In [21]:
```
#this problem is about finding out all names in the string

import re
text = "Tuba is Maya. Liza is Lily."
pattern = '[A-Z][\w]*(?=\W|\s)'
re.findall(pattern,text)
```

Out[21]:
```
['Tuba', 'Maya', 'Liza', 'Lily']
```

In [129]:
```
#this problem is about finding out all names in the string

text = "Amy is : B 5 years old, and her sister Mary is 2 years old. Ruth and
Peter, B their parents, have 3 kids. : B "
pattern = '[A-Z][\w]*'

re.findall(pattern,text)
```

Out[129]:
```
['Amy', 'B', 'Mary', 'Ruth', 'Peter', 'B', 'B']
```

In [1]:
```
#this problem is about finding out all 'B' in the string
import re
text = """Ronald Mayr: A
Bell Kassulke: B
Jacqueline Rupp: A
Alexander Zeller: C
Valentina Denk: C
Simon Loidl: B
Elias Jovanovic: B """

pattern = ': [B]*(?=\s)'
len(re.findall(pattern,text))
```

Out[1]:
```
3
```

In [130]:
```
text = "Bell KassJacqueline Rupp: A Alexander Zeller: C Valentina Denk: C Si
mon Loidl: B Elias Jovanovic: B Stefanie Weninger: A "

pattern = '[ \w]*(?=\: )'
re.findall(pattern,text)
```

Out[130]:
```
['Bell KassJacqueline Rupp',
 '',
 ' A Alexander Zeller',
 '',
 ' C Valentina Denk',
 '',
 ' C Simon Loidl',
 '',
 ' B Elias Jovanovic',
 '',
 ' B Stefanie Weninger',
 '']
```

In [7]:
```
#this problem is about finding out all 'B' in the string

text = """Ronald Mayr: A
Bell Kassulke: B
Jacqueline Rupp: A
```

```
Alexander Zeller: C
Valentina Denk: C
Simon Loidl: B
Elias Jovanovic: B
Stefanie Weninger: A
Fabian Peer: C
Hakim Botros: B
Emilie Lorentsen: B
Herman Karlsen: C
Nathalie Delacruz: C
Casey Hartman: C
Lily Walker : A
Gerard Wang: C
Tony Mcdowell: C
Jake Wood: B
Fatemeh Akhtar: B
Kim Weston: B
Nicholas Beatty: A
Kirsten Williams: C
Vaishali Surana: C
oby Mccormack: C
Yasmin Dar: B
Romy Donnelly: A
Viswamitra Upandhye: B
Bendrick Hilpert: A
Killian Kaufman: B
Elwood Page: B
Mukti Patel: A
Emily Lesch: B
Elodie Booker: B
Jedd Kim: A
Annabel Davies: A
Adnan Chen: B
Jonathan Berg: C
Hank Spinka: B
Bgnes Schneider: B
Kimberly Green: A
Lola-Rose Coates: C
Rose Christiansen: C
Shirley Hintz: B
Hannah Bayer: B"""

#pattern = '[A-Z][\w ]*(?=: [B])'
pattern = ': [B]*(?=\n|[B])'
len(re.findall(pattern,text))
```

Out[7]: 19

In [119]:
```
text = "hhhhkkkkk iiiio hhhh "
pattern = '[\w]*(?=\s)'

re.findall(pattern,text)
```

Out[119]: ['hhhhkkkkk', '', 'iiiio', '', 'hhhh', '']

In [121]:
```
import re
text = "Tuba is Maya. Liza is Lily "
pattern = '[A-Z][\w]*(?=\W|\s)'
re.findall(pattern,text)
```

Out[121]: ['Tuba', '', 'is', '', 'Maya', '', '', 'Liza', '', 'is', '', 'Lily', '']

In [128]:
```
#this problem is about finding all digits[0-9] in the string

s = 'tuba123tuba'
re.findall('[0-9]',s)
```

