Hanabython Documentation

Release 0.1.12

François Durand

Jun 27, 2019

Contents:

1 Hanabython **1**
- 1.1 Features . . . 1
- 1.2 Credits . . . 1

2 Installation **3**
- 2.1 Stable release . . . 3
- 2.2 From sources . . . 3

3 Usage **5**
- 3.1 Import . . . 5
- 3.2 Getting started (in a terminal) . . . 5
- 3.3 Getting started (in a notebook) . . . 5

4 Reference **7**
- 4.1 Manipulation of strings . . . 7
- 4.2 Colors . . . 9
- 4.3 Configuration . . . 11
- 4.4 Clues . . . 18
- 4.5 Cards . . . 19
- 4.6 Hands . . . 22
- 4.7 Draw Pile . . . 24
- 4.8 Discard Pile . . . 26
- 4.9 Board . . . 30
- 4.10 Actions . . . 32
- 4.11 Players . . . 33
- 4.12 Game . . . 47

5 Contributing **57**
- 5.1 Types of Contributions . . . 57
- 5.2 Get Started! . . . 58
- 5.3 Pull Request Guidelines . . . 59
- 5.4 Tips . . . 59
- 5.5 Deploying . . . 59

6 Credits **61**
- 6.1 Development Lead . . . 61
- 6.2 Contributors . . . 61

7 History **63**
7.1 0.1.12 (2019-06-27) 63
7.2 0.1.11 (2019-06-27) 63
7.3 0.1.10 (2018-02-26) 63
7.4 0.1.9 (2018-02-26) 63

8 Indices and tables **65**

Index **67**

CHAPTER 1

Hanabython

A Python implementation of Hanabi, a game by Antoine Bauza

- Free software: GNU General Public License v3
- Documentation: https://hanabython.readthedocs.io.

1.1 Features

- TODO

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install Hanabython, run this command in your terminal:

```
$ pip install hanabython
```

This is the preferred method to install Hanabython, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Hanabython can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/francois-durand/hanabython
```

Or download the tarball:

```
$ curl  -OL https://github.com/francois-durand/hanabython/tarball/master
```

Once you have a copy of the source, you can install it with:

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

CHAPTER 3

Usage

3.1 Import

To use Hanabython in a project:

```
import hanabython
```

3.2 Getting started (in a terminal)

```
from hanabython import Game, PlayerHumanText
Game(players=[
    PlayerHumanText('Antoine'),
    PlayerHumanText('Donald X'),
    PlayerHumanText('Uwe')
]).play()
```

3.3 Getting started (in a notebook)

```
from hanabython import Game, PlayerHumanText
Game(players=[
    PlayerHumanText('Antoine', ipython=True),
    PlayerHumanText('Donald X', ipython=True),
    PlayerHumanText('Uwe', ipython=True)
]).play();
```

CHAPTER 4

Reference

4.1 Manipulation of strings

class hanabython.**StringAnsi**

An ANSI escape code that modifies the printing aspect.

BLUE = '\x1b[94m'

BROWN = '\x1b[33m'

CYAN = '\x1b[96m'

GREEN = '\x1b[32m'

MAGENTA = '\x1b[35m'

RED = '\x1b[31m'

RESET = '\x1b[0;0m'

This escape code is special: it is used to return to default aspect.

STYLE_BOLD = '\x1b[1m'

STYLE_REVERSE_VIDEO = '\x1b[7m'

STYLE_UNDERLINE = '\x1b[4m'

WHITE = ''

This should be white on black background, and vice-versa.

YELLOW = '\x1b[93m'

hanabython.**str_from_iterable**(*l: Iterable[T_co]*) → str

Convert an iterable to a simple string.

There are two differences with the standard implementation of str:

1. No brackets.

2. For each `item` of the iterable, `str_from_iterable` uses `str(item)`, whereas `str` uses `repr(item)`.

Parameters **l** – an iterable.

Returns a simple string.

```
>>> print(str_from_iterable(['a', 'b', 'c']))
a b c
>>> print(['a', 'b', 'c'])
['a', 'b', 'c']
```

hanabython.**title**(*s: str*, *width: int*) → str

Format a string as a title.

Parameters

- **s** – the string
- **width** – the total width of the final layout (in number of characters).

Returns the string formatted as a title.

```
>>> title(s='Title', width=20)
'****** Title *******'
>>> title(s='A not-too-long title', width=20)
'A not-too-long title'
>>> title(s='A title that is really too long', width=20)
'A title that is r...'
```

hanabython.**uncolor**(*s: str*) → str

Remove ANSI escape codes from the string.

Parameters **s** (*string*) – a string.

Returns the same string without its ANSI escape codes.

```
>>> from hanabython import StringAnsi
>>> s = (StringAnsi.RED + "Hanabi" + StringAnsi.RESET + ', a game by '
...      + StringAnsi.BLUE + 'Antoine Bauza' + StringAnsi.RESET)
>>> uncolor(s)
'Hanabi, a game by Antoine Bauza'
```

class hanabython.**Colored**

An object with a colored string representation.

```
>>> from hanabython import StringAnsi
>>> class MyClass(Colored):
...     def colored(self):
...         return StringAnsi.RED + 'some text' + StringAnsi.RESET
>>> my_object = MyClass()
>>> my_object.colored()
'\x1b[31msome text\x1b[0;0m'
>>> str(my_object)
'some text'
>>> repr(my_object)
'<MyClass: some text>'
```

colored() → str

Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

test_str() → None
Test the string representations of the object.

Print the results of `__repr__()`, `__str__()` and *colored()*.

4.2 Colors

class hanabython.**Color**(*name: str*, *symbol: str*, *print_color: str*)
A color in Hanabi.

Parameters

- **name** – The full name of the color. In a game, two distinct colors must have different names.
- **symbol** – The short name of the color. For standard colors (defined as constants in *Colors*), it is always 1 character, and no two standard colors have the same symbol. For user-defined colors, it is recommended to do the same, but not necessary.
- **print_color** – an ANSI escape code that modifies the printing color. See *StringAnsi*.

```
>>> brown = Color(name='Brown', symbol='N', print_color=StringAnsi.BROWN)
>>> brown.name
'Brown'
>>> brown.symbol
'N'
>>> brown.print_color
'\x1b[33m'
```

color_str(*o: object*) → str
Convert an object to a colored string.

Parameters **o** – any object.

Returns the `__str__` of this object, with an ANSI color-modifying escape code at the beginning and its cancellation at the end.

```
>>> brown = Color(name='Brown', symbol='N',
...               print_color=StringAnsi.BROWN)
>>> brown.color_str('some text')
'\x1b[33msome text\x1b[0;0m'
>>> brown.color_str(42)
'\x1b[33m42\x1b[0;0m'
```

colored() → str
Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

is_cluable

Returns whether this color can be used for clues. For a normal color, it is True. This is different in *ColorMulticolor* and *ColorColorless*.

match(*clue_color: hanabython.Modules.Color.Color*) → bool
React to a color clue.

Parameters **clue_color** – the color of the clue.

Returns whether a card of the current color should react to a clue of color `clue_color`. A normal color matches simply if the color of the clue is the same. This is different in *ColorMulticolor* and *ColorColorless*.

```
>>> brown = Color(name='Brown', symbol='N',
...               print_color=StringAnsi.BROWN)
>>> pink = Color(name='Pink', symbol='P',
...              print_color=StringAnsi.MAGENTA)
>>> brown.match(clue_color=brown)
True
>>> brown.match(clue_color=pink)
False
```

class hanabython.**ColorMulticolor**(*name: str*, *symbol: str*, *print_color: str*)

is_cluable

Returns False. It is not allowed to give "multicolor" as a clue.

match(*clue_color: hanabython.Modules.Color.Color*) → bool
Multicolor matches any color clue.

```
>>> multicolor = ColorMulticolor(name='Multicolor', symbol='M',
...                              print_color=StringAnsi.MAGENTA)
>>> brown = Color(name='Brown', symbol='N',
...               print_color=StringAnsi.BROWN)
>>> multicolor.match(clue_color=brown)
True
```

class hanabython.**ColorColorless**(*name: str*, *symbol: str*, *print_color: str*)

is_cluable

Returns False. It is not allowed to give "colorless" as a clue.

match(*clue_color: hanabython.Modules.Color.Color*) → bool
Colorless matches no color clue.

```
>>> colorless = ColorColorless(name='Colorless', symbol='C',
...                            print_color=StringAnsi.MAGENTA)
>>> brown = Color(name='Brown', symbol='N',
...               print_color=StringAnsi.BROWN)
```

(continues on next page)

(continued from previous page)

```
>>> colorless.match(clue_color=brown)
False
```

class hanabython.**Colors**

Standard colors in Hanabi.

BLUE = <Color: B>

COLORLESS = <ColorColorless: C>

Use this for the colorless cards. As of now, it is brown but the display color might change in future implementations.

GREEN = <Color: G>

MULTICOLOR = <ColorMulticolor: M>

Use this for multicolor cards. As of now, it is cyan but the display color might change in future implementations.

RED = <Color: R>

SIXTH = <Color: P>

Use this for the sixth color. As of now, it is pink but the display color might change in future implementations.

WHITE = <Color: W>

YELLOW = <Color: Y>

classmethod from_symbol(*s: str*) → hanabython.Modules.Color.Color

Find one of the standard colors from its symbol.

Returns the corresponding color.

```
>>> color = Colors.from_symbol('B')
>>> print(color.name)
Blue
```

4.3 Configuration

class hanabython.**ConfigurationColorContents**(*contents: Iterable[int], name: str = None*)

The contents of a color in a deck of Hanabi.

This is essentially a list, stating the number of copies for each card. For example, [3, 2, 2, 2, 1] means there are 3 ones, 2 twos, etc. Each integer in this list must be strictly positive.

Parameters

- **contents** – an iterable used to create the list.
- **name** – the name of the configuration. Can be None (default value). Should not be capitalized (e.g. "my favorite configuration" and not "My favorite configuration").

```
>>> cfg = ConfigurationColorContents.NORMAL
>>> print(cfg.name)
normal
>>> print(cfg)
normal
>>> print(list(cfg))
```

(continues on next page)

(continued from previous page)

```
[3, 2, 2, 2, 1]
>>> cfg = ConfigurationColorContents([3, 2, 1])
>>> print(cfg.name)
None
>>> print(cfg)
[3, 2, 1]
```

NORMAL = <ConfigurationColorContents: normal>
Normal contents of a color (3 2 2 2 1).

SHORT = <ConfigurationColorContents: short>
Contents of a short color (1 1 1 1 1).

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

class hanabython.**ConfigurationDeck**(*contents: Iterable[Tuple[hanabython.Modules.Color.Color, hanabython.Modules.ConfigurationColorContents.ConfigurationColorContents]], name: str = None*)

The contents of the deck for a game of Hanabi.

This is essentially an OrderedDict. To each *Color* in the deck, it associates the contents of the color, an object of class *ConfigurationColorContents*.

The order of the colors is important: it will be used in many occasions (including for display).

Parameters

- **contents** – the iterable used to construct the ordered dictionary. Typically it is an OrderedDict or a list of pairs (*color*, *contents*).
- **name** – the name of the configuration. Can be None (default value). Should not be capitalized (e.g. "my favorite configuration" and not "My favorite configuration").

```
>>> cfg = ConfigurationDeck.NORMAL
>>> print(cfg.name)
normal
>>> print(cfg)
normal
>>> cfg = ConfigurationDeck(
...     contents=[
...         (Colors.BLUE, ConfigurationColorContents.NORMAL),
...         (Colors.RED, ConfigurationColorContents([3, 2, 1]))
...     ]
... )
>>> print(cfg.name)
None
>>> print(cfg)
B normal, R [3, 2, 1]
```

EIGHT_COLORS = <ConfigurationDeck: with sixth color, multicolor and colorless (10 car
Deck with 8 colors (6 colors + multicolor + colorless, all of 10 cards).

NORMAL = <ConfigurationDeck: normal>
Normal deck (5 colors of 10 cards).

W_MULTICOLOR = <ConfigurationDeck: with normal multicolor (10 cards)>
Deck with long multicolor (5 colors of 10 cards + 1 multi of 10 cards).

W_MULTICOLOR_SHORT = <ConfigurationDeck: with short multicolor (5 cards)>
Deck with short multicolor (5 colors of 10 cards + 1 multi of 5 cards).

W_SIXTH = <ConfigurationDeck: with normal sixth color (10 cards)>
Deck with long sixth color (6 colors of 10 cards).

W_SIXTH_SHORT = <ConfigurationDeck: with short sixth color (5 cards)>
Deck with short sixth color (5 colors of 10 cards + 1 color of 5 cards).

colored() → str
Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

copy() → hanabython.Modules.ConfigurationDeck.ConfigurationDeck
Copy the deck configuration.

Returns a copy of this deck configuration. You can modify the copy without modifying the original. However, it is not a deep copy, (most of time, it would not be useful).

