pycrypt Documentation

Release 0.2

Matěj Hlaváček

February 12, 2017

Contents

Pycrypt is a python suite for solving ciphers at (mostly Czech) cryptography games.

CHAPTER 1

Changelog

1.1 v0.2 - Apr 16, 2015

- `ThreadedGeneticSolver`, a new solver implementing the island model
- `crossovers` module with some crossover and selection strategies
- Weighted mutations based on letter frequencies
- Plotting the progress of genetic solvers
- Cached scoring for faster evolution
- Pycrypt can now be installed directly from github with pip
- Other fixes, tweaking and improvements

There's a longer post about v0.2 here

1.2 v0.1 - Mar 18, 2014

Initial release.

CHAPTER 2

Contents:

2.1 Introduction

2.1.1 What is it for?

Pycrypt was originally intended as a substitution cipher solver. When it did succeed, I wanted to bring to project a little bit further. It is meant to be used in cryptographic games, which often take place outside and at night. Therefore, pycrypt is (or at least is trying to be) fast to get started in, full of existing useful tools and easily extensible.

2.1.2 What does it do?

Pycrypt covers a range of most standard ciphers and usually even solves them. It comes with English and Czech dictionaries to measure cipher's proximity to being solved and right. It has a few solving algorithms and some analytical tools. In addition to some pycrypt's own graphical capability, external libraries fill in.

2.1.3 Who is it for?

Before you start using pycrypt, you should know basic to intermediate python programming. It does **not** come with a user-friendly graphical interface, as it is intended for python script and command line use only.

2.2 Getting started

2.2.1 Getting pycrypt's source

Pycrypt is on github here. You can clone it with:

```
$ git clone https://github.com/PrehistoricTeam/pycrypt.git
```

2.2.2 Installation

Pycrypt was developed on Python 2.7.5, but should work fine on previous versions as well.

If you don't have pip (you should), run this first:

```
$ curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py | python
```

You can now install pycrypt with pip:

```
$ pip install "git+https://github.com/PrehistoricTeam/pycrypt.git@master#egg=pycrypt"
```

If you want to hack on pycrypt's source, install it with:

```
$ pip install -e "git+https://github.com/PrehistoricTeam/pycrypt.git@master#egg=pycrypt"
```

It will download the source to the current directory and link it in the python installation.

Optional, but recommended packages are unidecode and ipython console for interactive use:

```
$ pip install unidecode
$ pip install ipython
```

2.2.3 Running tests

Pycrypt has some unit tests, you can run them in shell, while in the root directory of pycrypt, with:

```
$ python -m unittest discover
```

2.2.4 First script

To start solving a basic substitution cipher, try out this code:

```
import pycrypt as pc

cipher = "ERU NRIEU-GUFFIUH YUA UAMFU IY A FALMU HISLTAF GILH QX CLUB IT ERU XAWIFB APPICIELIHAU. A F
solver = pc.GeneticSolver(scorer=pc.EnglishScorer())

solver.solve(cipher)
```

Put it in a file (e.g. first_test.py) in the root directory and run it. You'll see some output and after some time, you *should* see the result close to:

```
The White-bellied Sea Eagle is a large diurnal bird of prey in the family Accipitridae. A distinctive
```

2.3 Structure

2.3.1 Class diagram

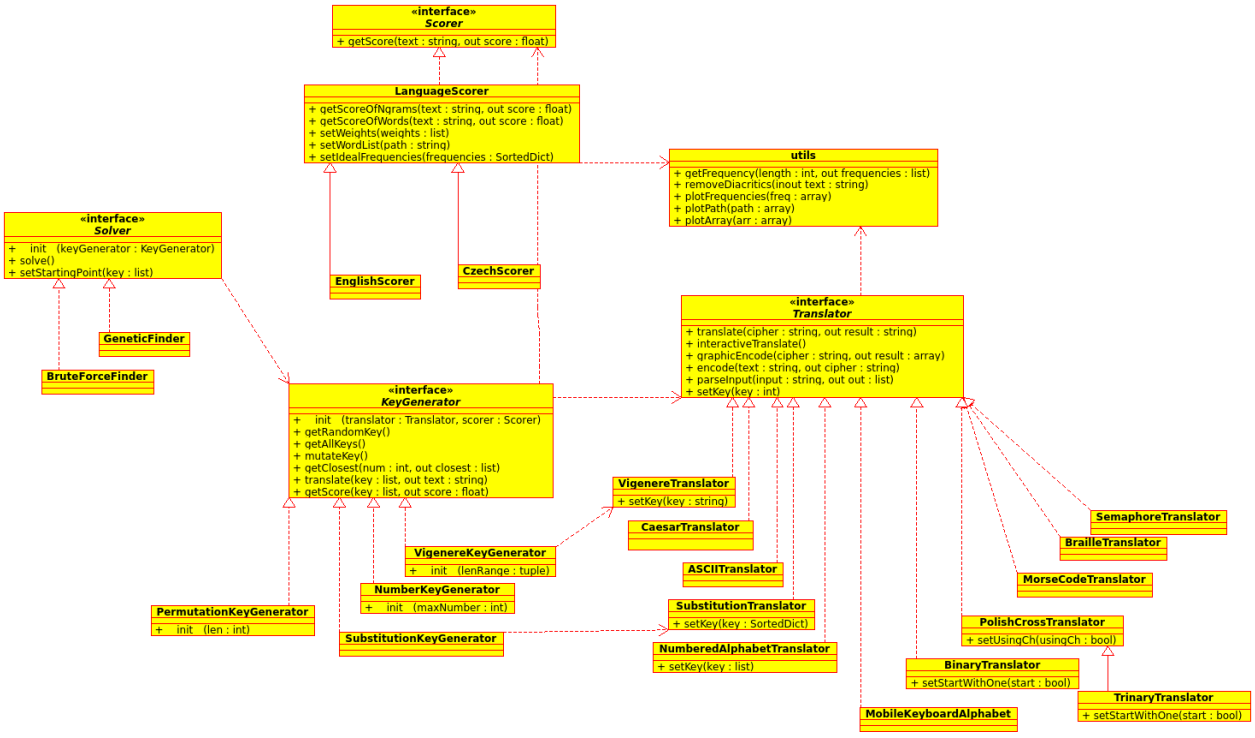
Pycrypt's overall structure consist of 4 main parts. But first, let's take a look at the UML diagram:

Warning: This diagram is just orientational. The project has evolved, while the diagram did, and probably will, not.

As you can see, there are 4 interfaces (which, because python doesn't support them, are just uninstantiated classes). These are the 4 basic building blocks for solving a cipher. They are:

- **Translators**

  which enable the basic encoding and decoding of a specific cipher (e.g. a Caesar cipher)



- **KeyGenerators**

  which generate keys for the Translators. They can implement generating all keys (for brute force solving) and mutating a key (for genetic algorithms)

- **Scorers**

  which calculate how *good* the solution is. Typically, you'll want to use them for scoring how close and similar they are to a specific language

- **Solvers**

  which glue everything together. They will get some keys with a KeyGenerator, apply these keys to the cipher with a Translator and finally score these solutions with a Scorer. They will also take care of printing out progress and optional interactions (during the solving process) from the user.

2.3.2 Why are you telling me all this??

Usually, pycrypt alone won't do too much during a typical cryptographic game. The cipher creators try hard to steer away from the standard ciphers. They'll try to make something creative and something that will require an idea. It would be impossible to cover all of these kinds of ciphers.

Pycrypt was developed with that in mind, and the user was meant to write a little bit of code during the actual solving. The structure of pycrypt is supposed to allow you to write code just for what is needed and take care of everything else (like printing and actual solving algorithms).

2.3.3 Next steps

You can either continue following this tutorial or jump ahead and dive into the API documentation:

See also:

pycrypt API documentation

2.4 Translators

Translators take care of translating to and from a specific cipher. There are some (over 10) already included in pycrypt.

