Py-Enigma Documentation

Release 0.1

Brian Neal

Jul 26, 2017

Contents

Author Brian Neal <bgneal@gmail.com>

Version 0.1

Date Jul 24, 2017

Home Page https://bitbucket.org/bgneal/enigma/

License MIT License (see LICENSE.txt)

Documentation http://py-enigma.readthedocs.org/

Support https://bitbucket.org/bgneal/enigma/issues

Py-Enigma is a historically accurate Enigma machine simulation library written in Python. Py-Enigma includes a simple command-line application to allow for quick experimenting and scripting.

Documentation contents:

CHAPTER 1

Overview

Introduction

Py-Enigma is a Python 3 library for simulating the Enigma machines used by the German armed forces (*Wehrmacht*) during World War II. Py-Enigma is historically accurate, meaning it can interoperate with actual Wehrmacht Enigma machines. In other words, Py-Enigma can decrypt coded messages created with an actual Enigma, and it can encrypt messages that an actual Enigma can decode.

It is hoped that this library will be useful to Enigma enthusiasts, historians, and students interested in cryptography.

Py-Enigma strives to be Pythonic, easy to use, and comes with both unit tests and documentation. Py-Enigma is a library for building applications for encrypting and decrypting Enigma messages. However, it also ships with a simple command-line application that can encrypt & decrypt messages for scripting and experimentation.

Scope

Currently, Py-Enigma simulates the Wehrmacht Enigma machines. This includes the 3 and 4 rotor machines used by the German Army (*Heer*), Air Force (*Luftwaffe*), and Navy (*Kriegsmarine*). Simulation of other Enigma models, including the various commercial, railroad, foreign market, and Abhwer (Military Intelligence) models may come later if there is enough interest and data available.

Quick Example

This example shows how the library can be used to decode a message using the procedure employed by the German army

```
from enigma.machine import EnigmaMachine

# setup machine according to specs from a daily key sheet:
```

```
machine = EnigmaMachine.from_key_sheet(
       rotors='II IV V',
       reflector='B',
       ring_settings=[1, 20, 11],
       plugboard_settings='AV BS CG DL FU HZ IN KM OW RX')

# set machine initial starting position
machine.set_display('WXC')

# decrypt the message key
msg_key = machine.process_text('KCH')

# decrypt the cipher text with the unencrypted message key
machine.set_display(msg_key)

ciphertext = 'NIBLFMYMLLUFWCASCSSNVHAZ'
plaintext = machine.process_text(ciphertext)

print(plaintext)
```

This program prints:

```
THEXRUSSIANSXAREXCOMINGX
```

Py-Enigma also includes a command-line application for processing messages. Assuming you have a proper key file that contains the same initial settings as the code above, the above example can be performed on the command-line:

```
$ pyenigma.py --key-file=keys.txt --start=WXC --text='KCH'
BLA
$ pyenigma.py --key-file=keys.txt --start=BLA --text='NIBLFMYMLLUFWCASCSSNVHAZ'
THEXRUSSIANSXAREXCOMINGX
```

The format of the key file can be found in *Key file format*.

Requirements

Py-Enigma is written in Python, specifically Python 3.2. It has no other requirements or dependencies.

Installation

Py-Enigma is available on the Python Package Index (PyPI). You can install it using pip:

```
$ pip install py-enigma             # install
$ pip install --upgrade py-enigma   # upgrade
```

You may also download a tarball or .zip file of the latest code using the "get source" link on the Py-Enigma Bitbucket page. Alternatively if you use Mercurial, you can clone the repository with the following command:

```
$ hg clone https://bitbucket.org/bgneal/enigma
```

If you did not use pip, you can install with this command:

```
$ python setup.py install
```

Support & Source

All support takes place at the Py-Enigma Bitbucket page. Please enter any feature requests or bugs into the issue tracker.

You may also clone the Mercurial source code repository:

```
$ hg clone https://bitbucket.org/bgneal/enigma
```

Acknowledgements & References

This software would not have been possible without the thorough and detailed descriptions of the Enigma machine on Dirk Rijmenants' incredible Cipher Machines and Cryptology website. In particular, his Technical Details of the Enigma Machine page was a gold mine of information.

Dirk has also written an Enigma simulator in Visual Basic. Although I did not look at his source code, I did use his simulator to check the operation of Py-Enigma.

I would also like to recommend the photos and video at Dr. Thomas B. Perera's Enigma Museum.

Another good website is The Enigma and the Bombe by Graham Ellsbury.

A nice video which shows the basic components and operation of the Enigma Machine is on YouTube: Nadia Baker & Enigma demo.

CHAPTER 2

User's guide

This short guide attempts to get you up and running with Py-Enigma quickly. For more detailed information, please see the *Reference manual*.

If you are new to Enigma machines

This guide assumes you know the basics of Enigma machines. Before proceeding with Py-Enigma please explore some of the links presented in the *Acknowledgements & References*. For the most complete and detailed description of how an Enigma machine works, please see Dirk Rijmenants' excellent Technical Details of the Enigma Machine.