Out[128]:
```
['1', '2', '3']
```

In [136]:
```
#this problem is about searching single digit[0-9] in the string


s = 'tuba023tuba'
if re.search('[0-9]',s):

    print("matched.found 1(the first one) digit in the string")
```

```
matched.found 1(the first one) digit in the string
```

In [137]:
```
#this problem is about searching three consecutive digits[0-9] in the string

s = 'tuba123tuba'
if re.search('[0-9][0-9][0-9]',s):
    print("found 3 consecutive digits in the string. 1,2,3")
```

```
found 3 consecutive digits in the string. 1,2,3
```

In [138]:
```
s = 'tuba12tuba3'
if re.search('[0-9][0-9][0-9]',s):
    print("found 3 consecutive digits in the string. 1,2,3")
else:
    print("digits are not consecutive here. it should be 123tuba. but found
 12tuba3")
```

```
digits are not consecutive here. it should be 123tuba. but found 12tuba3
```

In [142]:
```
#it searhes either tuba1,tuba4,tuba5 and finally gets tuba1

s = 'tuba123tuba'
re.search('tuba[145]',s)
```

Out[142]:
```
<re.Match object; span=(0, 5), match='tuba1'>
```

In [144]:
```
s = 'tuba123tuba'
re.search('tuba[451]',s)
```

Out[144]:
```
<re.Match object; span=(0, 5), match='tuba1'>
```

In [145]:
```
s = 'tuba123tuba'
re.search('[a-z]',s)  #matches the single character that is netween a and z
```

Out[145]:
```
<re.Match object; span=(0, 1), match='t'>
```

In [146]:
```
s = 'tuba123tuba'
re.search('[a-z][a-z][a-z]',s)
```

Out[146]:
```
<re.Match object; span=(0, 3), match='tub'>
```

In [147]:
```
s = 'tuba123tuba'
re.search('[0-9]',s)  #matches the single character that is  a number
```

Out[147]:
```
<re.Match object; span=(4, 5), match='1'>
```

In [148]:
```
s = 'tuba123tuba'
re.search('[^0-9]',s)

#matches the single character that is not a number.
#If a ^ character appears in a character class but isn't the first characte
r,
#then it has no special meaning and matches a literal '^' character:
```

Out[148]: <re.Match object; span=(0, 1), match='t'>

In [154]:
```
re.search('[-abc]','lk-a')
```

Out[154]: <re.Match object; span=(2, 3), match='-'>

In [155]:
```
re.search('[-abc]','blk-a')
```

Out[155]: <re.Match object; span=(0, 1), match='b'>

In [160]:
```
re.search('[ab\]cd]','[1]blk-a')
```

Out[160]: <re.Match object; span=(2, 3), match=']'>

In [163]:
```
re.search('[]]','[1]blk-a')
```

Out[163]: <re.Match object; span=(2, 3), match=']'>

In [169]:
```
re.search("tuba.tuba",'tuba1tuba')

#As a regex, tuba.tuba essentially means the characters 'tuba',
#then any character except newline, then the characters 'tuba'. The first st
ring shown above, 'tuba1tuba',
#fits the bill because the . metacharacter matches the '1'.
```

In [168]:
```
re.search("tuba.tuba",'tubatuba') #not matched
```

In [171]:
```
#\w matches any alphanumeric word character. Word characters are uppercase a
nd lowercase letters, digits, and the underscore (_)
#character, so \w is essentially shorthand for [a-zA-Z0-9_]:

re.search("\w",'@#!Kubatuba')
```

Out[171]: <re.Match object; span=(3, 4), match='K'>

In [174]:
```
#\W is the opposite. It matches any non-word character and is equivalent to
 [^a-zA-Z0-9_]:


re.search("\W",'@#!Kubatuba')
```

Out[174]: <re.Match object; span=(0, 1), match='@'>

In [175]:
```
#\d matches any decimal digit character. \D is the opposite. It matches any
 character that isn't a decimal digit:
#\d is essentially equivalent to [0-9], and \D is equivalent to [^0-9].

re.search("\d",'nhhjj123')
```

