The Performance Comparison of a Brute-Force Password Cracking Algorithm Using Regular Functions and Generator Functions in Python

Berker Tasoluk¹ and Zuhal Tanrikulu²

¹ Informatics, Istanbul University, Istanbul, Turkey
² Management Information Systems, Bogazici University, Istanbul, Turkey

Abstract

Python is used extensively in research, including algorithm testing. Python is a multi-paradigm programming language and supports both object-oriented programming and functional programming. In the functional side, it supports both regular functions and generator functions. This study tests both approaches in terms of usability cases and performance. A password-cracking algorithm is used for this tryout.

Keywords

Python functions, generator functions, regular functions, regular functions, in-memory functions, iterator, generator expressions, time efficiency, memory efficiency, python programming, general programming, password cracking

1. Introduction

In academic research field, Python is used extensively, including algorithm testing, Machine Learning [1], data science [2], web scraping [3], text and language processing [4, 5]. Python is a multi-paradigm programming language and supports both object-oriented programming [6] and functional programming [7]. Functional side of python supports both regular functions and generator functions [8]. This study tests both approaches in terms of usability cases and performance. A password-cracking algorithm is used for this tryout. Passwords are used extensively in the authentication process, by users to identify themselves to the systems they want to access [9]. Each password is associated with a specific user id in the system like email address, username, or phone number, and by providing password, users prove that they are who they claim to be [10]. Passwords connects identification and authentication processes.

In this study, python regular functions and generators are tested for their memory usage and duration taken to try a certain number of passwords based on a given set of criteria. At the discussion part, practical usage advice is given for both regular and generator functions.

2. Methods

In this study, our aim is to compare the two implementations of functions in the field, using a password cracking undertaking. We have a set of passwords, which eventually would be cracked,
which is called destination list passwords; and we have another set of passwords, which would be used to crack the destination list passwords, which we call source list passwords. Destination list passwords are composed of real-world revealed passwords. We try each entry in the source list with each entry in the destination list. When we find a match between source list entry and destination list entry, we suppose that the password is ‘broken’.

2.1. Destination (Target) List of Passwords

Destination list passwords, and the number of passwords included in them are as follows:

![Command Prompt - python]

Figure 1. Destination password list details

And the file size of each password file is as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>alypaa.txt</td>
<td>10/12/2011 10:23 ...</td>
<td>Text Document</td>
<td>12 KB</td>
</tr>
<tr>
<td>carders.cc.txt</td>
<td>10/12/2011 10:24 ...</td>
<td>Text Document</td>
<td>17 KB</td>
</tr>
<tr>
<td>elitehacker.txt</td>
<td>10/12/2011 10:23 ...</td>
<td>Text Document</td>
<td>7 KB</td>
</tr>
<tr>
<td>facebook-pastebay.txt</td>
<td>10/12/2011 10:23 ...</td>
<td>Text Document</td>
<td>1 KB</td>
</tr>
<tr>
<td>facebook-phished.txt</td>
<td>10/12/2011 10:24 ...</td>
<td>Text Document</td>
<td>26 KB</td>
</tr>
<tr>
<td>faithwriters.txt</td>
<td>10/12/2011 10:23 ...</td>
<td>Text Document</td>
<td>71 KB</td>
</tr>
<tr>
<td>hak5.txt</td>
<td>10/12/2011 10:23 ...</td>
<td>Text Document</td>
<td>25 KB</td>
</tr>
<tr>
<td>hotmail.txt</td>
<td>10/12/2011 10:23 ...</td>
<td>Text Document</td>
<td>86 KB</td>
</tr>
<tr>
<td>myspace.txt</td>
<td>10/12/2011 10:23 ...</td>
<td>Text Document</td>
<td>348 KB</td>
</tr>
<tr>
<td>porn-unknown.txt</td>
<td>10/12/2011 10:24 ...</td>
<td>Text Document</td>
<td>57 KB</td>
</tr>
<tr>
<td>singles.org.txt</td>
<td>10/12/2011 10:24 ...</td>
<td>Text Document</td>
<td>105 KB</td>
</tr>
<tr>
<td>tuscl.txt</td>
<td>10/12/2011 10:24 ...</td>
<td>Text Document</td>
<td>318 KB</td>
</tr>
</tbody>
</table>
2.2. Source List of Passwords

Here, the brute force passwords are used for password cracking. Uppercase, lowercase ascii letters and digits are used to form the source lists. Every possible combination of the following character sets are used in creating the password files. Python string module properties are used for the character sets.

```python
>>> import string
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (26 different characters)
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz' (26 different characters)
>>> string.digits
'0123456789' (10 different characters)
```

1, 2, 3, 4 and 5-character lists are created in this manner.