Building your Enigma machine

If you are interested in working with historically accurate Enigma machines, the easiest way to build your first machine is to use the "key sheet" shortcut functions. If instead you wish to experiment with custom designed rotors or configurations, you can build a machine out of separate components by hand. These two approaches are demonstrated in the following sections.

Using key sheet shortcuts

During the war, Enigma machine operators re-configured their machines every day according to a code book, or key sheet, to help increase security. Each key sheet contained daily Enigma settings for one month. Before transmitting the first message of the day, the operator looked up the current day on the key sheet for the given month and configured the machine accordingly. The key sheet specified:

- *Walzenlage*: what rotors to use, and what order to put them into the machine
- *Ringstellung*: the ring settings for each rotor
- *Steckerverbindungen*: the plugboard connections

- *Kenngruppen*: special text fragments that should be transmitted to identify the transmitter's key settings to any receiver. This is also known as the *message indicator*.

The reflector setting was usually fixed and not changed once in the field. The choice of reflector seems to have been decided at the unit level to establish different networks. Of course our simulation is not hindered by these logistical concerns, and our simulated key sheets will also specify reflector type.

When an Enigma machine operator received a message from a radio operator, probably his first task was to determine what key settings were used to transmit the message. For example, the message could have been transmitted the day before, and he was only handed the message just now. This was accomplished by transitting (in the clear) certain text fragments, the so-called *Kenngruppen*, at certain points in the message. By examining these text groups, the operator could scan the key sheet for today and perhaps the past few days and hopefully identify what day the message was sent. The operator would then reconfigure his Enigma machine accordingly and decode the message. The *Kenngruppen* was ignored when decrypting the actual message.

The *EnigmaMachine* class has two class methods for constructing machines from key sheet data. The first class method is called *from_key_sheet*:

```
from enigma.machine import EnigmaMachine

machine = EnigmaMachine.from_key_sheet(
       rotors='IV V I',
       reflector='B',
       ring_settings='21 15 16',
       plugboard_settings='AC LS BQ WN MY UV FJ PZ TR OK')
```

This is all well and good if you wish to simulate an army or air force Enigma machine. But what about navy (*Kriegsmarine*) models? Navy Enigma machines and key sheets have slightly different nomenclature. This is also no problem for Py-Enigma:

```
machine = EnigmaMachine.from_key_sheet(
       rotors='Beta VII IV V',
       reflector='B-Thin',
       ring_settings='G N O',
       plugboard_settings='18/26 17/4 21/6 3/16 19/14 22/7 8/1 12/25 5/9 10/15')
```

Some notes on the parameters:

- `rotors` can either be a space separated list of rotor names, or a list of rotor name strings. For a complete list of supported rotor names, see *Simulated rotor models*.
- `reflector` is a string that names the reflector to use. For a complete list of supported reflector names, see *Simulated reflector types*.
- `ring_settings` can be a space separated list of uppercase letters or numbers, as would be found on a key sheet. An empty string or `None` means ring settings of all 'A' or 1.
- `plugboard_settings` can either be space separated uppercase letter pairs, or slash separated numbers. Note that 'AB' is equivalent to '1/2', etc.

Warning: `ring_settings` can also take a list of integers, but these integers are **0-based**. Remember that when using a string of numbers they are **1-based** to correspond to actual historical key sheet data. In other words, these values produce identical ring settings: `[0, 5, 15]`, `'A F P'`, and `'1 6 16'`.

The second shortcut function allows you to keep your key settings stored in an external file:

```
from enigma.machine import EnigmaMachine
```

```
with open('my_enigma_keys.txt', 'r') as f:
   machine = EnigmaMachine.from_key_file(f, day=13)
```

The class method *from_key_file* builds an *EnigmaMachine* from settings stored in a simulated monthly key sheet file. The format of this file is explained in *Key file format*. The `day` argument allows you to specify the day of the month (1-31). If this parameter is omitted or `None`, the day value is obtained from the current date.

Constructing by hand

It is also possible to "build an Enigma machine by hand" by explicitly providing the component objects to the *EnigmaMachine* constructor. This makes it possible to invent different rotor and reflector types:

```
from enigma.rotors.rotor import Rotor
from enigma.plugboard import Plugboard
from enigma.machine import EnigmaMachine

r1 = Rotor('my rotor1', 'EKMFLGDQVZNTOWYHXUSPAIBRCJ', ring_setting=0, stepping='Q')
r2 = Rotor('my rotor2', 'AJDKSIRUXBLHWTMCQGZNPYFVOE', ring_setting=5, stepping='E')
r3 = Rotor('my rotor3', 'BDFHJLCPRTXVZNYEIWGAKMUSQO', ring_setting=15, stepping='V')

reflector = Rotor('my reflector', 'FVPJIAOYEDRZXWGCTKUQSBNMHL')

pb = Plugboard.from_key_sheet('PO ML IU KJ NH YT GB VF RE DC')

machine = EnigmaMachine([r1, r2, r3], reflector, pb)
```