2.4.1 Basic usage

Let's take a look at decoding a Caesar cipher with alphabet shift of 1:

```
import pycrypt as pc

t = pc.CaesarTranslator()
t.setKey(1)
print t.translate("GDKKN VNQKC!")
```

Which should output:

```
HELLO WORLD!
```

We have created a Translator, set its key (alphabet shift) to 1 and called the method `translate` to uncover the secret message.

Note: Since Translators are meant to be used on encrypted text, here the method `translate` actually shifted the alphabet by 1 back, not forward. You can also use the method `decode`, which is just an alias for `translate` and is maybe more semantically correct.

We can also revert the process with `encode`:

```
>>> t.encode("Hello World!")
'GDKKN VNQKC!'
```

And that's about it! But there are some more advanced uses too.

Some translators come with the `graphicEncode` method, which returns typically a 2d bool NumPy array that we can then draw with pycrypt's `plot_array` function:

```
t = pc.MorseCodeTranslator()
pc.plot_array(t.graphicEncode("SI\nRN\nSI\nNU\nWN\nSI"))
```

This will draw an image in a new window:

In this example, `MorseCodeTranslator`'s `graphicEncode` splits the input in lines and concatenates the Morse code characters, that represent the 1s and 0s (black and white squares). You can alter the functionality with some optional arguments.

You can also play around with the `interactiveTranslate` method, which just cyclically takes standard input, so you could see intermediate results.

And that's about all the functionality you can expect from Translators. Easy enough, isn't it?

2.4.2 Making your own Translator

Before we will extend the Translator interface, we should see its methods from the API:

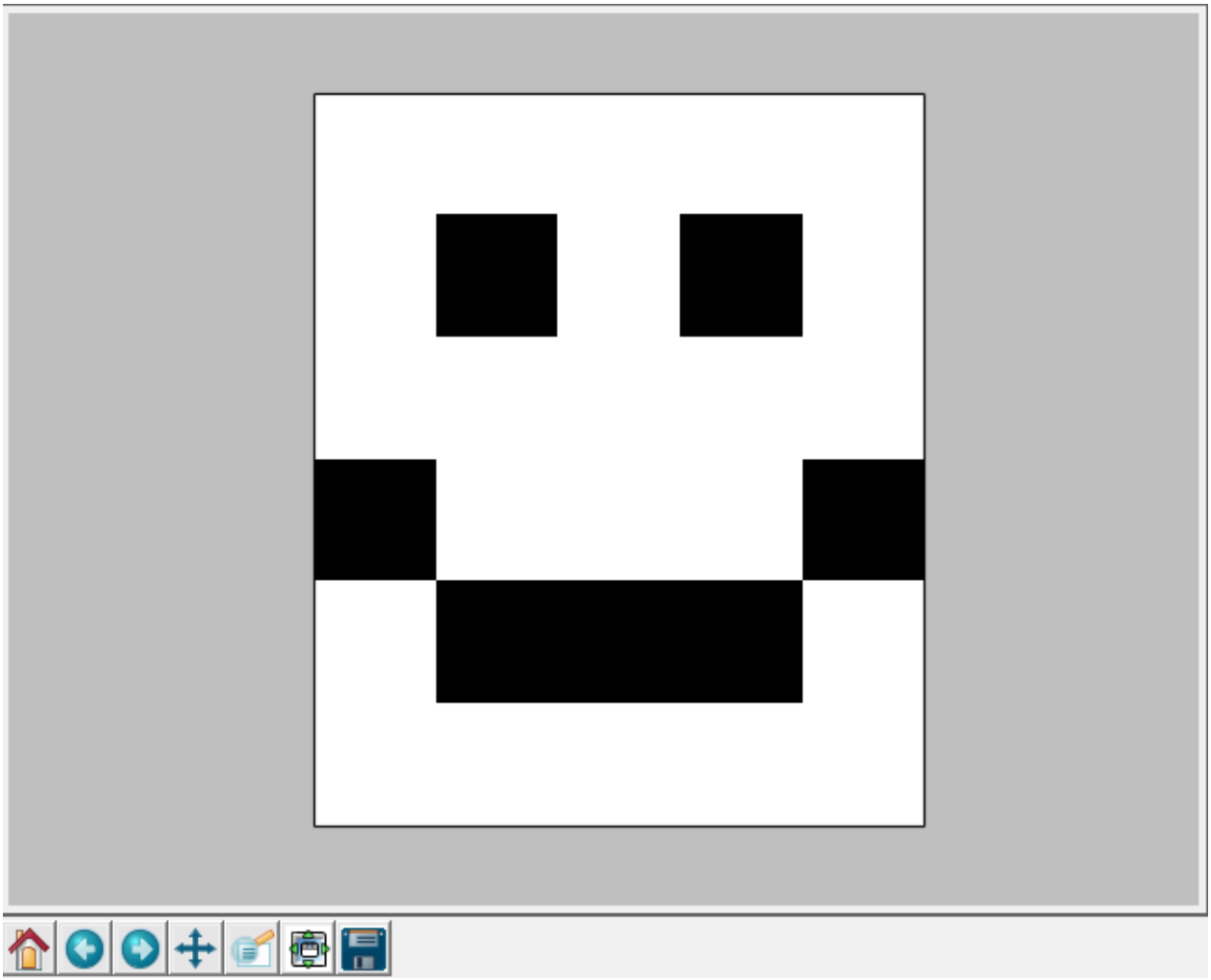


Fig. 2.1: SI RN SI NU WN SI

translator Module

class pycrypt.translators.translator.**Translator**
Abstract class for translating standard ciphers (i.e. Morse Code)

key = []

translate(**args*)
Base method for decoding a cipher

interactiveTranslate()
For quick translating with each character typed from the user, type ! to remove last characters

encode(**args*)
Reversed translation

decode(**args*)
Just and alias for translate

graphicEncode(**args*)
Return in numpy array for easy plotting

parseInput(*cipher*)
Standardize input to a list, values preferably integers indexed from 0

setKey(*key*)

All you have to do when inheriting from `Translator` is to implement the `translate` method. Optionally, you can implement `encode` and maybe even `graphicEncode`. `parseInput` is meant to be just an internal method and implementing it is optional, but pycrypt's standard is to always make the cipher uppercase. For examples, see to source of some implementations.

2.4.3 Further reading

To check out all Translators, see the API:

See also:

Translators

2.5 KeyGenerators

KeyGenerators handle generating keys (that was unexpected) for specific translators. They are intended to be used with Solvers, supplementing keys which the Solvers will then try out and process. They can implement generating all keys (for brute force solving), mutating a key (for genetic algorithms), or anything you would need for a specific Solver.

2.5.1 Basic usage

Most of pycrypt's KeyGenerators have a method `getAllKeys`, which usually returns a python generator:

```
>>> import pycrypt as pc
>>> import itertools

>>> kg = pc.CombinationKeyGenerator()
>>> for i in itertools.islice(kg.getAllKeys(), 30):
>>>     print i
```

```
('A',)
('B',)
('C',)
('D',)
('E',)
('F',)
('G',)
('H',)
('I',)
('J',)
('K',)
('L',)
('M',)
('N',)
('O',)
('P',)
('Q',)
('R',)
('S',)
('T',)
('U',)
('V',)
('W',)
('X',)
('Y',)
('Z',)
('A', 'A')
('A', 'B')
('A', 'C')
('A', 'D')
```

Tip: `itertools` is a great python module for working with iterators (generators in this case). It is really handy and has many different uses. You can see the docs here.