Out[175]: <re.Match object; span=(5, 6), match='1'>

In [176]:
```
re.search("\D",'@#!Kubatuba')
```

Out[176]:
```
<re.Match object; span=(0, 1), match='@'>
```

In [177]:
```
#\s matches any whitespace character:
#Note that, unlike the dot wildcard metacharacter, \s does match a newline c
haracter.

re.search("\s",'nhh  jj123')
```

Out[177]:
```
<re.Match object; span=(3, 4), match=' '>
```

In [178]:
```
re.search("\s",'nhh\n  jj123')
```

Out[178]:
```
<re.Match object; span=(3, 4), match='\n'>
```

In [181]:
```
#\S is the opposite of \s. It matches any character that isn't whitespace:

re.search("\S",'vhh\n  jj123')
```

Out[181]:
```
<re.Match object; span=(0, 1), match='v'>
```

In [184]:
```
#The character class sequences \w, \W, \d, \D, \s, and \S can appear inside
 a square bracket character class as well:
#In this case, [\d\w\s] matches any digit, word, or whitespace character.

re.search('[\s\d\w]','7jed 45')
```

Out[184]:
```
<re.Match object; span=(0, 1), match='7'>
```

In [187]:
```
#Occasionally, you'll want to include a metacharacter in your regex, except
 you won't want it to carry its special meaning.
#Instead, you'll want it to represent itself as a literal character.
#A metacharacter preceded by a backslash loses its special meaning and match
es the literal character instead.
#Consider the following examples:

print(re.search('.','tuba'))
print(re.search('\.','tuba.'))
```

```
<re.Match object; span=(0, 1), match='t'>
<re.Match object; span=(4, 5), match='.'>
```

In [194]:
```
#how to search '\' on a string?
s = r'tu\ba'
print(re.search('\\\\',s))

#another way is:
print(re.search(r'\\',s))  #good way
```

```
<re.Match object; span=(2, 3), match='\\'>
<re.Match object; span=(2, 3), match='\\'>
```

In [197]:
```
#ANCHOR

#Anchors are zero-width matches. They don't match any actual characters in t
he search string,
#and they don't consume any of the search string during parsing.
#Instead, an anchor dictates a particular location in the search string wher
```

```
e a match must occur.

#regex \Afoo or ^foo stipulates that 'foo' must be present not just any old
 place in the search string, but at the beginning:
s='tuba123'
print(re.search('^tu',s))
print(re.search('\Atub',s))
print(re.search('\A123',s))
```

```
<re.Match object; span=(0, 2), match='tu'>
<re.Match object; span=(0, 3), match='tub'>
None
```

In [201]:
```
#$ or \Z
#When the regex parser encounters $ or \Z, the parser's current position mus
t be at the end of the search string
#for it to find a match.

s='tuba123'
print(re.search('3$',s))
print(re.search('23\Z',s))
print(re.search('ba\Z',s))

#As a special case, $ (but not \Z) also matches just before a single newline
at the end of the search string:

s='tuba123\n'
print(re.search('123$',s))
print(re.search('123\Z',s))
```

```
<re.Match object; span=(6, 7), match='3'>
<re.Match object; span=(5, 7), match='23'>
None
<re.Match object; span=(4, 7), match='123'>
None
```

In [209]:
```
#\b
#\b asserts that the regex parser's current position must be at the beginnin
g or end of a word.
#A word consists of a sequence of alphanumeric characters or underscores ([a
-zA-Z0-9_]), the same as for the \w character class:
s='tuba123 moni'
print(re.search(r'\bmoni',s))

s='tuba123-moni'
print(re.search(r'\bmoni',s))
s='tuba123moni'
print(re.search(r'\bmoni',s))
s='tuba 123 moni'
print(re.search(r'\btuba',s))
s='tuba 123 moni'
print(re.search(r'\b123\b',s))
```

```
<re.Match object; span=(8, 12), match='moni'>
<re.Match object; span=(8, 12), match='moni'>
None
<re.Match object; span=(0, 4), match='tuba'>
<re.Match object; span=(5, 8), match='123'>
```

