crypto-enigma Documentation

Release 0.2.1

Roy Levien

December 22, 2015

Contents

Note: This documentation is in draft form. Reports of any errors or suggestions for improvement are welcomed and should be submitted as new issues.

An Enigma machine simulator with rich textual display functionality for Python 2.7.

Currently support is only provided for those machine models in most widespread general use during the war years: the I, M3, and M4.

No attempt is made here to describe the operation of an Enigma machine. For information about how an Enigma machine works see the excellent and extensive resources available at the Crypto Museum.

CHAPTER 1

Functionality and use

The package provides functionality for *generating a machine configuration* from a conventional specification, *examining the state* of a configuration, simulating the *operation* of a machine by stepping between states, and *encoding messages*:

Create a machine configuration (see the *config_enigma_from_string*):

```
>>> from crypto_enigma import *
>>> cfg = EnigmaConfig.config_enigma_from_string(u'B-I-III-I EMO UX.MO.AY 13.04.11')
```

Encode messages (see the *enigma_encoding*):

```
>>> cfg.enigma_encoding(u'TESTINGXTESTINGUD')
u'OZQKPFLPYZRPYTFVU'

>>> cfg.enigma_encoding(u'OZQKPFLPYZRPYTFVU')
u'TESTINGXTESTINGUD'
```

Show configuration details (see the *config_string*):

```
>>> print(cfg.config_string(letter=u'X', format='internal', mark_func=lambda c: '(' + c + ')'))
X > ABCDEFGHIJKLMNOPQRSTUVW(X)YZ
  P YBCDEFGHIJKLONMPQRSTXVW(U)AZ          UX.MO.AY
  1 HCZMRVJPKSUDTQOLWEXN(Y)FAGIB  O  05   I
  2 KOMQEPVZNXRBDLJHFSUWYACT(G)I  M  10   III
  3 AXIQJZ(K)RMSUNTOLYDHVBWEGPFC  E  19   I
  R YRUHQSLDPX(N)GOKMIEBFZCWVJAT          B
  3 ATZQVYWRCEGOI(L)NXDHJMKSUBPF          I
  2 VLWMEQYPZOA(N)CIBFDKRXSGTJUH          III
  1 WZBLRVXAYGIPD(T)OHNEJMKFQSUC          I
  P YBCDEFGHIJKLONMPQRS(T)XVWUAZ          UX.MO.AY
T < CNAUJVQSLEMIKBZRGPHXDFY(T)WO
```

Simulate machine operation (see the *print_operation*):

```
>>> cfg.print_operation(message=u'TESTING', show_step=True, mark_func=lambda c: '(' + c + ')')
0000        CNAUJVQSLEMIKBZRGPHXDFYTWO    EMO  19 10 05
0001  T > UNXKGVERLYDIQBTWMHZ(O)AFPCJS    EMP  19 10 06
0002  E > QTYJ(Z)XUPKDIMLSWHAVNBGROFCE    EMQ  19 10 07
0003  S > DMXAPTRWKYINBLUESG(Q)FOZHCJV    ENR  19 11 08
0004  T > IUSMHRPEAQTVDYWGJFC(K)BLOZNX    ENS  19 11 09
0005  I > WMVXQRLS(P)YOGBTKIEFHNZCADJU    ENT  19 11 10
0006  N > WKIQXNRSCVBOY(F)LUDGHZPJAEMT    ENU  19 11 11
0007  G > RVPTWS(L)KYXHGNMQCOAFDZBEJIU    ENV  19 11 12
```

Watch the machine as it runs for 500 steps:

```
>>> cfg.print_operation(steps=500, show_step=True, format='internal', overwrite=True)
```

CHAPTER 2

Command line functionality

A command line script, `enigma.py`, provides access to almost all the functionality of the API.

Encode messages:

```
$ enigma.py encode ``B-I-III-I EMO UX.MO.AY 13.04.11'' ``TESTINGXTESTINGUD''
OZQKPFLPYZRPYTFVU

$ enigma.py encode ``B-I-III-I EMO UX.MO.AY 13.04.11'' ``OZQKPFLPYZRPYTFVU''
TESTINGXTESTINGUD
```

Show configuration details (explained in more detail in the command line help):

```
$ enigma.py show ``B-I-III-I EMO UX.MO.AY 13.04.11'' -l `X' -H'()' -f internal
X > ABCDEFGHIJKLMNOPQRSTUVW(X)YZ
  P YBCDEFGHIJKLONMPQRSTXVW(U)AZ          UX.MO.AY
  1 HCZMRVJPKSUDTQOLWEXN(Y)FAGIB  O  05  I
  2 KOMQEPVZNXRBDLJHFSUWYACT(G)I  M  10  III
  3 AXIQJZ(K)RMSUNTOLYDHVBWEGPFC  E  19  I
  R YRUHQSLDPX(N)GOKMIEBFZCWVJAT          B
  3 ATZQVYWRCEGOI(L)NXDHJMKSUBPF          I
  2 VLWMEQYPZOA(N)CIBFDKRXSGTJUH          III
  1 WZBLRVXAYGIPD(T)OHNEJMKFQSUC          I
  P YBCDEFGHIJKLONMPQRS(T)XVWUAZ          UX.MO.AY
T < CNAUJVQSLEMIKBZRGPHXDFY(T)WO
```

Simulate machine operation (explained in more detail command line help):

```
$ enigma.py run ``B-I-III-I EMO UX.MO.AY 13.04.11'' -m ``TESTING'' -t -H'()'
0000       CNAUJVQSLEMIKBZRGPHXDFYTWO   EMO  19 10 05
0001  T > UNXKGVERLYDIQBTWMHZ(O)AFPCJS  EMP  19 10 06
0002  E > QTYJ(Z)XUPKDIMLSWHAVNBGROFCE  EMQ  19 10 07
0003  S > DMXAPTRWKYINBLUESG(Q)FOZHCJV  ENR  19 11 08
0004  T > IUSMHRPEAQTVDYWGJFC(K)BLOZNX  ENS  19 11 09
0005  I > WMVXQRLS(P)YOGBTKIEFHNZCADJU  ENT  19 11 10
0006  N > WKIQXNRSCVBOY(F)LUDGHZPJAEMT  ENU  19 11 11
0007  G > RVPTWS(L)KYXHGNMQCOAFDZBEJIU  ENV  19 11 12
```

Watch the machine as it runs for 500 steps:

```
$ enigma.py run  ``c-β-VIII-VII-VI QMLI `UX.MO.AY 01.13.04.11'' -s 500 -t -f internal -o
```

CHAPTER 3

Limitations

Note that the correct display of some characters used to represent components (thin Naval rotors) assumes support for Unicode, while some aspects of the display of machine state depend on support for combining Unicode. This is a known limitation that will be addressed in a future release.

Note also that at the start of any scripts that use this package, you should

```
from __future__ import unicode_literals
```

before any code that uses the API, or confiure IPython (in `ipython_config.py`) with

```
c.InteractiveShellApp.exec_lines += ["from __future__ import unicode_literals"]
```

or explicitly supply Unicode strings (e.g., as in many of the examples here with `u'TESTING'`).