```
>>> cfg = ConfigurationDeck.NORMAL.copy()
>>> cfg.name = None
>>> del(cfg[Colors.WHITE], cfg[Colors.YELLOW])
>>> print(cfg)
B normal, G normal, R normal
>>> print(ConfigurationDeck.NORMAL[Colors.WHITE])
normal
```

static normal_plus(*contents: Iterable[Tuple[hanabython.Modules.Color.Color, hanabython.Modules.ConfigurationColorContents.ConfigurationColorContents]], name: str = None*) → hanabython.Modules.ConfigurationDeck.ConfigurationDeck
Shortcut to define a deck configuration from the normal one.

Parameters

- **contents** – the additional contents (typically multicolor, etc.)
- **name** – the name of the configuration.

Returns the new configuration.

```
>>> cfg = ConfigurationDeck.normal_plus(contents=[
...     (Colors.SIXTH, ConfigurationColorContents.NORMAL),
...     (Colors.MULTICOLOR, ConfigurationColorContents.SHORT)
... ])
```

(continues on next page)

(continued from previous page)

```
>>> print(cfg)
B normal, G normal, R normal, W normal, Y normal, P normal, M short
```

class hanabython.**ConfigurationEmptyClueRule**(*i: int*, *name: str*)

A rule for "empty clues" in Hanabi.

An empty clue is a clue that corresponds to 0 cards in the hand of the concerned partner.

This class does not implement the rules themselves: they are hardcoded in the class *Game*.

Parameters

- **i** – a unique identifier of the rule.
- **name** – the name of the configuration. Should not be capitalized (e.g. "my favorite configuration" and not "My favorite configuration").

```
>>> cfg = ConfigurationEmptyClueRule.FORBIDDEN
>>> print(cfg)
empty clues are forbidden
>>> print(cfg==ConfigurationEmptyClueRule.FORBIDDEN)
True
>>> print(cfg==ConfigurationEmptyClueRule.ALLOWED)
False
```

ALLOWED = <ConfigurationEmptyClueRule: empty clues are allowed>

FORBIDDEN = <ConfigurationEmptyClueRule: empty clues are forbidden>

colored() → str

Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

class hanabython.**ConfigurationEndRule**(*i: int*, *name: str*)

A rule for the end of game in Hanabi.

This class does not implement the rules themselves: they are hardcoded in the class *Game*.

Parameters

- **i** – a unique identifier of the rule.
- **name** – the name of the configuration. Should not be capitalized (e.g. "my favorite configuration" and not "My favorite configuration"), except if it is seen as a title (e.g. "Crowning Piece").

```
>>> cfg = ConfigurationEndRule.NORMAL
>>> print(cfg)
normal
>>> print(cfg==ConfigurationEndRule.NORMAL)
True
```

(continues on next page)

(continued from previous page)

```
>>> print(cfg==ConfigurationEndRule.CROWNING_PIECE)
False
```

CROWNING_PIECE = <ConfigurationEndRule: Crowning Piece>
"Crowning piece" variant for the end of game. The game stops when a player starts her turn with no card in hand.

NORMAL = <ConfigurationEndRule: normal>
Default rule for the end of game. When a player draws the last card, all players play one last time (her included).

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

class hanabython.**ConfigurationHandSize**(*f: Callable[[int], int], name: str = None*)
A rule for the initial size of the players' hands.

Parameters

- **f** – a callable that, to a number of players, associates a number of cards.
- **name** – the name of the configuration. Can be None (default value). Should not be capitalized (e.g. "my favorite configuration" and not "My favorite configuration").

```
>>> cfg = ConfigurationHandSize.NORMAL
>>> print(cfg)
normal
>>> cfg = ConfigurationHandSize(f=lambda n: 9 - n)
>>> print(cfg)
7 for 2p, 6 for 3p, 5 for 4p, 4 for 5p
```

NORMAL = <ConfigurationHandSize: normal>
Normal rule for hand size (5 for 3- players, 4 for 4+ players).

VARIANT_6_3 = <ConfigurationHandSize: experimental (6 for 2 players, 3 for 5 players)
Experimental variant for hand size (6 for 2 players, 3 for 5+ players).

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

class hanabython.**Configuration**(*deck: hanabython.Modules.ConfigurationDeck.ConfigurationDeck = <ConfigurationDeck: normal>, n_clues: int = 8, n_misfires: int = 3, hand_size_rule: hanabython.Modules.ConfigurationHandSize.ConfigurationHandSize = <ConfigurationHandSize: normal>, empty_clue_rule: hanabython.Modules.ConfigurationEmptyClueRule.ConfigurationEmptyClueRule = <ConfigurationEmptyClueRule: empty clues are forbidden>, end_rule: hanabython.Modules.ConfigurationEndRule.ConfigurationEndRule = <ConfigurationEndRule: normal>, name: str = None*)

A configuration for a game of Hanabi.

Parameters

- **deck** – the configuration of the deck.
- **n_clues** – the number of clue chips that players have.
- **n_misfires** – the number of misfire chips that players have. If `n_misfires` misfire chips are used, then the game is lost immediately (it is not a final warning but really the end of the game).
- **hand_size_rule** – the rule used for the initial size of the hands.
- **empty_clue_rule** – the rule used about empty clues.
- **end_rule** – the rule used to determine when then game is finished.
- **name** – the name of the configuration. Can be None (default value). Should not be capitalized (e.g. "my favorite configuration" and not "My favorite configuration").

Variables

- **colors** (*list*) – a list of Color objects. It is the list of keys of `deck`.
- **n_colors** (*int*) – the number of colors.
- **highest** (*OrderedDict*) – For each color from `colors`, it gives the number on the highest card in that color.
- **n_values** (*int*) – the number on the highest card in the whole deck.
- **values** (*list*) – the list of possible values (from 1 to `n_values`).
- **deck_array** (*np.array*) – a numpy array of size `n_colors` * `n_values`. Each row represents the distribution of cards in a color. Typically, a row is [3, 2, 2, 2, 1], meaning that there are 3 ones, 2 twos, etc. Please note that column 0 corresponds to card value 1, etc.
- ***n_cards*** (*int*) – the total number of cards in the deck (50 in the standard configuration).
- **max_score** (*int*) – the maximum possible score (25 in the standard configuration).

```
>>> cfg = Configuration.W_MULTICOLOR_SHORT
>>> print(cfg.name)
with short multicolor (5 cards)
>>> print(cfg)
Deck: with short multicolor (5 cards).
Number of clues: 8.
Number of misfires: 3.
Clues rule: empty clues are forbidden.
End rule: normal.
>>> print(cfg.hand_size_rule)
normal
```

(continues on next page)

(continued from previous page)

```
>>> print(cfg.colors)
[<Color: B>, <Color: G>, <Color: R>, <Color: W>, <Color: Y>, <ColorMulticolor: M>]
>>> print(cfg.n_colors)
6
>>> print(cfg.highest)
OrderedDict([(<Color: B>, 5), (<Color: G>, 5), (<Color: R>, 5), (<Color: W>, 5), (
↪<Color: Y>, 5), (<ColorMulticolor: M>, 5)])
>>> print(cfg.n_values)
5
>>> print(cfg.values)
[1, 2, 3, 4, 5]
>>> print(cfg.deck_array)
[[3 2 2 2 1]
 [3 2 2 2 1]
 [3 2 2 2 1]
 [3 2 2 2 1]
 [3 2 2 2 1]
 [1 1 1 1 1]]
>>> print(cfg.n_cards)
55
>>> print(cfg.max_score)
30
```

Design a configuration manually:

```
>>> from hanabython import (ConfigurationDeck, ConfigurationColorContents,
...                         ConfigurationEmptyClueRule)
>>> cfg = Configuration(
...     deck=ConfigurationDeck(contents=[
...         (Colors.BLUE, ConfigurationColorContents([3, 2, 1, 1])),
...         (Colors.RED, ConfigurationColorContents([2, 1])),
...     ]),
...     n_clues=4,
...     n_misfires=1,
...     hand_size_rule=ConfigurationHandSize.VARIANT_6_3,
...     empty_clue_rule=ConfigurationEmptyClueRule.ALLOWED,
...     end_rule=ConfigurationEndRule.CROWNING_PIECE
... )
>>> print(cfg)
Deck: B [3, 2, 1, 1], R [2, 1].
Number of clues: 4.
Number of misfires: 1.
Clues rule: empty clues are allowed.
End rule: Crowning Piece.
```

EIGHT_COLORS = <Configuration: with sixth color, multicolor and colorless (10 cards e

STANDARD = <Configuration: standard>

W_MULTICOLOR = <Configuration: with normal multicolor (10 cards)>

W_MULTICOLOR_SHORT = <Configuration: with short multicolor (5 cards)>

W_SIXTH = <Configuration: with normal sixth color (10 cards)>

W_SIXTH_SHORT = <Configuration: with short sixth color (5 cards)>

colored() → str
Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

i_from_c(*c: hanabython.Modules.Color.Color*) → int
Finds index from a color (for example in `deck_array`).

Parameters **c** – a color.

Returns the corresponding index.

```
>>> Configuration.STANDARD.i_from_c(Colors.BLUE)
0
```

i_from_v(*v: int*) → int
Finds index from a value (for example in `deck_array`).

Parameters **v** – the value (typically 1 to 5).

Returns the corresponding index (typically 0 to 4).

```
>>> Configuration.STANDARD.i_from_v(1)
0
```

4.4 Clues

class hanabython.**Clue**(*x: Union[int, hanabython.Modules.Color.Color]*)
A clue.

Parameters **x** – the clue (value or color).

Variables **category** (*int*) – can be either Clue.VALUE or Clue.COLOR.

```
>>> clue = Clue(1)
>>> print(clue)
1
>>> clue.category == Clue.VALUE
True
>>> clue = Clue(Colors.RED)
>>> print(clue)
R
>>> clue.category == Clue.COLOR
True
```

COLOR = 1
Category for a clue by color.

VALUE = 0
Category for a clue by value.

colored() → str
Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

4.5 Cards

class hanabython.**Card**(**args*, ***kwargs*)

A card of Hanabi.

Parameters

- **c** (Color) – the color of the card.
- **v** (*int*) – the value of the card (usually between 1 and 5).
- **s** (*str*) – a short string representing the card. Must use one of the standard colors, cf. `Color.from_symbol()`.

You can provide either `c` and `v`, or `s`. The constructor accepts several types of syntax, as illustrated below.

```
>>> my_card = Card(c=Colors.BLUE, v=3)
>>> print(my_card)
B3
>>> my_card = Card(Colors.BLUE, 3)
>>> print(my_card)
B3
>>> my_card = Card(3, Colors.BLUE)
>>> print(my_card)
B3
>>> my_card = Card(s='B3')
>>> print(my_card)
B3
>>> my_card = Card('B3')
>>> print(my_card)
B3
>>> my_card = Card(s='3B')
>>> print(my_card)
B3
>>> my_card = Card('3B')
>>> print(my_card)
B3
```

N.B.: the string input works even if the v has several digits.

```
>>> my_card = Card('B42')
>>> print(my_card)
B42
>>> my_card = Card('51M')
>>> print(my_card)
M51
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

match(*clue: hanabython.Modules.Clue.Clue*) → bool
React to a clue.

Parameters **clue** – the clue.