In [216]:
```
#\B does the opposite of \b. It asserts that the regex parser's current posi
tion must not be at the start or end of a word:
```

```
s='tuba 123 moni'
print(re.search(r'\Btuba',s))
s='tuba 123 moni'
print(re.search(r'\Bmoni',s))
s='tuba 123 moni hjn'
print(re.search(r'\B123',s))
```

```
None
None

None
```

In [222]:
```
#Quantifiers

#A quantifier metacharacter immediately follows a portion of a <regex> and i
ndicates how many times that portion must occur
#for the match to succeed.
#For example, a* matches zero or more 'a' characters. That means it would ma
tch an empty string, 'a', 'aa', 'aaa', and so on.

print(re.search('tu*ba','tuba')) #1 u between tu and ba there is 1 a.
print(re.search('tu*ba','tuuba')) #2 u
print(re.search('tua*ba','tuba')) #0 u between tu and ba there is no a.
print(re.search('tu*ba','baba')) #0 u

# .* matches any character sequence up to a line break. (Remember that the .
wildcard metacharacter doesn't match a newline.)
#In this example, .* matches everything between 'foo' and 'bar':
```

```
<re.Match object; span=(0, 4), match='tuba'>
<re.Match object; span=(0, 5), match='tuuba'>
<re.Match object; span=(0, 4), match='tuba'>
None
```

In [225]:
```
# +  is similar to *, but the quantified regex must occur at least once:

print(re.search('tul+ba','tuba')) # no l between tu and ba

print(re.search('tu1+ba','tu11ba')) # two 1 between tu and ba

#The + metacharacter requires at least one occurrence.
```

```
None
<re.Match object; span=(0, 6), match='tu11ba'>
```

In [6]:
```
# ? Again, this is similar to * and +, but in this case there's only a match
if the preceding regex occurs once or not at all:
import re
print(re.search('tul?ba','tuba')) # no l between tu and ba
print(re.search('tu1?ba','tu11ba')) # two 1 between tu and ba
print(re.search('tu?ba','tuba')) #  1 u between t and ba
```

```
<re.Match object; span=(0, 4), match='tuba'>
None
<re.Match object; span=(0, 4), match='tuba'>
```

In [19]:
```
# *?
# +?
# ??
```

```
# the non-greedy(or lazy )version of .,+ and ? quantifiers
# When used alone, the quantifier metacharacters *, +, and ? are all greedy,
meaning they produce the longest possible match.

print(re.search('<.*>','<nms>lk+ <12hj> <foo>'))

# The regex <.*> effectively means:
# A '<' character
# Then any sequence of characters
# Then a '>' character
# But which '>' character? There are three possibilities:
# The one just after 'nms'
# The one just after '12hj'
# The one just after 'foo'
# Since the * metacharacter is greedy, it dictates the longest possible matc
h,
# which includes everything up to and including the '>' character that follo
ws 'foo'.
# You can see from the match object that this is the match produced.
# If you want the shortest possible match instead, then use the non-greedy m
etacharacter sequence *?:

print(re.search('<.*?>','<nms>lk+ <12hj> <foo>'))

# In this case, the match ends with the '>' character following 'nms'.
# There are lazy versions of the + and ? quantifiers as well:

print(re.search('<.+>','<nms>lk+ <12hj> <foo>'))
print(re.search('<.+?>','<nms>lk+ <12hj> <foo>'))

print(re.search('ba?','baaaa'))
print(re.search('ba??','baaa'))

# The last examples are a little different.
# In general, the ? metacharacter matches zero or one occurrences of the pre
ceding regex.
# The greedy version, ?, matches one occurrence, so ba? matches 'b' followed
by a single 'a'.
# The non-greedy version, ??, matches zero occurrences, so ba?? matches just
'b'.
```

```
<re.Match object; span=(0, 21), match='<nms>lk+ <12hj> <foo>'>
<re.Match object; span=(0, 5), match='<nms>'>
<re.Match object; span=(0, 21), match='<nms>lk+ <12hj> <foo>'>
<re.Match object; span=(0, 5), match='<nms>'>

None
<re.Match object; span=(0, 1), match='b'>
```

