pyCardDeck Documentation

Release 1.4.0

David Jetelina

Oct 30, 2018

Contents

We hope you'll find everything you'll ever need in here. If youn don't, why not submit a pull request :)

CHAPTER 1

API

1.1 pyCardDeck

Deck of cards with all the logic, so you don't have to!

copyright

3. 2016 David Jetelina

license MIT

1.2 Types

pyCardDeck isn't strict about types. It's however nice to use Python 3's type annotations. That's why we have custom types set up when needed

1.2.1 CardType

Can be either instance of an object, string or an integer. Basically, it's important that they aren't bool or NoneType. It's however recommended to inherit from one of the classes in *Cards*

1.3 Classes and Functions

1.3.1 Deck

class pyCardDeck.deck.**Deck**(*cards: Optional[List[object]] = None*, *reshuffle: object = True*, *name: str = None*, *discard: Optional[Deck] = None*)

Deck you will be using. Make sure to create the instance somewhere reachable :)

Parameters

- **cards** (List[*CardType*]) –

  Use this parameter if you don't plan to register your cards another way
  Cards can be either an instance of a object, string or an integer,
  the documentation will be calling this *CardType* (because of Python's rank hinting)

- **reshuffle** (*bool*) – Set reshuffle to false if you want your deck not to reshuffle after it's depleted
- **name** (*string*) – Name of the deck, used when converting the Deck instance into string
- **discard** (*Union[Deck, None]*) – optional Deck object to use as discard pile

Attributes

Deck.**name**

Returns The name of the deck

Return type str

Deck.**reshuffle**

Returns Whether the deck will be reshuffled when drawn out

Return type bool

Deck.**_cards**

Returns Cards in the deck

Return type list

Deck.**_discard_pile**

Note: Cards are not put in the discard pile automatically after drawing, the code assumes they went into a hand of sorts and must be discarded with `discard()` from there. This means that *reshuffle* doesn't work on one card deck as you can't reshuffle an empty deck (*errors.NoCards* would be raised).

Returns Cards in the discard pile

Return type list

Deck.**empty**

Returns Whether the deck is empty

Return type bool

Deck.**cards_left**
Cards left in the deck

Returns Number of cards in the deck

Return type int

Deck.**discarded**
Cards in the discard pile

Returns Number of cards in the discard pile

Return type int

Deck.**json**
Alternative to Deck.export("json")

Returns jsonpickled Deck

Return type str

Deck.**yaml**
Alternative to Deck.export("yaml")

Returns yaml dump of the Deck

Return type str

Card drawing

Deck.**draw**() → object
Draw the topmost card from the deck

Returns Card from the list

Return type *CardType*

Raises

- ***OutOfCards*** – when there are no cards in the deck
- ***NoCards*** – when the deck runs out of cards (no reshuffle)

Deck.**draw_bottom**() → object
Draw the bottommost card from the deck

Returns Card from the list

Return type *CardType*

Raises

- ***OutOfCards*** – when there are no cards in the deck
- ***NoCards*** – when the deck runs out of cards (no reshuffle)

Deck.**draw_random**() → object
Draw a random card from the deck

Returns Card from the list

Return type *CardType*

Raises

- ***OutOfCards*** – when there are no cards in the deck
- ***NoCards*** – when the deck runs out of cards (no reshuffle)

Deck.**draw_specific**(*specific_card: object*) → object
Draw a specific card from the deck

Note: For card instances to match, they should have *__eq__* method set to compare their equality. If you don't want to set those up, make sure their *__dict__* are the same and their name is the same.

If you are using a string or an integer, don't worry about this!

Parameters **specific_card** (*CardType*) – Card identical to the one you are looking for

Returns Card from the list

Return type *CardType*

Raises

- ***OutOfCards*** – when there are no cards in the deck
- ***NoCards*** – when the deck runs out of cards (no reshuffle)
- ***CardNotFound*** – when the card is not found in the deck

Card information

Deck.**card_exists**(*card: object*) → bool
Checks if a card exists in the deck

Note: For card instances to match, they should have *__eq__* method set to compare their equality. If you don't want to set those up, make sure their *__dict__* are the same and their name is the same.

If you are using a string or an integer, don't worry about this!