This example illustrates a few different things:

- When calling the *Rotor* constructor directly, the internal wiring is specified as a 26-character long string which specifies the cipher substitution. This notation is consistent with several online sources of Enigma information.
- *Rotor* `ring_setting` arguments are 0-based integers (0-25).
- *Rotor* `stepping` arguments specify when rotors turn their neighbors. For more information see the *Rotor* reference.
- Reflectors are simulated as rotors that have no ring setting or stepping capability.
- *Plugboard* objects have a convenient *from_key_sheet* class method constructor that works in exactly the same way as the previous example.
- When calling the *EnigmaMachine* constructor directly, the rotor assignment is specified by a list of rotors where order specifies the left-to-right order in the machine.

Note: If you decide to create your own reflector, and you desire to maintain reciprocal encryption & decryption (a fundamental characteristic of war-time Enigma machines), your connections must be made in pairs. Thus if you wire 'A' to 'G', you must also wire 'G' to 'A', and so on.

For more details on the various constructor arguments, please see the *Reference manual*.

Encrypting & Decrypting

Now that you have built your Enigma machine, you probably want to start using it to encrypt and decrypt text! The first step is to set your initial rotor positions. This is critical if you want someone else to understand your message!

```
machine.set_display('XYZ')        # set rotor positions
```

The value given to *set_display* is a simple string, which must have one uppercase letter per rotor in your machine. In this example, we are setting the leftmost rotor to 'X', the middle rotor to 'Y', and the rightmost rotor to 'Z'.

If you ever need to obtain the current rotor positions, you can use the *get_display* method:

```
position = machine.get_display()     # read rotor position
```

Note: The *set_display* method always takes letters for simulation convenience. If you are simulating an Enigma machine with numeric rotors, you'll have to translate the numbers to the appropriate letters. On actual Enigma machines, a label on the inside box lid had such a table to aid the operator.

Next, you can simulate a single key press:

```
c = machine.key_press('A')
```

The input to *key_press* is a string that consists of a single uppercase letter. Invalid input will raise an `EnigmaError` exception. The transformed text is returned.

To process a whole string of text:

```
c = machine.process_text('This is a test!', replace_char='X')
```

The *process_text* method accepts an arbitrary string and performs some processing on it before internally calling *key_press* on each element of the string.

First, all input is converted to uppercase. Next, any character not in the Enigma uppercase character set is either replaced or dropped from the input according to the `replace_char` parameter. If `replace_char` is a string of one character, it is used as the replacement character. If it is `None`, the invalid input character is removed from the message. Thus the previous example is equivalent to:

```
c = machine.process_text('THISXISXAXTESTX')
```

This is all you need to start creating encrypted and decrypted messages.

Example communication procedure

The Wehrmacht had various elaborate procedures for transmitting and receiving messages. These procedures varied by service branch and also changed during the course of the war. In general, the Kriegsmarine procedures were more elaborate and involved not only key sheets but other auxiliary documents. On top of this, each branch of the military had its own conventions for encoding abbreviations, numbers, space characters, place names, etc. Important words or phrases may need to be repeated or stressed in some way.

We will now present a simplified scenario based on a procedure employed by the army (*Heer*) after 1940. This example is based upon one found in Dirk Rijmenants' simulator manual, which is based upon a real-life example from Frode Weierud's Cryptocellar website.

Suppose a message needs to be transmitted. The operator of the transmitting machine consults his key sheet and configures his machine according to the daily settings found inside. Let's suppose the key sheet dictates the following initial parameters for the current day:

- Rotor usage and order is *II IV V*
- Ring settings for each rotor, in order, are: *B U L*

- Plugboard settings are: *AV BS CG DL FU HZ IN KM OW RX*
- One of the daily Kenngruppen possibilities is *UGZ*

Let us also assume the reflector employed by this army unit is 'B'.

The operator then configures his machine:

```
machine = EnigmaMachine.from_key_sheet(
       rotors='II IV V',
       reflector='B',
       ring_settings='B U L',
       plugboard_settings='AV BS CG DL FU HZ IN KM OW RX')
```

Suppose the Enigma operator was handed a message for transmit by an officer which reads "The Russians are coming!" The operator would first randomly decide two things:

- Initial rotor positions, say `WXC`
- A three letter *message key*, say `BLA`

The operator would then turn the rotor thumb wheels to set the initial rotor position and then type the three letter message key to produce an encrypted message key:

```
machine.set_display('WXC')     # set initial rotor positions
enc_key = machine.process_text('BLA')      # encrypt message key
```

In this example, the encrypted key turns out to be `KCH`. This is written down for later.