CHAPTER 4

Sitemap

Note: This documentation is in draft form. Reports of any errors or suggestions for improvement are welcomed and should be submitted as new issues.

4.1 Machine - `crypto_enigma.machine`

This module supports all of the functionality of an Enigma machine using an *EnigmaConfig* class and several utility functions, which support examination of its state (including the details of the cyphers used for encoding messages), stepping during operation, and encoding messages.

4.1.1 Overview

EnigmaConfig	An Enigma machine configuration.
config_enigma	Create an *EnigmaConfig* from strings specifying its state.
config_enigma_from_string	Create an *EnigmaConfig* from a single string specifying its state.
windows	The letters at the windows of an Enigma machine.
components	The identities of the components in the Enigma machine.
positions	The rotational positions of the components in the Enigma machine.
rings	The ring settings in the Enigma machine.
stage_mapping_list	The list of mappings for each stage of an Enigma machine.
enigma_mapping_list	The list of progressive mappings of an Enigma machine at each stage.
enigma_mapping	The mapping used by an Enigma machine for encoding.
config_string	A string representing a schematic of an Enigma machine's state.
step	Step the Enigma machine to a new machine configuration.
stepped_configs	Generate a series of stepped Enigma machine configurations.
print_operation	Show the operation of the Enigma machine as a series of configurations.
enigma_encoding	Encode a message using the machine configuration.
print_encoding	Show the conventionally formatted encoding of a message.

4.1.2 Machine configurations

Enigma machine configurations and their functionality are represented using single class:

class `crypto_enigma.machine.`**`EnigmaConfig`**
An Enigma machine configuration.

A class representing the state of an Enigma machine, providing functionality for

- *generating a machine configuration* from a conventional specification,
- *examining the state* of a configuration,
- simulating the *operation* of a machine by stepping between states, and
- *encoding messages*.

4.1.3 Creating configurations

static `EnigmaConfig.`**`config_enigma`**`(*args, **kwargs)`

Create an *`EnigmaConfig`* from strings specifying its state.

A (safe public, "smart") constructor that does validation and takes a conventional specification as input, in the form of four strings.

Following convention, the elements of these specifications are in physical machine order as the operator sees them, which is the reverse of the order in which they are encountered in processing.

Validation is permissive, allowing for ahistorical collections and numbers of rotors (including reflectors at the rotor stage, and trivial degenerate machines; e.g., `config_enigma("-", "A", "", "01")`, and any number of (non-contradictory) plugboard wirings (including none).

Parameters

- **`rotor_names`** (*unicode*) – The *Walzenlage*: The conventional letter or Roman numeral designations (its *`name`*) of the rotors, including reflector, separated by dashes (e.g. `'b-β-V-I-II'`). (See *`components`*.)
- **`window_letters`** (*unicode*) – The *Walzenstellung* (or, incorrectly, the *Grundstellung*): The letters visible at the windows (e.g. `'MQR'`). (See *`windows`*.)
- **`plugs`** (*unicode*) – The *Steckerverbindungen*: The plugboard specification (its *`name`*) as a conventional string of letter pairs separated by periods, (e.g., `'AU.ZM.ZL.RQ'`). (See *`components`*.)
- **`rings`** (*unicode*) – The *Ringstellung*: The location of the letter ring on each rotor (specifcially, the number on the rotor under ring letter **A**), separated by periods (e.g. `'22.11.16'`). (See *`rings`*.)

Returns A new Enigma machine configuration created from the specification arguments.

Return type *EnigmaConfig*

Raises `EnigmaValueError` – Raised when arguments do not pass validation.

Example

```
>>> cfg = EnigmaConfig.config_enigma("c-β-V-III-II", "LQVI", "AM.EU.ZL", "16.01.21.11")
```

static `EnigmaConfig.`**`config_enigma_from_string`**`(*args, **kwargs)`

Create an *`EnigmaConfig`* from a single string specifying its state.

Parameters **`string`** (*unicode*) – The elements of a conventional specification (as supplied to *`config_enigma`*) joined by spaces into a single string.

Returns A new Enigma machine configuration created from the specification argument.

Return type *EnigmaConfig*

Raises `EnigmaValueError` – Raised when argument does not pass validation.

Example

This is just a shortcut for invoking *config_enigma* using a sigle string:

```
>>> cfg_str = "c-β-V-III-II LQVI AM.EU.ZL 16.01.21.11"
>>> EnigmaConfig.config_enigma_from_string(cfg_str) == EnigmaConfig.config_enigma(*cfg_str.split
True
```

Note that the `string` argument corresponds to the string representation of an *EnigmaConfig*

```
>>> print(EnigmaConfig.config_enigma_from_string(cfg_str))
c-β-V-III-II LQVI AM.EU.ZL 16.01.21.11
```

so that this method is useful for instantiation of an *EnigmaConfig* from such strings (e.g., in files):

```
>>> unicode(EnigmaConfig.config_enigma_from_string(cfg_str)) == unicode(cfg_str)
True
```

4.1.4 State

The behavior of an Enigma machine, for both its *operation* and the *encodings* it performs is determined entirely by its state. This state is established when a machine is set to its initial configuration. Operation then produces a series of configurations each with new state

Formally, that state consists of *internal* elements not directly visible to the operator who can only indirectly *see changes* in the positions of the rotors as manifest in the rotor letters at the machine windows. This internal state is entirely responsible for determining the *mappings used by the machine* to encode messages.

Thes aspects of state can be used to costruct a varity of *representations* of the configuration of an Enigma machine.

Visible state

`EnigmaConfig.`**`windows`**`()`

The letters at the windows of an Enigma machine.

This is the (only) visible manifestation of configuration changes during *operation*.

Returns The letters at the windows in an *EnigmaConfig*, in physical, conventional order.

Return type unicode

Example

Using `cfg` as defined *above*:

```
>>> cfg.windows()
u'LQVI'
```

Internal state

The core properties of an *EnigmaConfig* embody a low level specification of an Enigma configuration.

The conventional historical specification of an Enigma machine (as used in *config_enigma*) includes redundant elements, and conceals properties that are directly relevant to the operation of the machine and the encoding it performs — notably the actual rotational positions of the components.

A complete "low level" formal characterization of the state of an Enigma machine consists of three elements: lists of *components*, their *positions*, and the settings of their *rings*. Here these lists are in *processing order* — as opposed to the physical order used in conventional specifications — and the have positions and ring settings generalized and "padded" for consistency to include the plugboard and reflector.

