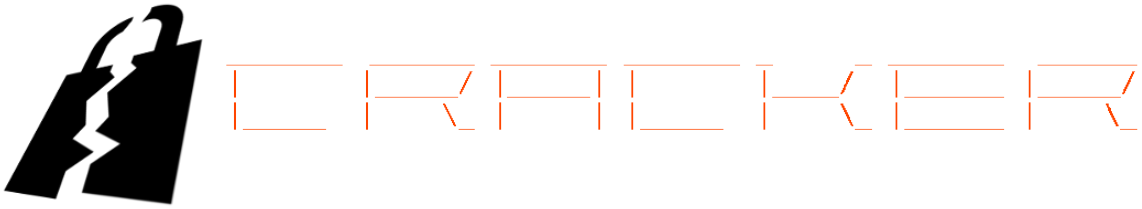
CONVENTION FOR THE DEVELOPERS OF CRACKER
(CDC)



Elerias, Lasercata

June 29, 2020

Contents

Chapter 1

Introduction

To do.

Chapter 2

Language Conventions

2.1 Code

The language of the code is English. All the name of variables, functions, classes or other objects, the documentation and the description of functions and modules should be written in English.

2.2 Annotations

Annotations can be written in English or in French because all the developers speak french, but the English is prefered for a more consistent code.

2.3 Communication with user

All the outputs in the program should be written in English. The program should take English and French inputs. After a closed question, Cracker should take the following answers :

- y ;
- Y ;
- yes ;
- Yes ;
- YES ;
- n ;
- N ;
- no ;
- No ;
- NO ;
- non ;
- Non ;
- NON.

Examples :
Text from file ? (y/n) yes
Text from file ? (y/n) Non

2.4 Documents

The documents should be written in English. French is however acceptable because all the developers speak french.

Chapter 3

Style Conventions

3.1 Names

Names of objets have to be short.
In Cracker, some variables have the same functions in different modules. They should have the same name :

- "`t`", "`txt`" or "`text`" represents the text that the program use. It is often asked to the user ;
- "`ret`" represents the text returned to the user ;
- "`t`" or "`t_`" + number represents a time ;
- "`alf`", "`alph`" or "`alphabet`" represents an alphabet ;
- "`f`", "`fn`", "`filename`" or "`f_name`" represents the name of a file ;
- "`f_ext`" represents the extension of the file ;
- "`w`", "`wrd`" or "`word`" represents a word ;
- "`wrdlst`" or "`wrdlt`" represent a wordlist ;
- "`D`", "`d`", "`dico`" or "`dct`" represents a dictionnary ;
- "`l`", "`lth`" or "`lenth`" or "`length`" represents a length ;
- "`n`" represents an integer ;
- "`p`" represents a prime number ;
- "`d`" or "`div`" represents a divisor ;
- "`L`" or "`lst`" represents a list ;
- "`r`" or "`rest`" represents the rest of a euclidean division ;
- "`M`" or "`msg`" represents a plain message ;
- "`C`" or "`msg_c`" represents a encrypted message ;
- "`key`" or "`k`" represents a key ;
- "`p`" or "`primality`" represents the primality ;
- "`x`" represents a preimage in a function ;
- "`y`" represents an image in a function ;
- "`f`" or "`func`" represents a function ;
- "`window`", "`wind`", "`win`", or "`w`" represent a window (Tkinter or PyQt5).

3.2 Whitespaces

Before and after the equal symbol (=), the test symbols (’==’ ’!=’ ’>’ ’>=’ ’<’ ’<=’) and ’;’, one whitespace should be placed.

Good examples :

```
a = 8 ; b = 7
c = 2

if a == 8:
   print(a)
```

Bad examples :

```
a    =    8; b= 7
c=2
if a==8 :
  print(a)
```

Before and after an operator (+ - * / // % **), one whitespace should but if there is a proritary operation.

Good examples :

```
a = 2 + 4 + 18
b = a*2 + 4 + 18/3
c = a * (b + 3**2)
c = a % (b + 3**2)
```

Bad examples :

```
a = 2+4+18
b = a * 2 + 4 + 18 / 3
c = a*(b + 3 ** 2)
c = a % (b+3 ** 2)
```

After ’:’ and ’,’, one whitespace should be placed. But not before.

Good examples :

```
if a == 2:
   d = {’a’:  14, 23:  ’test’}
   L = [1, ’hello’, 42]
```

Bad examples :

```
if a == 2 :
   d = {’a’ :  14, 23:’test’}
   L = [1 , ’hello’,42]
```

For other symbols (( ) { } [ ] ’ " .) , no whitespace should be placed.

Good examples :

```
L = [3, 90, ’bonjour’, (4, 2)]
a = 2 * (4+2)
b = f(2)
```

Bad examples :

```
L = [ 3, 90, ’bonjour’, ( 4, 2 ) ]
a = 2 * ( 4+2 )
b = f ( 2 )
```

3.3 Indentations

Four spaces, automatic indentations and tabulations can be used to indent.

3.4 Blank lines

Blanks lines improve *readability*, so they should be inserted between the differents *code portions*, i.e. between the import portion, the functions, the classes, the run portion. Blanks lines can also be inserted in the *functions* and *classes*, to separate the differents parts.

The number of blanks line to insert depends on the code's length. *Three* blanks lines should be inserted at maximum.

One blank line should be inserted after the *documentation* in the classes and functions.

3.5 Single and double quote

For lists, tuples and dictionnaries, single quotes are prefered to lighten the code. If there should be a quote in your string, it is prefered to use the others. Triple quotes are strongly discouraged.