The operator then sets the rotors to the unencrypted message key `BLA` and then types in the officer's message, performing various substitutions and transformations according to training and current procedures. In our simple case, he performs the following:

```
machine.set_display('BLA')     # use message key BLA
ciphertext = machine.process_text('THEXRUSSIANSXAREXCOMINGX')
print(ciphertext)
```

This produces the ciphertext `NIBLFMYMLLUFWCASCSSNVHAZ`.

Next, between the Enigma operator and the radio operator, a message is formed up. This message includes the following components:

- The time of transmission
- The station identification for transmitter and intended recipient(s)
- The message length; in our case this is 24
- The initial rotor positions in unencrypted form (`WXC`)
- The encrypted message key value (`KCH`)
- The unencrypted message indicator (*Kenngruppen*)
- The encrypted message contents

In our example, the message handed over to the radio operator to be transmitted by either Morse code or perhaps even voice would look something like this:

```
U6Z DE C 1500 = 24 = WXC KCH =

BNUGZ NIBLF MYMLL UFWCA
SCSSN VHAZ=
```

The top line indicates day 31, station C transmits to station U6Z, sent at 1500 hours and contains 24 letters. The starting position is WXC and the encrypted message key is KCH.

Next we have the body of the message. The army transmitted messages in 5 letter groups. The first group contains the Kenngruppen, or indicator. Procedure required the operator pick one of the Kenngruppen possibilities from the key sheet, and then pad it out with two random letters. Here the operator chose to prepend BN to the Kenngruppen value of UGZ. He could have also appended the two letters, or perhaps appended one and prepended the other.

After the message indicator group, the encrypted text follows in 5 letter groups.

Now at receiving station U6Z, the radio operator receives the over-the-air message and types or writes it up in the form shown and hands it to the Enigma operator.

The Enigma operator first looks for the message indicator. He uses the group BNUGZ and scans his key sheet for either BNU, NUG, or UGZ. He could presumably also use the date information found in the message preamble to help his search of the key sheet. If everything checks out the operator now knows which entry in his monthly key sheet to use. Thus, as was done at the transmitting station, he configures his Enigma according to the key sheet:

```
machine = EnigmaMachine.from_key_sheet(
       rotors='II IV V',
       reflector='B',
       ring_settings='B U L',
       plugboard_settings='AV BS CG DL FU HZ IN KM OW RX')
```

The receiving operator then must decrypt the message key:

```
machine.set_display('WXC')
msg_key = machine.process_text('KCH')
```

This should reveal that the message key is the original BLA. The rotors are then set to this value and the message can be decrypted, taking care to ignore the Kenngruppen:

```
machine.set_display(msg_key)      # original message key is BLA
plaintext = machine.process_text('NIBLFMYMLLUFWCASCSSNVHAZ')
print(plaintext)
```

The Enigma operator then decodes the message “THEXRUSSIANSXAREXCOMINGX”. He then uses his training and procedures to further process the message. Finally, the somewhat troubling message “The Russians are coming” is handed to his commanding officer.

CHAPTER 3

Reference manual

The Py-Enigma simulation is made up of several Python classes as described below.

EnigmaMachines

The `EnigmaMachine` class represents an assembled Enigma machine that consists of rotors, a plugboard, a keyboard, and indicator lamps. The keyboard and lamps act as input and outputs. The other components are represented by Python classes.

EnigmaMachine class reference

The top-level `EnigmaMachine` class represents an assembled Enigma machine. The `EnigmaMachine` class resides in the `enigma.machine` module.

class `enigma.machine.`**`EnigmaMachine`**(*rotors*, *reflector*, *plugboard*)

Top-level class that represents Enigma machines.

Parameters

- **`rotors`** – A list containing 3 or 4 (for the Kriegsmarine M4 version) *`Rotor`* objects. The order of the list is important. The first rotor is the left-most rotor, and the last rotor is the right-most (from the operator's perspective sitting at the machine).
- **`reflector`** – A *`Rotor`* object that represents the reflector (*UKW*).
- **`plugboard`** – A *`Plugboard`* object that represents the state of the plugboard (*Steckerbrett*).

classmethod `from_key_sheet`(*[rotors='I II III'[, ring_settings=None[, reflector='B'[, plugboard_settings=None]]]]*)

Convenience function to build an EnigmaMachine from the data as you might find it on a monthly key sheet (code book).

Parameters

- **rotors** – Either a list of strings naming the rotors from left to right or a single string: e.g. `["I", "III", "IV"]` or `"I III IV"`.
- **ring_settings** – Either a list/tuple of integers, a string, or `None` to represent the ring settings to be applied to the rotors in the rotors list (see below).
- **reflector** – A string that names the reflector to use.
- **plugboard_settings** – A string of plugboard settings as you might find on a key sheet (see below).

The `ring_settings` parameter can accept either:

•A list/tuple of integers with values between 0-25.

•A string; either space separated letters or numbers, e.g. `'B U L'` or `'1 20 11'`. Note that if numbers are used, they should be between 1-26 to match historical key sheet data.