In [22]:
```
# {m}
# This is similar to * or +, but it specifies exactly how many times the pre
ceding regex must occur for a match to succeed:
# matches exactly m repeatations

print(re.search('ba-{3}ba','ba---ba')) # exactly three dashes between ba and
ba
print(re.search('ba-{3}','ba---tuba')) # exactly three dashes after ba
print(re.search('ba-{3}ba','ba-----ba')) # expected three dashes but got fiv
e dashes
```

```
<re.Match object; span=(0, 7), match='ba---ba'>
<re.Match object; span=(0, 5), match='ba---'>
```

```
None
```

In [25]:
```
# {m,n}
# matches any number pf  repeatations from m to n inclusive

print(re.search('ba-{1,3}ba','ba---ba'))
print(re.search('ba-{3,6}ba','ba----ba'))
print(re.search('ba-{4,7}ba','ba---ba'))
```

```
<re.Match object; span=(0, 7), match='ba---ba'>
<re.Match object; span=(0, 8), match='ba----ba'>
None
```

In [28]:
```
# {,n}  Any number of repetitions of <regex> less than or equal to n
# {m,}  Any number of repetitions of <regex> greater than or equal to m
# {,}   Any number of repetitions of <regex>
# If you omit all of m, n, and the comma, then the curly braces no longer fu
nction as metacharacters.
# {} matches just the literal string '{}':

print(re.search('ba-{}ba','ba-{}ba'))
print(re.search('ba-{}','aaaba-{}baaaaa'))

# In fact, to have any special meaning, a sequence with curly braces must fi
t one of the following patterns
# in which m and n are nonnegative integers:
# {m,n}
# {m,}
# {,n}
# {,}
```

```
<re.Match object; span=(0, 7), match='ba-{}ba'>
<re.Match object; span=(3, 8), match='ba-{}'>
```

In [33]:
```
# {m,n}?
# {m,n} will match as many characters as possible, and {m,n}? will match as
 few as possible:

print(re.search('bam{3,5}','bammmmmmba'))
print(re.search('bam{3,5}?','bammmmba'))

# In this case, m{3,5} produces the longest possible match, so it matches fi
ve 'm' characters.
# m{3,5}? produces the shortest match, so it matches three.
```

```
<re.Match object; span=(0, 7), match='bammmmm'>
<re.Match object; span=(0, 5), match='bammm'>
```

In [34]:
```
# Grouping: A group represents a single syntactic entity. Additional metacha
racters apply to the entire group as a unit.

# (<regex>)
# This is the most basic grouping construct. A regex in parentheses just mat
ches the contents of the parentheses:

print(re.search('(bam)','bammmmmmba'))
print(re.search('bam','bammmmmmba'))

# As a regex, (bar) matches the string 'bar', the same as the regex bar woul
d without the parentheses.
```

```
<re.Match object; span=(0, 3), match='bam'>
```

```
<re.Match object; span=(0, 3), match='bam'>
<re.Match object; span=(0, 3), match='bam'>
```

In [17]:
```
# Treating a Group as a Unit

print(re.search('(bam)+','bammmmmmba'))
print(re.search('(bam)+','bambam'))
print(re.search('(a(bam))?','abambambam'))
print(re.search('bam+','bammmbam'))

# Here's a breakdown of the difference between the two regexes with and with
out grouping parentheses:
# bar+ = 'bar', 'barr', 'barrr'     (The + metacharacter applies only to the
character 'r'.)
# (bar)+ = 'bar', 'barbar', 'barbarbar'(The + metacharacter applies to the e
ntire string 'bar'.)
```

```
<re.Match object; span=(0, 3), match='bam'>
<re.Match object; span=(0, 6), match='bambam'>
<re.Match object; span=(0, 4), match='abam'>
<re.Match object; span=(0, 5), match='bammm'>
```