Parameters **card** (*CardType*) – Card identical to the one you are looking for

Returns

True if exists
False if doesn't exist

Return type bool

Deck Manipulation

Deck.**shuffle**() → None
Randomizes the order of cards in the deck

Raises ***NoCards*** – when there are no cards to be shuffled

Deck.**shuffle_back**() → None
Shuffles the discard pile back into the main pile

Deck.**discard**(*card: object*) → None
Puts a card into the discard pile

Parameters **card** (*CardType*) – Card to be discarded

Raises ***NotACard*** – When you try to insert False/None into a discard pile

Deck.**add_single**(*card: object*, *position: int = False*) → None
Shuffles (or inserts) a single card into the active deck

Parameters

- **card** (*CardType*) – Card you want to insert

- **position** (*int*) –

  If you want to let player insert card to a specific location, use position
  where 0 = top of the deck, 1 = second card from top etc.
  By default the position is random

Deck.**add_many**(*cards: List[object]*) → None
Shuffles a list of cards into the deck

Parameters **cards** (List[*CardType*]) – Cards you want to shuffle in

Deck.**show_top**(*number: int*) → List[object]
Selects the top X cards from the deck without drawing them

Useful for mechanics like scry in Magic The Gathering

If there are less cards left than you want to show, it will show only the remaining cards

Parameters **number** (*int*) – How many cards you want to show

Returns Cards you want to show

Return type List[*CardType*]

Import/Export

Deck.**export**(*fmt: str*, *to_file: bool = False*, *location: str = None*) → str
Export the deck. By default it returns string with either JSON or YaML, but if you set *to_file=True*, you can instead save the deck as a file. If no location (with filename) is provided, it'll save to the folder the script is opened from as *exported_deck* without an extension.

Parameters

- **fmt** (*str*) – Desired format, either YaML or JSON
- **to_file** (*bool*) – Whether you want to get a string back or save to a file
- **location** (*str*) – Where you want to save your file - include file name!

Raises ***UnknownFormat*** – When entered format is not supported

Returns Your exported deck as a string in your desired format

Return type str

Deck.**load**(*to_load: str*, *is_file: bool = False*) → None
Way to override a deck instance with a saved deck from either yaml, JSON or a file with either of those.

The library will first try to check if you have a save location saved, then verifies if the file exists as a path to a file. If it doesn't, it'l assume it's a string with one of the supported formats and will load from those.

Parameters

- **to_load** (*str*) –

  This should be either a path to a file or a string containing
  json/yaml generated by Deck.export(). It's not safe to trust your users
  with this, as they can provide harmful pickled JSON (see jsonpickle docs for more)

- **is_file** (*bool*) – whether to_load is a file path or actual data. Default is False

Raises ***UnknownFormat*** – When the entered yaml or json is not valid

Deck.**load_standard_deck**() → None
Loads a standard deck of 52 cards into the deck

Magic Methods

Deck.**__repr__**() → str
Used for representation of the object

called with repr(Deck)

Returns 'Deck of cards'

Return type string

Deck.**__str__**() → str
Used for representation of the object for humans

called with str(Deck)

This method is also called when you are providing arguments to str.format(), you can just provide your Deck instance and it will magically know the name, yay!

Returns Name of the deck if it has a name or 'Deck of cards' if it has none

Return type string

Deck.**__len__**() → int
Instead of doing len(Deck.cards) you can just check len(Deck)

It's however recommended to use the *cards_left* attribute

Returns Number of cards left in the deck

Return type int

Other Functions

pyCardDeck.deck.**_card_compare**(*card: object*, *second_card: object*) → bool
Function for comparing two cards. First it checks their *__eq__*, if that returns False, it checks *__dict__* and name of the Class that spawned them .

Parameters

- **card** (*CardType*) – First card to match
- **second_card** (*CardType*) – Second card to match

Returns Whether they are the same

Return type bool

pyCardDeck.deck.**_get_exported_string**(*format_stripped: str*, *deck: pyCardDeck.deck.Deck*) → str
Helper function to Deck.export()

Parameters

- **format_stripped** (*str*) – Desired format stripped of any spaces and lowercase
- **deck** (*Deck*) – instance of a Deck

Returns YAML/JSON string of the deck

Return type str

Raises ***UnknownFormat*** – when it doesn't recognize format_stripped

1.3.2 Cards

These classes are only recommended to inherit from, feel free to use your own!

class pyCardDeck.cards.**BaseCard**(*name: str*)
This is an example Card, showing that each Card should have a name.

This is good, because when we can show player their cards just by converting them to strings.

class pyCardDeck.cards.**PokerCard**(*suit: str*, *rank: str*, *name: str*)
Example Poker Card, since Poker is a a deck of Unique cards, we can say that if their name equals, they equal too.

1.3.3 Exceptions

exception pyCardDeck.errors.**DeckException**
Base exception class for pyCardDeck

exception pyCardDeck.errors.**NoCards**
Exception that's thrown when there are no cards to be manipulated.

exception pyCardDeck.errors.**OutOfCards**
Exception that's thrown when the deck runs out of cards. Unlike NoCardsException, this will happen naturally when reshuffling is disabled

exception pyCardDeck.errors.**NotACard**
Exception that's thrown when the manipulated object is False/None

exception pyCardDeck.errors.**CardNotFound**
Exception that's thrown when a card is not found

exception pyCardDeck.errors.**UnknownFormat**
Exception thrown when trying to export to a unknown format. Supported formats: YaML, JSON

CHAPTER 2

Examples

If you don't want to read through the whole documentation, you can just have a look at the examples we wrote to help you understand how to use pyCardDeck, enjoy!