•`None` means all ring settings are 0.

The `plugboard_settings` parameter can accept either:

•A string of space separated letter pairs; e.g. `'PO ML IU KJ NH YT GB VF RE DC'`.

•A string of slash separated number pairs; e.g. `'18/26 17/4 21/6 3/16 19/14 22/7 8/1 12/25 5/9 10/15'`.

•A value of `None` means no plugboard connections are made.

classmethod from_key_file(*fp*[, *day=None*])

Convenience function to build an EnigmaMachine by reading key parameters from a file.

Parameters

- **fp** – A file-like object that contains daily key settings, one day's settings per line.
- **day** – The line in the file labeled with the day number (1-31) will be used for the settings. If day is `None`, the day number will be determined from today's date.

For more information on the file format, see *Key File Format*.

set_display(*val*)

Sets the simulated rotor operator windows to *val*. This establishes a new starting position for a subsequent encrypt or decrypt operation. See also *get_display()*.

Parameters **val** – Must be a string or iterable containing uppercase letter values, one for each window from left to right. For example, a valid value for a 3 rotor machine would be `'ABC'`.

get_display(*val*)

This method returns the current position of the rotors as a string. See also *set_display()*.

Returns a string of uppercase letters, one for each rotor (left to right)

Return type string

get_rotor_count()

Returns a list of integers that represent the rotation counts for each rotor. The rotation counts are reset to 0 every time *set_display()* is called.

key_press(*key*)

Simulate a front panel key press. First the rotors are stepped by simulating the mechanical action of the machine. Next a simulated current is run through the machine. The lamp that is lit by this key press is returned as a string (a single uppercase letter A-Z).

Parameters **key** (*string*) – the letter pressed (A-Z)

Returns the lamp that is lit (A-Z)

Return type string

process_text (*text*[, *replace_char='X'*])

This is a convenience function for processing a string of text. For each character in the input text, *key_press()* is called. The output text is returned as a string.

This function performs some pre-processing of the input text, unlike *key_press()*. First, all input is converted to uppercase. Secondly, the parameter `replace_char` controls what is done to input characters that are not `A-Z`. If the input text contains a character not on the keyboard, it is replaced with `replace_char`. If `replace_char` is `None` the character is dropped from the input. `replace_char` defaults to `X`.

Parameters

- **text** (*string*) – the text to process
- **replace_char** – invalid input is replaced with this string or dropped if it is `None`

EnigmaMachine exceptions

EnigmaMachine operations may raise `enigma.machine.EnigmaError` under error conditions. The two `classmethod` constructors, *from_key_sheet* and *from_key_file* assemble an *EnigmaMachine* from parts, and those parts may raise these exceptions themselves:

- `rotor.rotors.RotorError`
- `plugboard.PlugboardError`

Rotors & Reflectors

The `Rotor` class represents the Enigma rotors, also known as the wheels or *Walzen* in German. They are the most important parts of the machine.

Rotors have little use on their own. They are placed inside an *EnigmaMachine* object, which then calls the public `Rotor` methods.

Rotor class reference

class `enigma.rotors.rotor.`**Rotor** (*model_name*, *wiring*[, *ring_setting=0*[, *stepping=None*]])

A rotor has 26 circularly arranged pins on the right (entry) side and 26 contacts on the left side. Each pin is connected to a single contact by internal wiring, thus establishing a substitution cipher. We represent this wiring by establishing a mapping from a pin to a contact (and vice versa for the return path). Internally we number the pins and contacts from 0-25 in a clockwise manner with 0 being the "top".

An alphabetic or numeric ring is fastened to the rotor by the operator. The labels of this ring are displayed to the operator through a small window on the top panel. The ring can be fixed to the rotor in one of 26 different positions; this is called the ring setting (*Ringstellung*). We will number the ring settings from 0-25 where 0 means no offset (e.g. the letter "A" is mapped to pin 0 on an alphabetic ring). A ring setting of 1 means the letter "B" is mapped to pin 0.

Each rotor can be in one of 26 positions on the spindle, with position 0 where pin/contact 0 is being indicated in the operator window. The rotor rotates towards the operator by mechanical means during normal operation as keys are being pressed during data entry. Position 1 is thus defined to be one step from position 0. Likewise, position 25 is the last position before another step returns it to position 0, completing 1 trip around the spindle.

Finally, a rotor has a "stepping" or "turnover" parameter. Physically this is implemented by putting a notch on the alphabet ring and it controls when the rotor will "kick" the rotor to its left, causing the neighbor rotor to rotate. Most rotors had one notch, but some Kriegsmarine rotors had 2 notches and thus rotated twice as fast.

Note that we allow the `stepping` parameter to be `None`. This indicates the rotor does not rotate. This allows us to model the entry wheel and reflectors as stationary rotors. The fourth rotor on the Kriegsmarine M4 models (*Beta* or *Gamma*) did not rotate.

The rotor constructor establishes the rotor characteristics.