In [113]:
```
# coursera problem

text = """146.204.224.152 - feest6811 [21/Jun/2019:15:45:24 -0700] "POST /in
centivize HTTP/1.1" 302 4622
197.109.77.178 - kertzmann3129 [21/Jun/2019:15:45:25 -0700] "DELETE /virtua
l/solutions/target/web+services HTTP/2.0" 203 26554
156.127.178.177 - okuneva5222 [21/Jun/2019:15:45:27 -0700] "DELETE /interact
ive/transparent/niches/revolutionize HTTP/1.1" 416 14701
233.187.15.207 - - [21/Jun/2019:15:46:14 -0700] "GET /harness/intuitive HTT
P/1.0" 304 21006"""

pattern = '.+(?= - )'

list = [i.split(' -', 1)[0] for i in re.findall(pattern,text)]
print(list)
pattern = ' - .*(?= \[)'

list = [i.split(' - ', 1)[1] for i in re.findall(pattern,text)]
print(list)
pattern = '\[.*(?=\])'

list = [i.split('[', 1)[1] for i in re.findall(pattern,text)]
print(list)
pattern = '\".*(?=\")'

list = [i.split('"', 1)[1] for i in re.findall(pattern,text)]
print(list)
```

```
['146.204.224.152', '197.109.77.178', '156.127.178.177', '233.187.15.20
7']
['feest6811', 'kertzmann3129', 'okuneva5222', '-']
['21/Jun/2019:15:45:24 -0700', '21/Jun/2019:15:45:25 -0700', '21/Jun/201
9:15:45:27 -0700', '21/Jun/2019:15:46:14 -0700']
['POST /incentivize HTTP/1.1', 'DELETE /virtual/solutions/target/web+serv
ices HTTP/2.0', 'DELETE /interactive/transparent/niches/revolutionize HTT
P/1.1', 'GET /harness/intuitive HTTP/1.0']
```

In [12]:
```
# Now take a look at a more complicated example.
# The regex (ba[rz]){2,4}(qux)? matches 2 to 4 occurrences of either 'bar' o
r 'baz', optionally followed by 'qux':
```

```
import re
print(re.search('(ba[rz]){2,4}(qux)?', 'bazbarbazbarbarqux'))
print(re.search('(ba[rz]){2,4}(qux)?', 'barbar'))
print(re.search('(ba[rz]){2,4}qux?', 'barbarbarbazbazbazqux')) # now look ca
refully. qux is not in (). so this is not optional.
```

```
<re.Match object; span=(0, 12), match='bazbarbazbar'>
<re.Match object; span=(0, 6), match='barbar'>
<re.Match object; span=(6, 21), match='barbazbazbazqux'>
```

In [14]:
```
# The following example shows that you can nest grouping parentheses:

print(re.search('(foo(bar)?)+(\d\d\d)?', 'foofoobarbar'))
```

```
<re.Match object; span=(0, 9), match='foofoobar'>
```

In [10]:
```
#grouping
# returns a tuple
import re
m = re.search('(\w+),(\w+),(\w+)', 'foo.quux,baz,sd')
print(m)
m = re.search('(\w+).(\w+).', 'foo.quux,baz.sd')
print(m.groups())
```

```
<re.Match object; span=(4, 15), match='quux,baz,sd'>
('foo', 'quux')
```

In [22]:
```
# m.group(<n>)

# the arguments are one-based, not zero-based. So, m.group(1) refers to the
 first captured match,
# m.group(2) to the second, and so on:

m = re.search('(\w+),(\w+),(\w+)', 'foo.quux,baz,jk')

print(m.groups()) # entire tuple
print(m.group(0)) # all values of the tuple. m.group(0) returns the entire m
atch, and m.group() does the same.
print(m.group(1)) # the first value
print(m.group(2))
```

```
('quux', 'baz', 'jk')
quux,baz,jk
quux
baz
```