Note that though it is not likely to be useful, these elements can be used to instantiate an *EnigmaConfig*:

```
>>> cfg_conv = EnigmaConfig.config_enigma("B-I-II-III", "ABC", "XO.YM.QL", "01.02.03")
>>> cfg_intl = EnigmaConfig(cfg_conv.components, cfg_conv.positions, cfg_conv.rings)
>>> cfg_conv == cfg_intl
True
```

They may also be useful in extending the functionality provided here, for example in constructing additional representations of configurations beyond those provided in `config_string`:

```
>>> [b'{} {}'.format(c, p) for c, p in zip(cfg_intl.components, cfg_intl.positions)[1:]]
['III 1', 'II 1', 'I 1', 'B 1']
```

`EnigmaConfig.`**`components`**

The identities of the components in the Enigma machine.

For rotors (including the reflector) these correspond to the the `rotor_names` supplied to *config_enigma*, while for the plugboard this is just the `plugs` argument.

Returns The *name* of each *Component* in an *EnigmaConfig*, in processing order.

Return type tuple

Example

Using `cfg` as defined *above*:

```
>>> cfg.components
(u'AM.EU.ZL', u'II', u'III', u'V', u'β', u'c')
```

`EnigmaConfig.`**`positions`**

The rotational positions of the components in the Enigma machine.

For rotors, this is to the number on the rotor (not letter ring) that is at the "window position", and is computed from the `window_letters` and `rings` parameters for *config_enigma*.

This (alone) determines permutations applied to components' *wiring* to produce the *mapping* for a configuration and thus the *message encoding* it performs.

Note that this is the only property of an enigma machine that changes when it is stepped (see *step*), and the changes in the letters visible at the *windows* are the (only) visible manifestation of this change.

Returns The generalized rotational position of each of the components in an *EnigmaConfig*, in machine processing order.

Return type tuple

Example

Using `cfg` as defined *above*:

```
>>> cfg.positions
(1, 25, 2, 17, 23, 1)
```

Note that for the plugboard and reflector, the position will always be **1** since the former cannot rotate, and the latter does not (neither will be different in a new configuration generated by *step*):

```
cfg.positions[0] == 1
cfg.positions[-1] == 1
```

EnigmaConfig.**rings**

The ring settings in the Enigma machine.

For rotors, these are the `rings` parameter for *config_enigma*.

Returns The generalized location of ring letter **A** on the rotor for each of the *components* in an *EnigmaConfig*, in machine processing order.

Return type tuple

Example

Using `cfg` as defined *above*:

```
>>> cfg.rings
(1, 11, 21, 1, 16, 1)
```

Note that for the plugboard and reflector, this will always be **1** since the former lacks a ring, and for latter ring position is irrelevant (the letter ring is not visible, and has no effect on when turnovers occur):

```
cfg.rings[0] == 1
cfg.rings[-1] == 1
```

Mappings

The Enigma machine's state determines the *mappings* it uses to perform *encodings*. Thes mappings can be examined in a number of ways:

EnigmaConfig.**stage_mapping_list**(**args*, ***kwargs*)

The list of mappings for each stage of an Enigma machine.

The list of *mappings* for each stage of in an *EnigmaConfig*: The encoding performed by the *Component* *at that point* in the progress through the machine.

These are arranged in processing order, beginning with the encoding performed by the plugboard, followed by the forward (see `Direction`) encoding performed by each rotor (see *mapping*), then the reflector, followed by the reverse encodings by each rotor, and finally by the plugboard again.

Returns A list of *mappings* preformed by the corresponding stage of the *EnigmaConfig* (see *mapping*).

Return type list of Mapping

Examples

This can be used to obtain lists of mappings for analysis:

```
>>> cfg = EnigmaConfig.config_enigma("b-γ-VII-V-IV", "VBOA", "NZ.AY.FG.UX.MO.PL", "05.16.11.21")
>>> cfg.stage_mapping_list()
[u'YBCDEGFHIJKPOZMLQRSTXVWUAN', u'DUSKOCLBRFHZNAEXWGQVYMIPJT', ...]
```

or more clearly

```
>>> for m in cfg.stage_mapping_list():
...     print(m)
YBCDEGFHIJKPOZMLQRSTXVWUAN
DUSKOCLBRFHZNAEXWGQVYMIPJT
CEPUQLOZJDHTWSIFMKBAYGRVXN
PCITOWJZDSYERHBNXVUFQLAMGK
UZYIGEPSMOBXTJWDNAQVKCRHLF
ENKQAUYWJICOPBLMDXZVFTHRGS
RKVPFZEXDNUYIQJGSWHMATOLCB
WOBILTYNCGZVXPEAUMJDSRFQKH
TSAJBPVKOIRFQZGCEWNLDXMYUH
NHFAOJRKWYDGVMEXSICZBTQPUL
YBCDEGFHIJKPOZMLQRSTXVWUAN
```

This list is a core part of the "internal" view of machine stage prduced by *config_string* (compare the second through the next-to-last lines with the above):

```
>>> print(cfg.config_string(format='internal'))
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
  P YBCDEGFHIJKPOZMLQRSTXVWUAN          NZ.AY.FG.UX.MO.PL
  1 DUSKOCLBRFHZNAEXWGQVYMIPJT  A   07  IV
  2 CEPUQLOZJDHTWSIFMKBAYGRVXN  O   05  V
  3 PCITOWJZDSYERHBNXVUFQLAMGK  B   13  VII
  4 UZYIGEPSMOBXTJWDNAQVKCRHLF  V   18  γ
  R ENKQAUYWJICOPBLMDXZVFTHRGS          b
  4 RKVPFZEXDNUYIQJGSWHMATOLCB          γ
  3 WOBILTYNCGZVXPEAUMJDSRFQKH          VII
  2 TSAJBPVKOIRFQZGCEWNLDXMYUH          V
  1 NHFAOJRKWYDGVMEXSICZBTQPUL          IV
  P YBCDEGFHIJKPOZMLQRSTXVWUAN          NZ.AY.FG.UX.MO.PL
    XZJVGSEMTCYUHWQROPFILDNAKB
```

Note that, because plugboard mapping is established by paired exchanges of letters it is always the case that:

```
>>> cfg.stage_mapping_list()[0] == cfg.stage_mapping_list()[-1]
True
```

EnigmaConfig.**enigma_mapping_list**(**args*, ***kwargs*)

The list of progressive mappings of an Enigma machine at each stage.

The list of *mappings* an *EnigmaConfig* has performed by each stage: The encoding performed by the *EnigmaConfig* as a whole *up to that point* in the progress through the machine.

These are arranged in processing order, beginning with the encoding performed by the plugboard, followed by the forward (see `Direction`) encoding performed up to each rotor (see *mapping*), then the reflector, followed by the reverse encodings up to each rotor, and finally by the plugboard again.

Returns A list of *mappings* preformed by the *EnigmaConfig* up to the corresponding stage of the *EnigmaConfig* (see *mapping*).

