  print("YES! We have a match!")
else:
  print("No match")
```

```
YES! We have a match!
```

In [29]:

```
s = 'software applications:programming languages'

match = re.search(r'programming', s)

print('Start Index:', match.start())
print('End Index:', match.end())
```

```
Start Index: 22
End Index: 33
```

In [30]:

```
# Using Backslash
```

In [31]:

```
s = 'mind.isstrong'

# without using \
match = re.search(r'.', s)
print(match)

# using \
match = re.search(r'\.', s)
print(match)
```

```
<re.Match object; span=(0, 1), match='m'>
<re.Match object; span=(4, 5), match='.'>
```

In [32]:

```
s = "Welcome to Guys"

# here x is the match object
res = re.search(r"\bG", s)

print(res.re)
print(res.string)
```

```
re.compile('\\bG')
Welcome to Guys
```

In [33]:
```
s = "smile to Gunpoint"

# here x is the match object
res = re.search(r"\bGun", s)

print(res.start())
print(res.end())
print(res.span())
```

```
9
12
(9, 12)
```

In [34]:
```
s = "Welcome to  Hyderabad"

# here x is the match object
res = re.search(r"\D{2} t", s)

print(res.group())
```

```
me t
```

In [35]:
```
str = 'an example word:cat!!'

match = re.search(r'word:\w\w\w', str)

# If-statement after search() tests if it succeeded

if match:
    print('found', match.group()) ## 'found word:cat'
else:
    print('did not find')
```

```
found word:cat
```

In [36]:
```
import re

# Search for pattern 'iii' in string 'piiig'.
# All of the pattern must match, but it may appear anywhere.
# On success, match.group() is matched text.

match = re.search(r'iii', 'piiig')  # found, match.group() == "iii"
match
```

Out[36]:
```
<re.Match object; span=(1, 4), match='iii'>
```

In [37]:
```
match = re.search(r'igs', 'piiig')  # not found, match == None
```

In [38]:
```
match
```

In [39]:
```
match = re.search(r'..g', 'piiig')  # found, match.group() == "iig"
```

In [40]:
```
match
```

Out[40]:
```
<re.Match object; span=(2, 5), match='iig'>
```

In [41]:

```
match = re.search(r'\d\d\d', 'p123g')  # found, match.group() == "123"
```

In [42]:

```
match
```

Out[42]:

```
<re.Match object; span=(1, 4), match='123'>
```

In [43]:

```
match = re.search(r'\w\w\w', '@@abcd!!')  # found, match.group() == "abc"
```

In [44]:

```
match
```

Out[44]:

```
<re.Match object; span=(2, 5), match='abc'>
```

In [45]:

```
line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print ("match --> matchObj.group() : ", matchObj.group())
else:
    print ("No match!!")
```

```
No match!!
```

In [46]:

```
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
    print ("search --> searchObj.group() : ", searchObj.group())
else:
    print ("Nothing found!!")
```

```
search --> searchObj.group() :  dogs
```

In [47]:

```
pattern = '^a...s$'
test_string = 'abyss'

result = re.match(pattern, test_string)

if result:
    print("Search successful.")
else:
    print("Search unsuccessful.")
```

```
Search successful.
```

In [48]:

```
string = "Python is fun"

# check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
  print("pattern found inside the string")
else:
  print("pattern not found")
```

```
pattern found inside the string
```

In [49]:

```
line = "Cats are smarter than dogs";

searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
    print ("searchObj.group() : ", searchObj.group())
    print ("searchObj.group(1) : ", searchObj.group(1))
    print ("searchObj.group(2) : ", searchObj.group(2))
else:
    print ('Nothing found!!')
```

```
searchObj.group() :  Cats are smarter than dogs
searchObj.group(1) :  Cats
searchObj.group(2) :  smarter
```

In [50]:

```
# Lets use a regular expression to match a date string
# in the form of Month name followed by day number
regex = r"([a-zA-Z]+) (\d+)"

match = re.search(regex, "I was born on June 24")

if match != None:

    # We reach here when the expression "([a-zA-Z]+) (\d+)"
    # matches the date string.

    # This will print [14, 21), since it matches at index 14
    # and ends at 21.
    print ("Match at index %s, %s" % (match.start(), match.end()))

    # We use group() method to get all the matches and

    # captured groups. The groups contain the matched values.

    # In particular:
    # match.group(0) always returns the fully matched string

    # match.group(1) match.group(2), ... return the capture

    # groups in order from left to right in the input string

    # match.group() is equivalent to match.group(0)

    # So this will print "June 24"
    print ("Full match: %s" % (match.group(0)))

    # So this will print "June"
    print ("Month: %s" % (match.group(1)))

    # So this will print "24"
    print ("Day: %s" % (match.group(2)))

else:
    print ("The regex pattern does not match.")
```

```
Match at index 14, 21
Full match: June 24
Month: June
Day: 24
```

In [ ]:

The split() Function

rather than "spilt." In Python, the split() function is a built-in method that allows you to split a string into a list of substrings based on a specified delimiter. It is not specific to regular expressions.

In [51]:
```
txt = "The rain in  Hyderabad"
x = re.split("\s", txt)
print(x)
```

```
['The', 'rain', 'in', '', 'Hyderabad']
```

In [52]:
```
string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

```
['Twelve:', ' Eighty nine:', '.']
```

In [53]:
```
# You can pass maxsplit argument to the re.split() method. It's the maximum number of sp

string = 'Twelve:12 Eighty nine:89 Nine:9.'
pattern = '\d+'

# maxsplit = 1
# split only at the first occurrence
result = re.split(pattern, string, 1)
print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']
```

```
['Twelve:', ' Eighty nine:89 Nine:9.']
```

In [54]:
```
z = 'Regular expressions can include literal characters'
x = re.split("\s", z)
print(z)
```

```
Regular expressions can include literal characters
```

In [55]:
```
# Split the string only at the first occurrence:

z = 'Regular expressions can include literal characters'
x = re.split("\s", z, 1)
print(x)
```

```
['Regular', 'expressions can include literal characters']
```

In [56]:
```
# split the string Second Occurance:

z = 'Regular expressions can include literal characters'
x = re.split("\s", z, 2)
print(x)
```

```
['Regular', 'expressions', 'can include literal characters']
```

In [57]:

```
from re import split

# '\W+' denotes Non-Alphanumeric Characters

# or group of characters Upon finding ','

# or whitespace ' ', the split(), splits the

# string from that point

print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))

# Here ':', ' ' ,',' are not AlphaNumeric thus,

# the point where splitting occurs

print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))

# '\d+' denotes Numeric Characters or group of

# characters Splitting occurs at '12', '2016',

# '11', '02' only

print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

```
['Words', 'words', 'Words']
['Word', 's', 'words', 'Words']
['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']
['On ', 'th Jan ', ', at ', ':', ' AM']
```

In [58]:

```
# Splitting will occurs only once, at

# '12', returned list will have length 2

print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))

# 'Boy' and 'boy' will be treated same when

# flags = re.IGNORECASE

print(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags=re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

```
['On ', 'th Jan 2016, at 11:02 AM']
['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']
['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']
```

In [ ]:

In [ ]:

The sub() Function

it is used to search for a pattern in a string and replace it with a specified replacement string. It performs

In [59]:

```
txt = "The King in the North claims he is worthy"
x = re.sub("\s", "9", txt)
print(x)
```

```
The9King9in9the9North9claims9he9is9worthy
```

In [60]:

```
# Replace the first 2 occurrences:

txt = "The King in the North claims he is worthy"
x = re.sub("\s", "9", txt, 2)
print(x)
```

```
The9King9in the North claims he is worthy
```

In [61]:

```
phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print ("Phone Num : ", num)

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print ("Phone Num : ", num)
```

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

In [62]:

```
# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.sub(pattern, replace, string)
print(new_string)

# Output: abc12de23f456
```

```
abc12de23f456
```