The number of entries for each list is calculated as follows:
1 character list: $62^{1} = 62$ entries
2 character list: $62^{2} = 3,844$ entries
3 character list: $62^{3} = 238,328$ entries
4 character list: $62^{4} = 14,776,336$ entries
5 character list: $62^{5} = 916,132,832$ entries

And these numbers are also obtained experimentally, using countline_dir() python function written.

```
>>> import os
>>> countline_dir('C:\Users\bt\OneDrive\TEZ\PYTHON\python codes\brutelist')
brulist-1uld.txt has 62 lines
brulist-2uld.txt has 3,844 lines
brulist-3uld.txt has 238,328 lines
brulist-4uld.txt has 14,776,336 lines
brulist-5uld.txt has 916,132,832 lines
```

![Figure 3. Experimentally calculated line counts](image)

And the space occupied in disk for each list is as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>brutelist-1uld.txt</td>
<td>✔️</td>
<td>4/26/2020 11:32 A...</td>
<td>Text Document</td>
<td>1 KB</td>
</tr>
<tr>
<td>brutelist-3uld.txt</td>
<td>✔️</td>
<td>4/26/2020 11:32 A...</td>
<td>Text Document</td>
<td>1,164 KB</td>
</tr>
<tr>
<td>brutelist-5uld.txt</td>
<td>✔️</td>
<td>4/26/2020 11:46 A...</td>
<td>Text Document</td>
<td>6,262,627 KB</td>
</tr>
</tbody>
</table>
The code ran on a machine with following specs:
- 16 GB Ram, Intel Core i7-8750H 2.20 Ghz processor, Windows 10 Enterprise LTSC Operating System, 512 GB SSD Disk

3. RESULTS

3.1. Regular Python Functions

Each destination list entry is tried to be matched with 1-char through 5-char lists; and the result is as follows:

1-char list took under 1 second to be completed, while 2-char list took under 1 min, 3-char list completed under 1 hour. 4-char list took 1.8 days to be completed, and 5-char list couldn’t be completed due to memory error. This memory error is the reason behind re-writing the algorithm in python using generator functions.

If we look at the relationship between time taken to try all entries in the list (y), with the number of entries in the list (x), then we get approximately \( y = \frac{1}{95}x \) linear relationship. From that, we may conclude that, if we had unlimited memory, it would take approximately 111 days for a 5-char password list to be completed.

<table>
<thead>
<tr>
<th>List name</th>
<th>Match count</th>
<th>Try count</th>
<th>Count per match</th>
<th>Line count</th>
<th>Duration (SECONDS)</th>
<th>Duration (HOURS)</th>
<th>Duration (DAYS)</th>
<th>Duration (YEARS)</th>
<th>Line count / Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>bruteList-500 txt</td>
<td>30</td>
<td>11,431,503</td>
<td>381,050</td>
<td>62</td>
<td>0.59</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>105</td>
</tr>
<tr>
<td>bruteList-1,000 txt</td>
<td>765</td>
<td>43,875,306,209</td>
<td>57,353,342</td>
<td>238,328</td>
<td>2.511.3</td>
<td>0.70</td>
<td>0.08</td>
<td>0.00</td>
<td>95</td>
</tr>
<tr>
<td>bruteList-2,000 txt</td>
<td>4505</td>
<td>1,704,852,164,340</td>
<td>592,528,404</td>
<td>14,770,236</td>
<td>155,652.00</td>
<td>43.25</td>
<td>1.80</td>
<td>0.00</td>
<td>95</td>
</tr>
<tr>
<td>bruteList-5,000 txt</td>
<td>916,132,852</td>
<td>9,602,824.00</td>
<td>2,694.12</td>
<td>111.84</td>
<td>0.306400</td>
<td>95</td>
<td>(APPROX)</td>
<td></td>
<td>(APPROX)</td>
</tr>
<tr>
<td>bruteList-50,000 txt</td>
<td>56,800,230,504</td>
<td>359,099,506.00</td>
<td>166.415.30</td>
<td>5,933.97</td>
<td>18.99718</td>
<td>95</td>
<td>(APPROX)</td>
<td></td>
<td>(APPROX)</td>
</tr>
<tr>
<td>bruteList-500,000 txt</td>
<td>5,221,614,606,208</td>
<td>37,143,990,436.00</td>
<td>30,937,248.74</td>
<td>429,939.20</td>
<td>1177.825</td>
<td>95</td>
<td>(APPROX)</td>
<td></td>
<td>(APPROX)</td>
</tr>
<tr>
<td>bruteList-1,000,000 txt</td>
<td>218,340,105,584,896</td>
<td>2,302,921,518,272.00</td>
<td>619,700,421.74</td>
<td>26,654,184.24</td>
<td>73025.16</td>
<td>95</td>
<td>(APPROX)</td>
<td></td>
<td>(APPROX)</td>
</tr>
</tbody>
</table>
3.2. Generator Python Functions

Generator functions use iterators, and instead of getting the source list into the memory at once, they use iterator objects and performs every operation one by one. The old ones are cleared up from the memory, and the next entry in the iterator is loaded into the memory.

The results from the generator function is as follows:

![Figure 6. Password cracking tryout using generator python functions](image)

It took nearly 4 times as much time as it took in using all in-memory python functions. From this graph, it may be concluded that 4-char source list would take around 1 week (1.8 days x 4), and 5-char source list would take around more than a year (111 days x 4).

4. DISCUSSION

As we can see from the above experiments and calculations; for our case; if we have resources available, especially the memory; it is better to have the function written as a regular python function which works all in-memory; instead of using generators. In our scenario; using generators is efficient in terms of memory, but also slower. If we have time, and not the resources; then using generator functions may be needed; and sometimes it may be the only solution we have in hand.
REFERENCES