Return type list of Mapping

Examples

This can be used to obtain lists of mappings for analysis:

```
>>> cfg = EnigmaConfig.config_enigma("b-γ-VII-V-IV", "VBOA", "NZ.AY.FG.UX.MO.PL", "05.16.11.21")
>>> cfg.enigma_mapping_list()
[u'YBCDEGFHIJKPOZMLQRSTXVWUAN', u'JUSKOLCBRFHXETNZWGQVPMIYDA', ...]
```

or more clearly

```
>>> for m in cfg.enigma_mapping_list():
...     print(m)
YBCDEGFHIJKPOZMLQRSTXVWUAN
JUSKOLCBRFHXETNZWGQVPMIYDA
DYBHITPEKLZVQASNROMGFWJXUC
TGCZDFNOYEKLXPUHVBRJWASMQI
VPYFIEJWLGBXHDKSCZAORUQTNM
TMGUJAIHOYNRWQCZKSELXFDVBP
MIEANRDXJCQWOSVBUHFYLZPTKG
XCLWPMIQGBUFEJROSNTKVHADZY
YAFMCQOEVSDPBIWGNZLRXKTJHU
UNJVFSEOTCAXHWQRMLGIPDZYKB
XZJVGSEMTCYUHWQROPFILDNAKB
```

Since these may be thought of as cumulative encodings by the machine, the final element of the list will be the mapping used by the machine for encoding:

```
>>> cfg.enigma_mapping() == cfg.enigma_mapping_list()[-1]
True
```

EnigmaConfig.**enigma_mapping**()

The mapping used by an Enigma machine for encoding.

The *mapping* used by an *EnigmaConfig* to encode a letter entered at the keyboard.

Returns The *mapping* used by the *EnigmaConfig* encode a single character.

Return type *Mapping*

Examples

This is the final element in the corresponding *enigma_mapping_list*:

```
>>> cfg.enigma_mapping()
u'XZJVGSEMTCYUHWQROPFILDNAKB'
```

State representations

EnigmaConfig.**config_string**(**args*, ***kwargs*)

A string representing a schematic of an Enigma machine's state.

A string representing the stat of an *EnigmaConfig* in a selected format (see examples), optionally indicating how specified character is encoded by the configuration.

Parameters

- **letter** (*unicode, optional*) – A character to indicate the encoding of by the *EnigmaConfig*.

- **format** (*str, optional*) – A string specifying the format used to display the *EnigmaConfig*.
- **show_encoding** (*bool, optional*) – Whether to indicate the encoding for formats that do not include it by default.
- **mark_func** (*function, optional*) – A `function` that highlights its argument by taking a single character as an argument and returning a string with additional characters added to (usually surrounding) that charater. Used in cases where default method of highlighting the encoded-to character (see *Mapping*) does not display correctly or clearly.

Returns A string schematically representing an *EnigmaConfig*

Return type str

Examples

A variety of formats are available for representing the state of the Enigma machine:

```
>>> cfg = EnigmaConfig.config_enigma("b-γ-V-VIII-II", "LFAQ", "UX.MO.KZ.AY.EF.PL",u"03.17.04.11"
>>> print(cfg.config_string(format='single'))
    CMAWFEKLNVGHBIUYTXZQOJDRPS  LFAQ  10 16 24 07
>>> print(cfg.config_string(format='internal'))
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
  P YBCDFEGHIJZPONMLQRSTXVWUAK         UX.MO.KZ.AY.EF.PL
  1 LORVFBQNGWKATHJSZPIYUDXEMC  Q  07  II
  2 BJYINTKWOARFEMVSGCUDPHZQLX  A  24  VIII
  3 ILHXUBZQPNVGKMCRTEJFADOYSW  F  16  V
  4 YDSKZPTNCHGQOMXAUWJFBRELVI  L  10  γ
  R ENKQAUYWJICOPBLMDXZVFTHRGS         b
  4 PUIBWTKJZSDXNHMFLVCGQYROAE         γ
  3 UFOVRTLCASMBNJWIHPYQEKZDXG         V
  2 JARTMLQVDBGYNEIUXKPFSOHZCW         VIII
  1 LFZVXEINSOKAYHBRGCPMUDJWTQ         II
  P YBCDFEGHIJZPONMLQRSTXVWUAK         UX.MO.KZ.AY.EF.PL
    CMAWFEKLNVGHBIUYTXZQOJDRPS
>>> print(cfg.config_string(format='windows'))
LFAQ
>>> print(cfg.config_string(format='config'))
b-γ-V-VIII-II LFAQ UX.MO.KZ.AY.EF.PL 03.17.04.11
>>> print(cfg.config_string(format='encoding', letter='K'))
K > G
```

Use `format='single'` or omit the argument to display a summary of the Enigma machine configuration as its *Mapping* (see *enigma_mapping*), the letters at the *windows*, and the *positions* of the rotors. If a valid message character is provided as a value for `letter`, that is indicated as input and the letter it is encoded to is highlighted.

For example,

```
>>> print(cfg.config_string(letter='K'))
K > CMAWFEKLNVGHBIUYTXZQOJDRPS  LFAQ  10 16 24 07
```

shows the process of encoding of the letter **K** to **G**.

The default method of highlighting the encoded-to character (see *Mapping*) may not display correctly on all systems, so the `marc_func` argument can be used to define a simpler marking that does:

```
>>> print(cfg.config_string(letter='K', mark_func=lambda c: '[' + c + ']'))
K > CMAWFEKLNV[G]HBIUYTXZQOJDRPS  LFAQ  10 16 24 07
>>> print(cfg.config_string(letter='K', mark_func=lambda c: '(' + c + ')'))
K > CMAWFEKLNV(G)HBIUYTXZQOJDRPS  LFAQ  10 16 24 07
```

Use `format='internal'` to display a summary of the Enigma machine configuration as a detailed schematic of each processing stage of the *EnigmaConfig* (proceeding from top to bottom), in which

- each line indicates the *Mapping* preformed by the component at that stage (see *stage_mapping_list*);
- each line begins with an indication of the stage (rotor number, **P** for plugboard, or **R** for reflector) at that stage, and ends with the specification (see *name*) of the component at that stage;
- rotors additionally indicate their window letter, and position; and
- if a valid `letter` is provided, it is indicated as input and its encoding at each stage is marked;

The schematic is followed by the mapping for the machine as a whole (as for the `'single'` format), and preceded by a (trivial, no-op) keyboard "mapping" for reference.