In [63]:

```
# You can pass count as a fourth parameter to the re.sub() method. If omited, it result

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'
replace = ''

new_string = re.sub(r'\s+', replace, string, 1)
print(new_string)
```

```
abc12de 23
 f45 6
```

In [64]:

```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

## re.sub(pat, replacement, str) -- returns new string with all replacements,

## \1 is group(1), \2 group(2) in the replacement

print(re.sub(r'([\w\.-]+)@([\w\.-]+)', r'\1@yo-yo-dyne.com', str))

## purple alice@yo-yo-dyne.com, blah monkey bob@yo-yo-dyne.com blah dishwasher
```

```
purple alice@yo-yo-dyne.com, blah monkey bob@yo-yo-dyne.com blah dishwasher
```

In [65]:

```
# Regular Expression pattern 'ub' matches the

# string at "Subject" and "Uber". As the CASE

# has been ignored, using Flag, 'ub' should

# match twice with the string Upon matching,

# 'ub' is replaced by '~*' in "Subject", and

# in "Uber", 'Ub' is replaced.

print(re.sub('ub', '~*', 'Subject has Uber booked already',
             flags=re.IGNORECASE))

# Consider the Case Sensitivity, 'Ub' in

# "Uber", will not be replaced.

print(re.sub('ub', '~*', 'Subject has Uber booked already'))

# As count has been given value 1, the maximum

# times replacement occurs is 1

print(re.sub('ub', '~*', 'Subject has Uber booked already',
             count=1, flags=re.IGNORECASE))

# 'r' before the pattern denotes RE, \s is for

# start and end of a String.

print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',
             flags=re.IGNORECASE))
```

```
S~*ject has ~*er booked already
S~*ject has Uber booked already
S~*ject has Uber booked already
Baked Beans & Spam
```

re.subn()

subn() is similar to sub() in all ways, except in its way of providing output. It returns a tuple with a count of the total of replacement and the new string rather than just the string.

In [66]:

```
print(re.subn('ub', '~*', 'Subject has Uber booked already'))

t = re.subn('ub', '~*', 'Subject has Uber booked already',
            flags=re.IGNORECASE)
print(t)
```

```
('S~*ject has Uber booked already', 1)
('S~*ject has ~*er booked already', 2)
```

In [67]:

```
print(len(t))
```

```
2
```

In [68]:

```
# This will give same output as sub() would have

print(t[0])
```

```
S~*ject has ~*er booked already
```

In [69]:

```

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.subn(pattern, replace, string)
print(new_string)
```

```
('abc12de23f456', 4)
```

re.escape()

Returns string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

syntax : re.escape(string)

In [70]:

```
# escape() returns a string with BackSlash '\',

# before every Non-Alphanumeric Character

# In 1st case only ' ', is not alphanumeric

# In 2nd case, ' ', caret '^', '-', '[]', '\'

# are not alphanumeric

print(re.escape("This is Awesome even 1 AM"))
```

```
This\ is\ Awesome\ even\ 1\ AM
```

In [71]:

```
print(re.escape("I Asked what is this [a-9], he said \t ^WoW"))
```

```
I\ Asked\ what\ is\ this\ \[a\-9\],\ he\ said\ \        \ \^WoW
```

In [ ]:

Match Object

a match object is returned by various functions like re.match(), re.search(), and re.findall() when a pattern is found in a string. The match object contains information about the match, such as the matching substring, the start and end positions of the match, and any captured groups

In [72]:
```
txt = "The gain in pain"
x = re.search("ai", txt)
print(x) #this will print an object
```

```
<re.Match object; span=(5, 7), match='ai'>
```

In [73]:
```
txt = "The King in the North claims he is worthy"
x = re.search("ai", txt)
print(x)
```

```
<re.Match object; span=(24, 26), match='ai'>
```

In [74]:
```
string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.
match = re.search(pattern, string)

if match:
    print(match.group())
else:
    print("pattern not found")

# Output: 801 35
```

```
801 35
```