Parameters

- **model_name** (*string*) – e.g. "I", "II", "III", "Beta", "Gamma"
- **wiring** (*string*) – This should be a string of 26 uppercase characters A-Z that represent the internal wiring transformation of the signal as it enters from the right side. This is the format used in various online resources. For example, for the Wehrmacht Enigma type I rotor the mapping is `"EKMFLGDQVZNTOWYHXUSPAIBRCJ"`.
- **ring_setting** (*integer*) – This should be an integer from 0-25, inclusive, which indicates the *Ringstellung*. A value of 0 means there is no offset; e.g. the letter `A` is fixed to pin `0`. A value of 1 means `B` is mapped to pin `0`.
- **stepping** – This is the stepping or turnover parameter. When it is an iterable, for example a string such as "Q", this indicates that when the rotor transitions from "Q" to "R" (by observing the operator window), the rotor will "kick" the rotor to its left, causing it to rotate. If the rotor has more than one notch, a string of length 2 could be used, e.g. "ZM". Another way to think of this parameter is that when a character in the stepping string is visible in the operator window, a notch is lined up with the pawl on the left side of the rotor. This will allow the pawl to push up on the rotor *and* the rotor to the left when the next key is depressed. A value of `None` means this rotor does not rotate.

Raises **RotorError** – when an invalid parameter is supplied

Note that for purposes of simulation, our rotors will always use alphabetic labels A-Z. In reality, the Heer & Luftwaffe devices used numbers 01-26, and Kriegsmarine devices used A-Z. Our usage of A-Z is simply for simulation convenience. In the future we may allow either display.

set_display(*val*)

Spin the rotor such that the string `val` appears in the operator window. This sets the internal position of the rotor on the axle and thus rotates the pins and contacts accordingly.

A value of 'A' for example puts the rotor in position 0, assuming an internal ring setting of 0.

Parameters **val** (*string*) – rotor position which must be in the range `A-Z`

Raises **RotorError** – when an invalid position value is supplied

get_display()

Returns current rotor position in the range `A-Z`

Return type string

signal_in(*n*)

Simulate a signal entering the rotor from the right at a given pin position n.

Parameters **n** (*integer*) – pin number between 0 and 25

Returns the contact number of the output signal (0-25)

Return type integer

signal_out(*n*)
Simulate a signal entering the rotor from the left at a given contact position n.

Parameters **n** (*integer*) – contact number between 0 and 25

Returns the pin number of the output signal (0-25)

Return type integer

notch_over_pawl()
Returns `True` if this rotor has a notch in the stepping position and `False` otherwise.

Return type Boolean

rotate()
Rotates the rotor forward.

A note on the entry wheel and reflectors

The entry wheel (*ETW*) is a special non-movable rotor that sits on the far right of the rotor array. It connects the rotor array with the plugboard wiring. On Wehrmacht Enigmas, the entry wheel performs a straight-through mapping. In other words, the wire from the 'A' key is passed to pin position 0, 'B' to pin position 1, etc. Thus there is no need to simulate the entry wheel given our current scope to model only military Enigmas.

The reflector, or *Umkehrwalze* (UKW), sits at the far left of the rotor array. It simply reflects the incoming signal coming from the right back through the left side of the rotors. We can thus model the reflector as a special non-movable rotor.

If you decide to create your own reflector, and you desire to maintain reciprocal encryption & decryption, your connections must be made in pairs. Thus if you wire 'A' to 'G', you must also wire 'G' to 'A', and so on.

Rotor & reflector factory functions

While it is possible to create your own rotor type, for convenience two factory functions have been created to return rotors and reflectors used by the Wehrmacht. These factory functions let you refer to the rotors and reflectors by name instead of providing their internal wiring every time you need one (which would be both tedious and error prone).

The following table lists the names of the rotors we currently simulate.

Table 3.1: Simulated rotor models

Rotor names	Enigma Models
I, II, III, IV, V	All Wehrmacht models
VI, VII, VIII	Kriegsmarine M3 & M4
Beta, Gamma	Kriegsmarine M4 (with thin reflectors)

Any of the names in the first column of the above table can be used by the factory function *enigma.rotors.factory.create_rotor()*, described below.

Likewise there exists a factory function to create reflectors by name. The following table lists the names of the supported reflectors.

Table 3.2: Simulated reflector types

Reflector names	Enigma Models
B, C	All Wehrmacht models
B-Thin, C-Thin	Kriegsmarine M4 (with Beta & Gamma rotors)

The two factory functions are described next:

enigma.rotors.factory.**create_rotor**(*model*[, *ring_setting=0*])
Create and return a *Rotor* object with the given ring setting.

Parameters

- **model** (*string*) – the model name to create; see the *Simulated rotor models* table
- **ring_setting** (*integer*) – the ring setting (0-25) to use

Returns the newly created *Rotor*

Raises **RotorError** – when an unknown model name is provided

enigma.rotors.factory.**create_reflector**(*model*)
Create and return a *Rotor* object that is meant to be used in the reflector role.

