**Python Final Project - ZIP and RAR Password Cracker**

**Page**  **Details**

2……………………………....……………………………………………Process vs Thread

3………………………………..…………………………………………………..….Modules

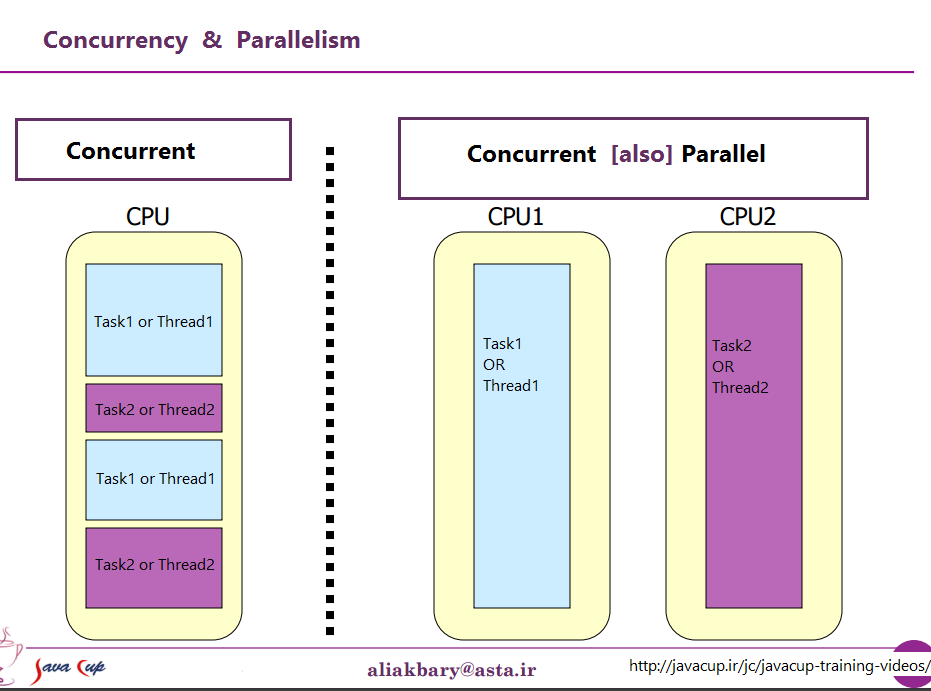
4…….……………………………………………………………….…Functions breakdown

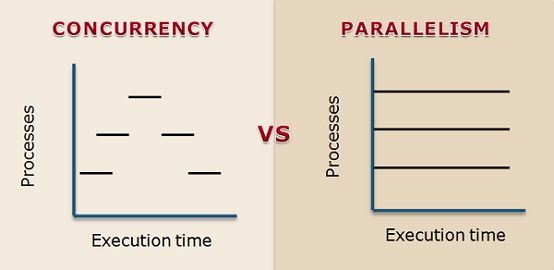
14…….…………...…………………………………………………………..…..Main Function

16…………………………………………………………………………………………..Code

Process vs Thread

* **Why did I choose multiprocessing instead of multithreading?**
* I decided that instead of searching through one list of 100k or even 1m passwords, I’ll cut down the list into smaller chunks, and have each process search in a unique sub-list so it will be much faster.
* Because of the GIL (Global Interpreter Lock) in python, threads cannot work in parallel, so true parallelism doesn’t happen with them. The GIL makes sure threads work in separate times so they won’t corrupt or destroy data they may manipulate.
* Processes can work in parallel (as long as they read data and not write data), and achieve faster results. Each process has its own interpreter, and much more computing power goes into them.
* Threads are concurrent, meaning they are ideal for jumping between tasks (which isn’t ideal in my case, and will be the same as running through one big list). While processes are parallel (and concurent), meaning they can work at the same time.
* Since searching for the passwords in multiple lists should be fast and happen at the same time for all lists, Processes are way better for this role, cutting the time down by 4 or even 8 times! (depends on PC)
* Threads and Processes can be switched to see the difference in the resulting execution time: just change ProcessPoolExecutor to ThreadPoolExecutor and see for yourself!





Modules

|  |
| --- |
| import concurrent.futures import time import rarfile import requests import os import psutil import pyzipper |

* **concurrent.futures** - managing the multiprocessing.
* **time** - measuring the time it takes to finish cracking.
* **rarfile**, **pyzipper** - archive related methods: extracting, file reading, and password guessing.
* **requests** - getting the most common passwords list from the internet.
* **psutil** - accessing the hardware, to know core count and CPU load.
* **os** - managing paths and helping extraction.