Returns whether the card should be pointed when giving this clue.

```
>>> from hanabython import Colors
>>> card_blue = Card('B3')
>>> card_blue.match(Clue(Colors.BLUE))
True
>>> card_blue.match(Clue(Colors.RED))
False
>>> card_blue.match(Clue(3))
True
>>> card_blue.match(Clue(4))
False
>>> card_multi = Card('M3')
>>> card_multi.match(Clue(Colors.BLUE))
True
>>> card_colorless = Card('C3')
>>> card_colorless.match(Clue(Colors.BLUE))
False
```

class hanabython.**CardPublic**(*cfg: hanabython.Modules.Configuration.Configuration*)
The "public" part of a card.

An object of this class represents what is known by all players, including the owner of the card.

Parameters **cfg** – the configuration of the game.

Variables

- **can_be_c** (*np.array*) – a coefficient is True iff the card can be of the corresponding color.
- **can_be_v** (*np.array*) – a coefficient is True iff the card can be of the corresponding value.
- **yes_clued_c** (*np.array*) – a coefficient is True iff the card was explicitly clued as the corresponding color *and* it can be of this color (this precision is important for multicolor).
- **yes_clued_v** (*np.array*) – a coefficient is True iff the card was explicitly clued as value v.

```
>>> from hanabython import Configuration
>>> card = CardPublic(Configuration.EIGHT_COLORS)
>>> print(card)
BGRWYPMC 12345
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

match(*clue: hanabython.Modules.Clue.Clue*, *b: bool*) → None
React to a clue.

Updates the internal variables of the card.

Parameters

- **clue** – the clue.
- **b** – whether the card matches or not.

```
>>> from hanabython import Configuration
>>> cfg = Configuration.EIGHT_COLORS
>>> card = CardPublic(cfg)
>>> print(card)
BGRWYPMC 12345
>>> card.match(clue=Clue(Colors.RED), b=False)
>>> print(card)  #doctest: +NORMALIZE_WHITESPACE
BGWYPC 12345
>>> card.match(clue=Clue(Colors.BLUE), b=True)
>>> print(card)  #doctest: +NORMALIZE_WHITESPACE
B 12345
```

Let us try with the clues in the opposite order:

```
>>> from hanabython import Configuration
>>> card = CardPublic(Configuration.EIGHT_COLORS)
>>> print(card)
BGRWYPMC 12345
>>> card.match(clue=Clue(Colors.BLUE), b=True)
>>> print(card)  #doctest: +NORMALIZE_WHITESPACE
BM 12345
>>> card.match(clue=Clue(Colors.RED), b=False)
>>> print(card)  #doctest: +NORMALIZE_WHITESPACE
B 12345
```

Now with clues by value:

```
>>> from hanabython import Configuration
>>> card = CardPublic(Configuration.EIGHT_COLORS)
>>> print(card)
BGRWYPMC 12345
>>> card.match(clue=Clue(3), b=False)
>>> print(card)  #doctest: +NORMALIZE_WHITESPACE
BGRWYPMC 1245
>>> card.match(clue=Clue(5), b=True)
```

(continues on next page)

(continued from previous page)

```
>>> print(card)  #doctest: +NORMALIZE_WHITESPACE
BGRWYPMC 5
```

4.6 Hands

class hanabython.**Hand**(*source: Iterable[Union[hanabython.Modules.Card.Card, str]] = None*)

The hand of a player.

We use the same convention as in Board Game Arena: newest cards are on the left (i.e. at the beginning of the list) and oldest cards are on the right (i.e. at the end of the list).

Basically, a Hand is a list of Card objects. It can be constructed as such, or using a list of strings which will be automatically converted to cards.

Parameters **source** – an iterable used to construct the hand. N.B.: this parameter is mostly used for examples and tests. In contrast, at the beginning of a game, the hand should be initialized with no cards, because cards will be given one by one to the players during the initial dealing of hands.

```
>>> hand = Hand([Card('Y3'), Card('M1'), Card('B2'), Card('R4')])
>>> print(hand)
Y3 M1 B2 R4
>>> hand = Hand(['Y3', 'M1', 'B2', 'R4'])
>>> print(hand)
Y3 M1 B2 R4
```

colored() → str

Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

give(*k: int*) → hanabython.Modules.Card.Card

Give a card.

Parameters **k** – the position of the card in the hand (0 = newest).

Returns the card given.

```
>>> hand = Hand(['Y3', 'B1', 'M1', 'B2', 'R4'])
>>> card = hand.give(1)
>>> print(card)
B1
>>> print(hand)
Y3 M1 B2 R4
```

match(*clue: hanabython.Modules.Clue.Clue*) → List[bool]

React to a clue.

Parameters **clue** – the clue.

Returns a list of booleans. The *i*-th coefficient is *True* iff the *i*-th card of the hand matches the clue given.

```
>>> hand = Hand(['G2', 'Y3', 'M1', 'B2', 'R4'])
>>> hand.match(Clue(Colors.RED))
[False, False, True, False, True]
>>> hand.match(Clue(2))
[True, False, False, True, False]
```

receive(*card: hanabython.Modules.Card.Card*) → None
Receive a card.

Parameters **card** – the card received.

The card is added on the left, i.e. at the beginning of the list.

```
>>> hand = Hand(['Y3', 'M1', 'B2', 'R4'])
>>> hand.receive(Card('G2'))
>>> print(hand)
G2 Y3 M1 B2 R4
```

class hanabython.**HandPublic**(*cfg: hanabython.Modules.Configuration.Configuration*, *n_cards: int = 0*)
The "public" part of a hand.

An object of this class represents what is known by all players, including the owner of the hand.

We use the same convention as in Board Game Arena: newest cards are on the left (i.e. at the beginning of the list) and oldest cards are on the right (i.e. at the end of the list).

Basically, a HandPublic is a list of CardPublic objects.

Parameters

- **cfg** – the configuration of the game.
- **n_cards** – the number of cards in the hand. N.B.: this parameter is mostly used for examples and tests. In contrast, at the beginning of a game, the hand should be initialized with 0 cards, because cards will be given one by one to the players during the initial dealing of hands.

```
>>> hand = HandPublic(cfg=Configuration.STANDARD, n_cards=4)
>>> print(hand)
BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

give(*k: int*) → None
Give a card.

Parameters **k** – the position of the card in the hand (0 = newest).

The card is simply suppressed from the hand.

```
>>> hand = HandPublic(cfg=Configuration.STANDARD, n_cards=4)
>>> hand.match(clue=Clue(5), bool_list=[False, True, False, False])
>>> hand.match(clue=Clue(4), bool_list=[True, False, False, False])
>>> print(hand)  #doctest: +NORMALIZE_WHITESPACE
BGRWY 4  ,   BGRWY 5  ,  BGRWY 123 ,  BGRWY 123
>>> hand.give(1)
>>> print(hand)  #doctest: +NORMALIZE_WHITESPACE
BGRWY 4  ,  BGRWY 123 ,  BGRWY 123
```

match(*clue: hanabython.Modules.Clue.Clue, bool_list: List[bool]*)

React to a clue

Parameters

- **clue** – the clue.
- **bool_list** – a list of booleans. The *i*-th coefficient is *True* iff the *i*-th card of the hand matches the clue given.

Updates the internal variables of the hand.

```
>>> hand = HandPublic(cfg=Configuration.STANDARD, n_cards=4)
>>> hand.match(clue=Clue(3), bool_list=[False, True, False, False])
>>> print(hand)  #doctest: +NORMALIZE_WHITESPACE
BGRWY 1245 ,   BGRWY 3  , BGRWY 1245 , BGRWY 1245
>>> hand.match(clue=Clue(Colors.RED),
...            bool_list=[False, True, False, False])
>>> print(hand)  #doctest: +NORMALIZE_WHITESPACE
BGWY 1245 ,    R3     ,  BGWY 1245 ,  BGWY 1245
```

receive() → None

Receive a card.

An unknown card is added on the left, i.e. at the beginning of the list.

```
>>> hand = HandPublic(cfg=Configuration.STANDARD, n_cards=4)
>>> hand.match(clue=Clue(5), bool_list=[True, True, False, False])
>>> print(hand)  #doctest: +NORMALIZE_WHITESPACE
BGRWY 5  ,   BGRWY 5  , BGRWY 1234 , BGRWY 1234
>>> hand.receive()
>>> print(hand)  #doctest: +NORMALIZE_WHITESPACE
BGRWY 12345,   BGRWY 5  ,   BGRWY 5  , BGRWY 1234 , BGRWY 1234
```

4.7 Draw Pile

class hanabython.**DrawPile**(*cfg: hanabython.Modules.Configuration.Configuration*)

The draw pile of a game of Hanabi.

Parameters **cfg** – the configuration of the game.

At initialization, the draw pile is generated with the parameters in `cfg`, then it is shuffled.

Basically, a DrawPile is a list of cards. The top of the pile, where cards are drawn, is represented by the end of the list (not that we care much, but it could have an influence someday in some not-yet-implemented non-official variants).

```
>>> from hanabython import Configuration
>>> draw_pile = DrawPile(Configuration.STANDARD)
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

give() → Optional[hanabython.Modules.Card.Card]
Give the card from the top of pile.

Returns the card drawn. If the pile is empty, return None.

```
>>> from hanabython import Configuration
>>> draw_pile = DrawPile(cfg=Configuration.STANDARD)
>>> card = draw_pile.give()
>>> type(card)
<class 'hanabython.Modules.Card.Card'>
>>> while draw_pile.n_cards >= 1:
...     _ = draw_pile.give()
>>> print(draw_pile.give())
None
```

n_cards
Number of cards in the pile.

Returns the number of cards.

```
>>> from hanabython import Configuration
>>> draw_pile = DrawPile(Configuration.STANDARD)
>>> draw_pile.n_cards
50
```

class hanabython.**DrawPilePublic**(*cfg: hanabython.Modules.Configuration.Configuration*)
The public part of a draw pile.

An object of this class represents what is known by all players. In the normal version of the game and all official variants, it is only the number of cards left.

Parameters **cfg** – the configuration of the game.

```
>>> from hanabython import Configuration
>>> draw_pile = DrawPilePublic(cfg=Configuration.STANDARD)
>>> print(draw_pile)
50 cards left
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

give() → None
Give the card from the top of pile.

Updates the internal variables of the pile.

```
>>> from hanabython import Configuration
>>> draw_pile = DrawPilePublic(cfg=Configuration.STANDARD)
>>> print(draw_pile)
50 cards left
>>> while draw_pile.n_cards >= 2:
...     draw_pile.give()
>>> print(draw_pile)
1 card left
>>> draw_pile.give()
>>> print(draw_pile)
No card left
```

4.8 Discard Pile

class hanabython.**DiscardPile**(*cfg: hanabython.Modules.Configuration.Configuration*)
The discard pile in a game of Hanabi.

Parameters **cfg** – the configuration of the game.

Variables

- **chronological** (*list*) – a list a cards discarded, by chronological order.
- **array** (*np.array*) – each row represents a color, each column a card value. The coefficient is the number of copies of this card in the discard pile.
- **not_discarded** (*np.array*) – is equal to `Configuration.deck_array - array`. Number of copies left for each card (including everything except the discard pile: the draw pile, the players' hand and the board).
- **scorable** (*np.array*) – each row represents a color, each column a card value. The coefficient is True it is possible to have a such card on the board at the end of the game (whether it is already on the board or not). For example, if the two G4's are discarded, then G4 and G5 are not "scorable". Note that a 1 always is considered "scorable", whether it is on the board or not.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> print(discard_pile)
No card discarded yet
```

Check that scorable cards are counted correctly with unusual configurations:

```
>>> from hanabython import (Configuration, ConfigurationDeck,
...                         Colors, ConfigurationColorContents)
>>> discard_pile = DiscardPile(Configuration(
...     deck=ConfigurationDeck(contents=[
...         (Colors.BLUE, ConfigurationColorContents([3, 2, 1, 1])),
...         (Colors.RED, ConfigurationColorContents([2, 1])),
...     ])
... ))
>>> print(discard_pile)
No card discarded yet
>>> print(discard_pile.array)
[[0 0 0 0]
 [0 0 0 0]]
>>> print(discard_pile.not_discarded)
[[3 2 1 1]
 [2 1 0 0]]
>>> print(discard_pile.scorable)
[[ True  True  True  True]
 [ True  True False False]]
>>> print(discard_pile.max_score_possible)
6
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

colored_as_array() → str
Colored version of *str_as_array()*.

colored_compact_chronological() → str
Colored version of *str_compact_chronological()*.

colored_compact_factorized() → str
Colored version of *str_multi_line_compact()*.

colored_compact_ordered() → str
Colored version of *str_compact_ordered()*.

colored_multi_line() → str
Colored version of *str_multi_line()*.

colored_multi_line_compact() → str
Colored version of *str_multi_line_compact()*.

list_reordered
List of discarded cards, ordered by color and value.