In [75]:
```
# Here, match variable contains a match object.

# Our pattern (\d{3}) (\d{2}) has two subgroups (\d{3}) and (\d{2}). You can get the pa

match.group(1)
```

Out[75]:
```
'801'
```

In [76]:
```
match.group(2)
```

Out[76]:
```
'35'
```

In [77]:

```
match.group(1, 2)
```

Out[77]:

```
('801', '35')
```

In [78]:

```
match.groups()
```

Out[78]:

```
('801', '35')
```

The Match object has properties and methods used to retrieve information about the search, and the result:

- span() returns a tuple containing the start-, and end positions of the match.
- string returns the string passed into the function
- group() returns the part of the string where there was a match

In [79]:

```
# Print the position (start- and end-position) of the first match occurrence.

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

```
(12, 17)
```

In [80]:

```
txt = "The gain is in the pain"
x = re.search(r"\bi\w+", txt)

if x is not None:
    print("Match found:", x.group())
    print("Match span:", x.span())
else:
    print("No match found")
```

```
Match found: is
Match span: (9, 11)
```

In [81]:

```
txt = "The gain is in the pain"
x = re.search(r"\bt\w+", txt)

if x is not None:
    print("Match found:", x.group())
    print("Match span:", x.span())
else:
    print("No match found")
```

```
Match found: the
Match span: (15, 18)
```

In [82]:

```python
str = 'purple alice-b@google.com monkey dishwasher'

match = re.search(r'\w+@\w+', str)

if match:
    print(match.group())  ## 'b@google'
```

```
b@google
```

In [83]:

```python
#Square Brackets

match = re.search(r'[\w.-]+@[\w.-]+', str)
if match:
    print(match.group())  ## 'alice-b@google.com'
```

```
alice-b@google.com
```

In [84]:

```python
# Group Match (Extraction)

str = 'purple alice-b@google.com monkey dishwasher'

match = re.search(r'([\w.-]+)@([\w.-]+)', str)

if match:
    print(match.group())   ## 'alice-b@google.com' (the whole match)
    print(match.group(1))  ## 'alice-b' (the username, group 1)
    print(match.group(2))  ## 'google.com' (the host, group 2)
```

```
alice-b@google.com
alice-b
google.com
```

In [85]:

```python
line = "Cats are smarter than dogs"

matchObj = re.match(r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
    print("matchObj.group() : ", matchObj.group())
    print("matchObj.group(1) : ", matchObj.group(1))
    print("matchObj.group(2) : ", matchObj.group(2))
else:
    print("No match!!")
```

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

In [ ]:

Extracting number from text

```
In [86]: text ='he is a good boy his number is  7090902299202 and his usa number is (700)-228-99
         pattern = "\(\d{3}\)-\d{3}-\d{4}|\d{10}"
         re.findall(pattern,text)
```

```
Out[86]: ['7090902299', '(700)-228-9904']
```

Extracting headlies from text

```
In [87]: text =  '''
         Note 1 - Overview
         Tesla, Inc. (“Tesla”, the “Company”, “we”, “us” or “our”) was incorporated in the State
         products. Our Chief Executive Officer, as the chief operating decision maker (“CODM”),
         Beginning in the first quarter of 2021, there has been a trend in many parts of the wor
         against COVID-19, as well as an easing of restrictions on social, business, travel and
         rates and regulations continue to fluctuate in various regions and there are ongoing gl
         and increases in costs for logistics and supply chains, such as increased port congesti
         supply. We have also previously been affected by temporary manufacturing closures, empl
         administrative activities supporting our product deliveries and deployments.
         Note 2 - Summary of Significant Accounting Policies
         Unaudited Interim Financial Statements
         The consolidated balance sheet as of September 30, 2021, the consolidated statements of
         comprehensive income, the consolidated statements of redeemable noncontrolling interest
         30, 2021 and 2020 and the consolidated statements of cash flows for the nine months ende
         disclosed in the accompanying notes, are unaudited. The consolidated balance sheet as o
         consolidated financial statements as of that date. The interim consolidated financial st
         conjunction with the annual consolidated financial statements and the accompanying note
         ended December 31, 2020.
         '''
```