Here we use `itertools.islice` to look at the first 30 results that our `CombinationKeyGenerator` provides. As you might expect, it returns every possible combination of letters. Usually, you can also set some rules for the keys generated:

```
>>> kg.length_range = (3, 10)
>>> for i in itertools.islice(kg.getAllKeys(), 5):
>>>     print i
('A', 'A', 'A')
('A', 'A', 'B')
('A', 'A', 'C')
('A', 'A', 'D')
('A', 'A', 'E')
```

You can use `getRandomKey` to ... well, guess:

```
>>> print kg.getRandomKey()
('Y', 'Q', 'L', 'U', 'Q')
>>> print kg.getRandomKey()
('C', 'H', 'M', 'I')
>>> print kg.getRandomKey()
('Z', 'C', 'W', 'M', 'F', 'N', 'J', 'C', 'D')
>>> print kg.getRandomKey()
('C', 'M', 'Y')
```

Notice, how the rule we set before (length_range) also applies to this (and all other) method.

Now let's take a look at `mutateKey`, which is mainly used by the `GeneticSolver`. `mutateKey` returns a similar key based on the random number generator. The entropy can be changed with the randFunc lambda function passed as an optional argument:

```
>>> kg.mutateKey("HELLO")
('H', 'E', 'L', 'L', 'J', 'Z')
>>> kg.mutateKey("HELLO")
('H', 'E', 'L', 'P', 'O')
>>> kg.mutateKey("HELLO")
('H', 'E', 'L', 'L', 'O', 'M')
>>> kg.mutateKey("HELLO")
('H', 'N', 'L', 'L', 'O')
>>> kg.mutateKey("HELLO")
('H', 'E', 'L', 'L', 'L')
```

2.5.2 Making your own KeyGenerator

If you're trying to solve a simpler cipher and all of the possible keys can be tried out in a reasonable time, you can implement only the `getAllKeys` method. It is preferred to return a generator, as its lazy evaluation uses almost no memory. For the more complicated ciphers (like the substitution cipher), you should implement `getRandomKey` and `mutateKey`.

Tip: It's great to make some applicable rules to the KeyGenerator. You can then change them interactively during the actual cipher solving and help the solving process head the right way.

2.5.3 Further reading

To check out all KeyGenerators, see the API:

See also:

KeyGenerators

2.6 Scorers

Scorers are used by Solvers to see how good a solution is. It can be anything like scoring a Sudoku grid, but usually you'll be using them to score similarity to some language. Pycrypt comes with a Scorer for English and Czech.

2.6.1 Basic usage

Let's see our EnglishScorer in action!

```
>>> import pycrypt as pc

>>> s = pc.EnglishScorer()

>>> s.score("asdsdghuioz")
0.5275818181818182
```

```
>>> s.score("Hello World")
1.0342818181818183
```

As you can see, the jumbled text scored a half of what the English text did. You might expect a bit larger difference, but this example uses just too short text. There is no normalization of the score, so you could see scores around 1 just as well as scores over 5. Usually, jumbled text scores only a small fraction.

2.6.2 Making your own Scorer

Just extend `Scorer`'s `score` method and you're good to go!

If you want a `LanguageScorer` on the other hand, you'll need some frequency statistics, but first, let's look at the `CzechScorer` implementation:

```
import languagescorer
import czechfrequencies as cze

class CzechScorer(languagescorer.LanguageScorer):
        """Czech scorer, credits for frequencies go to MFF"""
        def __init__(self):
                self.setIdealNgramFrequencies([cze.monograms, cze.bigrams, cze.trigrams, cze.tetragra
                self.setWeights([10, 100, 1000, 10000, 100000])
```

As you can see, all you have to do is call the `setIdealNgramFrequencies` method to load frequency dictionaries. The `setWeights` just multiplies the score got from their respective n-gram frequencies (pentagrams are more relevant than monograms and pentagrams usually score much lower because of their limited dictionaries).

The frequency dictionaries are just python `dict` s, which have the n-grams as a key and their probability distribution as a value. The values, if all possible keys are referenced, should sum up to 1. The Czech data is generated from the files here. There are only Czech and English statistics to date, but more languages are to come. Should you want to process them, you can use the `ngram_converter.py` script, which comes with pycrypt.

Keep in mind, that a good Scorer should not only give good score to correct results and bad score to incorrect. It should also give half the score (or log half or something) to half correct results. This is essential, when using the genetic algorithms (and several others), to let the algorithm know, that it is on the right track. You should avoid making too big local maxima as well.

2.6.3 Further reading

To check out Scorers' source, check out the API:

See also:

Scorers

2.7 Solvers

Solvers glue everything we have learned so far together. They will get some keys from a KeyGenerator, apply these keys to the cipher with a Translator and finally score these solutions with a Scorer. They will also take care of printing out progress and optional interactions (during the solving process) from the user.

To date, there are only two Solvers. Since they are so essential for pycrypt's use, we'll go over both of them.

2.7.1 Basic usage

BruteForceSolver

We'll be trying to solve a Vigenère cipher. First, we will make the actual cipher:

```
import pycrypt as pc

text = "The White-bellied Sea Eagle is a large diurnal bird of prey in the family Accipitridae. A dis
t = pc.VigenereTranslator(key="EGG")
cipher = t.encode(text)

print cipher
```

We will get the encoded output:

```
OAX RABOX-UZEEDXW NXT ZTZGX BN T EVKZZ WBPKGVE UDKW JY IMXR DG MCX YVFBGR TXVBKBMMBWVX. T YBLOBGXMBQX
```

Since the Vigenère cipher key is only 3 characters long, the `BruteForceSolver` should suffice:

```
s = pc.BruteForceSolver(keyGenerator=pc.CombinationKeyGenerator(length_range=(1, 3)),
        translator=pc.VigenereTranslator(), scorer=pc.EnglishScorer())
s.solve(cipher)
```

The first line sets up our `BruteForceSolver`. `CombinationKeyGenerator` with small `length_range`, so that we can try out all the keys, obviously `VigenereTranslator` as the specified Translator and `EnglishScorer`.

Tip: You can set the default scorer in the conf.py file

When you'll run this, you should see all of the possible keys with their respective solution previews. In 15 seconds or so, the final output will look like this:

```
...
Score: 0.35820      Key: ZZU      Text: OAS RAWOX-PZEZDXR NXO ZTUGX WN T ZVKUZ WWPKBVE PDKR JY DMXM D
Score: 0.36785      Key: ZZV      Text: OAT RAXOX-QZEADXS NXP ZTVGX XN T AVKVZ WXPKCVE QDKS JY EMXN D
Score: 0.29618      Key: ZZW      Text: OAU RAYOX-RZEBDXT NXQ ZTWGX YN T BVKWZ WYPKDVE RDKT JY FMXO D
Score: 0.33593      Key: ZZX      Text: OAV RAZOX-SZECDXU NXR ZTXGX ZN T CVKXZ WZPKEVE SDKU JY GMXP D
Score: 0.41876      Key: ZZY      Text: OAW RAAOX-TZEDDXV NXS ZTYGX AN T DVKYZ WAPKFVE TDKV JY HMXQ D
Score: 0.33509      Key: ZZZ      Text: OAX RABOX-UZEEDXW NXT ZTZGX BN T EVKZZ WBPKGVE UDKW JY IMXR D

=====Best Solution=====
Score: 2.89494237918
Key: EGG
Text: THE WHITE-BELLIED SEA EAGLE IS A LARGE DIURNAL BIRD OF PREY IN THE FAMILY ACCIPITRIDAE. A DISTI
```

If we would know, that the key was a meaningful word, we could use for instance some sort of word list KeyGenerator (which, as of now, doesn't exist).