Returns the list of discarded cards, by lexicographic order. The order on the colors is the one specified in cfg.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('R4'))
>>> discard_pile.receive(Card('B1'))
>>> discard_pile.list_reordered
[<Card: B1>, <Card: B3>, <Card: R4>]
```

max_score_possible
Maximum possible score, considering the discard pile.

Returns the maximum score that is still possible.

receive(*card*) → None
Receive a card.

Parameters **card** – the card discarded.

Update the internal variables of the discard pile.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('B2'))
>>> discard_pile.receive(Card('B3'))
>>> print(discard_pile)
B2 B3 B3
>>> print(discard_pile.not_discarded)
[[3 1 0 2 1]
 [3 2 2 2 1]
 [3 2 2 2 1]
 [3 2 2 2 1]
 [3 2 2 2 1]]
>>> print(discard_pile.scorable)
[[ True  True False False False]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]]
>>> print(discard_pile.max_score_possible)
22
```

str_as_array() → str
Convert to string in an array-style layout.

Returns a representation of the discard pile.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('R4'))
>>> discard_pile.receive(Card('B1'))
>>> print(discard_pile.str_as_array())
   1 2 3 4 5
B [1 0 1 0 0]
G [0 0 0 0 0]
R [0 0 0 1 0]
W [0 0 0 0 0]
Y [0 0 0 0 0]
```

str_compact_chronological() → str
Convert to string in a list-style layout, by chronological order.

Returns a representation of the discard pile.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('R4'))
>>> discard_pile.receive(Card('B1'))
>>> print(discard_pile.str_compact_chronological())
B3 R4 B1
```

str_compact_factorized() → str
Convert to nice string.

Returns a representation of the discard pile.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('R4'))
>>> discard_pile.receive(Card('B1'))
>>> print(discard_pile.str_compact_factorized())
B 1 3  R 4
```

str_compact_ordered() → str
Convert to string in a list-style layout, ordered by color and value.

Returns a representation of the discard pile.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('R4'))
>>> discard_pile.receive(Card('B1'))
>>> print(discard_pile.str_compact_ordered())
B1 B3 R4
```

str_multi_line() → str
Convert to nice string.

Returns a representation of the discard pile.

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('R4'))
>>> discard_pile.receive(Card('B1'))
>>> print(discard_pile.str_multi_line())
B1 B3
-
R4
-
-
```

str_multi_line_compact() → str
Convert to nice string.

Returns a representation of the discard pile. As of now, it is the one used for the standard method `__str__()` (this behavior might be modified in the future).

```
>>> from hanabython import Configuration
>>> discard_pile = DiscardPile(Configuration.STANDARD)
>>> discard_pile.receive(Card('B3'))
>>> discard_pile.receive(Card('R4'))
>>> discard_pile.receive(Card('B1'))
>>> print(discard_pile.str_multi_line_compact())
B1 B3
R4
```

4.9 Board

class hanabython.**Board**(*cfg: hanabython.Modules.Configuration.Configuration*)

The board (cards successfully played) in a game of Hanabi.

Parameters **cfg** – the configuration of the game.

Variables **altitude** (*np.array*) – indicates the highest card played in each color. E.g. with color `c` of index `i`, `altitude[i]` is the value of the highest card played in color `c`. The correspondence between colors and indexes is the one provided by `cfg`.

```
>>> from hanabython import Configuration
>>> board = Board(Configuration.STANDARD)
>>> print(board.altitude)
[0 0 0 0 0]
```

colored() → str

Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

colored_compact() → str

Colored version of *str_compact()*.

colored_fixed_space() → str

Colored version of *str_fixed_space()*.

colored_multi_line() → str

Colored version of *str_multi_line()*.

colored_multi_line_compact() → str

Colored version of *str_multi_line_compact()*.

score

The current score.

Returns the sum of the altitudes reached in all colors.

```
>>> from hanabython import Configuration
>>> cfg = Configuration.STANDARD
>>> board = Board(cfg)
>>> for s in ['G1', 'G2', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5']:
...     _ = board.try_to_play(Card(s))
>>> print(board.score)
7
```

str_compact() → str
Convert to string in "compact" layout.

Returns a representation of the board.

```
>>> from hanabython import Configuration
>>> board = Board(Configuration.STANDARD)
>>> for s in ['G1', 'G2', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5']:
...     _ = board.try_to_play(Card(s))
>>> print(board.str_compact())
G1 G2 Y1 Y2 Y3 Y4 Y5
```

str_fixed_space() → str
Convert to string in "fixed-space" layout.

Returns a representation of the board.

```
>>> from hanabython import Configuration
>>> board = Board(Configuration.STANDARD)
>>> for s in ['G1', 'G2', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5']:
...     _ = board.try_to_play(Card(s))
>>> print(board.str_fixed_space())
B -         G 1 2       R -         W -         Y 1 2 3 4 5
```

str_multi_line() → str
Convert to string in "multi-line" layout.

Returns a representation of the board.

```
>>> from hanabython import Configuration
>>> board = Board(Configuration.STANDARD)
>>> for s in ['G1', 'G2', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5']:
...     _ = board.try_to_play(Card(s))
>>> print(board.str_multi_line())
-
G1 G2
-
-
Y1 Y2 Y3 Y4 Y5
```

str_multi_line_compact() → str
Convert to string in "compact multi-line" layout.

Returns a representation of the board.

```
>>> from hanabython import Configuration
>>> board = Board(Configuration.STANDARD)
>>> for s in ['G1', 'G2', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5']:
...     _ = board.try_to_play(Card(s))
>>> print(board.str_multi_line_compact())
```

(continues on next page)

(continued from previous page)

```
G1 G2
Y1 Y2 Y3 Y4 Y5
```

try_to_play(*card: hanabython.Modules.Card.Card*) → bool
Try to play a card on the board.

Parameters **card** – the card.

Returns True if the card is successfully played on the board, False otherwise (i.e. if it leads to a misfire).

```
>>> from hanabython import Configuration, Card
>>> board = Board(Configuration.STANDARD)
>>> for s in ['B1', 'B2', 'Y1', 'Y3', 'B1']:
...     board.try_to_play(Card(s))
True
True
True
False
False
>>> print(board.str_compact())
B1 B2 Y1
```

4.10 Actions

class hanabython.**Action**(*category: int*)
An action performed by a player (Throw, Play a card, Clue or Forfeit).

In the end-user interfaces (including methods `colored`), "throw" should be called "discard" and "play a card" can be called "play" (to be consistent with the official rules). In the code however, we prefer to use "throw" (to distinguish from other forms of discards, for example after a misfire) and "play a card" (to distinguish from simply playing in general).

Parameters **category** – can be *Action.THROW*, *Action.PLAY_CARD*, *Action.CLUE* or *Action.FORFEIT*.

Generally, only subclasses are instantiated. Cf. `ActionThrow`, `ActionPlayCard`, *ActionClue* and *ActionForfeit*.

CATEGORIES = {0, 1, 2, 3}
Possibles categories of action.

CLUE = 2

FORFEIT = 3

PLAY_CARD = 1

THROW = 0

class hanabython.**ActionClue**(*i: int*, *clue: hanabython.Modules.Clue.Clue*)
An action of a player: give a clue.

Parameters

- **i** – the relative position of the concerned player (i.e. 1 for next player, 2 for second next player, etc.).
- **clue** – the clue.

```
>>> from hanabython import Colors
>>> action = ActionClue(i=1, clue=Clue(2))
>>> print(action)
Clue 2 to player in relative position 1
>>> action = ActionClue(i=2, clue=Clue(Colors.BLUE))
>>> print(action)
Clue B to player in relative position 2
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

class hanabython.**ActionForfeit**
An action of a player: forfeit (lose the game immediately).

```
>>> action = ActionForfeit()
>>> print(action)
Forfeit
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

4.11 Players

class hanabython.**Player**(*name: str*)
A player for Hanabi.

Parameters **name** – the name of the player.

To define a subclass, the only real requirement is to implement the function *choose_action()*.

```
>>> antoine = Player('Antoine')
>>> print(antoine)
Antoine
```

choose_action() → hanabython.Modules.Action.Action
Choose an action.

Returns the action chosen by the player.

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

receive_action_illegal(*s: str*) → None
Receive a message: the action chosen is illegal.

Parameters **s** – a message explaining why the action is illegal.

receive_action_legal() → None
Receive a message: the action chosen is legal.

receive_begin_dealing() → None
Receive a message: the initial dealing of hands begins.

receive_end_dealing() → None
Receive a message: the initial dealing of hands is over.

The hands themselves are not communicated in this message. Drawing cards, including for the initial hands, is always handled by *receive_i_draw()* or *receive_partner_draws()*.

receive_game_exhausted(*score: int*) → None
Receive a message: the game is over and is neither really lost (misfires, forfeit) nor a total victory (maximal score). Typically, this happens a bit after the deck ran out of cards (it depends on the end-of-game rule that is used).

Parameters **score** – the final score.

receive_i_draw() → None
Receive a message: this player tries to draw a card.

A card is actually drawn only if the draw pile is not empty.

receive_init(*cfg: hanabython.Modules.Configuration.Configuration, player_names: List[str]*) → None
Receive a message: the game starts.

Parameters

- **cfg** – the configuration of the game.
- **player_names** – the names of the players, rotated so that this player corresponds to index 0.

receive_lose(*score: int*) → None
Receive a message: the game is lost (misfires or forfeit).

Parameters **score** – the final score (0 in that case).

receive_partner_draws(*i_active: int*, *card: hanabython.Modules.Card.Card*) → None
Receive a message: another player tries to draw a card.

A card is actually drawn only if the draw pile is not empty.

Parameters

- **i_active** – the position of the player who draws (relatively to this player).
- **card** – the card drawn.

receive_remaining_turns(*remaining_turns: int*) → None
Receive a message: the number of remaining turns is now known.

This happens with the normal rule for end of game: as soon as the discard pile is empty, we know how many turns are left. N.B.: the word "turn" means that one player gets to play (not all of them).

Parameters **remaining_turns** – the number of turns left.

receive_someone_clues(*i_active: int, i_clued: int, clue: hanabython.Modules.Clue.Clue, bool_list: List[bool]*) → None
Receive a message: a player gives a clue to another one.

It is not necessary to check whether this action is legal: the *Game* will only send this message when it is the case.

Parameters

- **i_active** – the position of the player who gives the clue (relatively to this player).
- **i_clued** – the position of the player who receives the clue (relatively to this player).
- **clue** – the clue.
- **bool_list** – a list of boolean that indicates what cards match the clue given.

receive_someone_forfeits(*i_active: int*) → None
Receive a message: a player forfeits.

Parameters **i_active** – the position of the player who forfeits (relatively to this player).

receive_someone_plays_card(*i_active: int*, *k: int*, *card: hanabython.Modules.Card.Card*) → None
Receive a message: a player tries to play a card on the board.

This can be a success or a misfire.

Parameters

- **i_active** – the position of the player who plays the card (relatively to this player).
- **k** – position of the card in the hand.
- **card** – the card played.

receive_someone_throws(*i_active: int*, *k: int*, *card: hanabython.Modules.Card.Card*) → None
Receive a message: a player throws (discards a card willingly).

It is not necessary to check whether this action is legal: the *Game* will only send this message when it is the case.

Parameters

- **i_active** – the position of the player who throws (relatively to this player).
- **k** – position of the card in the hand.
- **card** – the card thrown.

receive_turn_begin() → None
Receive a message: the turn of the player begins.

receive_turn_finished() → None
Receive a message: the turn of the player is finished.

receive_win(*score: int*) → None
Receive a message: the game is won (total victory).

Parameters **score** – the final score.

class hanabython.**PlayerPuppet**(*name*, *speak=False*)
A player for Hanabi that serves only for testing purposes.

Parameters **speak** – if True, then each time this player receives a message, she prints a acknowledgement.

Variables **next_action** (Action) – this variable makes it possible to control this player's action.