In [25]:
```
# m.group(<n1>, <n2>, ...)

print(m.groups())
print(m.group(1,2))
print(m.group(3,2,1))
```

```
('quux', 'baz', 'jk')
('quux', 'baz')
('jk', 'baz', 'quux')
```

In [10]:
```
# \<n>
import re
regex = r'(\w+),\1'
```

```
m = re.search(regex, 'foo,foo,foo')
print(m.groups())
```

```
('foo',)
```

In [8]:
```
import re
wiki = "tuba ok. you are ok .now ok .lazy girl ok"
for item in re.finditer("(?P<title>[\w ]*)(?P<edit_link>ok)",wiki):
    print(item.groupdict())

pattern = """ (?P<title>[\w ]*)
(?P<edit_link>ok)"""

for item in re.finditer(pattern,wiki,re.VERBOSE):

    print(item.groupdict())
```

```
{'title': 'tuba ', 'edit_link': 'ok'}
{'title': ' you are ', 'edit_link': 'ok'}
{'title': 'now ', 'edit_link': 'ok'}
{'title': 'lazy girl ', 'edit_link': 'ok'}
{'title': 'tuba ', 'edit_link': 'ok'}
{'title': ' you are ', 'edit_link': 'ok'}
{'title': 'now ', 'edit_link': 'ok'}
{'title': 'lazy girl ', 'edit_link': 'ok'}
```

In [19]:
```
text = """146.204.224.152 - feest6811 [21/Jun/2019:15:45:24 -0700] "POST /in
centivize HTTP/1.1" 302 4622
197.109.77.178 - kertzmann3129 [21/Jun/2019:15:45:25 -0700] "DELETE /virtua
l/solutions/target/web+services HTTP/2.0" 203 26554
156.127.178.177 - okuneva5222 [21/Jun/2019:15:45:27 -0700] "DELETE /interact
ive/transparent/niches/revolutionize HTTP/1.1" 416 14701
233.187.15.207 - - [21/Jun/2019:15:46:14 -0700] "GET /harness/intuitive HTT
P/1.0" 304 21006"""

pattern = '.+(?= - )'

list = [i.split(' -', 1)[0] for i in re.findall(pattern,text)]
print(list)
pattern = ' - .*(?= \[)'

list = [i.split(' - ', 1)[1] for i in re.findall(pattern,text)]
print(list)
pattern = '\[.*(?=\])'

list = [i.split('[', 1)[1] for i in re.findall(pattern,text)]
print(list)
pattern = '\".*(?=\")'

list = [i.split('"', 1)[1] for i in re.findall(pattern,text)]
print(list)
```

```
['146.204.224.152', '197.109.77.178', '156.127.178.177', '233.187.15.20
7']
['feest6811', 'kertzmann3129', 'okuneva5222', '-']
['21/Jun/2019:15:45:24 -0700', '21/Jun/2019:15:45:25 -0700', '21/Jun/201
9:15:45:27 -0700', '21/Jun/2019:15:46:14 -0700']
['POST /incentivize HTTP/1.1', 'DELETE /virtual/solutions/target/web+serv
ices HTTP/2.0', 'DELETE /interactive/transparent/niches/revolutionize HTT
P/1.1', 'GET /harness/intuitive HTTP/1.0']
```

In [ ]:
```
def logs():
    logs = []
    pattern = '(?P<host>(?:\d+\.){3}\d+)\s+(?:\S+)\s+(?P<user_name>\S+)\s+\
[(?P<time>[-+\w\s:/]+)\]\s+"(?P<request>.+?.+?)"'
    with open("assets/logdata.txt", "r") as f:
        logdata = f.read()
    for m in re.finditer(pattern, logdata):
        logs.append(m.groupdict())
    return logs
```