GeneticSolver

3 character long keys take about 20 seconds with the `BruteForceSolver`, but 4 characters would take 26 times that! That is over 8 minutes. To try out all the possible 8 character keys, it would take over 6000 years. That's where the `GeneticSolver` comes in. It uses a very basic genetic algorithm. But first, let's make a more complex Vigenère cipher from our sample text:

```
t.setKey("SPAMANDEGGS")
cipher = t.encode(text)

print cipher
```

We'll get this:

```
ARD JGUPZ-UXSSSDQ RQW ZTZSL SR N KMNBX WPBBMNK NEMW HM WBDL HZ PCX YHTSKL ZOYDIBAYSCND. M ZDLMPUMSVUQ
```

Now let's try to solve it:

```
s = pc.GeneticSolver(keyGenerator=pc.CombinationKeyGenerator(length_range=(1, 11)),
        translator=pc.VigenereTranslator(), scorer=pc.EnglishScorer())
s.solve(cipher)
```

You *should* see output similar (but maybe very different) to this:

```
 1.      Score: 0.74231      Text: HLE ENNWT-VSZLZXR MXP GNANS LY H LHUUE QQWIFUE OZTP OG XWKE OT QXE
 2.      Score: 0.85933      Text: THE QZOSP-KMFLIEX KKZ PJOFE IS U DGQRN LCURNUD HHCM WZ PRES AT SSN
 3.      Score: 0.93790      Text: THE QZOSP-KMILIEX KKZ PJOIE IS U DGQRN LFURNUD HHCM WC PRES AT SSN
 4.      Score: 1.02072      Text: THE QZOSV-KMLLIEX KKZ VJOLE IS U DGQXN LIURNUD HHIM WF PRES AT SYN
 5.      Score: 1.11349      Text: THE QZOSE-BMILIEX KKZ EAOIE IS U DGQGE LFURNUD HHRD WC PRES AT SHE
 6.      Score: 1.13169      Text: THE QOOSB-KMLLIEX ZKZ BJOLE IS U SGQDN LIURNUS HHOM WF PRES PT SEN
 7.      Score: 1.36420      Text: THE QZOTE-BMILIEX KKA EAOIE IS U DGRGE LFURNUD HIRD WC PRES AT THE
 8.      Score: 1.36962      Text: THE QZOTE-BHILIEX KKA EAJIE IS U DGRGE GFURNUD HIRD RC PRES AT THE
 9.      Score: 1.74856      Text: THE QZITE-BMILIEX KEA EAOIE IS U DARGE LFURNUD BIRD WC PRES AN THE
10.      Score: 1.88447      Text: THE QZITE-BEILIEX KEA EAGIE IS U DARGE DFURNUD BIRD OC PRES AN THE
11.      Score: 2.20848      Text: THE QZITE-BELLIEX KEA EAGLE IS U DARGE DIURNUD BIRD OF PRES AN THE
12.      Score: 2.31031      Text: THE WZITE-BELLIED KEA EAGLE IS A DARGE DIURNAD BIRD OF PREY AN THE
13.      Score: 2.34455      Text: THE WTITE-BELLIED EEA EAGLE IS A XARGE DIURNAX BIRD OF PREY UN THE
14.      Score: 2.63445      Text: THE QHITE-BELLIEX SEA EAGLE IS U LARGE DIURNUL BIRD OF PRES IN THE
15.      Score: 2.63445      Text: THE QHITE-BELLIEX SEA EAGLE IS U LARGE DIURNUL BIRD OF PRES IN THE
16.      Score: 2.63445      Text: THE QHITE-BELLIEX SEA EAGLE IS U LARGE DIURNUL BIRD OF PRES IN THE
17.      Score: 2.89494      Text: THE WHITE-BELLIED SEA EAGLE IS A LARGE DIURNAL BIRD OF PREY IN THE
18.      Score: 2.89494      Text: THE WHITE-BELLIED SEA EAGLE IS A LARGE DIURNAL BIRD OF PREY IN THE
```

If you'll stop the process with Ctrl-C (you have to be in some sort of interactive shell), you'll see the last evolution:

```
18.      Score: 2.89494      Text: THE WHITE-BELLIED SEA EAGLE IS A LARGE DIURNAL BIRD OF PREY IN THE
Evolution interrupted! Setting starting point to continue

=====Best Solution=====
Score: 2.89494237918
Key: ['S', 'P', 'A', 'M', 'A', 'N', 'D', 'E', 'G', 'G', 'S']
Text: THE WHITE-BELLIED SEA EAGLE IS A LARGE DIURNAL BIRD OF PREY IN THE FAMILY ACCIPITRIDAE. A DISTI
```

Warning: Right now, it is not unusual for the genetic algorithm to get stuck in a local maxima. It does not happen often, but when it does, just restart the script. It shouldn't happen in the future, as many improvements are planned to the actual algorithm as well as some more tools to help to resolve this problem.

As you can see, the `GeneticSolver` can prove to be highly effective. You'll want to use them in most cases, however, if you can try out all the keys in a reasonable time, `BruteForceSolver` is a better choice, as the `GeneticSolver` can prove unreliable sometimes.

2.7.2 Advanced usage

Let's move on to a more complex case of a cipher, such as a substitution cipher. Again, we'll make the encoded text first:

```
t = pc.SubstitutionTranslator()
t.setKey(dict(zip(pc.alphabet, reversed(pc.alphabet))))
cipher = t.encode(text)

print cipher
```

We set the `SubstitutionTranslator` key to a reversed alphabet (which produces a very simple cipher), but we could have chosen any possible unordered alphabet, this is just for illustration. We'll end up with this cipher:

```
GSV DSRGV-YVOORVW HVZ VZTOV RH Z OZITV WRFIMZO YRIW LU KIVB RM GSV UZNROB ZXXRKRGIRWZV. Z WRHGRMXGREV
```

Now we will attempt to solve it with the `GeneticSolver`:

```
s = pc.GeneticSolver(keyGenerator=pc.SubstitutionKeyGenerator(),
        translator=pc.SubstitutionTranslator(), scorer=pc.EnglishScorer())
s.solve(cipher)
```

Unless you are very lucky, you will see that the substitution cipher is much harder to solve. You might even want to restart a few times. Let's see an example output:

```
  1.      Score: 1.04425      Text: END PNTED-KDMMTDV HDZ DZFMD TH Z MZIFD VTRIAZM KTIV CB YIDQ TA EN
  2.      Score: 1.78308      Text: THE KHOTE-NECCOEB WEF EFUCE OW F CFAUE BOPAZFC NOAB LV DAEI OZ TH
  3.      Score: 1.98144      Text: THE KHOTE-NECCOEB WES ESUCE OW S CSAUE BOPAZSC NOAB LV DAEI OZ TH
  4.      Score: 2.03995      Text: THE KHOTE-BECCOEN WES ESUCE OW S CSAUE NOPAZSC BOAN LV DAEI OZ TH
  5.      Score: 2.11829      Text: THE KHOTE-BECCOEN WES ESUCE OW S CSAUE NOPARSC BOAN LV DAEI OR TH
  6.      Score: 2.18511      Text: THE KHOTE-BECCOEN WES ESUCE OW S CSRUE NOPRASC BORN LV DREI OA TH
  7.      Score: 2.21979      Text: THE CHOTE-LEJJOEN WES ESBJE OW S JSABE NOPAISJ LOAN VU DAER OI TH
  8.      Score: 2.27611      Text: THE KHOTE-BECCOEN WES ESUCE OW S CSRUE NOPRFSC BORN LV IRED OF TH
  9.      Score: 2.34155      Text: THE WHOTE-QEVVOEB RES ESGVE OR S VSAGE BOIANSV QOAB YC PAED ON TH
 10.      Score: 2.38612      Text: THE WHITE-QEVVIEB RES ESGVE IR S VSAGE BIOANSV QIAB YK PAED IN TH
 11.      Score: 2.40644      Text: THE WHOTE-QEVVOEU AES ESGVE OA S VSRGE UOIRNSV QORU YC PRED ON TH
 12.      Score: 2.46465      Text: THE VHOTE-QERROED FEA EAGRE OF A RASGE DOISNAR QOSD YC PSEB ON TH
 13.      Score: 2.48524      Text: THE WHOTE-QERROED FES ESGRE OF S RSIGE DOAINSR QOID YC PIEB ON TH
 Evolution interrupted! Setting starting point to continue

=====Best Solution=====
Score: 2.46465315985
Key:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
KBWVLITFSXPYNZRJMOHGCEDUQA
Text: THE VHOTE-QERROED FEA EAGRE OF A RASGE DOISNAR QOSD YC PSEB ON THE CAZORB AUUOPOTSODAE. A DOFTC
```

At the end, we have stopped the process with Ctrl-C. If you are using an interactive python shell (e.g. regular command-line python, ipython or IDLE's python shell), you should be able to continue issuing commands.

