Python Final Project - ZIP and RAR Password Cracker

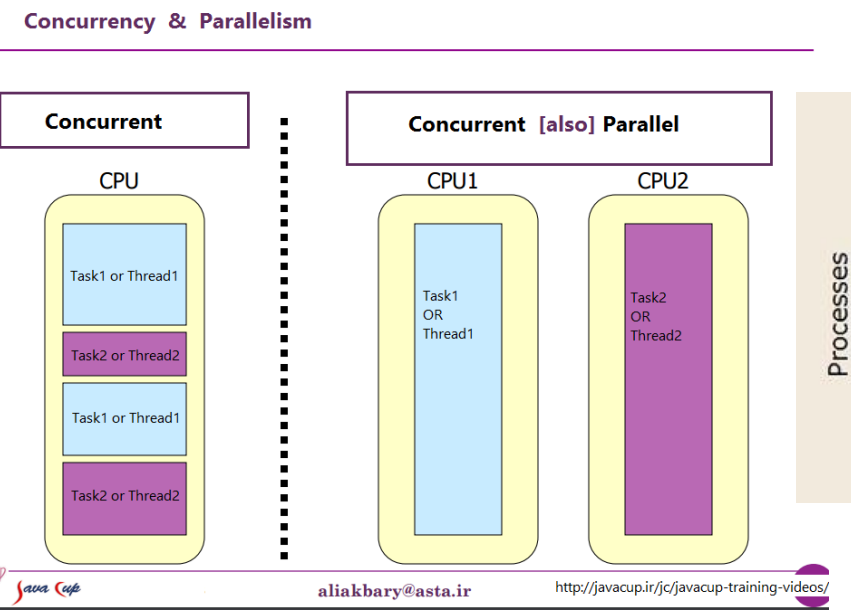
Made by: Yuval (Kozi) Kozlovski

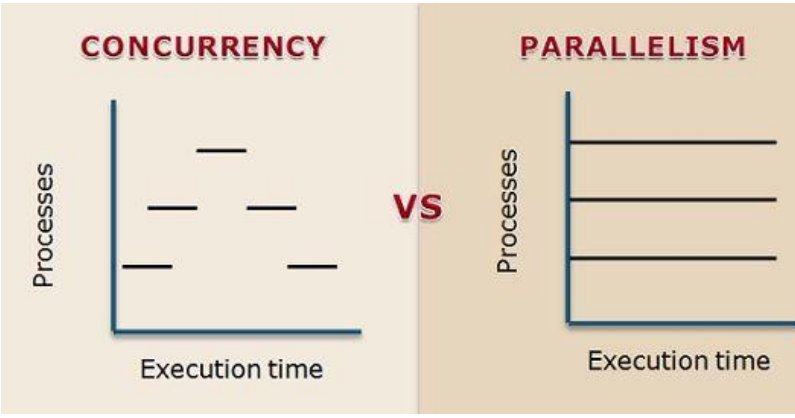
Page **Details**
2……Process vs Thread
3……Security aspect
4……Modules
5……Functions breakdown
15……Main Function
17……Code

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 concurrent), 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!





Weak passwords and Cybersecurity

Weak passwords are still one of the easiest ways to lose control of data. People use short words, names, or simple patterns because they're easy to remember, but that makes them easy to guess. When attackers get a copy of an encrypted file, they can try guesses offline and aren't slowed by lockouts or rate limits, so weak passwords quickly become a practical vulnerability, as they are an important brick of security between classified data and confidentiality exposure.

When attackers target a protected file, code is a beneficial partner for them. At a higher level the software does three jobs - queue possible passwords, test them against the encrypted data, and view the results - while optimizations in the code like distributing work across threads or machines make the process far faster than manual guessing. Developers also encode heuristics and priorities into the tool (which wordlists to try first, which mutation rules to run, when to fall back to wider searches), and they use checks so the program can quickly detect a successful decryption without wasting cycles. All of this is why attacker tooling can rapidly turn a low-entropy secret into a broken one.

Defenses are straightforward. Use long passphrases (12-16+ characters or several random words), don't reuse passwords, and store credentials in a password manager. For archives, prefer modern encryption like AES and strong key-derivation settings so each guess costs the attacker time. The lesson for defenders is that good software design benefits both sides, so strong cryptography and complex passwords remain the best deterrents.

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.

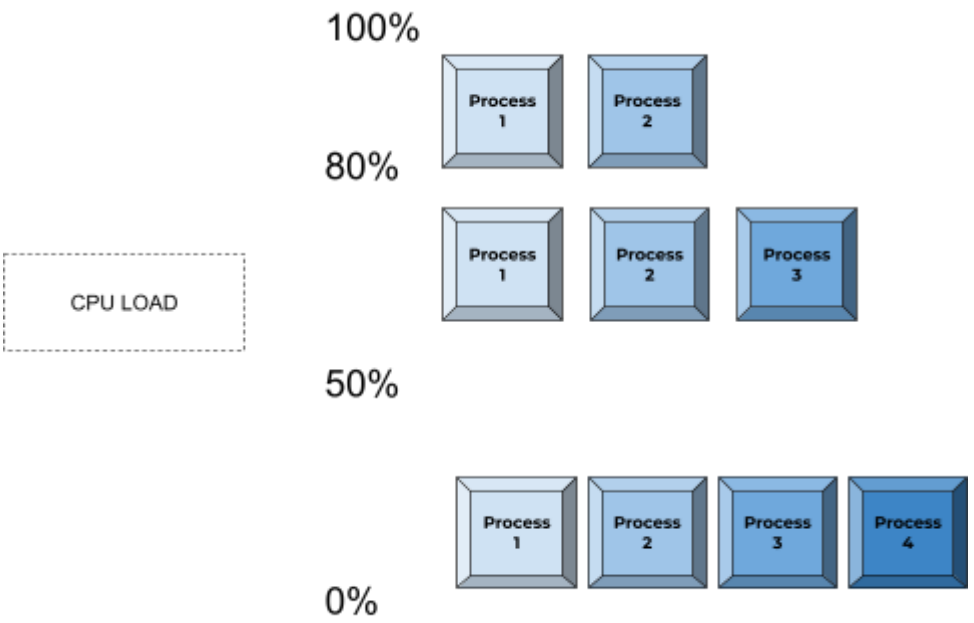
Function Breakdown

- **`get_optimal_cpu()`**
- Input: None
- Output: int (number of processes that will run in parallel)

```python
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.



- **`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/maste
r/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]

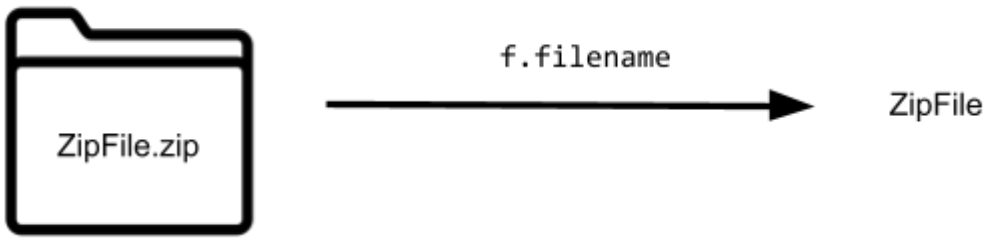
...Turns into this...

[[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 ""
```

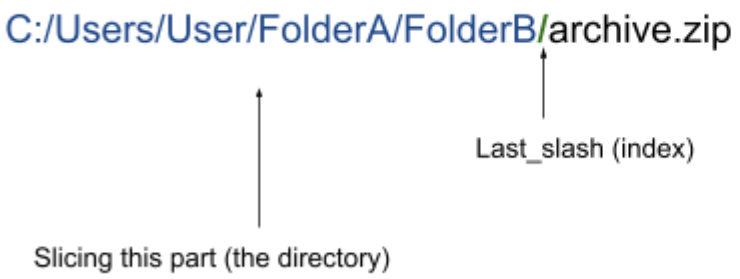
- `get_name_for_extract()` will take the path of the archive file and get its name for later extraction. It differentiates between .zip and .rar files, using respective libraries but the same methods.



- **`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.

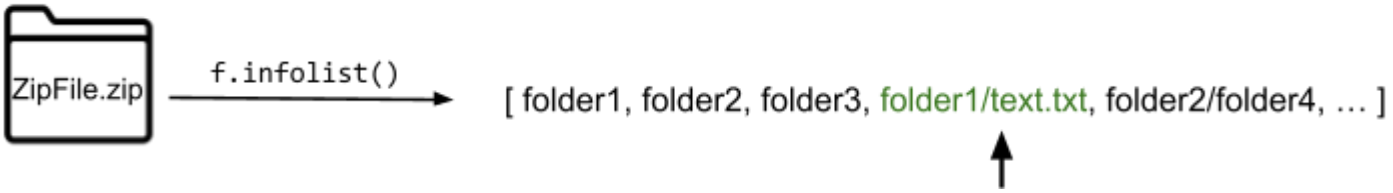


- **`get_filename_for_testing(path)`**
- Input: str (path of the archive)
- Output: str (name of the first file the function finds inside the archive)

```python
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.

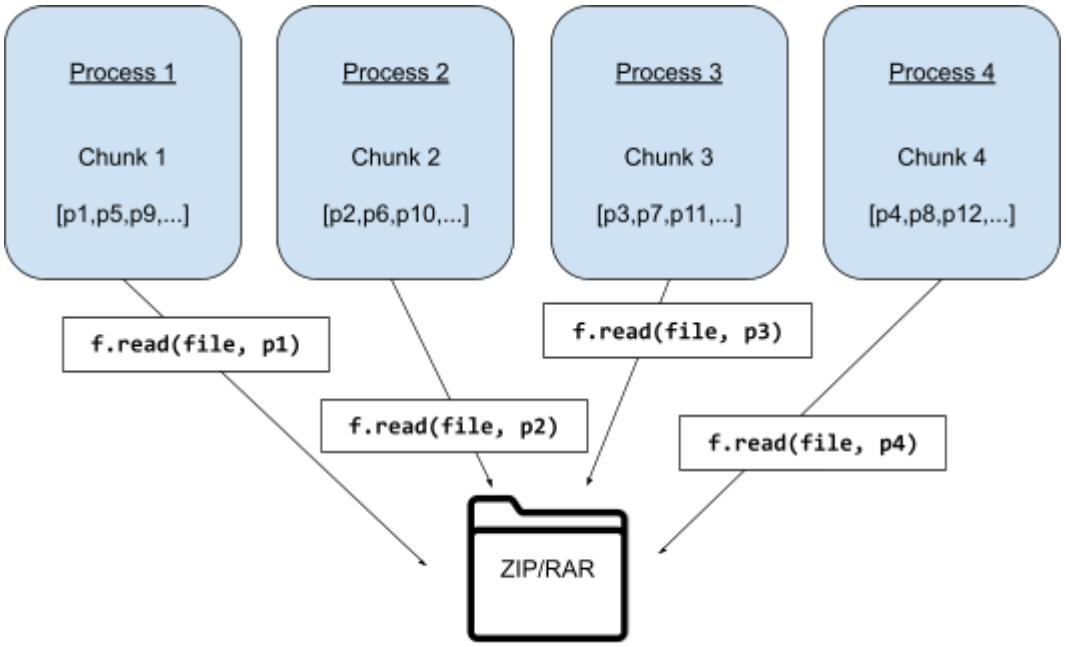


- **`try_pass(chunk, path)`**
- Input: list[str] (chunk of passwords), str (path of the archive)
- Output: str (correct password, empty if wasn't found)

```
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 ""
```

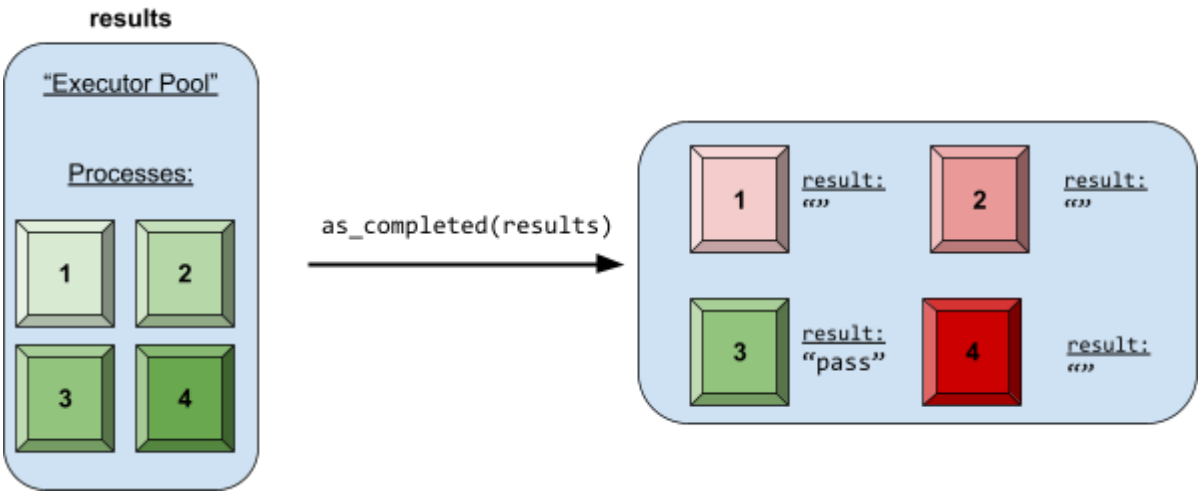
- `try_pass()` is a function that runs by each process. It takes a list of passwords (named chunk) and the path of the archive. First it takes the name of a file from the archive (using the earlier function) and for each type of archive (zip or rar) will try a password by reading the file with the read() 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,...]
f.read(file, p1)
f.read(file, p2)
f.read(file, p3)
f.read(file, p4)
ZIP/RAR

- **`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 ""
```

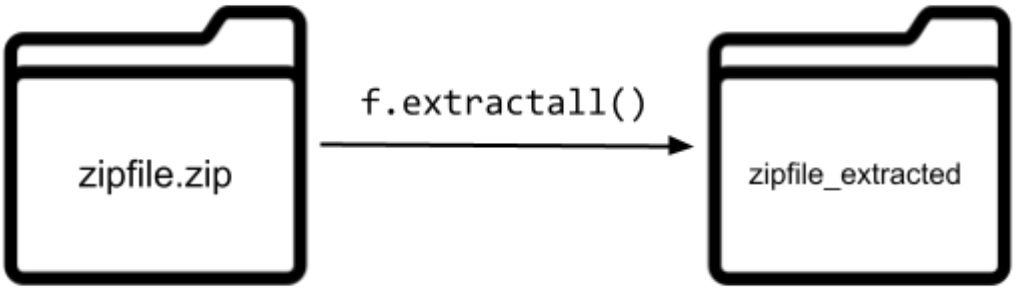
`brute_cracking()` is the function that manages all the processes. With `ProccessPoolExecuter()` it creates a list containing all the processes that are created. Every element of the list is a Future object that manages the results/status of its own `try_pass()` function of the process submitted to the 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.



- **`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.



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.

Code

```
import concurrent.futures
import time
import rarfile
import requests
import os
import psutil
import pyzipper

#function that returns the optimal maximum of processes the script can run
#based on user's hardware and current load on the CPU
def get_optimal_cpu() -> int:
   total_cpus : int = os.cpu_count() #total number of available CPU's in user's hardware
   current_cpu_load : float = psutil.cpu_percent(interval=1) #current % of load on the CPU's
   optimal_maximum_processes : int = 1 #saving here the maximum number of processes the PC
can run concurrently
   if current_cpu_load > 80:
       #if the current load is more than 80%, use half
       optimal_maximum_processes = max(1, total_cpus // 2)
   elif current_cpu_load > 50:
       #if the current load is more than 50%, use 3/4 of the cores
       optimal_maximum_processes = max(1, (total_cpus * 3) // 4)
   else:
       #if the current load is less than 50%, use all cores
       optimal_maximum_processes = total_cpus
   return optimal_maximum_processes

#function that splits the password list based on the number of processes the PC can manage
#returns a list with lists as elements, element for each process
def split_passwords_list(max_processes) -> list[list[str]]:
   response =
requests.get("https://raw.githubusercontent.com/danielmiessler/SecLists/refs/heads/master/Pa
sswords/Common-Credentials/10-million-password-list-top-100000.txt")
   passwords_list : list[str] = response.text.splitlines() #taking all the passwords from
the raw page, shoving them into a list
   divided_passwords_list : list[list[str]] = [] #declaring the divided list to return
   for _ in range(max_processes):
       #appending sub-lists based on the number of processes
       divided_passwords_list.append([])

   chunk : int = 0 #iteratable for appending the passwords
       #iterate on the password list, and for every element, add it to the next "chunk"
(chunk = sub-list = process)
```

```
    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


#function that returns the name of the zip file in a string format
#returns a string of the archive name
def get_name_for_extract(path) -> str:
    #methods for .zip files, using pyzipper module
    if path.endswith(".zip"):
        try:
            with pyzipper.AESZipFile(path, 'r') as f:
                return f.filename #returning the archive name
        except:
            print("error getting file name")
            return ""
    #methods for .rar files, using rarfile module
    else:
        try:
            with rarfile.RarFile(path, "r") as f:
                return f.filename #returning the archive name
        except:
            print("error getting file name")
            return ""

#function that gets the directory the archive exists in
#returns a string of the directory
def get_archive_directory(path):
    last_slash : int = path.rfind("/") #the index of the last slash, before the file name
    if last_slash == -1:
        last_slash = path.rfind("\\") #the index of the last slash, before the file name
    Path_name : str = path[0:last_slash] #slicing the string from the start until the last
folder in the path
    return path_name

#function that gets the name of the first file in the archive to test the password on.
#returns a string of the file name/directory of the file inside the archive
def get_filename_for_testing(path) -> str:
    #methods for .zip files, using pyzipper module
    if path.endswith(".zip"):
```

```
        with pyzipper.AESZipFile(path, "r") as f:
            for i in f.infolist(): #look at every element of the archive
                if not i.is_dir(): #if it's a file
                    return i.filename
    #methods for .rar files, using rarfile module
    else:
        with rarfile.RarFile(path, "r") as f:
            for i in f.infolist(): #look at every element of the archive
                if i.is_file(): #if it's a file
                    return i.filename
    return ""

#function that tries every password in the chunk it receives on the archive
#returns a correct password if not, else an empty string
def try_pass(chunk, path) -> str:
    name = get_filename_for_testing(path) #getting the file for testing
    #if the archive is not empty:
    if name:
        # methods for .zip files, using pyzipper module
        if path.endswith(".zip"):
            for password in chunk:
                with pyzipper.AESZipFile(path, "r") as f:
                    try:
                        f.read(name, pwd=bytes(password.encode())) #for the password given,
try reading the file
                        return password
                    except:
                        #if the password is incorrect, continue to the next password
                        continue
        else:
            # methods for .rar files, using rarfile module
            for password in chunk:
                with rarfile.RarFile(path, "r") as f:
                    try:
                        f.read(name, pwd=password) #for the password given, try reading the
file
                        return password
                    except:
                        # if the password is incorrect, continue to the next password
                        continue

    return ""

#functiong that creates different processes, to crack the code faster
```

```
#returns the correct password, else returns an empty string
def brute_cracking(path) -> str:
   optimal_max_processes : int = get_optimal_cpu() #getting the number of optimal maximum
number of processes
   passwords_list_chunks : list[list[str]] = split_passwords_list(optimal_max_processes)
#getting the divided password list relative to the number of processes
   with concurrent.futures.ProcessPoolExecutor() as executor:
       results = [executor.submit(try_pass, passwords_list_chunks[i], path) for i in
range(optimal_max_processes)]
       # ^    list of all processes, started and joined together by the ProcessPoolExecutor
       for f in concurrent.futures.as_completed(results):
           #look in every process and check the result
           if f.result() != "":
               #if a password was found (not an empty string)
               return f.result()
   return ""

#function that extracts all the data from the archive
def extract_zip(path, password, path_to_extract) -> None:
   # methods for .zip files, using pyzipper module
   if path.endswith(".zip"):
       with pyzipper.AESZipFile(path, "r") as f:
           f.extractall(path_to_extract, pwd=bytes(password.encode()))
   # methods for .rar files, using rarfile module
   else:
       with rarfile.RarFile(path, "r") as f:
           f.extractall(path_to_extract, pwd=password.encode)


if __name__ == '__main__':
   while True:
       path_to_zip : str = input("Enter the path of the zip or rar file: ") #asking the user
for the path of the compressed file
       # looping until user gives a correct format and correct directory
       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")

   start : float = time.perf_counter() #starting measuring the time until the
brute_cracking() finishing
```

```
    print("Please wait, this can take some time...")
    correct_password : str = brute_cracking(path_to_zip) #actual brute force cracking
    finish : float = time.perf_counter() #finish measuring the time
    print(f"Finished in: {finish - start} seconds") #printing total time for the cracking
    if not correct_password:
        print("couldn't crack password or compressed file is empty without files")
    else:
        #option to extract archive starts here
        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 choice.lower() == 'y': #user chose to extract data
            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")  # path to extracted folder
            i = 1
            while True:
                try:
                    os.makedirs(extracted_content_path) #trying to create the folder
                    break
                except:
                    #a folder with the same name already existing, so making a new folder
name
                    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) #extracting
the data to the new folder
            print("done!!")
```