For example,

```
>>> print(cfg.config_string(letter='K', format='internal', mark_func=lambda c: '(' + c + ')'))
K > ABCDEFGHIJ(K)LMNOPQRSTUVWXYZ
  P YBCDFEGHIJ(Z)PONMLQRSTXVWUAK          UX.MO.KZ.AY.EF.PL
  1 LORVFBQNGWKATHJSZPIYUDXEM(C)  Q  07   II
  2 BJ(Y)INTKWOARFEMVSGCUDPHZQLX  A  24   VIII
  3 ILHXUBZQPNVGKMCRTEJFADOY(S)W  F  16   V
  4 YDSKZPTNCHGQOMXAUW(J)FBRELVI  L  10   γ
  R ENKQAUYWJ(I)COPBLMDXZVFTHRGS          b
  4 PUIBWTKJ(Z)SDXNHMFLVCGQYROAE          γ
  3 UFOVRTLCASMBNJWIHPYQEKZDX(G)          V
  2 JARTML(Q)VDBGYNEIUXKPFSOHZCW          VIII
  1 LFZVXEINSOKAYHBR(G)CPMUDJWTQ          II
  P YBCDFE(G)HIJZPONMLQRSTXVWUAK          UX.MO.KZ.AY.EF.PL
G < CMAWFEKLNV(G)HBIUYTXZQOJDRPS
```

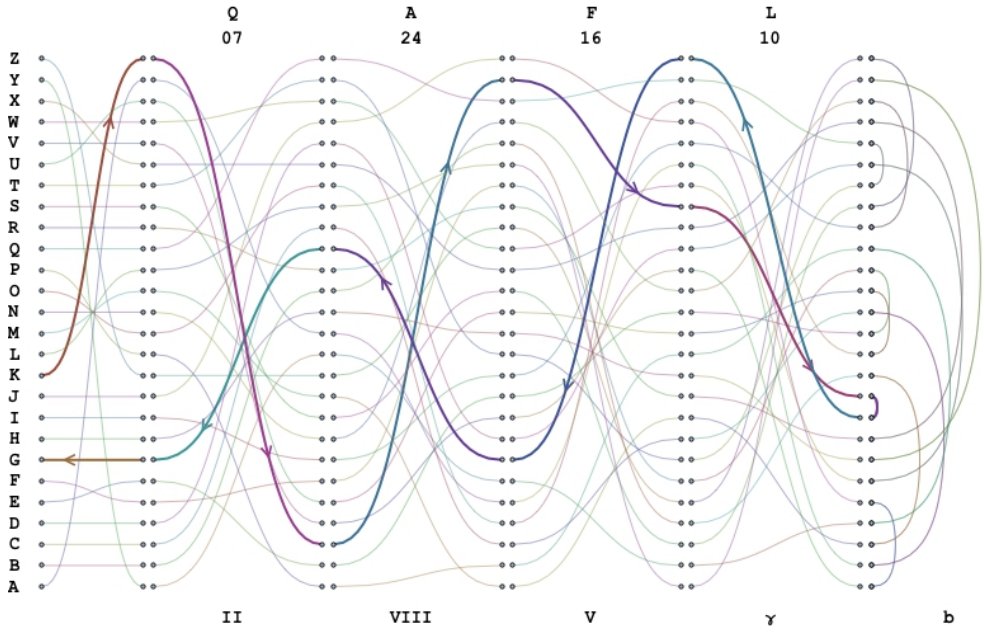
shows the process of encoding of the letter **K** to **G**:

- **K** is entered at the keyboard, which is then
- encoded by the plugboard (**P**), which includes **KZ** in its specification (see Name), to **Z**, which is then
- encoded by the first rotor (**1**), a **II** rotor in the 06 position (and **Q** at the window), to **C**, which is then
- encoded by the second rotor (**2**), a **VIII** rotor in the 24 position (and **A** at the window), to **Y**, which is then
- encoded by the third rotor (**3**), a **V** rotor in the 16 position (and **F** at the window), to **S**, which is then
- encoded by the fourth rotor (**4**), a γ rotor in the 10 position (and **L** at the window), to **J**, which is then
- encoded by the reflector rotor (**U**), a **b** reflector, to **I**, which reverses the signal sending it back through the rotors, where it is then
- encoded in reverse by the fourth rotor (**4**), to **Z**, which is then
- encoded in reverse by the third rotor (**3**), to **G**, which is then
- encoded in reverse by the second rotor (**2**), to **Q**, which is then
- encoded in reverse by the first rotor (**1**), to **G**, which is then
- left unchanged by the plugboard (**P**), and finally

•displayed as **G**.

Note that (as follows from Mapping) the position of the marked letter at each stage is the alphabetic position of the marked letter at the previous stage.

This can be represented schematically (with input arriving and output exiting on the left) as



Use `format='windows'` to simply show the letters at the *windows* as the operator would see them.

```
>>> print(cfg.config_string(format='windows'))
LFAQ
```

And use `format='config'` to simply show a conventional specification of an *EnigmaConfig* (as used for *config_enigma_from_string*):

```
>>> print(cfg.config_string(format='config'))
b-γ-V-VIII-II LFAQ UX.MO.KZ.AY.EF.PL 03.17.04.11
```

For both of the preceeding two formats, it is possible to also indicate the encoding of a character (not displayed by default) by setting `show_encoding` to `True`:

```
>>> print(cfg.config_string(format='windows', letter='K'))
LFAQ
>>> print(cfg.config_string(format='windows', letter='K', show_encoding=True))
LFAQ  K > G
>>> print(cfg.config_string(format='config', letter='K'))
b-γ-V-VIII-II LFAQ UX.MO.KZ.AY.EF.PL 03.17.04.11
>>> print(cfg.config_string(format='config', letter='K', show_encoding=True))
b-γ-V-VIII-II LFAQ UX.MO.KZ.AY.EF.PL 03.17.04.11  K > G
```

Use `format='encoding'` to show this encoding alone:

```
>>> print(cfg.config_string(format='encoding', letter='K'))
K > G
```

Note that though the examples above have been wrapped in `print` for clarity, these functions return strings:

```
>>> cfg.config_string(format='windows', letter='K', show_encoding=True)
u'LFAQ  K > G'
>>> cfg.config_string(format='internal').split('\n')
[u'    ABCDEFGHIJKLMNOPQRSTUVWXYZ', u'  P YBCDFEGHIJZPONMLQRSTXVWUAK         UX.MO.KZ.AY.EF.PL',
```

4.1.5 State transitions and operation

EnigmaConfig.**step**()

Step the Enigma machine to a new machine configuration.

Step the Enigma machine by rotating the rightmost (first) rotor one position, and other rotors as determined by the *positions* of rotors in the machine, based on the positions of their *components.Component.turnovers*. In the physical machine, a step occurs in response to each operator keypress, prior to processing that key's letter (see *enigma_encoding*).

Stepping leaves the *components* and *rings* of a configuration unchanged, changing only *positions*, which is manifest in changes of the letters visible at the *windows*:

Returns A new Enigma configuration.

Return type *EnigmaConfig*

Examples

Using the initial configuration

```
>>> cfg = EnigmaConfig.config_enigma("c-γ-V-I-II", "LXZO", "UX.MO.KZ.AY.EF.PL", "03.17.04.01")
```

the consequences of the stepping process can be observed by examining the *windows* of each stepped configuration:

```
>>> print(cfg.windows())
LXZO
>>> print(cfg.step().windows())
LXZP
>>> print(cfg.step().step().windows())
LXZQ
>>> print(cfg.step().step().step().windows())
LXZR
>>> print(cfg.step().step().step().step().windows())
LXZS
>>> print(cfg.step().step().step().step().step().windows())
LXZT
```

This, and the fact that only positions (and thus window letters) change as the result of stepping, can be visualized in more detail using *print_operation*:

```
>>> cfg.print_operation(steps=5, format='config')
c-γ-V-I-II LXZO UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZP UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZQ UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZR UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZS UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZT UX.MO.KZ.AY.EF.PL 03.17.04.01
```

EnigmaConfig.**stepped_configs**(*steps=None*)

Generate a series of stepped Enigma machine configurations.