```
>>> from hanabython import ActionThrow
>>> antoine = PlayerPuppet('Antoine', speak=True)
>>> antoine.next_action = ActionThrow(k=4)
>>> _ = antoine.choose_action()
Antoine: Choose an action
Antoine: action = Discard card in position 5
```

choose_action() → hanabython.Modules.Action.Action

Returns the value of `next_action`

receive_action_illegal(*s: str*) → None
Receive a message: the action chosen is illegal.

Parameters **s** – a message explaining why the action is illegal.

receive_action_legal() → None
Receive a message: the action chosen is legal.

receive_begin_dealing() → None
Receive a message: the initial dealing of hands begins.

receive_end_dealing() → None
Receive a message: the initial dealing of hands is over.

The hands themselves are not communicated in this message. Drawing cards, including for the initial hands, is always handled by *receive_i_draw()* or *receive_partner_draws()*.

receive_game_exhausted(*score: int*) → None
Receive a message: the game is over and is neither really lost (misfires, forfeit) nor a total victory (maximal score). Typically, this happens a bit after the deck ran out of cards (it depends on the end-of-game rule that is used).

Parameters **score** – the final score.

receive_i_draw() → None
Receive a message: this player tries to draw a card.

A card is actually drawn only if the draw pile is not empty.

receive_init(*cfg: hanabython.Modules.Configuration.Configuration, player_names: List[str]*) → None
Receive a message: the game starts.

Parameters

- **cfg** – the configuration of the game.

- **player_names** – the names of the players, rotated so that this player corresponds to index 0.

receive_lose(*score: int*) → None
Receive a message: the game is lost (misfires or forfeit).

Parameters **score** – the final score (0 in that case).

receive_partner_draws(*i_active: int*, *card: hanabython.Modules.Card.Card*) → None
Receive a message: another player tries to draw a card.

A card is actually drawn only if the draw pile is not empty.

Parameters

- **i_active** – the position of the player who draws (relatively to this player).
- **card** – the card drawn.

receive_remaining_turns(*remaining_turns: int*) → None
Receive a message: the number of remaining turns is now known.

This happens with the normal rule for end of game: as soon as the discard pile is empty, we know how many turns are left. N.B.: the word "turn" means that one player gets to play (not all of them).

Parameters **remaining_turns** – the number of turns left.

receive_someone_clues(*i_active: int, i_clued: int, clue: hanabython.Modules.Clue.Clue, bool_list: List[bool]*) → None
Receive a message: a player gives a clue to another one.

It is not necessary to check whether this action is legal: the *Game* will only send this message when it is the case.

Parameters

- **i_active** – the position of the player who gives the clue (relatively to this player).
- **i_clued** – the position of the player who receives the clue (relatively to this player).
- **clue** – the clue.
- **bool_list** – a list of boolean that indicates what cards match the clue given.

receive_someone_forfeits(*i_active: int*) → None
Receive a message: a player forfeits.

Parameters **i_active** – the position of the player who forfeits (relatively to this player).

receive_someone_plays_card(*i_active: int*, *k: int*, *card: hanabython.Modules.Card.Card*) → None
Receive a message: a player tries to play a card on the board.

This can be a success or a misfire.

Parameters

- **i_active** – the position of the player who plays the card (relatively to this player).
- **k** – position of the card in the hand.
- **card** – the card played.

receive_someone_throws(*i_active: int*, *k: int*, *card: hanabython.Modules.Card.Card*) → None
Receive a message: a player throws (discards a card willingly).

It is not necessary to check whether this action is legal: the *Game* will only send this message when it is the case.

Parameters

- **i_active** – the position of the player who throws (relatively to this player).
- **k** – position of the card in the hand.
- **card** – the card thrown.

receive_turn_begin() → None
Receive a message: the turn of the player begins.

receive_turn_finished() → None
Receive a message: the turn of the player is finished.

receive_win(*score: int*) → None
Receive a message: the game is won (total victory).

Parameters **score** – the final score.

class hanabython.**PlayerBase**(*name: str*)
A player for Hanabi with basic features.

This class is meant to serve as a mother class for most AIs and interface for human players. It provides all basic features, such as keeping track of the number of cards in the draw pile, the cards in the other players' hands, the clues given, etc.

Note that all the variables are "personal" to this player: the *Game* does not share access to its internal variables with the players.

Note also that most methods are not supposed to work before *receive_init()* is run at least once, which initializes all the variables for a new game.

Parameters **name** (*str*) – the name of the player.

Variables

- **player_names** (*list*) – a list of strings, each with a player's name. By convention, the list is always rotated to that this player has position 0, the next player has position 1, etc.
- **n_players** (*int*) – the number of players.
- **cfg** (Configuration) – the configuration of the game.
- **board** (Board) – the board.
- **draw_pile** (DrawPilePublic) – the draw pile.
- **discard_pile** (DiscardPile) – the discard pile.
- **n_clues** (*int*) – the number of clues left.
- **n_misfires** (*int*) – the number of misfires (initially 0).
- **hand_size** (*int*) – the initial hand size.
- **hands** (*list*) – a list of Hand objects. The hand in position 0, corresponding to this player, is never updated because the player does not know what she has.
- **hands_public** (*list*) – a list of HandPublic objects. This allow the player to keep track, not only of her own clues, but also of the clues received by her partners.
- **remaining_turns** (*int*) – the number of remaining turns (once the draw pile is empty, in the normal rule for end of game). As long as the draw pile contains cards, this variable is *None*.

- **recent_events** (*str*) – things that happened "recently" (typically, since this player's last turn). Subclasses may typically print and/or empty this variable from time to time. Cf. *log()*.
- **dealing_is_ongoing** (*bool*) – True only during the initial dealing of hands. Avoid useless verbose messages in `recent events`. Cf. *log()*.
- **display_width** (*int*) – the width of the display on the terminal (in number of characters).

```
>>> antoine = PlayerBase(name='Antoine')
```

colored() → str
Colored version of `__str__()`.

In the subclasses, the principle is to override only this method. `__str__()` is automatically defined as the uncolored version of the same string, and `__repr__()` as the same with the class name added.

Of course, it is also possible to override `__str__()` and/or `__repr__()` if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

colored_hands() → str
A string used to display the hands of all players.

Returns the string (whose width is usually `display_width`, except maybe in the end of game when the hands are shorter).

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.demo_game()
>>> from hanabython import uncolor
>>> print(uncolor(antoine.colored_hands()))  #doctest: +NORMALIZE_WHITESPACE
Antoine
BGRWY 12345, BGRWY 2345 ,   BGRWY 1  ,   BGRWY 1  , BGRWY 2345
<BLANKLINE>
Donald X
    Y2     ,     R1     ,     R3     ,     G3     ,     Y4
BGRWY 2345 ,   BGRWY 1  , BGRWY 2345 , BGRWY 2345 , BGRWY 2345
<BLANKLINE>
Uwe
    G4     ,     B4     ,     W4     ,     G5     ,     W1
BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345
```

demo_game() → None
Mimic the beginning of a game.

This method is meant to be used for tests and examples.

```
>>> from hanabython import uncolor
>>> antoine = PlayerBase('Antoine')
>>> antoine.demo_game()
>>> print(uncolor(antoine.recent_events))
Configuration
-------------
Deck: normal.
Number of clues: 8.
```

(continues on next page)

(continued from previous page)

```
Number of misfires: 3.
Clues rule: empty clues are forbidden.
End rule: normal.
<BLANKLINE>
First moves
-----------
The game begins.
Antoine clues Donald X about 1.
Donald X clues Antoine about 1.
Uwe discards R3.
Uwe draws G4.
Antoine plays W1.
Antoine draws a card.
<BLANKLINE>
```

log(*o: object*) → None
Log events for the player.

Parameters **o** – an object. The method adds *str(o)* to the variable `recent_events`, except during the initial dealing of cards (to avoid useless messages about each card dealt). Do not forget the end-of-line character when relevant (it is not added automatically).

This is for the player herself: it is used, in particular, in the subclass *PlayerHumanText* to inform the player of the most recent events in a relatively user-friendly form.

N.B.: this is totally different from the use of the standard package `logging`, which is essentially used for debugging purposes.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.log_init()
>>> antoine.log('Something happens.\n')
>>> antoine.dealing_is_ongoing = True
>>> antoine.log('Many useless messages.\n')
>>> antoine.dealing_is_ongoing = False
>>> antoine.log('Something else happens.\n')
>>> print(antoine.recent_events)
Something happens.
Something else happens.
<BLANKLINE>
>>> antoine.log_forget()
>>> antoine.log('Something new happens.')
>>> print(antoine.recent_events)
Something new happens.
```

log_forget() → None
Forget old events (during the game).

Empties `recent_events`. In this base class, this method has the same implementation as *log_init()*, but it could be different in some subclasses.

log_init() → None
Initialize the log process (at the beginning of a game).

Empties `recent_events`.

receive_begin_dealing() → None
The log is turned off to avoid having a message for each card dealt. Cf. *log()*.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.receive_begin_dealing()
>>> antoine.dealing_is_ongoing
True
>>> antoine.receive_end_dealing()
>>> antoine.dealing_is_ongoing
False
```

receive_end_dealing() → None
The log is turned back on. Cf. *log()* and *receive_begin_dealing()*.

receive_game_exhausted(*score: int*) → None
We just log the event for the player.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_game_exhausted(score=23)
>>> print(antoine.recent_events)
Antoine's team has reached the end of the game.
Score: 23.
<BLANKLINE>
```

receive_i_draw() → None
If there are cards in the draw pile, a card is drawn. There is one card less in `drawpile`, and one more in this player's hand in `hands_public`.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> for _ in range(4):
...     antoine.receive_i_draw()
>>> print(antoine.draw_pile)
46 cards left
>>> print(antoine.hands_public[0])
BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345
```

If there are no cards in the draw pile, nothing happens.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> for _ in range(50):
...     antoine.receive_i_draw()
>>> len(antoine.hands_public[0])
50
>>> print(antoine.draw_pile)
No card left
>>> antoine.receive_i_draw()
>>> len(antoine.hands_public[0])
50
>>> print(antoine.draw_pile)
No card left
```

receive_init(*cfg: hanabython.Modules.Configuration.Configuration, player_names: List[str]*) → None

Initialize all the instance variables for a new game.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> print(repr(antoine))  #doctest: +NORMALIZE_WHITESPACE
<PlayerBase:
************************* board *************************
B -         G -         R -         W -         Y -
************************** cfg **************************
Deck: normal.
Number of clues: 8.
Number of misfires: 3.
Clues rule: empty clues are forbidden.
End rule: normal.
****************** dealing_is_ongoing *******************
False
********************** discard_pile **********************
No card discarded yet
********************** display_width **********************
63
*********************** draw_pile ************************
50 cards left
*********************** hand_size ************************
5
************************* hands **************************
[<Hand: >, <Hand: >]
********************** hands_public **********************
[<HandPublic: >, <HandPublic: >]
************************ n_clues *************************
8
********************** n_misfires ************************
0
********************** n_players *************************
2
************************* name ***************************
Antoine
********************** player_names **********************
['Antoine', 'Donald X']
********************** recent_events *********************
Configuration
-------------
Deck: normal.
Number of clues: 8.
Number of misfires: 3.
Clues rule: empty clues are forbidden.
End rule: normal.
<BLANKLINE>
******************** remaining_turns *********************
None
>
```

receive_lose(*score: int*) → None

We just log the event for the player.

```
>>> antoine = PlayerBase('Antoine')
```

(continues on next page)

(continued from previous page)

```
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_lose(score=0)
>>> print(antoine.recent_events)
Antoine's team loses.
Score: 0.
<BLANKLINE>
```

receive_partner_draws(*i_active: int*, *card: hanabython.Modules.Card.Card*) → None

If there are cards in the draw pile, a card is drawn. There is one card less in `drawpile`, one more in the partner's hand in `hands_public`, and the actual card is added in the partner's hand in `hands`.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> for s in ['B1', 'G3', 'Y1', 'W1', 'R5']:
...     antoine.receive_partner_draws(i_active=1, card=Card(s))
>>> print(antoine.draw_pile)
45 cards left
>>> print(antoine.hands[1])
R5 W1 Y1 G3 B1
>>> print(antoine.hands_public[1])
BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345
```

If there are no cards in the draw pile, nothing happens.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> for _ in range(50):
...     antoine.receive_partner_draws(i_active=1, card=Card('B1'))
>>> len(antoine.hands[1])
50
>>> len(antoine.hands_public[1])
50
>>> print(antoine.draw_pile)
No card left
>>> antoine.receive_i_draw()
>>> len(antoine.hands[1])
50
>>> len(antoine.hands_public[1])
50
>>> print(antoine.draw_pile)
No card left
```

receive_remaining_turns(*remaining_turns: int*) → None

We update `remaining_turns` and log the event for the player.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_remaining_turns(remaining_turns=2)
>>> antoine.remaining_turns
2
```

(continues on next page)

(continued from previous page)

```
>>> print(antoine.recent_events)
2 turns remaining!
<BLANKLINE>
```

receive_someone_clues(*i_active: int, i_clued: int, clue: hanabython.Modules.Clue.Clue, bool_list: List[bool]*) → None

We remove a clue chip, and we update the clued player's hand in `hands_public`.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> for s in ['B1', 'G3', 'Y1', 'W1', 'R5']:
...     antoine.receive_partner_draws(i_active=1, card=Card(s))
>>> print(antoine.hands[1])
R5 W1 Y1 G3 B1
>>> antoine.n_clues
8
>>> antoine.receive_someone_clues(
...     i_active=0, i_clued=1, clue=Clue(1),
...     bool_list=[False, True, True, False, True])
>>> print(antoine.hands_public[1])  #doctest: +NORMALIZE_WHITESPACE
BGRWY 2345 ,   BGRWY 1  ,   BGRWY 1  , BGRWY 2345 ,   BGRWY 1
>>> antoine.n_clues
7
```

receive_someone_forfeits(*i_active: int*) → None

We just log the event for the player.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_someone_forfeits(i_active=1)
>>> print(antoine.recent_events)
Donald X forfeits.
<BLANKLINE>
```

receive_someone_plays_card(*i_active: int, k: int, card: hanabython.Modules.Card.Card*) → None

If the action succeeds, the card goes on the board.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> for s in ['B1', 'G3', 'Y1', 'W1', 'R5']:
...     antoine.receive_partner_draws(i_active=1, card=Card(s))
>>> print(antoine.hands[1])
R5 W1 Y1 G3 B1
>>> antoine.receive_someone_plays_card(i_active=1, k=1, card=Card('W1'))
>>> print(antoine.hands[1])
R5 Y1 G3 B1
>>> print(antoine.hands_public[1])
BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345
>>> print(antoine.board)  #doctest: +NORMALIZE_WHITESPACE
B -         G -         R -         W 1         Y -
```

If the action fails, the card goes in the discard pile and players get a misfire.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> for s in ['B1', 'G3', 'Y1', 'W1', 'R5']:
...     antoine.receive_partner_draws(i_active=1, card=Card(s))
>>> print(antoine.hands[1])
R5 W1 Y1 G3 B1
>>> antoine.receive_someone_plays_card(i_active=1, k=0, card=Card('R5'))
>>> print(antoine.hands[1])
W1 Y1 G3 B1
>>> print(antoine.hands_public[1])
BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345
>>> print(antoine.board)  #doctest: +NORMALIZE_WHITESPACE
B -         G -         R -         W -         Y -
>>> print(antoine.discard_pile)
R5
>>> antoine.n_misfires
1
```

receive_someone_throws(*i_active: int*, *k: int*, *card: hanabython.Modules.Card.Card*) → None

The card goes in the discard pile, and players regain a clue chip.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.n_clues = 3
>>> for s in ['B1', 'G3', 'Y1', 'W1', 'R5']:
...     antoine.receive_partner_draws(i_active=1, card=Card(s))
>>> print(antoine.hands[1])
R5 W1 Y1 G3 B1
>>> antoine.receive_someone_throws(i_active=1, k=4, card=Card('B1'))
>>> print(antoine.hands[1])
R5 W1 Y1 G3
>>> print(antoine.hands_public[1])
BGRWY 12345, BGRWY 12345, BGRWY 12345, BGRWY 12345
>>> print(antoine.discard_pile)
B1
>>> antoine.n_clues
4
```

receive_win(*score: int*) → None

We just log the event for the player.

```
>>> antoine = PlayerBase('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_win(score=25)
>>> print(antoine.recent_events)
Antoine's team wins!
Score: 25.
<BLANKLINE>
```

class hanabython.**PlayerHumanText**(*name: str*, *ipython=False*)

User interface for a human player in text mode (terminal or notebook).

Parameters **ipython** – use *True* when using the player in a notebook. This fixes a problem between `clear_output` and `input`.

```
>>> antoine = PlayerHumanText('Antoine', ipython=True)
```

choose_action() → hanabython.Modules.Action.Action
The human player gets to choose an action.

receive_action_illegal(*s: str*) → None
Receive a message: the action chosen is illegal.

Parameters **s** – a message explaining why the action is illegal.

receive_action_legal() → None
We forget the previous events.

```
>>> from hanabython import Configuration
>>> antoine = PlayerHumanText('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.log('Donald does something.')
>>> antoine.recent_events
'Donald does something.'
>>> # Here, Antoine would choose his own action. Then...
>>> antoine.receive_action_legal()
>>> antoine.log("Antoine's action has such and such consequences.")
>>> antoine.recent_events
"Antoine's action has such and such consequences."
```

receive_game_exhausted(*score: int*) → None
We print and forget the recent events.

```
>>> from hanabython import Configuration
>>> antoine = PlayerHumanText('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_game_exhausted(score=23)
Antoine's team has reached the end of the game.
Score: 23.
<BLANKLINE>
>>> antoine.recent_events
''
```

receive_lose(*score: int*) → None
We print and forget the recent (unfortunate) events.

```
>>> from hanabython import Configuration
>>> antoine = PlayerHumanText('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_lose(score=0)
Antoine's team loses.
Score: 0.
<BLANKLINE>
>>> antoine.recent_events
''
```

receive_turn_begin() → None
We pause, then we inform the player of the most recent events.

receive_turn_finished() → None
We inform the player of the most recent events, i.e. the consequences of her actions. Then we pause (unless this string was empty). Finally, we forget these recent events.

receive_win(*score: int*) → None
We print and forget the recent (cheerful) events.

```
>>> from hanabython import Configuration
>>> antoine = PlayerHumanText('Antoine')
>>> antoine.receive_init(Configuration.STANDARD,
...                      player_names=['Antoine', 'Donald X'])
>>> antoine.log_forget()
>>> antoine.receive_win(score=25)
Antoine's team wins!
Score: 25.
<BLANKLINE>
>>> antoine.recent_events
''
```

4.12 Game

class hanabython.**Game**(*players: List[hanabython.Modules.Player.Player], cfg: hanabython.Modules.Configuration.Configuration = <Configuration: standard>*)
A game of Hanabi.

Parameters

- **cfg** – the configuration.
- **players** – the list of players. They will play in this order, starting with the first player in this list.

Variables

- **n_players** (*int*) – the number of players.
- **board** (Board) – the board.
- **draw_pile** (DrawPile) – the draw pile.
- **discard_pile** (DiscardPile) – the discard pile.
- **n_clues** (*int*) – the number of clue chips that players currently have.
- **n_misfires** (*int*) – the number of misfires chips that players currently have.
- **hand_size** (*int*) – the initial size of the hands.
- **hands** (*list*) – a list of *Hand* objects (in the same order as players).
- **remaining_turns** (*int*) – the number of remaining turns (once the draw pile is empty, in the normal rule for end of game). As long as the draw pile contains cards, this variable is *None*.
- **b_lose** (*bool*) – the game is lost.
- **b_win** (*bool*) – the game is won.
- ***i_active*** (*int*) – the index of the active player.

- **active** (Player) – the active player. It is automatically updated when *i_active* is updated.

```
>>> game = Game(players=[PlayerHumanText('Antoine'),
...                      PlayerHumanText('Donald X')])
```

ATTEMPTS_BEFORE_FORFEIT = 100
Number of attempts that a player has to choose her action. If she provides illegal actions as many times, she is automatically considered to forfeit (and this issues a warning).

check_game_exhausted() → bool
Check if the game is exhausted.

Typically, the game end by exhaustion a bit after the deck ran out of cards (the exact moment depends on the end-of-game rule that is used).

This method is called at the beginning of each player's turn.

We do not check here whether the current score is equal to the maximum score still possible (considering what is discarded), which would also end the game. This verification is done in *play()*.

Returns True iff the game must end.

If the normal end-of-game rule is used, and remaining_turns is an integer: it is updated, then the end-of-game condition is checked.

```
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.players[1].speak = True
>>> game.remaining_turns = 1
>>> # Here, previous player would play her turn.. Then...
>>> game.check_game_exhausted()
Donald X: The number of remaining turns is now known.
Donald X: remaining_turns = 0
True
>>> print(game.remaining_turns)
0
```

If the normal end-of-game rule is used, and remaining_turns is *None*: it is not updated (the final countdown has not started).

```
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.players[1].speak = True
>>> game.check_game_exhausted()
False
>>> print(game.remaining_turns)
None
```

If "Crowning piece" rule is used: if the active player has no card in hand, the game is over.

```
>>> game = Game(
...     players=[PlayerPuppet('Antoine'),
...              PlayerPuppet('Donald X'),
...              PlayerPuppet('Uwe')],
...     cfg=Configuration(end_rule=ConfigurationEndRule.CROWNING_PIECE)
... )
```

(continues on next page)

(continued from previous page)

```
>>> game.players[1].speak = True
>>> game.i_active = 1
>>> game.check_game_exhausted()
True
>>> game.hands[1].receive(Card('B1'))
>>> game.check_game_exhausted()
False
```

colored() → str
Colored version of __str__().

In the subclasses, the principle is to override only this method. __str__() is automatically defined as the uncolored version of the same string, and __repr__() as the same with the class name added.

Of course, it is also possible to override __str__() and/or __repr__() if a different behavior is desired.

Returns a string representing the object, possibly with ANSI escape codes to add colors and style.

Return type str

deal() → None
Deal the initial hands.

i_active should be -1 before dealing and will be -1 at the end (modulo the number of players).

```
>>> game = Game(players=[PlayerHumanText('Antoine'),
...                      PlayerHumanText('Donald X'),
...                      PlayerHumanText('Uwe')])
>>> game.i_active = -1
>>> game.deal()
>>> [len(hand) for hand in game.hands]
[5, 5, 5]
>>> game.i_active
2
```

draw() → None
The active player draws a card.

- Draw a card and put it in hand (unless the discard pile is empty).
- If the discard pile becomes empty, launch countdown for end of game by setting variable remaining_turns to value n_players + 1. It will be decremented at the beginning of next player's turn (before testing the end-of-game condition). Cf. *check_game_exhausted()*.

```
>>> game = Game(players=[PlayerHumanText('Antoine'),
...                      PlayerHumanText('Donald X'),
...                      PlayerHumanText('Uwe')])
>>> game.i_active = -1
>>> for _ in range(50):
...     game.i_active += 1
...     game.draw()
>>> [len(hand) for hand in game.hands]
[17, 17, 16]
>>> game.i_active += 1
>>> print(game.draw())
None
```

(continues on next page)

(continued from previous page)

```
>>> game.remaining_turns
4
```

execute_action (*action: hanabython.Modules.Action.Action*) → bool
Execute the action (by the active player).

Parameters **action** – the action.

Returns True iff the action is legal. If not, it will be necessary to choose another action.

This method dispatches to the auxiliary methods *execute_clue()*, *execute_forfeit()*, *execute_play_card()* and *execute_throw()*. Each of these methods has the responsability to:

- Check if the action is legal,
- Inform the active player whether it is the case or not,
- Perform the action,
- Update the relevant variables, in particular `b_lose` and `b_win`.
- Inform all players of the result of the action,
- Make the active player draw a new card if necessary,
- Return the boolean stating whether the action is legal.

execute_clue (*i_clued: int*, *clue: hanabython.Modules.Clue.Clue*) → bool
Execute the action: give a clue.

Parameters

- **i_clued** – the index of the player who receives the clue.
- **clue** – the clue.

Returns True iff the action is legal.

```
>>> import random
>>> random.seed(0)
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')],
...             cfg=Configuration.W_MULTICOLOR)
>>> game.i_active = -1
>>> game.deal()
>>> print(game.hands[2])
G2 W1 W1 B1 Y4
>>> game.players[1].speak = True
>>> game.i_active = 1
>>> game.n_clues = 0
>>> game.execute_clue(2, Clue(1))
Donald X: The action chosen is illegal.
Donald X: You cannot give a clue because you have do not have any clue chip.
False
>>> game.n_clues = 3
>>> game.execute_clue(1, Clue(1))
Donald X: The action chosen is illegal.
Donald X: You cannot give a clue to yourself.
False
>>> game.execute_clue(2, Clue(6))
```

(continues on next page)

(continued from previous page)

```
Donald X: The action chosen is illegal.
Donald X: This value does not exist: 6.
False
>>> game.execute_clue(2, Clue(Colors.COLORLESS))
Donald X: The action chosen is illegal.
Donald X: This color is not in the deck: C.
False
>>> game.execute_clue(2, Clue(Colors.MULTICOLOR))
Donald X: The action chosen is illegal.
Donald X: You cannot clue this color: M.
False
>>> game.execute_clue(2, Clue(3))
Donald X: The action chosen is illegal.
Donald X: You cannot give this clue because it does not correspond to any␣
↪card.
False
>>> game.execute_clue(2, Clue(1))
Donald X: The action chosen is legal.
Donald X: A player gives a clue to another one.
Donald X: i_active = 0
Donald X: i_clued = 1
Donald X: clue = 1
Donald X: bool_list = [False, True, True, True, False]
True
>>> game.n_clues
2
```

execute_forfeit() → bool

Execute the action: forfeit.

Returns True (meaning that this action is always legal).

```
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.players[1].speak = True
>>> game.i_active = 1
>>> is_legal = game.execute_forfeit()
Donald X: The action chosen is legal.
Donald X: A player forfeits.
Donald X: i_active = 0
>>> is_legal
True
>>> game.b_lose
True
>>> game.i_active = 2
>>> is_legal = game.execute_forfeit()
Donald X: A player forfeits.
Donald X: i_active = 1
>>> is_legal
True
>>> game.b_lose
True
```

execute_play_card(*k: int*) → bool

Execute the action: try to play a card.

Parameters **k** – the index of the card in the active player's hand.

Returns True (meaning that this action is always legal).

The action can fail, in the sense that it leads to a misfire, but it is legal anyway. If it leads to the last misfire, then the players lose:

```
>>> import random
>>> random.seed(0)
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.i_active = -1
>>> game.deal()
>>> game.i_active = 2
>>> game.n_misfires = 2
>>> print(game.hands[2])
B4 W4 G5 W1 R3
>>> game.players[1].speak = True
>>> is_legal = game.execute_play_card(2)
Donald X: A player tries to play a card on the board.
Donald X: i_active = 1
Donald X: k = 2
Donald X: card = G5
>>> is_legal
True
>>> print(game.board)  #doctest: +NORMALIZE_WHITESPACE
B -         G -         R -         W -         Y -
>>> print(game.discard_pile)
G5
>>> game.n_misfires
3
>>> game.b_lose
True
```

If the highest card in a color is played, then the players gain a clue:

```
>>> import random
>>> random.seed(0)
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.i_active = -1
>>> game.deal()
>>> for s in ['G1', 'G2', 'G3', 'G4']:
...     _ = game.board.try_to_play(card=Card(s))
>>> game.n_clues = 3
>>> game.i_active = 2
>>> print(game.hands[2])
B4 W4 G5 W1 R3
>>> game.players[1].speak = True
>>> is_legal = game.execute_play_card(2)
Donald X: A player tries to play a card on the board.
Donald X: i_active = 1
Donald X: k = 2
Donald X: card = G5
Donald X: Another player tries to draw a card.
Donald X: i_active = 1
Donald X: card = G4
>>> is_legal
```

(continues on next page)

(continued from previous page)

```
True
>>> print(game.board)  #doctest: +NORMALIZE_WHITESPACE
B -         G 1 2 3 4 5 R -         W -         Y -
>>> game.n_clues
4
```

But players cannot gain a clue if they already have the maximum number of clues:

```
>>> import random
>>> random.seed(0)
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.i_active = -1
>>> game.deal()
>>> for s in ['G1', 'G2', 'G3', 'G4']:
...     _ = game.board.try_to_play(card=Card(s))
>>> game.n_clues
8
>>> game.i_active = 2
>>> print(game.hands[2])
B4 W4 G5 W1 R3
>>> game.players[1].speak = True
>>> is_legal = game.execute_play_card(2)
Donald X: A player tries to play a card on the board.
Donald X: i_active = 1
Donald X: k = 2
Donald X: card = G5
Donald X: Another player tries to draw a card.
Donald X: i_active = 1
Donald X: card = G4
>>> is_legal
True
>>> print(game.board)  #doctest: +NORMALIZE_WHITESPACE
B -         G 1 2 3 4 5 R -         W -         Y -
>>> game.n_clues
8
```

If the card completes the board, then the players win the game.

```
>>> import random
>>> random.seed(0)
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.i_active = -1
>>> game.deal()
>>> for c in ['B', 'R', 'W', 'Y']:
...     for v in range(1, 6):
...         _ = game.board.try_to_play(card=Card(c + str(v)))
>>> for s in ['G1', 'G2', 'G3', 'G4']:
...     _ = game.board.try_to_play(card=Card(s))
>>> game.i_active = 2
>>> print(game.hands[2])
B4 W4 G5 W1 R3
>>> game.players[1].speak = True
>>> _ = game.execute_play_card(2)
```

(continues on next page)

(continued from previous page)

```
Donald X: A player tries to play a card on the board.
Donald X: i_active = 1
Donald X: k = 2
Donald X: card = G5
>>> print(game.board)  #doctest: +NORMALIZE_WHITESPACE
B 1 2 3 4 5 G 1 2 3 4 5 R 1 2 3 4 5 W 1 2 3 4 5 Y 1 2 3 4 5
>>> game.b_win
True
```

execute_throw(*k: int*) → bool

Execute the action: throw (= discard willingly).

Parameters **k** – the index of the card in the active player's hand.

Returns True iff the action is legal, i.e. except if players have all the clue chips.

```
>>> import random
>>> random.seed(0)
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.players[1].speak = True
>>> game.i_active = 1
>>> game.draw()
Donald X: This player tries to draw a card.
>>> is_legal = game.execute_throw(0)
Donald X: The action chosen is illegal.
Donald X: You cannot discard because you have all the clue chips.
>>> is_legal
False
>>> game.n_clues = 3
>>> game.i_active = 2
>>> game.draw()
Donald X: Another player tries to draw a card.
Donald X: i_active = 1
Donald X: card = Y4
>>> is_legal = game.execute_throw(0)
Donald X: A player throws (discards a card willingly).
Donald X: i_active = 1
Donald X: k = 0
Donald X: card = Y4
Donald X: Another player tries to draw a card.
Donald X: i_active = 1
Donald X: card = R3
>>> is_legal
True
>>> game.n_clues
4
>>> print(game.discard_pile)
Y4
>>> print(game.hands[2])
R3
```

game_exhausted() → int

The game is exhausted.

Inform the players. The game is over and is neither really lost (misfires, forfeit) nor a total victory (maximal score). Typically, this happens a bit after the deck ran out of cards (it depends on the end-of-game rule that

is used).

Returns the final score.

```
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> _ = game.board.try_to_play(Card('B1'))
>>> game.players[1].speak = True
>>> score = game.game_exhausted()
Donald X: The game is exhausted.
Donald X: score = 1
>>> score
1
```

i_active
Index of the active player.

Returns this index is automatically set modulo the number of players.

```
>>> game = Game(players=[PlayerHumanText('Antoine'),
...                      PlayerHumanText('Donald X'),
...                      PlayerHumanText('Uwe')])
>>> game.i_active = 2
>>> game.i_active += 1
>>> print(game.i_active)
0
```

lose() → int
Lose the game (forfeit or too many misfires).

Inform the players.

Returns the final score, i.e. 0.

```
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> game.players[1].speak = True
>>> score = game.lose()
Donald X: The game is lost.
Donald X: score = 0
>>> score
0
```

play() → int
Main method: play the game.

Note: it is only possible to "play" once with a *Game* object. If you want to launch a game with the same player, it is necessary to define a new *Game*.

Returns the final score of the game.

rel(*who: int*, *fro: int*) → int
Relative position of a player from the point of view of another one.

Parameters

- **who** – the player we talk about.
- **fro** – the player to whom we talk.

Returns the relative position of `who` from the point of view of `fro`, i.e. `who` - `fro` (modulo `n_players`).

```
>>> game = Game(players=[PlayerHumanText('Antoine'),
...                      PlayerHumanText('Donald X'),
...                      PlayerHumanText('Uwe')])
>>> game.rel(who=1, fro=2)
2
```

win() → int

Win the game (maximum score).

Inform the players.

Returns the final score.

```
>>> game = Game(players=[PlayerPuppet('Antoine'),
...                      PlayerPuppet('Donald X'),
...                      PlayerPuppet('Uwe')])
>>> for c in ['B', 'G', 'R', 'W', 'Y']:
...     for v in range(1, 6):
...         _ = game.board.try_to_play(card=Card(c + str(v)))
>>> game.players[1].speak = True
>>> score = game.win()
Donald X: The game is won.
Donald X: score = 25
>>> score
25
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/francois-durand/hanabython/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

Hanabython could always use more documentation, whether as part of the official Hanabython docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/francois-durand/hanabython/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *hanabython* for local development.

1. Fork the *hanabython* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/hanabython.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv hanabython
$ cd hanabython/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 hanabython tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/francois-durand/hanabython/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_hanabython
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Credits

6.1 Development Lead

- François Durand <fradurand@gmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.1.12 (2019-06-27)

- Test release for PyPI deployment.

7.2 0.1.11 (2019-06-27)

- Test release for PyPI deployment.

7.3 0.1.10 (2018-02-26)

- Correct a display bug of white cards on some terminals.

7.4 0.1.9 (2018-02-26)

- Game engine.
- Text interface for a human player.
- Patch import problems from previous versions 0.1.*.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

A

Action (*class in hanabython*), 32
ActionClue (*class in hanabython*), 32
ActionForfeit (*class in hanabython*), 33
ALLOWED (*hanabython.ConfigurationEmptyClueRule attribute*), 14
ATTEMPTS_BEFORE_FORFEIT (*hanabython.Game attribute*), 48

B

BLUE (*hanabython.Colors attribute*), 11
BLUE (*hanabython.StringAnsi attribute*), 7
Board (*class in hanabython*), 30
BROWN (*hanabython.StringAnsi attribute*), 7

C

Card (*class in hanabython*), 19
CardPublic (*class in hanabython*), 20
CATEGORIES (*hanabython.Action attribute*), 32
check_game_exhausted() (*hanabython.Game method*), 48
choose_action() (*hanabython.Player method*), 33
choose_action() (*hanabython.PlayerHumanText method*), 46
choose_action() (*hanabython.PlayerPuppet method*), 36
Clue (*class in hanabython*), 18
CLUE (*hanabython.Action attribute*), 32
Color (*class in hanabython*), 9
COLOR (*hanabython.Clue attribute*), 18
color_str() (*hanabython.Color method*), 9
ColorColorless (*class in hanabython*), 10
Colored (*class in hanabython*), 8
colored() (*hanabython.ActionClue method*), 33
colored() (*hanabython.ActionForfeit method*), 33
colored() (*hanabython.Board method*), 30
colored() (*hanabython.Card method*), 19
colored() (*hanabython.CardPublic method*), 20
colored() (*hanabython.Clue method*), 18
colored() (*hanabython.Color method*), 9
colored() (*hanabython.Colored method*), 8
colored() (*hanabython.Configuration method*), 17
colored() (*hanabython.ConfigurationColorContents method*), 12
colored() (*hanabython.ConfigurationDeck method*), 13
colored() (*hanabython.ConfigurationEmptyClueRule method*), 14
colored() (*hanabython.ConfigurationEndRule method*), 15
colored() (*hanabython.ConfigurationHandSize method*), 15
colored() (*hanabython.DiscardPile method*), 27
colored() (*hanabython.DrawPile method*), 25
colored() (*hanabython.DrawPilePublic method*), 25
colored() (*hanabython.Game method*), 49
colored() (*hanabython.Hand method*), 22
colored() (*hanabython.HandPublic method*), 23
colored() (*hanabython.Player method*), 34
colored() (*hanabython.PlayerBase method*), 39
colored_as_array() (*hanabython.DiscardPile method*), 27
colored_compact() (*hanabython.Board method*), 30
colored_compact_chronological() (*hanabython.DiscardPile method*), 27
colored_compact_factorized() (*hanabython.DiscardPile method*), 27
colored_compact_ordered() (*hanabython.DiscardPile method*), 27
colored_fixed_space() (*hanabython.Board method*), 30
colored_hands() (*hanabython.PlayerBase method*), 39
colored_multi_line() (*hanabython.Board method*), 30
colored_multi_line() (*hanabython.DiscardPile method*), 27
colored_multi_line_compact() (*han-*

abython.Board method), 30
colored_multi_line_compact() (*hanabython.DiscardPile method*), 27
COLORLESS (*hanabython.Colors attribute*), 11
ColorMulticolor (*class in hanabython*), 10
Colors (*class in hanabython*), 11
Configuration (*class in hanabython*), 15
ConfigurationColorContents (*class in hanabython*), 11
ConfigurationDeck (*class in hanabython*), 12
ConfigurationEmptyClueRule (*class in hanabython*), 14
ConfigurationEndRule (*class in hanabython*), 14
ConfigurationHandSize (*class in hanabython*), 15
copy() (*hanabython.ConfigurationDeck method*), 13
CROWNING_PIECE (*hanabython.ConfigurationEndRule attribute*), 15
CYAN (*hanabython.StringAnsi attribute*), 7

D

deal() (*hanabython.Game method*), 49
demo_game() (*hanabython.PlayerBase method*), 39
DiscardPile (*class in hanabython*), 26
draw() (*hanabython.Game method*), 49
DrawPile (*class in hanabython*), 24
DrawPilePublic (*class in hanabython*), 25

E

EIGHT_COLORS (*hanabython.Configuration attribute*), 17
EIGHT_COLORS (*hanabython.ConfigurationDeck attribute*), 12
execute_action() (*hanabython.Game method*), 50
execute_clue() (*hanabython.Game method*), 50
execute_forfeit() (*hanabython.Game method*), 51
execute_play_card() (*hanabython.Game method*), 51
execute_throw() (*hanabython.Game method*), 54

F

FORBIDDEN (*hanabython.ConfigurationEmptyClueRule attribute*), 14
FORFEIT (*hanabython.Action attribute*), 32
from_symbol() (*hanabython.Colors class method*), 11

G

Game (*class in hanabython*), 47
game_exhausted() (*hanabython.Game method*), 54
give() (*hanabython.DrawPile method*), 25
give() (*hanabython.DrawPilePublic method*), 26
give() (*hanabython.Hand method*), 22
give() (*hanabython.HandPublic method*), 23
GREEN (*hanabython.Colors attribute*), 11
GREEN (*hanabython.StringAnsi attribute*), 7

H

Hand (*class in hanabython*), 22
HandPublic (*class in hanabython*), 23

I

i_active (*hanabython.Game attribute*), 55
i_from_c() (*hanabython.Configuration method*), 18
i_from_v() (*hanabython.Configuration method*), 18
is_cluable (*hanabython.Color attribute*), 10
is_cluable (*hanabython.ColorColorless attribute*), 10
is_cluable (*hanabython.ColorMulticolor attribute*), 10

L

list_reordered (*hanabython.DiscardPile attribute*), 27
log() (*hanabython.PlayerBase method*), 40
log_forget() (*hanabython.PlayerBase method*), 40
log_init() (*hanabython.PlayerBase method*), 40
lose() (*hanabython.Game method*), 55

M

MAGENTA (*hanabython.StringAnsi attribute*), 7
match() (*hanabython.Card method*), 20
match() (*hanabython.CardPublic method*), 21
match() (*hanabython.Color method*), 10
match() (*hanabython.ColorColorless method*), 10
match() (*hanabython.ColorMulticolor method*), 10
match() (*hanabython.Hand method*), 22
match() (*hanabython.HandPublic method*), 24
max_score_possible (*hanabython.DiscardPile attribute*), 28
MULTICOLOR (*hanabython.Colors attribute*), 11

N

n_cards (*hanabython.DrawPile attribute*), 25
NORMAL (*hanabython.ConfigurationColorContents attribute*), 12
NORMAL (*hanabython.ConfigurationDeck attribute*), 13
NORMAL (*hanabython.ConfigurationEndRule attribute*), 15
NORMAL (*hanabython.ConfigurationHandSize attribute*), 15
normal_plus() (*hanabython.ConfigurationDeck static method*), 13

P

play() (*hanabython.Game method*), 55
PLAY_CARD (*hanabython.Action attribute*), 32

Player (*class in hanabython*), 33
PlayerBase (*class in hanabython*), 38
PlayerHumanText (*class in hanabython*), 45
PlayerPuppet (*class in hanabython*), 36

R

receive() (*hanabython.DiscardPile method*), 28
receive() (*hanabython.Hand method*), 23
receive() (*hanabython.HandPublic method*), 24
receive_action_illegal() (*hanabython.Player method*), 34
receive_action_illegal() (*hanabython.PlayerHumanText method*), 46
receive_action_illegal() (*hanabython.PlayerPuppet method*), 36
receive_action_legal() (*hanabython.Player method*), 34
receive_action_legal() (*hanabython.PlayerHumanText method*), 46
receive_action_legal() (*hanabython.PlayerPuppet method*), 36
receive_begin_dealing() (*hanabython.Player method*), 34
receive_begin_dealing() (*hanabython.PlayerBase method*), 40
receive_begin_dealing() (*hanabython.PlayerPuppet method*), 36
receive_end_dealing() (*hanabython.Player method*), 34
receive_end_dealing() (*hanabython.PlayerBase method*), 41
receive_end_dealing() (*hanabython.PlayerPuppet method*), 36
receive_game_exhausted() (*hanabython.Player method*), 34
receive_game_exhausted() (*hanabython.PlayerBase method*), 41
receive_game_exhausted() (*hanabython.PlayerHumanText method*), 46
receive_game_exhausted() (*hanabython.PlayerPuppet method*), 36
receive_i_draw() (*hanabython.Player method*), 34
receive_i_draw() (*hanabython.PlayerBase method*), 41
receive_i_draw() (*hanabython.PlayerPuppet method*), 36
receive_init() (*hanabython.Player method*), 34
receive_init() (*hanabython.PlayerBase method*), 41
receive_init() (*hanabython.PlayerPuppet method*), 36
receive_lose() (*hanabython.Player method*), 34
receive_lose() (*hanabython.PlayerBase method*), 42
receive_lose() (*hanabython.PlayerHumanText method*), 46
receive_lose() (*hanabython.PlayerPuppet method*), 37
receive_partner_draws() (*hanabython.Player method*), 34
receive_partner_draws() (*hanabython.PlayerBase method*), 43
receive_partner_draws() (*hanabython.PlayerPuppet method*), 37
receive_remaining_turns() (*hanabython.Player method*), 35
receive_remaining_turns() (*hanabython.PlayerBase method*), 43
receive_remaining_turns() (*hanabython.PlayerPuppet method*), 37
receive_someone_clues() (*hanabython.Player method*), 35
receive_someone_clues() (*hanabython.PlayerBase method*), 44
receive_someone_clues() (*hanabython.PlayerPuppet method*), 37
receive_someone_forfeits() (*hanabython.Player method*), 35
receive_someone_forfeits() (*hanabython.PlayerBase method*), 44
receive_someone_forfeits() (*hanabython.PlayerPuppet method*), 37
receive_someone_plays_card() (*hanabython.Player method*), 35
receive_someone_plays_card() (*hanabython.PlayerBase method*), 44
receive_someone_plays_card() (*hanabython.PlayerPuppet method*), 37
receive_someone_throws() (*hanabython.Player method*), 35
receive_someone_throws() (*hanabython.PlayerBase method*), 45
receive_someone_throws() (*hanabython.PlayerPuppet method*), 37
receive_turn_begin() (*hanabython.Player method*), 35
receive_turn_begin() (*hanabython.PlayerHumanText method*), 46
receive_turn_begin() (*hanabython.PlayerPuppet method*), 38
receive_turn_finished() (*hanabython.Player method*), 35
receive_turn_finished() (*hanabython.PlayerHumanText method*), 47
receive_turn_finished() (*hanabython.PlayerPuppet method*), 38
receive_win() (*hanabython.Player method*), 36
receive_win() (*hanabython.PlayerBase method*), 45

receive_win() (*hanabython.PlayerHumanText method*), 47
receive_win() (*hanabython.PlayerPuppet method*), 38
RED (*hanabython.Colors attribute*), 11
RED (*hanabython.StringAnsi attribute*), 7
rel() (*hanabython.Game method*), 55
RESET (*hanabython.StringAnsi attribute*), 7

S

score (*hanabython.Board attribute*), 30
SHORT (*hanabython.ConfigurationColorContents attribute*), 12
SIXTH (*hanabython.Colors attribute*), 11
STANDARD (*hanabython.Configuration attribute*), 17
str_as_array() (*hanabython.DiscardPile method*), 28
str_compact() (*hanabython.Board method*), 31
str_compact_chronological() (*hanabython.DiscardPile method*), 28
str_compact_factorized() (*hanabython.DiscardPile method*), 29
str_compact_ordered() (*hanabython.DiscardPile method*), 29
str_fixed_space() (*hanabython.Board method*), 31
str_from_iterable() (*in module hanabython*), 7
str_multi_line() (*hanabython.Board method*), 31
str_multi_line() (*hanabython.DiscardPile method*), 29
str_multi_line_compact() (*hanabython.Board method*), 31
str_multi_line_compact() (*hanabython.DiscardPile method*), 29
StringAnsi (*class in hanabython*), 7
STYLE_BOLD (*hanabython.StringAnsi attribute*), 7
STYLE_REVERSE_VIDEO (*hanabython.StringAnsi attribute*), 7
STYLE_UNDERLINE (*hanabython.StringAnsi attribute*), 7

T

test_str() (*hanabython.Colored method*), 9
THROW (*hanabython.Action attribute*), 32
title() (*in module hanabython*), 8
try_to_play() (*hanabython.Board method*), 32

U

uncolor() (*in module hanabython*), 8

V

VALUE (*hanabython.Clue attribute*), 18
VARIANT_6_3 (*hanabython.ConfigurationHandSize attribute*), 15

W

W_MULTICOLOR (*hanabython.Configuration attribute*), 17
W_MULTICOLOR (*hanabython.ConfigurationDeck attribute*), 13
W_MULTICOLOR_SHORT (*hanabython.Configuration attribute*), 17
W_MULTICOLOR_SHORT (*hanabython.ConfigurationDeck attribute*), 13
W_SIXTH (*hanabython.Configuration attribute*), 17
W_SIXTH (*hanabython.ConfigurationDeck attribute*), 13
W_SIXTH_SHORT (*hanabython.Configuration attribute*), 17
W_SIXTH_SHORT (*hanabython.ConfigurationDeck attribute*), 13
WHITE (*hanabython.Colors attribute*), 11
WHITE (*hanabython.StringAnsi attribute*), 7
win() (*hanabython.Game method*), 56

Y

YELLOW (*hanabython.Colors attribute*), 11
YELLOW (*hanabython.StringAnsi attribute*), 7