Interactive mode

The ability to interrupt the process is very useful, as we can *help* the Solver. You might want to play around with different settings for the algorithm (like population size or the randomness of mutations). But we can have a more direct control. For instance, if we take a look at the last evolution from our last example:

```
13.      Score: 2.48524      Text: THE WHOTE-QERROED FES ESGRE OF S RSIGE DOAINSR QOID YC PIEB ON THE
```

We can tell, that the "THE" is probably right. We can then lock it in place, so further evolution doesn't change it.

```
>>> s.lock("THE")
```

`GeneticSolver`'s `lock` processes the arguments and the just calls its keyGenerator's `lock` to add some rules. If no key is set (as an optional argument), it locks according to the key from the last evolution. If we, for example, would know that A translates to Z (which it does), we could call `SubstitutionKeyGenerator`'s `lock` directly:

```
>>> s.keyGenerator.lock('A', 'Z')
```

Also now that we have some readable results, we can increase the randomness a bit:

```
>>> s.keyGenerator.randFunc = lambda x: x ** 3
```

When the `SubstitutionKeyGenerator` calculates how many elements to swap around, it gets a random value between 0 and 1. It is then put through its randFunc. The default is `lambda x: x ** 6`, so now, it will tend to swap more characters.

Tip: If, for any reason, you want to start the evolution again while keeping the locks, you can do:

```
>>> s.setStartingPoint(None)
```

Now, let's continue the evolution:

```
>>> s.solve(cipher)
```

You may have to set up some more locks, but in the end, you should end up with this:

```
...
 17.      Score: 2.89556      Text: THE WHITE-BELLIED SEA EAGLE IS A LARGE DIURNAL BIRD OF PREY IN TH
 Evolution interrupted! Setting starting point to continue

=====Best Solution=====
Score: 2.89555799257
Key:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQPONMLKJIHGFEDCBA
Text: THE WHITE-BELLIED SEA EAGLE IS A LARGE DIURNAL BIRD OF PREY IN THE FAMILY ACCIPITRIDAE. A DISTI
```

As we can see, the correct key is in fact the reversed alphabet.

2.7.3 Making your own Solver

All you have to do is to implement the `solve` method. You should be supporting the `startingPoint` variable, as it is a useful feature. For printing, there are prepared the `printer` and `lastPrint` methods. (TODO)

2.7.4 Next steps

We have covered Solvers, which is the last part of pycrypt. You should be now able to use it efficiently.

Next, we will go over some useful external modules, which could come in handy.

If you want more guidelines, you can see example uses on ciphers from real cryptography game (hopefully regularly updated).

2.7.5 Further reading

To see the source code of Solvers, you can refer to the API:

See also:

Solvers

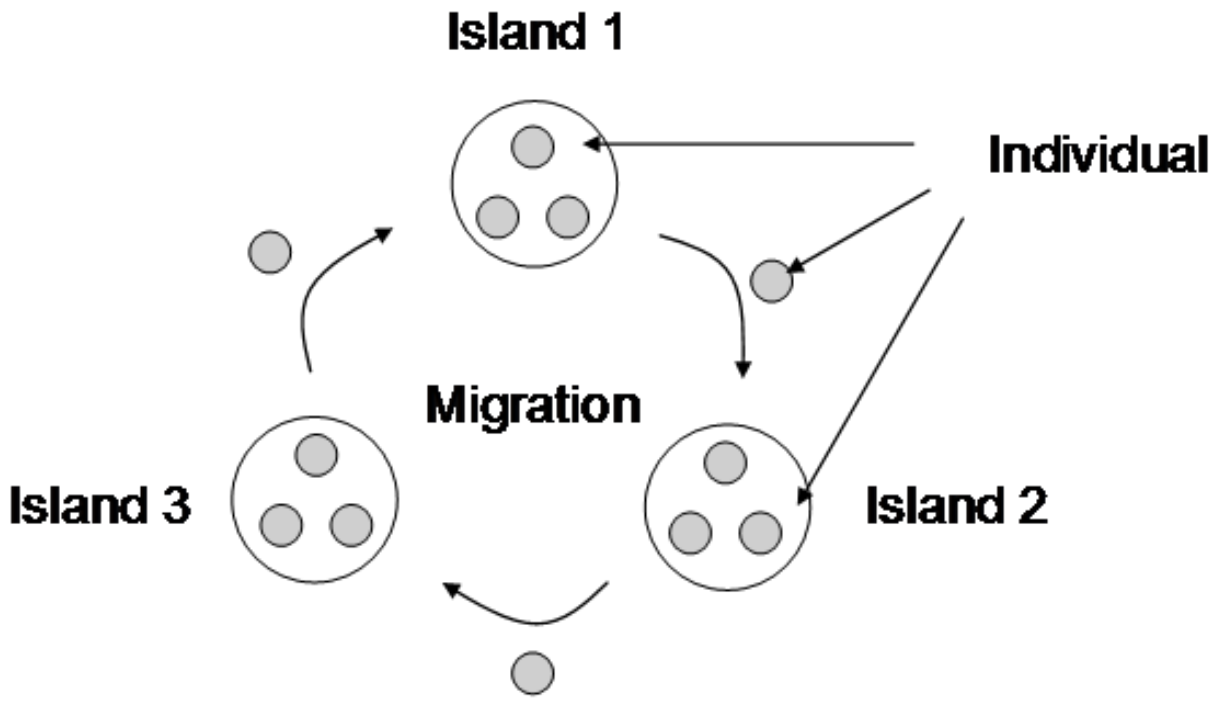
2.8 What's new in v0.2

Apart from some bug fixes and improvements, version 0.2 of pycrypt came with some new features. Let's break them down:

2.8.1 The island model

The biggest improvement in pycrypt is that it's now multi-threaded. It can now eat up all your processing power and is about 4 (or equivalent to your number of cores) times faster.

The new `ThreadedGeneticSolver` has the same interface as `GeneticSolver`, but runs as many instances of `GeneticSolver` as you have cores (you can change that with the `num_processes` argument). They don't just run separated though, every 10 iterations (set by `migration_iterations`) the "islands" exchange their best individuals in a cyclic pattern.