```
In [88]: pattern = "Note \d - [^\n]+"
```

```
In [89]: re.findall(pattern,text)
```

```
Out[89]: ['Note 1 - Overview', 'Note 2 - Summary of Significant Accounting Policies']
```

Extracting the series of code in the text

```
In [90]: text = '''The gross cost of operating lease vehicles in FY2021 Q1 was $4.85 billion.
         In previous quarter i.e. FY2020 Q4 it was $3 billion. FY2022 Q5.
         '''
         pattern = "FY\d{4} Q[1-4]"
         re.findall(pattern,text)
```

```
Out[90]: ['FY2021 Q1', 'FY2020 Q4']
```

remove capital and small letters

In [91]:
```
text = '''The gross cost of operating lease vehicles in FY2021 Q1 was $4.85 billion.
In previous quarter i.e. fy2020 Q4 it was $3 billion. FY2022 Q5.
'''
pattern = "FY\d{4} Q[1-4]"
re.findall(pattern,text,flags=re.IGNORECASE)
```

Out[91]:
```
['FY2021 Q1', 'fy2020 Q4']
```

extract only dollar 4.85, dollar 3

In [92]:
```
text = '''The gross cost of operating lease vehicles in FY2021 Q1 was $4.85 billion.
In previous quarter i.e. fy2020 Q4 it was $3 billion. FY2022 Q5.
'''
pattern = "\$[0-9\.]+"
re.findall(pattern,text,flags=re.IGNORECASE)
```

Out[92]:
```
['$4.85', '$3']
```

Extract phone number

In [93]:
```
chat1 = 'codebasics: you ask lot of questions 😡  1235678912,abcP@xyz.com'
chat2 = 'codebasics: here it is: (123)-567-8912, abc_29@xyz.in'
chat3 = 'codebasics: yes, phone: 1235678912 email: abc@xyz.com'
```

In [94]:
```
pattern = '\d{10}'
matches = re.findall(pattern,chat1)
matches
```

Out[94]:
```
['1235678912']
```

In [95]:
```
pattern = '\(\d{3}\)-\d{3}-\d{4}|\d{10}'
matches = re.findall(pattern,chat2)
matches
```

Out[95]:
```
['(123)-567-8912']
```

In [96]:
```
pattern = '\(\d{3}\)-\d{3}-\d{4}|\d{10}'
matches = re.findall(pattern,chat3)
matches
```

Out[96]:
```
['1235678912']
```

email extraction

In [97]:
```
pattern = '[a-z0-9A-Z_]*@[a-z]*\.[a-z0-9A-Z]*'
re.findall(pattern,chat1)
```

Out[97]:
```
['abcP@xyz.com']
```

In [98]:

```
pattern = '[a-z0-9A-Z_]*@[a-z]*\.[a-z0-9A-Z]*'
re.findall(pattern,chat2)
```

Out[98]:

```
['abc_29@xyz.in']
```

In [99]:

```
pattern = '[a-z0-9A-Z_]*@[a-z]*\.[a-z0-9A-Z]*'
re.findall(pattern,chat3)
```

Out[99]:

```
['abc@xyz.com']
```

extraction of order number

In [100]:

```
chat1='codebasics: Hello, I am having an issue with my order # 412882'
chat2 ='codebasics: I have a problem with my order number 412889912'
chat3='codebasics: My order 49912 is having an issue, I was charged 300$ when online it


pattern = 'order[^\d]*(\d*)'
```

In [101]:

```
re.findall(pattern,chat3)
```

Out[101]:

```
['49912']
```

In [102]:

```
re.findall(pattern,chat2)
```

Out[102]:

```
['412889912']
```

In [103]:

```
re.findall(pattern,chat1)
```

Out[103]:

```
['412882']
```

In [ ]: