Guide_encadrantes.md

Questions/Remarques un peu générales

Le README

J'ai écrit le README essentiellement pour les encadrantes, s'y référer pour tout ce qui est relatif au package, aux programmes en ligne de commande que je propose d'utiliser etc. C'est là qu'il y a toutes les infos

La place des femmes dans le hacking

J'ai fait une sorte d'introduction pour faire le lien entre code et féminisme (c'est l'un des PDF décrit dans le README). Je n'ai pas réellement eu le temps de la finir et j'en suis pas vraiment satisfait (trop long !!)

Au pire, celles qui veulent le lire chez elles auront tout le loisir de le faire. **Il ne faut pas qu'elles passent trop de temps dessus pendant la séance (qui est déjà courte) !**

> *Avec France nous avons pensé que ça pourrait être mieux si c'était dans un format "présenter les images au projecteur et discuter entre filles de ce qu'elles évoquent". C'est comme vous le sentez*

Prérequis: Quid du notebook ?

J'imagine que le jupyter notebook sera installé avec les ordinateurs. Si c'est le cas, je pense que ça peut peut-être intéressant de leur présenter si c'est possible. De mon expérience ça fait gagner beaucoup de temps aux débutant.es

A propos de l'ouverture du fichier RAR en python

Le package comprend la fonction suivante qui est ***assz sale***:

```
def essayer_mdp(chemin_vers_fichier_RAR, mdp, nom_fichier=None):
    """ Permet de tester le mot de passe d'un fichier RAR
    Renvoit True si le mot de passe est bon, et renvoit False si on n'a pas
    pas réussi à ouvrir le fichier RARself.

    Si le nom du fichier (nom_fichier) n'est pas donné en détail, alors par
    défaut la fonction essaye d'ouvrir le premier fichier de la listeself.

    ATTENTION: Si la fonction renvoit False, cela peut également être dû
    à une mauvaise variable nom_fichier"""

    fichierRar = RarFile(os.path.realpath(chemin_vers_fichier_RAR))
    mdp=str(mdp)

    txt_list = {}
    to_return = True
    if not nom_fichier:
        fichier = sorted(fichierRar.namelist())[0]
    else:
        fichier = nom_fichier

    try:
        txt = fichierRar.open(fichier,psw=mdp).read().decode().strip()
        #txt_list[fichier] = txt
    except BadRarFile:
        to_return = False

    return to_return
```

Je vous ferai pas l'affront de la commenter davantage. Elle est très (trop ?) minimaliste mais au moins ça permet de rentrer rapidement dans le vif du sujet sans s'embêter avec les histoires d'encodage, de cryptage etc

Néanmoins si vous arrivez à faire plus rapide (Il me semble que c'est difficile) n'hésitez pas à apporter une contribution sur le github car c'est vraiment une étape assez longue.

Etapes/Exercices de Niveau 1

Installer crackrar

Voir README.md pour l'installation

On peut en profiter pour présenter pip très sobrement: Un programme python qui permet d'installer facilement d'autres programmes python.

Utiliser crackrar pour cracker un mot de passe

Cf le README

> N'hésitez pas à passer un peu de temps sur la ligne de commande parce que c'est une façon intuitive (avec les chiffres qui défilent) de voir ce qu'elles doivent essayer de réaliser.

Utilisez plutôt **brutegen** au début car il **affiche par défaut** toutes les combinaisons et c'est beaucoup plus **intuitif** (et beaucoup plus **rapide**, parce qu'on ne fait qu'afficher les combinaisons, on ne les essaie pas (**voir README.md: "Notes sur la performance"**)

> **On peut utiliser crackrar sur le fichier qui se trouve dans "data" et qui s'appelle "notes.rar". Le code est : 1789 (on peut donc le cracker rapidement avec des chiffres puisque c'est un code PIN)**

Tester un mot de passe en python sur un .rar

IL va s'agir ensuite de leur apprendre à utiliser le package non plus en ligne de commande, mais directement en python pour avoir plus de liberté d'expression.

Cf la fonction ci-dessus qu'on peut importer du package

```
from crackrar import essayer_mdp

if essayer_mdp(chemin_vers_fichier_RAR="~/crackrar/crackrar/data/notes.rar",
               mdp="1789", nom_fichier=None):

    print('Ouverture du fichier réussie !')
```

> RQ: Avec cette fonction on ne retourne pas le contenu, on vérifie juste si le mot de passe était le bon

Note : Ce fichier existe réellement dans le sous-dossier "data" et le mot de passe est vraiment 1789

Tester à partir d'une entrée utilisateur

Pour celles qui sont les plus débutantes, j'imagine qu'avant de faire des boucles on peut simplement utiliser la fonction **input()** avec le code précédent. Ca peut être un très bon exercice pour débutanter (Demander à l'utilisateur un chemin vers un fichier .rar, puis lui demander en boucle le mot de passe tant qu'il ne l'a pas trouvé)

Tester tous les nombres possibles

Je pensais juste à faire une façon très simple:

```
from crackrar import essayer_mdp

i = 0

while not essayer_mdp(chemin_vers_fichier_RAR="~/crackrar/crackrar/data/notes.rar",
               mdp=str(i), nom_fichier=None):
    i += 1
```

```
print(i)
```

Il y a aussi l'alternative bornée à un nombre maximum (exemple: 10000)

```
for i in range(10000):
    if essayer_mdp(chemin_vers_fichier_RAR="~/crackrar/crackrar/data/notes.rar",
                   mdp=str(i), nom_fichier=None):

        print(i)
        break # Ça peut être l'occasion de faire le point sur "break"
```

Tester tous les mots dans un fichier

C'est l'occasion de présenter les fonctions **os.path.realpath()**, **os.getcwd()**, **open()**, **strip()**, **readline()**, le retour à la ligne ***"\n"***, voir même éventuellement le context manager **with [...] as xxx :** pour éviter la fuite de mémoire quand on laisse le fichier ouvert (ou à défaut, présenter **.close()**)

```
with open(os.path.realpath(os.getcwd()+"/"+"filedictionnary.txt"),"r") as filin:
    for line in filin:
        word = line.strip()
        if essayer_mdp(chemin_vers_fichier_RAR="~/crackrar/crackrar/data/notes.rar",
                       mdp=str(word), nom_fichier=None):

            print(word)
            break
```

De vraies attaques par dictionnaire

En ré-utilisant les codes suivant on peut déjà faire des vraies attaques par dictionnaire de mots de passe. En effet dans ***~/crackrar/crackrar/data/*** on trouvera plusieurs dictionnaires au format texte (une ligne par mot):

- 10 millions de mots de passe aléatoires
- 1000 mots parmi les plus fréquents en français
- Une liste de 10.000 mots de passe parmi les plus utilisés au monde
- Un dictionnaire de 330.000 mots français
- Les 75 mots de passe les plus fréquets de tous les temps (pas nécessairement dans l'ordre)
- Une liste de prénoms français (en majuscules)
- Un .csv qui reprend ces prénoms mais permet aussi de les classer par abondance

Je les ai nettoyés/bien reformatés et ils sont prêts à l'emploi. On trouve aussi: Les mono, bi, tri et quadgrams de la langue française avec leur abondance. Cela permet de générer des mots qui pourront "sonner français" sans pour autant être des mots de français

Cela peut servir de base pour la suite du travail

Etapes / Exercices de niveau 2

Avant de commencer les exercices de niveau 2

Les exercices de "niveau 2" sont **tous corrigés** dans le fichier **crackrar/attack_dicts.py**

Je pense que le but c'est qu'elles fassent des codes sans fonction/organisation particulière (exemple: dans des scripts séparés, des notebooks...) jusqu'à ce qu'elles **se rendent compte par elle-mêmes que ça devient le bordel.**

Puis quand ça devient trop le dawa (trop de choses différentes à gérer) on peut les aider à mettre leur code sous forme de **fonctions génératrices**

par exemple

```
def generer_chiffres_jusque(n):
    for i in range(10000):
        if essayer_mdp(chemin_vers_fichier_RAR="~/crackrar/crackrar/data/notes.rar",
                       mdp=str(i), nom_fichier=None):

            yield(i)
            break # Ça peut être l'occasion de faire le point sur "break"
```

Tous les exercices de "niveau 2" ici sont en fait "créer des générateurs"; à chaque fois un peu plus compliqués que les précédents.

Comprendre, utiliser et créer des itérateurs

J'imagine qu'il n'y aura pas le temps pour voir autre chose que les générateurs de la forme def func() / yield

```
def my_generator():
    for elt in "abdcefghijklmnopqrstuvwxyz":
        yield elt
```

Néanmoins pour celles qui apprendraient plus vite, ça peut être intéressant de voir la forme générable d'un itérable

```
>>> test = iter([1,2,3])
>>> next(test)
1
>>> next(test)
2
>>> next(test)
3
>>> next(test)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

S'il est un peu tôt pour s'attarder sur la façon dont python gère les itérables dans une boucle For (en récupérant un **StopIteration** dans un bloc **try/except**), on peut même parler du fait que tous les objets qui sont itérables implémentent la fonction ***_iter_*()** et ***_next_*()** ? Il n'y a pas besoin de comprendre en détail ce qu'est un objet pour l'utiliser: A ce stade on utilise déjà des fonctions (qui sont des objets), des "range" (qui sont des générateurs) ou des listes (qui sont itérables) sans en avoir conscience : Pour certain.es ça peut être intuitif.

> A la fin du README.md quand je parle de "combiner les stratégies" en les rendant itérables c'est à ça que je pensais [https://www.programmiz.com/python-programming/iterator]. Je dirais que c'est plus un challenge qu'autre chose.

Quelques commandes sont également intéressantes à taper dans l'interpréteur:

```
>>> list(range(4))
[0, 1, 2, 3]
```

ou présenter l'opérateur d'unpacking

```
>>> numbers = [1,2,3,4]
>>> more_numbers = [*numbers,5,6]
>>> more_numbers
[1, 2, 3, 4, 5, 6]
```

On peut en profiter si on a le temps pour parler des listes en intentions, qui à ce stade sont des curiosités qui paraissent superflues, mais qui prendront sûrement sens au cours des séances.

```
>>>[x for x in range(4) if x%2]
[0,3]
```

Comme littéralement tout le projet consiste à fabriquer des bons itérateurs, je pense qu'on peut aller assez dans les détails.

strings

Le module est pas trop dur à utiliser, il permet d'itérer sur les lettres, les caractères spéciaux etc.

itertools

Prise de tête garantie entre combinations, permutations, product, combinations_with_replacement

Je conseille fortement la doc et quelques essais dans un interpréteur parce que ça peut être déroutant

[https://docs.python.org/3.5/library/itertools.html]

Tester tous les codes PIN à 4 chiffres

> **ATTENTION** : Ici l'exercice est différent puisqu'on ne veut pas tester 0, 1, 2... Mais bien 0000, 0001, 0002, etc

Il y a une infinité de façons de faire. L'une de ces façons (pas la plus simple !) est la suivante:

```
def PIN_4():
    default_PIN = "0000"
    for i in range(10000):
        yield str(i)
        if i < 1000: # To return 0000 0001 instead of 0, 1...
            yield default_PIN[:-len(str(i))]+str(i)
```

La plus simple est sûrement le **string formatting**

Chronométrer son code

On peut utiliser le module **datetime**,

```
import datetime

now = datetime.now()
dt_string = now.strftime("%d/%l/%Y %H:%M:%S")
print(df_string)

for in range(10000000):
    pass

now = datetime.now()
dt_string = now.strftime("%d/%l/%Y %H:%M:%S")
print(df_string)
```

... Mais c'est un peu fastidieux !! Je pense que **tqdm** est **nettement plus pratique**

tqdm est installé par pîp lors de l'installation de crackrar. On peut donc profiter de son interface très sympa:

```
from tqdm import tqdm

for elt in tqdm(range(10000000)):
    pass
```

ce qui donne une barre de progression avec une vitesse moyenne:

```
>>> for elt in tqdm(range(10000000)):
...     pass
...
100%|████████████████████████| 10000000/10000000 [00:01<00:00, 6073549.49it/s]
>>>
```

> Joli non ?

Il y a aussi l'interface en ligne de commande qui est montrée dans le README.md, et qui a l'avantage de marcher également sur les générateurs donc on ne connaît pas la taille

```
user@computer:~/crackrar/crackrar/test_package/testdata$ brutegen -l 4  | python -m tqdm | wc -l
112550881it [01:42, 1095235.63it/s]
112550881
```

> Pratique !

Tester toutes les dates de naissance possibles

Là encore, tout est imaginable. L'une des solutions les plus propres est la suivante:

```
from datetime import date, timedelta

def birth_dates():
    def daterange(start_date, end_date):
        for n in range(int ((end_date - start_date).days)):
            yield start_date + timedelta(n)

    start_date = date(1, 1, 1900)
    end_date = date.today()
    for single_date in daterange(start_date, end_date):
        year = int("".join(single_date.strftime("%Y")[2:]))
        month = int(single_date.strftime("%m"))
        day = int(single_date.strftime("%d"))
        yield(str(year)) # ex: 91
        yield(str(day)+str(month)) # ex: 0702
        yield(str(day)+str(month)+str(year)) # ex: 070291
```

Tester tous les digicodes possibles

Pour moins de migraines, il faut utiliser itertools !

```
import itertools

def digicode():
    list1 = [str(x) for x in [0,1,2,3,4,5,6,7,8,9,"#","*","A","B"]]
    permut_digicode = ["".join(x) for x in list(itertools.product(list1,repeat=4))]
    for elt in permut_digicode:
        yield elt


>>> import crackrar as cr
>>> test = [x for x in cr.digicode()]
>>> test[100:110]
['0072', '0073', '0074', '0075', '0076', '0077', '0078', '0079', '007#', '007*']
>>> test[1000:1030]
['0516', '0517', '0518', '0519', '051#', '051*', '051A', '051B', '0520', '0521', '0522', '0523', '0524', '0525', '0526
```

> Remarque: Ca fait beaucoup de digicodes !

```
>>> len(test)
38416
```

> Ca fait bien 14*14*14*14 = 38416 possibilités :)

Et il n'y a pas de doublon

```
>>> import numpy as np
>>> len(np.unique(test))
38416
```

Générer des lettres de l'alphabet

Soit en utilisant strings, soit en utilisant des solutions plus simples. La chaîne de caractères **"abcdefghijklmnopqrstuvwxyz"** est un itérable !

Muter les majuscules en minuscules

Voir le principe de l'objet mutation dans "README.md".

Dans la façon dont le package est codé, la mutation n'est pas un objet en tant que tel. C'est simplement **n'importe quelle fonction génératrice**. Une mutation prend en entrée une chaîne de caractères, puis elle génère toutes les variations envisagées. Dans crackrar elle n'est définie qu'**indirectement *via*** l'attackDict

```
class attackDict(object):
    def __init__(self,generator,*args_generator,**kwargs_generator):
        self.mutations_list = []
        self.generator = generator
        self.generator_kwargs = kwargs_generator
        self.generator_args = args_generator

    def add_mutation(self,function): # Drops the original
        self.mutations_list.append(function)

    def generate(self):
        for elt in self.generator(*self.generator_args,**self.generator_kwargs):
            if self.mutations_list:
                for mutation in self.mutations_list:
                    mutated_elt = mutation(elt)
                    yield mutated_elt
            else:
                yield elt
```

... Donc **n'importe quelle génératrice traîtant une chaîne de caractères est valide** !

Voila quelques idées pour faires des mutations de majuscules/minuscules utiles pour notre projet

```
def up_down_up_down(seq,start = "up"):
    seq = seq.lower()
    newseq = []
    current = start
    for elt in seq:
        if current == "up":
            newseq.append(elt.upper())
            current = "down"
        elif current == "down":
            newseq.append(elt.lower())
            current="up"
    return "".join(newseq)

def startby_maj(seq):
    seq = seq.lower()
    seq = seq[0].upper()+"".join(seq[1:])
    return seq

def maj_modif(elt):
    yield elt
    yield up_down_up_down(elt,start="up")
    yield up_down_up_down(elt,start="down")
    yield elt.lower()
    yield elt.upper()
    yield startby_maj(elt)
```

Niveau 3

On va à partir des exercices du niveau 3 utiliser les objets attackDict et Strategy du package crackrar. Ces objets sont décrits dans la dernière section du README.md, donc la totalité des explications s'y reportent.

A la fin de la partie 3, nous sommes en mesure de faire un brute force qui commence à devenir modulaire.

Niveau 4

Aucune correction disponible pour les challenges -- Trop chronophage dans l'immédiat

J'en rajouterai peut-être plus tard