Parameters **steps** (*int, optional*) – An optional limit on the number of steps to take in generating configurations.

Yields *EnigmaConfig* – The `EnigmaConfig` resulting from applying `step` to the previous one.

Examples

This allows the examples above to be rewritten as

```
>>> for c in cfg.stepped_configs(5):
...     print(c.windows())
LXZO
LXZP
LXZQ
LXZR
LXZS
LXZT

>>> for c in cfg.stepped_configs(5):
...     print(c)
c-γ-V-I-II LXZO UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZP UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZQ UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZR UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZS UX.MO.KZ.AY.EF.PL 03.17.04.01
c-γ-V-I-II LXZT UX.MO.KZ.AY.EF.PL 03.17.04.01
```

EnigmaConfig.**print_operation**(**args*, ***kwargs*)

Show the operation of the Enigma machine as a series of configurations.

Print out the operation of the Enigma machine as a series of `EnigmaConfig`, as it encodes a `message` and/or for a specified number of `steps`.

Parameters

- **message** (*unicode*) – A message to encode. Characters that are not letters will be replaced with standard *Kriegsmarine* substitutions or be removed (see `make_message`). Each character will be used as a `letter` in the `config_string` specified by the `format`.
- **steps** (*int, optional*) – A number of steps to run; if omitted when a `message` is provided, will default to the length of the message; otherwise defaults to 1
- **overwrite** (*bool, optional*) – Whether to overwrite the display of each step after a pause. (May result in garbled output on some systems.)
- **format** (*str, optional*) – A string specifying the format used to display the `EnigmaConfig` at each step of message processing; see `config_string`.
- **initial** (*bool, optional*) – Whether to show the initial starting step; the `EnigmaConfig` before encoding begins.
- **delay** (*float, optional*) – The number of seconds to wait (see `time.sleep`) between the display of each processing step; defaults to 0.2.
- **show_step** (*bool, optional*) – Whether to include the step number in the display.
- **show_encoding** (*bool, optional*) – Whether to indicate the encoding of each character for formats that do not include it by default; see `config_string`.
- **mark_func** (*function, optional*) – A `function` that highlights its argument by taking a single character as an argument and returning a string with additional characters added to

(usually surrounding) that charater. Used in cases where default method of highlighting the encoded-to character (see *Mapping*) does not display correctly or clearly.

Examples

(For details on differences among formats used for displaying each step, see the examples for *config_string*.)

Show the operation of a machine for 10 steps, indicating step numbers:

```
>>> cfg = EnigmaConfig.config_enigma("B-I-III-I", "EMO", "UX.MO.AY", "13.04.11")
>>> cfg.print_operation(format='single', steps=10, show_step=True)
0000      CNAUJVQSLEMIKBZRGPHXDFYTWO  EMO  19 10 05
0001      UNXKGVERLYDIQBTWMHZOAFPCJS  EMP  19 10 06
0002      QTYJZXUPKDIMLSWHAVNBGROFCE  EMQ  19 10 07
0003      DMXAPTRWKYINBLUESGQFOZHCJV  ENR  19 11 08
0004      IUSMHRPEAQTVDYWGJFCKBLOZNX  ENS  19 11 09
0005      WMVXQRLSPYOGBTKIEFHNZCADJU  ENT  19 11 10
0006      WKIQXNRSCVBOYFLUDGHZPJAEMT  ENU  19 11 11
0007      RVPTWSLKYXHGNMQCOAFDZBEJIU  ENV  19 11 12
0008      IYTKRVSMALDJHZWXUEGCQFOPBN  ENW  19 11 13
0009      PSWGMODULZVIERFAXNBYHKCQTJ  ENX  19 11 14
0010      IVOWZKHGARFSPUCMXJLYNBDQTE  ENY  19 11 15
```

Show the operation of a machine as it encodes a message, with step numbers:

```
>>> cfg.print_operation(format='single', message='TESTING', show_step=True)
0000      CNAUJVQSLEMIKBZRGPHXDFYTWO  EMO  19 10 05
0001  T > UNXKGVERLYDIQBTWMHZOAFPCJS  EMP  19 10 06
0002  E > QTYJZXUPKDIMLSWHAVNBGROFCE  EMQ  19 10 07
0003  S > DMXAPTRWKYINBLUESGQFOZHCJV  ENR  19 11 08
0004  T > IUSMHRPEAQTVDYWGJFCKBLOZNX  ENS  19 11 09
0005  I > WMVXQRLSPYOGBTKIEFHNZCADJU  ENT  19 11 10
0006  N > WKIQXNRSCVBOYFLUDGHZPJAEMT  ENU  19 11 11
0007  G > RVPTWSLKYXHGNMQCOAFDZBEJIU  ENV  19 11 12
```

Show the same process, but just what the operator would see:

```
>>> cfg.print_operation(format='windows', message='TESTING', show_encoding=True, show_step=True)
0000  EMO
0001  EMP  T > O
0002  EMQ  E > Z
0003  ENR  S > Q
0004  ENS  T > K
0005  ENT  I > P
0006  ENU  N > F
0007  ENV  G > L
```

Show detailed internal version of the same process:

```
>>> cfg.print_operation(format='internal', message='TESTING', show_step=True)
0000
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
  P YBCDEFGHIJKLONMPQRSTXVWUAZ          UX.MO.AY
  1 HCZMRVJPKSUDTQOLWEXNYFAGIB  O  05  I
  2 KOMQEPVZNXRBDLJHFSUWYACTGI  M  10  III
  3 AXIQJZKRMSUNTOLYDHVBWEGPFC  E  19  I
  R YRUHQSLDPXNGOKMIEBFZCWVJAT          B
  3 ATZQVYWRCEGOILNXDHJMKSUBPF          I
```

```
      2 VLWMEQYPZOANCIBFDKRXSGTJUH          III
      1 WZBLRVXAYGIPDTOHNEJMKFQSUC          I
      P YBCDEFGHIJKLONMPQRSTXVWUAZ          UX.MO.AY
        CNAUJVQSLEMIKBZRGPHXDFYTWO

    0001
    T > ABCDEFGHIJKLMNOPQRSTUVWXYZ
      P YBCDEFGHIJKLONMPQRSTXVWUAZ          UX.MO.AY
      1 BYLQUIOJRTCSPNKVDWMXEZFHAG  P  06   I
      2 KOMQEPVZNXRBDLJHFSUWYACTGI  M  10   III
      3 AXIQJZKRMSUNTOLYDHVBWEGPFC  E  19   I
      R YRUHQSLDPXNGOKMIEBFZCWVJAT          B
      3 ATZQVYWRCEGOILNXDHJMKSUBPF          I
      2 VLWMEQYPZOANCIBFDKRXSGTJUH          III
      1 YAKQUWZXFHOCSNGMDILJEPRTBV          I
      P YBCDEFGHIJKLONMPQRSTXVWUAZ          UX.MO.AY
    O < UNXKGVERLYDIQBTWMHZOAFPCJS

    0002
    E > ABCDEFGHIJKLMNOPQRSTUVWXYZ
      P YBCDEFGHIJKLONMPQRSTXVWUAZ          UX.MO.AY
      1 XKPTHNIQSBROMJUCVLWDYEGZFA  Q  07   I
      2 KOMQEPVZNXRBDLJHFSUWYACTGI  M  10   III
      3 AXIQJZKRMSUNTOLYDHVBWEGPFC  E  19   I
      R YRUHQSLDPXNGOKMIEBFZCWVJAT          B
    ...
```

