Python: find regexp in a file

Asked 12 years, 11 months ago Modified 8 years, 6 months ago Viewed 14k times

13

Have:

```
f = open(...)
r = re.compile(...)
```

Need:
Find the position (start and end) of a first matching regexp in a big file?
(starting from `current_pos=...`)

How can I do this?

I want to have this function:

```
def find_first_regex_in_file(f, regexp, start_pos=0):
   f.seek(start_pos)

   .... (searching f for regexp starting from start_pos) HOW?

   return [match_start, match_end]
```

File 'f' is expected to be big.

python regex

Share Edit Follow

edited Feb 14, 2011 at 5:58

asked Feb 14, 2011 at 5:44

Sergey
20.1k 13 44 70

Could you show a more complete example of what you want to do? With some sample inputs and outputs along with it. – Jeff Mercado Feb 14, 2011 at 5:50

3 Answers

Sorted by: Highest score (default)

33

One way to search through big files is to use the `mmap` library to map the file into a big memory chunk. Then you can search through it without having to explicitly read it.

For example, something like:

```
size = os.stat(fn).st_size
f = open(fn)
data = mmap.mmap(f.fileno(), size, access=mmap.ACCESS_READ)

m = re.search(r"867-?5309", data)
```

This works well for very big files (I've done it for a file 30+ GB in size, but you'll need a 64-bit OS if your file is more than a GB or two).

Share Edit Follow

answered Feb 14, 2011 at 6:08

Greg Hewgill
966k 186 1155 1289

looks great, I'll check it as soon as I can – Sergey Feb 14, 2011 at 6:13

What about searching for data coming from socket ?! – socketpair Jan 3, 2016 at 19:54

1 If you use a size parameter of `0`, it will use the entire file. – mbomb007 Oct 2, 2017 at 14:14

4

The following code works reasonably well with test files around 2GB in size.

```
def search_file(pattern, filename, offset=0):
    with open(filename) as f:
        f.seek(offset)
        for line in f:
            m = pattern.search(line)
            if m:
                search_offset = f.tell() - len(line) - 1
                return search_offset + m.start(), search_offset + m.end()
```

Note that the regular expression must not span multiple lines.

Share Edit Follow

edited Feb 14, 2011 at 22:46

answered Feb 14, 2011 at 7:07

Adeel Zafar Soomro
1,492 9 15

0

NOTE: this has been tested on python2.7. You may have to tweak things in python 3 to handle strings vs bytes but it shouldn't be too painful hopefully.

Memory mapped files may not be ideal for your situation (32-bit mode increases chance there isn't enough contiguous virtual memory, can't read from pipes or other non-files, etc).

Here is a solution that reads 128k blocks at a time and as long as your regex matches a string smaller than that size, this will work. Also note you are not restricted by using single-line regexes. This solution works plenty fast, although I suspect it will be marginally slower than using mmap. It probably depends more on what you're doing with the matches, as well as the size/complexity of the regex you're searching for.

The method will make sure to only keep a maximum of 2 blocks in memory. You might want to enforce at least 1 match per block as a sanity check in some use cases, but this method will truncate in order to keep the maximum of 2 blocks in memory. It also makes sure that any regex match that eats to the end of the current block is NOT yielded and instead the last position is saved for when either the true input is exhausted or we have another block that the regex matches before the end of, in order to better match patterns like "[^\n]+" or "xxx$". You may still be able to break things if you have a lookahead at the end of the regex like xx(?!xyz) where yz is in the next block, but in most cases you can work around using such patterns.

```
import re

def regex_stream(regex,stream,block_size=128*1024):
    stream_read=stream.read
    finditer=regex.finditer
    block=stream_read(block_size)
    if not block:
        return
    lastpos = 0
    for mo in finditer(block):
        if mo.end()!=len(block):
            yield mo
            lastpos = mo.end()
        else:
            break
    while True:
        new_buffer = stream_read(block_size)
        if not new_buffer:
            break
        if lastpos:
            size_to_append=len(block)-lastpos
            if size_to_append > block_size:
                block='%s%s'%(block[-block_size:],new_buffer)
            else:
                block='%s%s'%(block[lastpos:],new_buffer)
        else:
            size_to_append=len(block)
            if size_to_append > block_size:
                block='%s%s'%(block[-block_size:],new_buffer)
            else:
                block='%s%s'%(block,new_buffer)
        lastpos = 0
        for mo in finditer(block):
            if mo.end()!=len(block):
                yield mo
                lastpos = mo.end()
            else:
                break
    if lastpos:
        block=block[lastpos:]
    for mo in finditer(block):
        yield mo
```

To test / explore, you can run this:

```
# NOTE: you can substitute a real file stream here for t_in but using this as a
test
t_in=cStringIO.StringIO('testing this is a 1regexxx\nanother 2regexx\nmore
3regexes')
```

```
block_size=len('testing this is a regex')
re_pattern=re.compile(r'\dregex+',re.DOTALL)
for match_obj in regex_stream(re_pattern,t_in,block_size=block_size):
    print 'found regex in block of len %s/%s: "%s[[[%s]]]%s"'%(
        len(match_obj.string),
        block_size,match_obj.string[:match_obj.start()].encode('string_escape'),
        match_obj.group(),
        match_obj.string[match_obj.end():].encode('string_escape'))
```

Here is the output:

```
found regex in block of len 46/23: "testing this is a [[[1regexxx]]]\nanother
2regexx\nmor"
found regex in block of len 46/23: "testing this is a 1regexxx\nanother
[[[2regexx]]]\nmor"
found regex in block of len 14/23: "\nmore [[[3regex]]]es"
```

This can be useful in conjunction with quick-parsing a large XML where it can be split up into mini-DOMs based on a sub element as root, instead of having to dive into handling callbacks and states when using a SAX parser. It also allows you to filter through XML faster as well. But I've used it for tons of other purposes as well. I'm kind of surprised recipes like this aren't more readily available on the net!

One more thing: Parsing in unicode should work as long as the passed in stream is producing unicode strings, and if you're using the character classes like \w, you'll need to add the re.U flag to the re.compile pattern construction. In this case block_size actually means character count instead of byte count.

Share Edit Follow

answered Jul 16, 2015 at 1:41

parity3

673 10 18