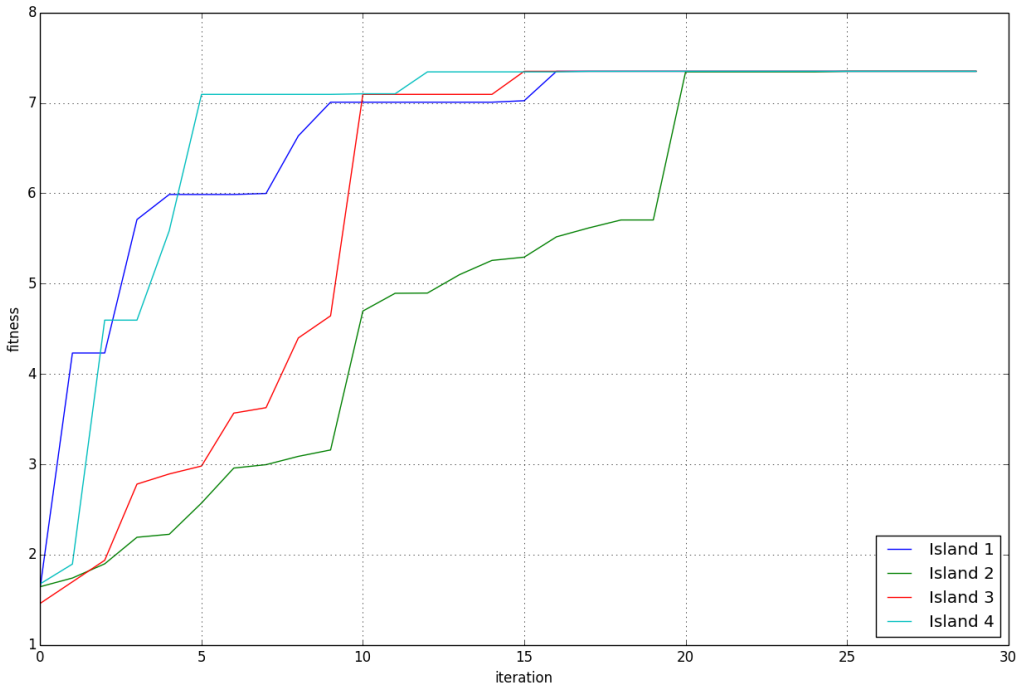


This strategy makes it a lot better, mainly because the evolution doesn't get stuck in local maxima as much now. If we look at a plot of an evolution with `ThreadedGeneticSolver`:

Even though island 2 got behind, the other islands helped it get back up to speed on iteration 20 (migrations occur on every tenth iteration).

> **Warning:** The interactive interruption of `ThreadedGeneticSolver` doesn't work so well. On different OSes happen different problems, the safest bet is just to set the iteration max and wait for the evolution to actually finish.

This brings us to another feature:



2.8.2 Evolution plotting

Both `ThreadedGeneticSolver` and `GeneticSolver` now support the `plotLog` method, you just need to enable logging with `log=True`. Let's take a look at `GeneticSolver` plot:

This proved to be very helpful during development, as it revealed some quirks the algorithms had.

2.8.3 Crossovers

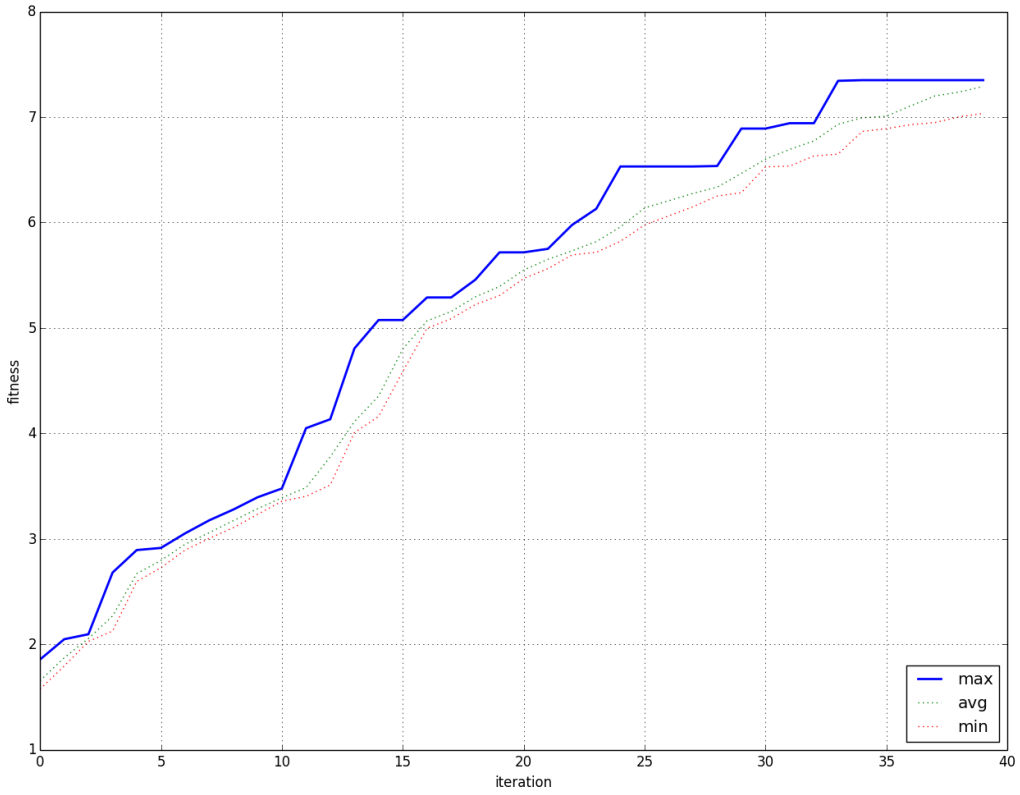
Initially, pycrypt didn't include crossovers in its genetic algorithms. That was because permutations (which are used as keys for substitution ciphers) aren't easily crossed over. Inspired by the algorithm described here (the order crossover 1), along with some standard algorithms as 1 and 2 point crossovers and tournament selection, they are now implemented.

I think that this added some not insignificant boost, but it's not easily measured.

2.8.4 Temperature scaling

Another experimental feature, which is based on cooling down the mutations as the fitness gets better. It works a bit different in pycrypt - with high temperature, the most frequent letters like 'E', 'T' and 'A' get switched. As the evolution progresses, less frequent letters get switched, so that the finishing touches on the solution are made.

This approach didn't prove very useful though, so I turned it off by default. I think it is because he biggest problems are local maxima and they don't necessarily have the infrequent letters wrong, so the evolution gets stuck even more.


fitness
iteration
max
avg
min
1
2
3
4
5
6
7
8
0
5
10
15
20
25
30
35
40

2.8.5 Cached scoring

The scoring is the performance bottleneck of pycrypt. All scoring is now cached, so if you score an individual twice, the score gets computed only once. This is managed by the `cache` decorator in the `utils` module and it can be applied to any function or method you want.

2.8.6 Easier installation

Pycrypt is now structured as a legit python package with requirements done finally right, so you can install it quickly with:

```
$ pip install "git+https://github.com/PrehistoricTeam/pycrypt.git@master#egg=pycrypt"
```

I might even consider getting pycrypt on PyPI in the near future.

2.9 API

2.9.1 pycrypt package

pycrypt.keygenerators package

Submodules

pycrypt.keygenerators.combinationkeygenerator module

class pycrypt.keygenerators.combinationkeygenerator.**CombinationKeyGenerator**(*alphabet='ABCDEFGH*
rand_func=<function
<lambda>>,
length_range=(1,
6),
***kwargs*)

Bases: *pycrypt.keygenerators.keygenerator.KeyGenerator*

getRandomKey(*length=None*)
If length is None, random from range is set

getAllKeys()
Generator of all combinations from shortest to longest from length_range

mutateKey(*key*)
Changes random number of elements, randomly changes length by 1

pycrypt.keygenerators.crossovers module

pycrypt.keygenerators.crossovers.**point1**(*parent1*, *parent2*)
Basic 1 point crossover for lists

pycrypt.keygenerators.crossovers.**point2**(*parent1*, *parent2*)
Basic 2 point crossover for lists

pycrypt.keygenerators.crossovers.**permutation**(*parent1*, *parent2*)
Crossover for permutations, parents should be dicts. Inspired by order crossover 1 from http://www.cs.colostate.edu/~genitor/1995/permutations.pdf

Note that crossing over two same individuals won't always return the same.

class pycrypt.keygenerators.crossovers.**Tournament**(*crossover_func=<function point2>*, *tournament_size=20*, *crossovers=6*)

Basic tournament selector for crossovers

crossover(*population*)
Returns a list of new offsprings from population

pycrypt.keygenerators.keygenerator module

class pycrypt.keygenerators.keygenerator.**KeyGenerator**(*crossover=<pycrypt.keygenerators.crossovers.Tournamen instance>*, ***kwargs*)