4.1.6 Encoding

Message encoding

EnigmaConfig.**enigma_encoding**(**args*, ***kwargs*)

Encode a message using the machine configuration.

Encode a string, interpreted as a message (see *make_message*), using the (starting) machine configuration, by stepping (see *step*) the configuration prior to processing each character of the message. This produces a new configuration (with new *positions* only) for encoding each character, which serves as the "starting" configuration for subsequent processing of the message.

Parameters **message** (*unicode*) – A message to encode.

Returns The machine-encoded message.

Return type unicode

Examples

Given machine configuration

```
>>> cfg = EnigmaConfig.config_enigma("b-γ-V-VIII-II", "LFAP", "UX.MO.KZ.AY.EF.PL", "03.17.04.11"
```

the message `'KRIEG'` is encoded to `'GOWNW'`:

```
>>> cfg.enigma_encoding('KRIEG')
u'GOWNW'
```

The details of this encoding and its relationship to stepping from one configuration to another are illustrated using *print_operation*:

```
>>> cfg.print_operation("KRIEG", format='windows', show_encoding=True, show_step=True)
0000  LFAP
0001  LFAQ  K > G
0002  LFAR  R > O
0003  LFAS  I > W
0004  LFAT  E > N
0005  LFAU  G > W
```

Note that because of the way the Enigma machine is designed, it is always the case (provided that `msg` is all uppercase letters) that:

```
cfg.enigma_encoding(cfg.enigma_encoding(msg)) == msg
```

`EnigmaConfig.`**`print_encoding`**`(*args, **kwargs)`

Show the conventionally formatted encoding of a message.

Print out the encoding of a message by an (initial) *EnigmaConfig*, formatted into conventional blocks of four characters.

Parameters **`message`** (*unicode*) – A message to encode. Characters that are not letters will be replaced with standard *Kriegsmarine* substitutions or be removed (see *make_message*).

Examples

```
>>> cfg = EnigmaConfig.config_enigma("c-β-V-VI-VIII", "CDTJ", "AE.BF.CM.DQ.HU.JN.LX.PR.SZ.VW", "
>>> cfg.print_encoding("FOLGENDES IST SOFORT BEKANNTZUGEBEN")
RBBF PMHP HGCZ XTDY GAHG UFXG EWKB LKGJ
```

static `EnigmaConfig.`**`make_message`**`(*args, **kwargs)`

Convert a string to valid Enigma machine input.

Replace any symbols for which there are standard *Kriegsmarine* substitutions, remove any remaining non-letter characters, and convert to uppercase. This function is applied automatically to `message` arguments for functions defined here (*enigma_encoding*).

Parameters **`string`** (*unicode*) – A string to convert to valid Enigma machine input.

Returns A string of valid Enigma machine input characters.

Return type unicode

Note: This documentation is in draft form. Reports of any errors or suggestions for improvement are welcomed and should be submitted as new issues.

4.2 Components - `crypto_enigma.components`

This is a supporting module that defines the components used to construct an Enigma machine. It will not generally be used directly.

4.2.1 Overview

Component	A component of an Enigma machine.
name	The specification a component of an Enigma machine.
wiring	The physical wiring of a component, expressed as a *mapping*.
turnovers	The turnover positions for a rotor.
mapping	The mapping performed by a component based on its rotational position.
Direction	The direction that a signal flows through a component.
component	Retrieve a specified component.
rotors	The list of valid (non-reflector) rotor names.
reflectors	The list of valid reflector rotor names

4.2.2 Machine components

class crypto_enigma.components.**Component**(*name*, *wiring*, *turnovers*)

A component of an Enigma machine.

A component used to construct an Enigma machine (as embodied in an *EnigmaConfig*) identified by its specification (see *name*), and characterized by its physical *wiring* and additionally — for rotors other than the reflector — by *turnovers* which govern the stepping (see *step*) of the machine in which it is installed.

There is no reason to construct a component directly, and no directly instantiated component can be used in an *EnigmaConfig*. The properties of components "outside of" an *EnigmaConfig* can be *examined using component*.

4.2.3 Component properties

Component.**name**

The specification a component of an Enigma machine.

For rotors (including the reflector) this is one of the conventional letter or Roman numeral designations (e.g., `'IV'` or `'`β`'`) or rotor "names". For the plugboard this is the conventional string of letter pairs, indicating letters wired together by plugging (e.g., `'AU.ZM.ZL.RQ'`). Absence or non-use of a plugboard can be indicated with `'~'` (or almost anything that isn't a valid plugboard spec).

Returns A string uniquely specifying a *Component*.

Return type unicode

Component.**wiring**

The physical wiring of a component, expressed as a *mapping*.

Returns The *mapping* established by the physical wiring of a *Component*: the forward mapping when **01** is at the window position for rotors; by the plug arrangement for the plugboard.

Return type *Mapping*

Examples

A rotor's wiring is fixed by the physical connections of the wires inside the rotor:

```
>>> cmp = component('V')
>>> cmp.wiring
u'VZBRGITYUPSDNHLXAWMJQOFECK'
>>> component('VI').wiring
u'JPGVOUMFYQBENHZRDKASXLICTW'
```

For plugboards, it is established by the specified connections:

```
>>> component('AZ.BY').wiring
u'ZYCDEFGHIJKLMNOPQRSTUVWXBA'
```

Component.**turnovers**
The turnover positions for a rotor.

Returns The letters on a rotor's ring that appear at the window (see *windows*) when the ring is in the turnover position. Not applicable (and empty) for the plugboard and for reflectors. (See *step*.)

Return type unicode

Examples

Only "full-width" rotors have turnovers:

```
>>> component('V').turnovers
u'Z'
>>> component('VI').turnovers
u'ZM'
>>> component('I').turnovers
u'Q'
```

Reflectors, "half-width" rotors, and the plugboard never do:

```
>>> component('B').turnovers
u''
>>> component('β').turnovers
u''
>>> component('AG.OI.LM.ER.KU').turnovers
u''
```

4.2.4 The component mapping

Component.**mapping**(**args*, ***kwargs*)
The mapping performed by a component based on its rotational position.