2.1 Blackjack

Blackjack game made using pyCardDeck. This is an example of pyCardDeck; it's not meant to be complete blackjack game, but rather a showcase of pyCardDeck's usage.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import pyCardDeck
from typing import List
from pyCardDeck.cards import PokerCard

class Player:

    def __init__(self, name: str):
        self.hand = []
        self.name = name

    def __str__(self):
        return self.name

class BlackjackGame:

    def __init__(self, players: List[Player]):
        self.deck = pyCardDeck.Deck()
        self.deck.load_standard_deck()
        self.players = players
```

(continues on next page)

(continued from previous page)

```
        self.scores = {}
        print("Created a game with {} players.".format(len(self.players)))
```

2.1.1 The main blackjack game sequence

Each player takes an entire turn before moving on. If each player gets a turn and no one has won, the player or players with the highest score below 21 are declared the winner.

```
def blackjack(self):

    print("Setting up...")
    print("Shuffling...")
    self.deck.shuffle()
    print("All shuffled!")
    print("Dealing...")
    self.deal()
    print("\nLet's play!")
    for player in self.players:
        print("{}'s turn...".format(player.name))
        self.play(player)
    else:
        print("That's the last turn. Determining the winner...")
        self.find_winner()
```

Dealing.

Deals two cards to each player.

```
def deal(self):
    for _ in range(2):
        for p in self.players:
            newcard = self.deck.draw()
            p.hand.append(newcard)
            print("Dealt {} the {}.".format(p.name, str(newcard)))
```

Determining the winner.

Finds the highest score, then finds which player(s) have that score, and reports them as the winner.

```
def find_winner(self):

    winners = []
    try:
        win_score = max(self.scores.values())
        for key in self.scores.keys():
            if self.scores[key] == win_score:
                winners.append(key)
            else:
                pass
        winstring = " & ".join(winners)
        print("And the winner is...{}!".format(winstring))
```

(continues on next page)

(continued from previous page)

```
    except ValueError:
        print("Whoops! Everybody lost!")
```

Hit.

Adds a card to the player's hand and states which card was drawn.

```
def hit(self, player):

    newcard = self.deck.draw()
    player.hand.append(newcard)
    print("   Drew the {}.".format(str(newcard)))
```

An individual player's turn.

If the player's cards are an ace and a ten or court card, the player has a blackjack and wins.

If a player's cards total more than 21, the player loses.

Otherwise, it takes the sum of their cards and determines whether to hit or stand based on their current score.

```
def play(self, player):

    while True:
        points = sum_hand(player.hand)
        if points < 17:
            print("   Hit.")
            self.hit(player)
        elif points == 21:
            print("   {} wins!".format(player.name))
            sys.exit(0) # End if someone wins
        elif points > 21:
            print("   Bust!")
            break
        else:  # Stand if between 17 and 20 (inclusive)
            print("   Standing at {} points.".format(str(points)))
            self.scores[player.name] = points
            break
```

2.1.2 Sum of cards in hand.

Converts ranks of cards into point values for scoring purposes. 'K', 'Q', and 'J' are converted to 10. 'A' is converted to 1 (for simplicity), but if the first hand is an ace and a 10-valued card, the player wins with a blackjack.

```
def sum_hand(hand: list):

    vals = [card.rank for card in hand]
    intvals = []
    while len(vals) > 0:
        value = vals.pop()
        try:
            intvals.append(int(value))
```

(continues on next page)

(continued from previous page)

```
        except ValueError:
            if value in ['K', 'Q', 'J']:
                intvals.append(10)
            elif value == 'A':
                intvals.append(1)  # Keep it simple for the sake of example
    if intvals == [1, 10] or intvals == [10, 1]:
        print("   Blackjack!")
        return(21)
    else:
        points = sum(intvals)
        print("   Current score: {}".format(str(points)))
        return(points)
```