Parameters **model** (*string*) – the model name to create; see the *Simulated reflector types* table

Returns the newly created reflector, which is actually of type *Rotor*

Raises **RotorError** – when an unknown model name is provided

Rotor exceptions

Rotor objects may raise `enigma.rotors.RotorError` when an invalid constructor argument is given, or if the rotor object is given an invalid parameter during a *set_display* operation.

Plugboards

The plugboard, or *Steckerbrett* in German, allows the operator to swap up to 10 keys and indicator lamps for increased key strength.

Plugboards have little use on their own. They are placed inside an *EnigmaMachine* object, which then calls the public `Plugboard` methods.

Plugboard class reference

class enigma.plugboard.**Plugboard**([*wiring_pairs=None*])
The plugboard allows the operator to swap letters before and after the entry wheel. This is accomplished by connecting cables between pairs of plugs that are marked with letters (Heer & Luftwaffe models) or numbers (Kriegsmarine). Ten cables were issued with each machine; thus up to 10 of these swappings could be used as part of a machine setup.

Each cable swaps both the input and output signals. Thus if A is connected to B, A crosses to B in the keyboard to entry wheel direction and also in the reverse entry wheel to lamp direction.

The constructor configures the plugboard according to a list or tuple of integer pairs, or None.

Parameters **wiring_pairs** – A value of `None` or an empty list/tuple indicates no plugboard connections are to be used (i.e. a straight mapping). Otherwise `wiring_pairs` must be an iterable of integer pairs, where each integer is between 0-25, inclusive. At most 10 such pairs can be specified. Each value represents an input/output path through the plugboard. It is invalid to specify the same path more than once in the list.

Raises **PlugboardError** – If an invalid `wiring_pairs` parameter is given.

classmethod **from_key_sheet**(*[settings=None]*)

This is a convenience function to build a plugboard according to a settings string as you may find on a key sheet.

Two syntaxes are supported, the Heer/Luftwaffe and Kriegsmarine styles:

In the Heer syntax, the settings are given as a string of alphabetic pairs. For example: `'PO ML IU KJ NH YT GB VF RE DC'`.

In the Kriegsmarine syntax, the settings are given as a string of number pairs, separated by a '/'. Note that the numbering uses 1-26, inclusive. For example: `'18/26 17/4 21/6 3/16 19/14 22/7 8/1 12/25 5/9 10/15'`.

To specify no plugboard connections, settings can be `None` or an empty string.

Parameters **settings** – A settings string as described above, or `None`.

Raises **PlugboardError** – If the settings string is invalid, or if it contains more than 10 pairs. Each plug should be present at most once in the settings string.

signal(*n*)

Simulate a signal entering the plugboard on wire n, where n must be an integer between 0 and 25.

Parameters **n** (*integer*) – The wire number the input signal is on (0-25).

Returns The wire number of the output signal (0-25).

Return type integer

Note that since the plugboard always crosses pairs of wires, it doesn't matter what direction (keyboard -> entry wheel or vice versa) the signal is coming from.

Plugboard exceptions

Plugboard objects may raise `enigma.plugboard.PlugboardError` when an invalid constructor argument is given.

CHAPTER 4

pyenigma command-line application

Py-Enigma includes a simple application, *pyenigma.py*, to let you perform Enigma text transformations on the command-line. This allows for quick experimentation and scripting of operations.

Getting help

To get help and see all the available options, invoke pyenigma.py with the `--help` option:

```
$ python pyenigma.py --help

usage: pyenigma.py [-h] [-k KEY_FILE] [-d DAY] [-r ROTOR [ROTOR ...]]
                   [-i RING_SETTING [RING_SETTING ...]]
                   [-p PLUGBOARD [PLUGBOARD ...]] [-u REFLECTOR] [-s START]
                   [-t TEXT] [-f FILE] [-x REPLACE_CHAR] [-z] [-v]

Encrypt/decrypt text according to Enigma machine key settings

optional arguments:
  -h, --help            show this help message and exit
  -k KEY_FILE, --key-file KEY_FILE
                        path to key file for daily settings
  -d DAY, --day DAY     use the settings for day DAY when reading key file
  -r ROTOR [ROTOR ...], --rotors ROTOR [ROTOR ...]
                        rotor list ordered from left to right; e.g III IV I
  -i RING_SETTING [RING_SETTING ...], --ring-settings RING_SETTING [RING_SETTING ...]
                        ring setting list from left to right; e.g. A A J
  -p PLUGBOARD [PLUGBOARD ...], --plugboard PLUGBOARD [PLUGBOARD ...]
                        plugboard settings
  -u REFLECTOR, --reflector REFLECTOR
                        reflector name
  -s START, --start START
                        starting position
  -t TEXT, --text TEXT  text to process
  -f FILE, --file FILE  input file to process
```

```
  -x REPLACE_CHAR, --replace-char REPLACE_CHAR
                        if the input text contains chars not found on the
                        enigma keyboard, replace with this char [default: X]
  -z, --delete-chars    if the input text contains chars not found on the
                        enigma keyboard, delete them from the input
  -v, --verbose         provide verbose output; include final rotor positions

Key settings can either be specified by command-line arguments, or read
from a key file. If reading from a key file, the line labeled with the
current day number is used unless the --day argument is provided.

Text to process can be supplied 3 ways:

   if --text=TEXT is present TEXT is processed
   if --file=FILE is present the contents of FILE are processed
   otherwise the text is read from standard input

Examples:

    $ pyenigma.py --key-file=enigma.keys -s XYZ -t HELLOXWORLDX
    $ pyenigma.py -r III IV V -i 1 2 3 -p AB CD EF GH IJ KL MN -u B -s XYZ
    $ pyenigma.py -r Beta III IV V -i A B C D -p 1/2 3/4 5/6 -u B-Thin -s WXYZ
```