The *mapping* performed by a *Component* as a function of its position (see *positions*) in an Enigma machine and the *Direction* of the signal passing through it.

For all other positions of rotors, the mapping is a cyclic permutation this wiring's inputs (backward) and outputs (forward) by the rotational offset of the rotor away from the **01** position.

Parameters

- **position** (*int*) – The rotational offset of the *Component* in the Enigma machine.
- **direction** (Direction) – The direction of signal passage through the component.

Returns The *mapping* performed by the component in the `direction` when `position` is at the window position.

Return type *Mapping*

Examples

Note that because the wiring of reflectors generates mappings that consist entirely of paired exchanges of letters, reflectors (at any position) produce the same mapping in both directions (the same is true of the plugboard):

```
>>> all(c.mapping(p, Direction.FWD) == c.mapping(p, Direction.REV) for c in map(component, refle
True
```

For rotors in their base position, with **01** at the window position, and for the plugboard, this is just the *wiring*:

```
>>> cmp.wiring == cmp.mapping(1, Direction.FWD)
True
```

class crypto_enigma.components.**Direction**
The direction that a signal flows through a component.

During encoding of a character, the signal passes first through the wiring of each *Component*, from right to left in the machine, in a forward (FWD) direction, then through the reflector, and then, from left to right, through each component again, in reverse (REV). This direction affects the encoding performed by the component (see mapping).

4.2.5 Getting components

crypto_enigma.components.**component**(**args*, ***kwargs*)
Retrieve a specified component.

Parameters **name** – The name of a *Component*

Returns The component with the specified name.

Return type *Component*

Examples

Components are displayed as a string consisting of their properties:

```
>>> print(component('VI'))
VI JPGVOUMFYQBENHZRDKASXLICTW ZM
```

Components with the same name are always identical:

```
>>> component('AG.OI.LM.ER.KU') is component('AG.OI.LM.ER.KU')
True
```

crypto_enigma.components.**rotors**
The list of valid (non-reflector) rotor names.

```
>>> rotors
[u'I', u'II', u'III', u'IV', u'V', u'VI', u'VII', u'VIII', u'β', u'γ']
```

crypto_enigma.components.**reflectors**
The list of valid reflector rotor names

```
>>> reflectors
[u'A', u'B', u'C', u'b', u'c']
```

Note: This documentation is in draft form. Reports of any errors or suggestions for improvement are welcomed and should be submitted as new issues.

4.3 Cypher - crypto_enigma.cypher

This is a supporting module that implements the simple substitution cypher employed by the Enigma machine to encode messages. It will not generally be used directly.

4.3.1 Overview

Mapping	A substitution cypher mapping.
encode_string	Encode a string using the mapping.
encode_char	Encode a single character using the mapping.

4.3.2 Substitution cypher mappings

All encoding functionality is built upon a single class:

class crypto_enigma.cypher.**Mapping**

Bases: unicode

A substitution cypher mapping.

The Enigma machine, and the components from which it is constructed, use **mappings** to perform a simple substitution encoding. Mappings describe

- the cryptographic effects of each component's fixed *wiring*;
- the encoding they perform individually in a machine based on their rotational positions and the direction in which a signal passes through them (see *mapping*); and,
- the progressive (stage_mapping_list) and overall (enigma_mapping_list and enigma_mapping) encoding performed by the machine as a whole.

4.3.3 Mapping encoding

Mappings are expressed as a string of letters indicating the mapped-to letter for the letter at that position in the alphabet — i.e., as a permutation of the alphabet. For example, the mapping **EKMFLGDQVZNTOWYHXUSPAIBRCJ** encodes **A** to **E**, **B** to **K**, **C** to **M**, ..., **Y** to **C**, and **Z** to **J**:

```
>>> mpg = Mapping(u'EKMFLGDQVZNTOWYHXUSPAIBRCJ')
>>> mpg.encode_string(u'ABCYZJ')
u'EKMCJZ'
>>> mpg.encode_string(u'ABCDEFGHIJKLMNOPQRSTUVWXYZ') == mpg
True
```

Note that there is no way to directly create *Mapping* for use by an EnigmaMachine or Component. The closest one can get is to configure a plugboard with component:

```
>>> component(u'AE.BK.CM.FD').wiring
u'EKMF...'
```

For reference, two functions are provided to perform a mappings substitution cypher, though these will rarely be used directly:

Mapping.**encode_string**(*string*)

Encode a string using the mapping.

Parameters **string** (*str*) – A string to encode using the *Mapping*.

Returns A string consisting of each of the characters replaced with the corresponding character in the *Mapping*.

Return type str

Examples

This just the collected results of applying *encode_char* to each letter of the string:

```
>>> component(u'AE.BK.CM.FD').wiring.encode_string(u'ABKCFEKMD')
u'EKBMDABCF'
>>> ''.join(component(u'AE.BK.CM.FD').wiring.encode_char(c) for c in u'ABKCFEKMD')
u'EKBMDABCF'
```

Note that, critically, the mapping used by an Enigma machine *changes before each character is encoded* so that:

```
>>> cfg.enigma_mapping().encode_string(str) != cfg.enigma_encoding(str)
True
```

Mapping.**encode_char**(*ch*)

Encode a single character using the mapping.

Parameters **ch** (*char*) – A character to encode using the *Mapping*.

Returns The character, replaced with the corresponding characters in the *Mapping*.

Return type chr

Example

In the context of this package, this is most useful in low level analysis of the encoding process:

```
>>> wng = component(u'AE.BK.CM.FD').wiring
>>> wng.encode_char(u'A')
u'E'
>>> wng.encode_char(u'K')
u'B'
>>> wng.encode_char(u'Q')
u'Q'
```

For example, it can be used to confirm that only letters connected in a plugboard are unaltered by the encoding it performs:

```
>>> pbd = component(u'AE.BK.CM.FD')
>>> all(pbd.wiring.encode_char(c) == c for c in filter(lambda c: c not in pbd.name,  u'ABCDEFGHI
True
>>> all(pbd.wiring.encode_char(c) != c for c in filter(lambda c: c in pbd.name,  u'ABCDEFGHIJKLM
True
```

Note: This documentation is in draft form. Reports of any errors or suggestions for improvement are welcomed and should be submitted as new issues.

4.4 Exceptions - crypto_enigma.exceptions

This module defines some placeholders for custom exceptions and errors.

4.4.1 Exceptions

exception crypto_enigma.exceptions.**EnigmaError**
Bases: exceptions.Exception

exception crypto_enigma.exceptions.**EnigmaValueError**
Bases: *crypto_enigma.exceptions.EnigmaError*, exceptions.ValueError

exception crypto_enigma.exceptions.**EnigmaDisplayError**
Bases: *crypto_enigma.exceptions.EnigmaError*

CHAPTER 5

Indices and tables

- genindex
- modindex

c

e

m

C

D

E

M

N

P

R

S

T

W