Bases: object

Abstract class for generating keys for specific Translator

getRandomKey()
Random key i.e. for starting genetic population

getAllKeys()
Get all possible keys, python generator preferably

mutateKey(*key*)
For genetics - get similar key

crossover(*population*)
For genetics - get some new offsprings

pycrypt.keygenerators.numberkeygenerator module

class pycrypt.keygenerators.numberkeygenerator.**NumberKeyGenerator**(*max_number=26*, *rand_func=<function <lambda>>*, ***kwargs*)

Bases: *pycrypt.keygenerators.keygenerator.KeyGenerator*

getRandomKey()

getAllKeys()

mutateKey(*key*)
Change randFunc for different transformation number after random.random

pycrypt.keygenerators.permutationkeygenerator module

class pycrypt.keygenerators.permutationkeygenerator.**PermutationKeyGenerator**(*sequence='ABCDEFGH rand_func=<function <lambda>>*, ***kwargs*)

Bases: *pycrypt.keygenerators.substitutionkeygenerator.SubstitutionKeyGenerator*

getRandomKey()

getAllKeys()
Returns all permutations in lexicographic order (according to indexing in the given sequence)

mutateKey(*key*)
Swaps random number of elements around

lock(*indx*, *value*)
Lock an index of the key, so that the other functions return only keys with the set value on the given index

unlock(*indx*)

pycrypt.keygenerators.substitutionkeygenerator module

class pycrypt.keygenerators.substitutionkeygenerator.**SubstitutionKeyGenerator**(*alphabet='ABCDEFG rand_func=<function <lambda>>, weighted=None, crossover=<pycrypt.k in- stance>, **kwargs*)

Bases: *pycrypt.keygenerators.keygenerator.KeyGenerator*

getRandomKey(*_return_list=False*)

getAllKeys(*_return_list=False*)
Generator of all keys in lexicographic order (according to indexing in the given alphabet)

mutateKey(*key*, *_return_list=False*, *temp=1*)
Swaps random number of elements around

lock(*element*, *value=None*, *key=None*)
Lock an element of the key, so that the other functions return only keys with the set value

unlock(*element*)

clearLock()

pycrypt.keygenerators.test_crossovers module

class pycrypt.keygenerators.test_crossovers.**TestCrossovers**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_point1()

test_point2()

test_permutation()

Module contents

pycrypt.scorers package

Submodules

pycrypt.scorers.cgetngramfrequencies module

pycrypt.scorers.czechfrequencies module

Czech frequencies, extract from http://ufal.mff.cuni.cz/~hajic/courses/npfl067/stats/czech.html data from 564532247 characters, kept only most relevant for speed

pycrypt.scorers.czechscorer module

class pycrypt.scorers.czechscorer.**CzechScorer**
Bases: *pycrypt.scorers.languagescorer.LanguageScorer*

Czech scorer, credits for frequencies go to MFF

pycrypt.scorers.englishfrequencies module

pycrypt.scorers.englishscorer module

class pycrypt.scorers.englishscorer.**EnglishScorer**
Bases: *pycrypt.scorers.languagescorer.LanguageScorer*

English scorer, frequencies got from interwebz

pycrypt.scorers.languagescorer module

class pycrypt.scorers.languagescorer.**LanguageScorer**
Bases: *pycrypt.scorers.scorer.Scorer*

Scorer for languages based on N-grams and words

words = None

minWordLen = 3

maxWordLen = 10

log = False

ngramWeights = None

wordWeight = 0

unidec = True

setIdealNgramFrequencies (*freqs*)

loadWordList (*path*, *minwordlen=3*, *maxwordlen=10*)
Load words from file, 1 word per line

setWeights (*ngram_weights*, *word_weight=0*)
Score multipliers, ngram_weights is list corresponding to ideal frequencies when something is 0, it's ignored when scoring

getNgramFrequencies (*text*, *length*)
Get dictionary of frequencies of N-grams (of given length)

scoreNgrams (*text*)

scoreWords (*text*)

score (**args*, ***kwargs*)

pycrypt.scorers.ngram_converter module

pycrypt.scorers.scorer module

class pycrypt.scorers.scorer.**Scorer**
Abstract class for scoring strings (i.e. language resemblance)

score (*text*)
Get score of a string

Module contents

pycrypt.solvers package

Submodules

pycrypt.solvers.bruteforcesolver module

class pycrypt.solvers.bruteforcesolver.**BruteForceSolver** (*keyGenerator=<pycrypt.keygenerators.numberkeygene object>*, *translator=<pycrypt.translators.caesartranslator.CaesarTran instance>*, *scorer=<pycrypt.scorers.czechscorer.CzechScorer instance>*, *quiet=False*)
Bases: *pycrypt.solvers.solver.Solver*

Tries out all possible solutions

solve (*text=None*, *return_all_keys=False*)

lastPrint (*key*, *score*, *text=None*)

setKeyGenerator (*keyGenerator*)

setStartingPoint (*startingPoint*)

pycrypt.solvers.geneticsolver module

class pycrypt.solvers.geneticsolver.**GeneticSolver**(*keyGenerator=None*, *translator=<pycrypt.translators.substitutiontranslator.SubstitutionTran instance>*, *scorer=<pycrypt.scorers.czechscorer.CzechScorer instance>*, *population_size=20*, *mutations=20*, *random_starting_population=1000*, *quiet=False*, *exclude_tried=False*, *log=False*, *crossover=True*, *temperature=False*, *temperature_func=<function <lambda>>*)

Bases: *pycrypt.solvers.solver.Solver*

Uses own genetic algorithm, calls KeyGenerators mutateKey method

solve(*text=None*, *iterations=0*, *return_all_keys=False*)
Set iterations to 0 for infinite loop

printer(*key*, *score*, *text=None*, *iterations=None*)
Gets the best sample in population in every cycle

setStartingPoint(*startingPoint*)
Starting population -> can be list

lock(*string*, *key=None*)
Lock character in the keyGenerator for the given key, if None, startingPoint key is used

plotLog()

pycrypt.solvers.solver module

class pycrypt.solvers.solver.**Solver**(*keyGenerator*, *translator=None*, *scorer=<pycrypt.scorers.czechscorer.CzechScorer instance>*)

Bases: object

Abstract class for connecting KeyGenerators, Scorers and optionally Translators

solve(*text=None*)
Find best scored key for the given text (if None, the key itself will be scored) Returns best (score, key) pair

setStartingPoint(*startingPoint*)
Set where the solve method should start (useful for continuing genetics)

score(*key*, *text=None*, *return_ciphered=True*)

printer(*key*, *score*, *text=None*)
Callback method for every key generated and scored

lastPrint(*key*, *score*, *text=None*)
Callback method for last and best result

pycrypt.solvers.threadedgeneticsolver module

pycrypt.solvers.threadedgeneticsolver.**mapper**(*solver*)

class pycrypt.solvers.threadedgeneticsolver.**ThreadedGeneticSolver**(*keyGenerator=<pycrypt.keygenerators.object>*, *translator=<pycrypt.translators.substitutiontr instance>*, *scorer=<pycrypt.scorers.czechscorer.C instance>*, *num_processes=None*, *migration_iterations=10*, *migration_size=10*, *quiet=False*, *log=False*, ***kwargs*)

Bases: *pycrypt.solvers.solver.Solver*

Implements the island model using GeneticSolver

solve(*text=None*, *iterations=0*, *return_all_keys=False*)
Paralelized GeneticSolver's solve. Note that you can't interrupt the evolution as you could normally.

printer(*key*, *score*, *text=None*, *iterations=None*)
Gets the best sample in population in every cycle

setStartingPoint(*startingPoint*)

lock(*string*, *key=None*)

plotLog()