Function Breakdown

* **get\_optimal\_cpu()**
* Input: None
* Output: int (number of processes that will run in parallel)

|  |
| --- |
| def get\_optimal\_cpu() -> int:  total\_cpus : int = os.cpu\_count()  current\_cpu\_load : float = psutil.cpu\_percent(interval=1)  optimal\_maximum\_processes : int = 1   if current\_cpu\_load > 80:  optimal\_maximum\_processes = max(1, total\_cpus // 2)  elif current\_cpu\_load > 50:  optimal\_maximum\_processes = max(1, (total\_cpus \* 3) // 4)  else:  optimal\_maximum\_processes = total\_cpus  return optimal\_maximum\_processes |

* get\_optimal\_cpu() is a function that returns an int value, which is the number of processes the os can run efficiently. First, it gets the number of available CPU cores the user has with os.cpu\_count(). Then, the current CPU load is stored with psutil.cpu\_percent().
* With those values, the program determines how many processes should be used in order to crack the password. If the CPU load is more than 80% - use half the cores. If the CPU load is between 50%-80% - use three quarters of the cores. Else, use all of the cores.

**100%**

**80%**

**50%**

**0%**

**Process 1**

**Process 2**

**Process 3**

**Process 2**

**Process 1**

**Process 2**

**Process 4**

**Process 3**

**Process 1**

CPU LOAD

* **split\_passwords\_list(max\_processes)**
* Input: int (number of processes)
* Output: list[list[str]] (divided list of passwords)

|  |
| --- |
| def split\_passwords\_list(max\_processes) -> list[list[str]]:  response = requests.get("https://raw.githubusercontent.com/danielmiessler/SecLists/refs/heads/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt")  passwords\_list : list[str] = response.text.splitlines()   divided\_passwords\_list : list[list[str]] = []   for \_ in range(max\_processes):  divided\_passwords\_list.append([])  chunk : int = 0   for i in passwords\_list:  if chunk == len(divided\_passwords\_list):  chunk = 0  divided\_passwords\_list[chunk].append(i)  chunk += 1  return divided\_passwords\_list |

* split\_function\_list() is a function that takes the calculated number of processes that will run, and divides the list of passwords in a way that every process will handle a different chunk of it.
* The list of passwords is retrieved from the link inserted into the request.get() method, and it is stored in a list. Then, the program declares the divided\_passwords\_list, which is a list that contains sub-lists with the str type. For every number of processes, the program appends an empty list into the divided one (an example for 4 processes below).

[ [] , [] , [] , [] ]

* Iterating over the passwords list, every password is inserted into a different “chunk” (chunk = sublist = process), that way every process will have different passwords to try. The function returns that divided list of passwords (an example for 4 processes below).

**[ p1, p2 , p3 , p4 , p5 , p6 , p7 , p8 , p9 , p10 , p11 , p12]**

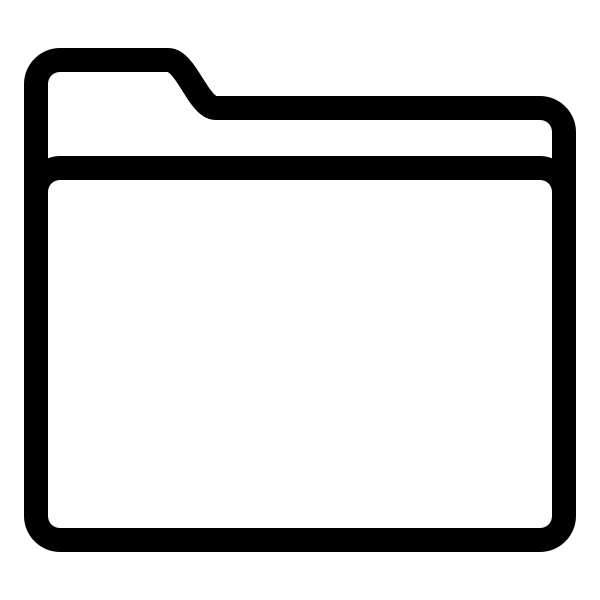
…Turns into this…

**[** [**p1, p5, p9**] , [**p2, p6, p10**] , [**p3, p7, p11**] , [**p4, p8, p12**] **]**

* **get\_name\_for\_extract(path)**
* Input: str (path of the archive)
* Output: str (name of the archive, empty string if wasn’t found)

|  |
| --- |
| def get\_name\_for\_extract(path) -> str:  if path.endswith(".zip"):  try:  with pyzipper.AESZipFile(path, 'r') as f:  return f.filename  except:  print("error getting file name")  return ""  else:  try:  with rarfile.RarFile(path, "r") as f:  return f.filename  except:  print("error getting file name")  return "" |

* get\_name\_for\_extract() will take the path of the archive file and get its name for later extraction. It differentiates between .zip and .rar files, using respective libraries but the same methods.



ZipFile.zip

f.filename

ZipFile

* **get\_archive\_directory(path)**
* Input: str (path of the archive)
* Output: str (directory of the archive, without file name)

|  |
| --- |
| def get\_archive\_directory(path):  last\_slash : int = path.rfind("/")  if last\_slash == -1:  last\_slash = path.rfind("\\")  Path\_name : str = path[0:last\_slash]  return path\_name |

* get\_archive\_directory() takes the path of the archive and returns the path to the directory the archive is in, in a string format.

C:/Users/User/FolderA/FolderB/archive.zip

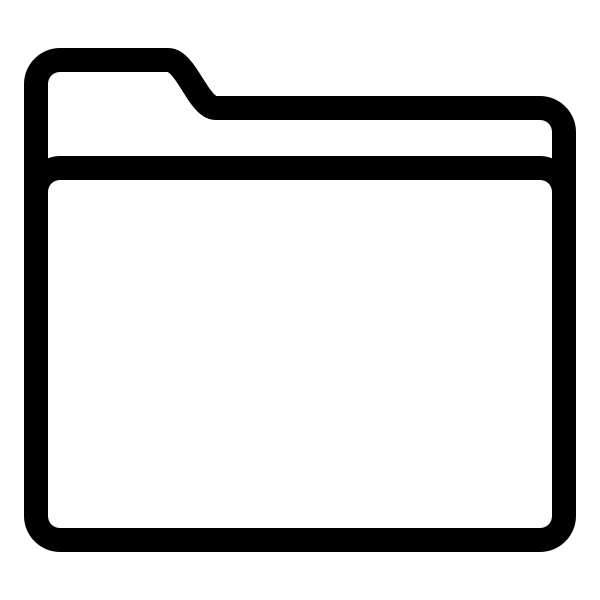
Slicing this part (the directory)

Last\_slash (index)

* **get\_filename\_for\_testing(path)**
* Input: str (path of the archive)
* Output: str (name of the first file the function finds inside the archive)

|  |
| --- |
| def get\_filename\_for\_testing(path) -> str:  if path.endswith(".zip"):  with pyzipper.AESZipFile(path, "r") as f:  for i in f.infolist():  if not i.is\_dir():  return i.filename  else:  with rarfile.RarFile(path, "r") as f:  for i in f.infolist():  if i.is\_file():  return i.filename   return "" |

* get\_filename\_for\_testing() will return the name of the first file the script finds. The file name is essential for the next function, which will test a password on the given file.



ZipFile.zip

f.infolist()

[ folder1, folder2, folder3, folder1/text.txt, folder2/folder4, … ]

* **try\_pass(chunk, path)**
* Input: list[str] (chunk of passwords), str (path of the archive)
* Output: str (correct password, empty if wasn’t found)

|  |
| --- |
| def try\_pass(chunk, path) -> str:  name = get\_filename\_for\_testing(path)  if name:  if path.endswith(".zip"):  for password in chunk:  with pyzipper.AESZipFile(path, "r") as f:  try:  f.read(name, pwd=bytes(password.encode()))  return password  except:  continue  else:  for password in chunk:  with rarfile.RarFile(path, "r") as f:  try:  f.read(name, pwd=password)  return password  except:  continue   return "" |

* try\_pass() is a function that runs by each process. It takes a list of passwords (named chunk) and the path of the archive. First it takes the name of a file from the archive (using the earlier function) and for each type of archive (zip or rar) will try a password by reading the file with the read() function of each library. If the password is not correct it raises an error, and to avoid it the script continues to the next password with exception handling. If it doesn’t find a password it returns an empty string (None).

Process 1

Chunk 1

[p1,p5,p9,...]

Process 2

Chunk 2

[p2,p6,p10,...]

Process 3

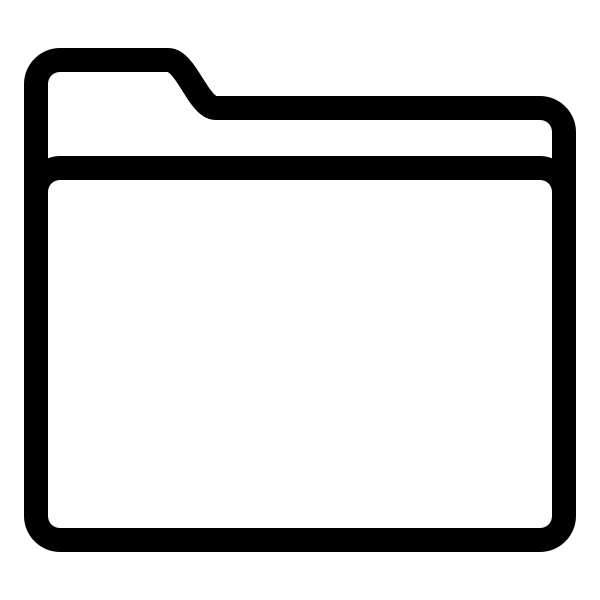
Chunk 3

[p3,p7,p11,...]

Process 4

Chunk 4

[p4,p8,p12,...]



ZIP/RAR

**f.read(file, p1)**

**f.read(file, p2)**

**f.read(file, p4)**

**f.read(file, p3)**

* **brute\_cracking(path)**
* Input: str (path of the archive)
* Output: str (correct password, empty string if wasn’t found)

|  |
| --- |
| def brute\_cracking(path) -> str:  optimal\_max\_processes : int = get\_optimal\_cpu()  passwords\_list\_chunks : list[list[str]] = split\_passwords\_list(optimal\_max\_processes)  with concurrent.futures.ProcessPoolExecutor() as executor:  results = [executor.submit(try\_pass, passwords\_list\_chunks[i], path) for i in range(optimal\_max\_processes)]  for f in concurrent.futures.as\_completed(results):  if f.result() != "":  return f.result()  return "" |

brute\_cracking() is the function that manages all the processes. With ProccessPoolExecuter() it creates a list containing all the processes that are created. Every element of the list is a Future object that manages the results/status of its own try\_pass() function of the process submitted to the executor. The context manager starts them and joins them all together, and then it waits for the results from all the processes by calling the as\_completed() method, which in turn will return a tuple inside a set of all futures (results of the processes). Then the code looks inside every Future object and its returned value. If the value is not None (an empty string), it means a correct password was found, and it is returned to the main function.

“Executor Pool”

Processes:

**1**

**2**

**3**

**4**

as\_completed(results)

**results**

**1**

**4**

**3**

**2**

result**:**

“”

result:

“”

result:

“pass”

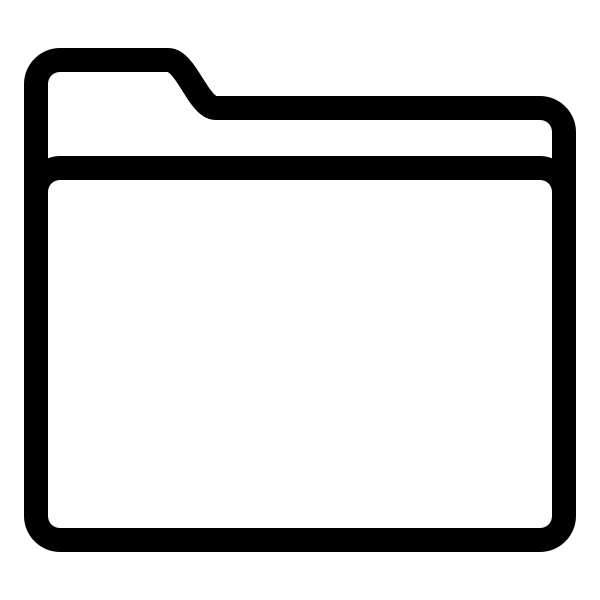
result:

“”

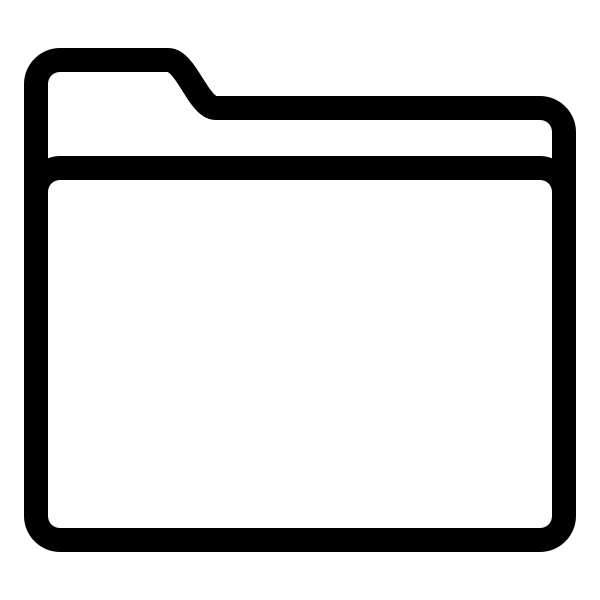
* **extract\_zip(path, password, path\_to\_extract)**
* Input: str (path of the archive), str (correct password), str (path of extraction folder)
* Output: None

|  |
| --- |
| def extract\_zip(path, password, path\_to\_extract) -> None:  if path.endswith(".zip"):  with pyzipper.AESZipFile(path, "r") as f:  f.extractall(path\_to\_extract, pwd=bytes(password.encode()))  else:  with rarfile.RarFile(path, "r") as f:  f.extractall(path\_to\_extract, pwd=password.encode) |

extract\_zip() is an extra function that extracts all of the data inside of the archive into a folder, using the methods belonging to the corresponding modules. It takes the path of the archive, the correct password and the path to extract to.



zipfile.zip



zipfile\_extracted

f.extractall()

Main Function

|  |
| --- |
| if \_\_name\_\_ == '\_\_main\_\_':  while True:  path\_to\_zip : str = input("Enter the path of the zip or rar file: ")  if path\_to\_zip.endswith(".zip") or path\_to\_zip.endswith(".rar"):  if os.path.exists(path\_to\_zip):  break  else:  print(f"Directory {path\_to\_zip} doesn't exist")  else:  print("incorrect file type: not a zip or rar file") |

* First the script asks the user for the path of the archive. The path must be valid, and be of a zip or rar type.

|  |
| --- |
| start : float = time.perf\_counter()  print("Please wait, this can take some time...") correct\_password : str = brute\_cracking(path\_to\_zip) finish : float = time.perf\_counter()  print(f"Finished in: {finish - start} seconds") |

* Then, a timer starts measuring the time and the brute force cracking starts. At this time the user waits for the results until the processes finish. The total time it took for the cracking is printed to the user.

|  |
| --- |
| if not correct\_password:  print("couldn't crack password or compressed file is empty without files") else:  print(f"Password cracked: {correct\_password}")  choice = input("Would you like to extract the content of the compressed file? (Y/<any other key to exit>]): ") |

* If a correct password was found or not, the user gets to see it printed in the console. If a password was found, the user gets asked if they want to extract the data from the compressed archive.

|  |
| --- |
| if choice.lower() == 'y':  print("making folder...")  extracted\_content\_path : str = os.path.join(get\_archive\_directory(path\_to\_zip), f"{get\_name\_for\_extract(path\_to\_zip)}\_extracted")   i = 1  while True:  try:  os.makedirs(extracted\_content\_path)  break  except:  extracted\_content\_path = os.path.join(get\_archive\_directory(path\_to\_zip), f"{get\_name\_for\_extract(path\_to\_zip)}\_extracted\_{i}")  i += 1  print("extracting data...")  extract\_zip(path\_to\_zip, correct\_password, extracted\_content\_path)  print("done!!") |

* If the user chose yes (entered ‘y’) then a path for the extracted folder is created with the os.path.join() method, which takes the directory of the archive (with get\_archive\_directory()) and the name of the folder to create. The code enters a while loop and tries the actual creation of the folder with os.makedirs() method. If by chance, the user already extracted the data or a folder with a similar name is existing, it raises an error and to avoid it, a counter is updated for every try of the os.makedirs() method. The iterator i is being updated by 1 for every error being raised, and the number is added to the folder that is going to be created (much like “New Folder”, “New Folder (1)”, “New Folder (2)”...)
* After the folder is created, the data from the archive is extracted into it, and the code exits the main function.