Good examples :
```
d = {'b':  8, '23':  'hello', 'logo' :  "Cracker's logo"}
L = [7, 'afternoon', ('monday', 'wednesday'), '"quoted"']
```

Bad examples :
```
d = {"b":  8, "23":  "hello"}
L = [7, "test", "\"quoted\"", 'Cracker\'s logo']
```

Very bad example :
```
L = [7, '''afternoon''', ("""monday""", """wednesday""")]
```

To frame a word, single quotes are also recommended.

Good examples :
```
a = 'hello'
b = "Cracker's logo"
c = '"This" is quoted'
```

Bad examples :
```
a = """hello"""
b = 'Cracker\'s logo'
c = "\"This\" is quoted"
```

To define an empty character string, 2 double quotes are prefered.

Good example :
```
a = ""
```
Rather not :
```
a = ''
```
Absolutely not :
```
a = """"""
```

3.6 Docstrings

Both single and double quoted are accepted in the docstrings, but only one type of quote should be used within a same file, to have a consistent code.

If the description line measures one line, the 2 docstrings should be on the same line, else they have to be on the same column.

Good examples :

```
def f(x):
    """Return the square of x."""

    return x ** 2

def g(x):
    ‘‘‘Return the inverse of x.
    x :  a number different of 0.
    ’’’

    return 1 / x

def h(x, n):
    """
    Return the nth root of x.

    x :  a float number ;
    n :  an other float number.
    """

    return x **(1 / n)
```

Bad examples :

```
def f(x):
    """Return the square of x.
    """

    return x ** 2

def g(x):
    """Return the inverse of x.
    x :  a number different of 0"""

    return 1 / x

def h(x, n):
    """
    Return the nth root of x.

    x :  a float number ;
    n :  an other float number."""

    return x **(1 / n)
```

3.7 Annotations

Todo.

3.8 Imports

Todo.

3.9 Modules

Todo.

Chapter 4

Using of Craker functions

Todo

Chapter 5

Documents about Cracker

5.1 Updates

An update note should be fill every time the program is modified. It should be saved in the folder `Cracker_v[version]/updates/updates_notes`, and it should be named "`update_YYYY-MM-DD_[new version].txt`" (with the brackets, this time).

The update note should begin by the date at the format "`YYYY.MM.DD`" (or can be "`YYYY.MM.[day begin]-[day finish]`" (example : `2020.05.01-09`) if the update took more than one day, but it is not needed), and should be followed, the line after, by "`Cracker v[new version] <-- v[old version]`". There should be a part for every module improved, begining by "`[module name] v[new version] <-- v[old version] :`", and ending by "-"×3. The first part should be consecreted to the general improvements, begin by "`General improvements (from [old Cracker version]) :`", and end by "-"×6.
The update note should end by a separation followed the line after by "`By [author]`"

The update notes should be written in raw text, with the "`.txt`" extension.

Example :
file name : "`update_2020-05-02_[2.4.0].txt`"

```
2020.05.02
Cracker v2.4.0 <-- v2.3.2
---------------------------------
General improvements (from 2.3.2) :
   - Small bug corrections ;
   - Improving Crypta and Prima.
------

prima v3.0 <-- v2.4 :
   - New algorithms ;
   - Pollard's rho algorithm is the fastest.
---

crypta v2.9 <-- v2.8 :
   - Joining AES in the menu.
---

---------------------------------
By Elerias
```

5.2 History

The file "`history.txt`" contain all the majors improvements of *Cracker*. It is situated at `Cracker_v[version]/updates`. If this software is improved enough, i.e. if the version pass from `x.y.z` to `x+1.0.0` or `x.y+1.0` (First or second digit incremented), the file should be updated. To update this file, the `General improvements` part of the update note should be added to the `history.txt` file.

5.3 Version

The file "`Version.txt`" contain the actual *Cracker* version. The version should contain **three** numbers, separeted by two dots, i.e. at the format `x.y.z`, where `x`, `y`, and `z` are positive integers.

Example : `2.4.0`

If the actual version is `x.y.9`, the next version should not be `x.y+1.0`, but `x.y.10`. The same rule applies to the others digits.

If only some small bugs were fixed, whitout major improvements, only the *last digit* (`z`) should be incremented. If some improvements were added, and maybe new modules added, the *second digit* (`y`) should be incremented, and the last should be set to 0. If the program was improved generaly, maybe with a total re-organisation, a lot of functionnal upturns, and a lot of modules improved, then the *first digit* (`x`) should be incremented.

Good exemples :
- `2.3.9` → `2.3.10` (small bux corrections)
- `2.4.3` → `2.5.0` (new module, launchers improved)
- `1.5.2` → `2.0.0` (rewriting some parts, put the project in modules)

Bad examples :
- `2.3.9` → `2.4.0` (small bux corrections)
- `2.4.3` → `2.5.3` (new module)
- `1.5.2` → `2.5.0` (bug fix)

5.4 Version_modules

The file "`Versions_modules.txt`" contain the actual *Cracker*'s modules versions. They should correspond to the variables set at the top of the modules (`[module name]__ver`). They should be composed of **two** numbers, separated by one dot. If the version need to contain more than two numbers, separate the lasts with a comma (`,`).

Exampe :

```
 Version of cracker_launcher :  1.0,1
Version of cracker_gui_launcher :  1.1
Version of cracker_parser :  2.0

Version of cracker_console_functions :  1.1
Version of base_functions :  2.2
Version of color :  2.2
Version of matrix :  1.1

Version of Hasher :  3.3
Version of hash_crack_2 :  1.0
Version of wordlist_generator :  6.4
Version of Crypta :  2.9
Version of Prima :  3.0
Version of Base convert :  2.1,2
Version of P@ssw0rd_Test0r :  1.0
```

When a module is created, its version should be `1.0`.

5.5 CDC

Todo