Module contents

pycrypt.translators package

Submodules

pycrypt.translators.asciitranslator module

class pycrypt.translators.asciitranslator.**ASCIITranslator**
Bases: *pycrypt.translators.translator.Translator*

Simple ASCII translation using unichr

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

pycrypt.translators.binarytranslator module

class pycrypt.translators.binarytranslator.**BinaryTranslator**(*start_with_one=False*)
Bases: *pycrypt.translators.translator.Translator*

startWithOne = False

setStartWithOne(*b*)

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

graphicEncode(*cipher*)

pycrypt.translators.brailletranslator module

class pycrypt.translators.brailletranslator.**BrailleTranslator**
Bases: *pycrypt.translators.translator.Translator*

Braille, translation formats: swza is T (qw as zx)

key = {‘’: ‘ ‘, ‘qzx’: ‘U’, ‘azws’: ‘T’, ‘qzs’: ‘O’, ‘qzw’: ‘M’, ‘qzwx’: ‘X’, ‘aw’: ‘I’, ‘q’: ‘A’, ‘qaw’: ‘F’, ‘qas’: ‘H’, ‘qaz’: ‘

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

graphicEncode(*cipher*)

pycrypt.translators.caesartranslator module

class pycrypt.translators.caesartranslator.**CaesarTranslator**(*key=13*)
Bases: *pycrypt.translators.translator.Translator*

Simple alphabet rotation, default ROT13

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

pycrypt.translators.morsecodetranslator module

class pycrypt.translators.morsecodetranslator.**MorseCodeTranslator**
Bases: *pycrypt.translators.translator.Translator*

Morse Code, translation formats: .-//-... ; ., ,... ; [[0,1],[1,0,0,0]]

key = {‘’: ‘ ‘, ‘–..–‘: ‘,’, ‘....-‘: ‘4’, ‘.....’: ‘5’, ‘-...’: ‘B’, ‘-..-‘: ‘X’, ‘.-.’: ‘R’, ‘.–‘: ‘W’, ‘..—‘: ‘2’, ‘.-‘: ‘A’, ‘..’: ‘I’, ‘...–‘: ‘3’, ‘

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

graphicEncode(*cipher, gkey={‘-‘: [1], ‘.’: [0]}*)
change gkey dict to other . and - representations (i.e. ‘-‘ can be [1, 1, 1])

pycrypt.translators.numberedalphabettranslator module

class pycrypt.translators.numberedalphabettranslator.**NumberedAlphabetTranslator**
Bases: *pycrypt.translators.translator.Translator*

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

pycrypt.translators.polishcrosstranslator module

class pycrypt.translators.polishcrosstranslator.**PolishCrossTranslator**(*using_ch=True*)
Bases: *pycrypt.translators.translator.Translator*

Polish cross, Ch optional as argument, input: q1 -> A, c3 -> Z

key = {'a': 3, 'c': 8, 'e': 2, 'd': 5, 'q': 0, 's': 4, 'w': 1, 'x': 7, 'z': 6}

setUsingCh(*using_ch*)

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

graphicEncode(*cipher*, *three_by_three_grid=False*)
Splits input to words, draws letters in words over each other. If three_by_three_grid argument is False, 9x3 grid with individual letters in the polish cross will be used

pycrypt.translators.semaphoretranslator module

class pycrypt.translators.semaphoretranslator.**SemaphoreTranslator**
Bases: *pycrypt.translators.translator.Translator*

Semaphore, translation format: zx is A (qwe a d zxc)

key = {'': ' ', 'ac': 'S', 'ad': 'R', 'xc': 'G', 'ea': 'Q', 'ec': 'X', 'zc': 'N', 'zx': 'A', 'ex': 'E', 'ez': 'L', 'ax': 'B', 'az': 'H', '

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

graphicEncode(*cipher*)

pycrypt.translators.substitutiontranslator module

class pycrypt.translators.substitutiontranslator.**SubstitutionTranslator**(*key='ZYXWVUTSRQPONMLK*
Bases: *pycrypt.translators.translator.Translator*

Basic substitution, default key reversed alphabet

setKey(*key*)

parseInput(*cipher*)

translate(*cipher*)

encode(*cipher*)

pycrypt.translators.test_binarytranslator module

class pycrypt.translators.test_binarytranslator.**TestBinaryTranslator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

test_graphicEncode()

pycrypt.translators.test_brailletranslator module

class pycrypt.translators.test_brailletranslator.**TestBrailleTranslator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

test_graphicEncode()

pycrypt.translators.test_caesartranslator module

class pycrypt.translators.test_caesartranslator.**TestCaesarTranslator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

test_parseInput()

pycrypt.translators.test_morsecodetranslator module

class pycrypt.translators.test_morsecodetranslator.**TestMorseCodeTranslator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

test_graphicEncode()

pycrypt.translators.test_polishcrosstranslator module

class pycrypt.translators.test_polishcrosstranslator.**TestPolishCrossTranslator**(*methodName='runT*
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

test_graphicEncode()

pycrypt.translators.test_semaphoretranslator module

class pycrypt.translators.test_semaphoretranslator.**TestSemaphoreTranslator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

test_graphicEncode()

pycrypt.translators.test_substitutiontranslator module

class pycrypt.translators.test_substitutiontranslator.**TestSubstitutionTranslator**(*methodName='ru*
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

test_parseInput()

test_setKey()

pycrypt.translators.test_vigeneretranslator module

class pycrypt.translators.test_vigeneretranslator.**TestVigenereTranslator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_translate()

test_encode()

pycrypt.translators.translator module

class pycrypt.translators.translator.**Translator**
Abstract class for translating standard ciphers (i.e. Morse Code)

key = []

translate(**args*)
Base method for decoding a cipher

interactiveTranslate()
For quick translating with each character typed from the user, type ! to remove last characters

encode(**args*)
Reversed translation

decode(**args*)
Just and alias for translate

graphicEncode(**args*)
Return in numpy array for easy plotting

parseInput(*cipher*)
Standardize input to a list, values preferably integers indexed from 0

setKey(*key*)

pycrypt.translators.vigeneretranslator module

class pycrypt.translators.vigeneretranslator.**VigenereTranslator**(*key='A'*, *ignore_nonletters=True*)
Bases: *pycrypt.translators.translator.Translator*

Adds perpetually key letters to text (Caesar with longer keys)

parseInput(*cipher*)

translate(*cipher*, *a_is_one=True*)

encode(*cipher*)

pycrypt.translators.xortranslator module

class pycrypt.translators.xortranslator.**XorTranslator**
Bases: *pycrypt.translators.translator.Translator*

One time pad translator

translate(*cipher*)

encode(*cipher*)

Module contents

2.9.2 pycrypt.utils module

pycrypt.utils.**split**(*string*)

pycrypt.utils.**line_split**(*string*)

pycrypt.utils.**array_concat**(*raw_arrays*)
Concats 2d numpy arrays to one big one, lines don't have to be the same size

pycrypt.utils.**plot_array**(*arr*)
Plots binary 2d numpy array

pycrypt.utils.**get_frequency**(*string*, *freq_alphabet='ABCDEFGHIJKLMNOPQRSTUVWXYZ'*, *ratio=False*)
Count frequency of given alphabet, if None, count every char. Set ratio True to divide by length

pycrypt.utils.**plot_dict**(*d*)
Plots bar graph of dict (usually used with get_frequency)

pycrypt.utils.**pprint_dict**(*d*)
Prints dicts keys and values on top of each other

pycrypt.utils.**plot_genetic_log**(*log*)
Plots the max, min and avg fitness of the population

pycrypt.utils.**plot_genetic_log_threaded**(*log*)
Plots each island individually

pycrypt.utils.**cache**(*func*)
General decorator for function caching, if called with same arguments, it is bypassed

2.9.3 Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

A

B

C

D

E

G

S

T

U

V

W

X