There are numerous options, but most are hopefully self-explanatory. There are two ways to invoke *pyenigma.py*:

1. Explicitly specifying all initial key settings
2. Using a key file to initialize the Enigma machine

Specifying all key settings

Here are some examples of specifying all the key settings on the command-line:

```
$ python pyenigma.py --rotors I IV V --ring-settings 5 17 8 \
  --plugboard AV BS CG DL FU HZ IN KM OW RX --reflector C \
  --start=DRX

$ python pyenigma.py -r I IV V -i 5 17 8 \
  -p AV BS CG DL FU HZ IN KM OW RX -u C -s DRX
```

These two invocations create the same settings, the first uses long form option names, while the second uses short form.

If no `--text` or `--file` options are provided, *pyenigma.py* will prompt for input:

```
$ python pyenigma.py -r I IV V -i 5 17 8 -p AV BS CG DL FU HZ IN KM OW RX -u C -s DRX
--> THIS IS MY SECRET MESSAGE
QAWYWZBVCDEZWOHPVCKFMMFLY
```

Using a key file for settings

It is often unwieldy to type so many options on the command-line, so *pyenigma.py* provides a way to store key settings in a simulated key sheet file:

```
$ python pyenigma.py --key-file keyfile --start='AAB' --day=29 --text='HERE IS MY␣
↪MESSAGE'
OCJNFADTCMQIBJLYWW
```

If the `--day` option is omitted, the day is determined from the current date.

The format of the key sheet file is described in *Key file format*.

Verbose output

The `--verbose` or `-v` option is useful if you wish to view the final rotor positions and view how many times the rotors stepped while processing your text:

```
$ python pyenigma.py --key-file keyfile --start='XHC' --day=29 --file msg.txt --
↪verbose
Final rotor positions: YXY
Rotor rotation counts: [1, 16, 412]
Output:
TOSCKAVFTVPONPBJZQPZFBFJXNMCLCZEVDHNEGNPGBWTYTRXJUVOKWBCBFVXIMURRDWNQTHEWTBHMPLKLPLVSJLNLNUOZDCSWAOY(
```

CHAPTER 5

Key file format

Specifying key settings can become tedious and error-prone, so Py-Enigma allows you to store key settings in a simulated monthly key sheet file. This is a simple text file that you can create with your favorite text editor. Each line of this file represents one days settings. Within the line, whitespace separates each item. The columns for each line are as follows:

1. The first column is the day number for the setting, similar to a real key sheet. The day should be an integer in the range 1-31.
2. The next 3 or 4 columns are rotor names. See *Simulated rotor models* for a list of valid rotor names.
3. The next 3 or 4 columns are the ring settings for each rotor. These can be a list of numbers (1-26) or letters (A-Z).
4. The next 10 columns are the plugboard settings. These can be a list of 2-letter pairs (e.g. AB CD, etc.) or slash separated number pairs (e.g. 1/20 3/22).
5. The last column is the reflector name. See *Simulated reflector types* for a list of the valid reflector names.

Please note the following about the file format:

- Each line must have either 18 or 20 columns, depending on if you are simulating a 3 or 4 rotor Enigma.
- It is possible to mix 3 and 4 rotor settings in the same file.
- You do not have to supply settings for every day in the month.
- Py-Enigma will simply scan the file from top to bottom until it finds the line that corresponds to the day number it is looking for. Thus duplicate day settings are allowed, but keep in mind the first line will be used.
- The file can contain blank lines.
- The file can contain comment lines. Comment lines begin with a # character in the first column and extend to the end of the line.

Example file:

```
# My sample settings file
29 II IV V 1 16 10 AV BS CG DL FU HZ IN KM OW RX B
30 Beta II IV I A A A V 1/20 2/12 4/6 7/10 8/13 14/23 15/16 17/25 18/26 22/24 B-Thin
```

CHAPTER 6

Indices and tables

- genindex
- search

Index