```
if __name__ == "__main__":
    game = BlackjackGame([Player("Kit"), Player("Anya"), Player("Iris"),
        Player("Simon")])
    game.blackjack()
```

2.2 Hearthstone Arena

This shows how simple something like drafting can be with pyCardDeck. Although not much more complicated with just a list :D

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
This is an example of pyCardDeck, it's not meant to be complete poker script,
but rather a showcase of pyCardDeck's usage.
"""

import pyCardDeck
import random
import requests

arena_deck = pyCardDeck.Deck(reshuffle=False, name="Awesome arena deck!")
rarity = {"Common": 100, "Rare": 50, "Epic": 15, "Legendary": 1}


def card_choice() -> list:
    """
    Picks a rarity, then lets you make a choice

    :return:    List with the card information
    """
    pick_rarity = random.choice([k for k in rarity for _ in range(rarity[k])])
    # This api doesn't provide an easy way to get class and rarity filter at the same␣
↪time
    # and I'm too lazy to look for another, reminder: this is an example
    cards = requests.get("https://omgvamp-hearthstone-v1.p.mashape.com/cards/
↪qualities/{}".format(pick_rarity),
                         headers={"X-Mashape-Key":
↪"GkQg9DFiZWmshWn6oYqlfXXlXeK9p1QuB6QjsngIi1sHnJiJqv"}).json()
    first, second, third = [random.choice(cards)] * 3
    while second == first:
```

(continues on next page)

(continued from previous page)

```
        second = random.choice(cards)
    while third == first or third == second:
        third = random.choice(cards)
    choice = input("Which one would you like?\n 1: {0}, 2: {1}, 3: {2}\n".format(
        first['name'], second['name'], third['name']))
    while choice not in ["1", "2", "3"]:
        if choice == "1":
            return first
        elif choice == "2":
            return second
        elif choice == "3":
            return third


def draft():
    """
    Simple draft logic
    """
    for _ in range(30):
        arena_deck.add_single(card_choice())
    print(arena_deck)


if __name__ == '__main__':
    draft()
```

2.3 Poker example

This is a poker example of pyCardDeck, it's not meant to be complete poker script, but rather a showcase of pyCardDeck's usage.

```
import pyCardDeck
from typing import List
from pyCardDeck.cards import PokerCard
```

For python 3.3 and 3.4 compatibility and type hints, we import typing.List - this is not needed, however the package itself and PokerCard are recommended here

```
class Player:

    def __init__(self, name: str):
        self.hand = []
        self.name = name

    def __str__(self):
        return self.name


class PokerTable:

    def __init__(self, players: List[Player]):
        self.deck = pyCardDeck.Deck(
            cards=generate_deck(),
            name='Poker deck',
```

(continues on next page)

(continued from previous page)

```
            reshuffle=False)
        self.players = players
        self.table_cards = []
        print("Created a table with {} players".format(len(self.players)))
```

We define our Player class, to have a hand and a name, and our PokerTable which will hold all the information and will have following methods:

```
def texas_holdem(self):
    """
    Basic Texas Hold'em game structure
    """
    print("Starting a round of Texas Hold'em")
    self.deck.shuffle()
    self.deal_cards(2)
    # Imagine pre-flop logic for betting here
    self.flop()
    # Imagine post-flop, pre-turn logic for betting here
    self.river_or_flop()
    # Imagine post-turn, pre-river logic for betting here
    self.river_or_flop()
    # Imagine some more betting and winner decision here
    self.cleanup()
```

This is the core "loop" of Texas Hold'em

```
def deal_cards(self, number: int):
    for _ in range(0, number):
        for player in self.players:
            card = self.deck.draw()
            player.hand.append(card)
            print("Dealt {} to player {}".format(card, player))
```

Dealer will go through all available players and deal them x number of cards.

```
def flop(self):
    # Burn a card
    burned = self.deck.draw()
    self.deck.discard(burned)
    print("Burned a card: {}".format(burned))
    for _ in range(0, 3):
        card = self.deck.draw()
        self.table_cards.append(card)
        print("New card on the table: {}".format(card))
```

Burns a card and then shows 3 new cards on the table

```
def river_or_flop(self):
    burned = self.deck.draw()
    self.deck.discard(burned)
    print("Burned a card: {}".format(burned))
    card = self.deck.draw()
    self.table_cards.append(card)
    print("New card on the table: {}".format(card))
```

Burns a card and then shows 1 new card on the table

```
def cleanup(self):
    for player in self.players:
        for card in player.hand:
            self.deck.discard(card)
    for card in self.table_cards:
        self.deck.discard(card)
    self.deck.shuffle_back()
    print("Cleanup done")
```

Cleans up the table to gather all the cards back

```
def generate_deck() -> List[PokerCard]:
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
    ranks = {'A': 'Ace',
             '2': 'Two',
             '3': 'Three',
             '4': 'Four',
             '5': 'Five',
             '6': 'Six',
             '7': 'Seven',
             '8': 'Eight',
             '9': 'Nine',
             '10': 'Ten',
             'J': 'Jack',
             'Q': 'Queen',
             'K': 'King'}
    cards = []
    for suit in suits:
        for rank, name in ranks.items():
            cards.append(PokerCard(suit, rank, name))
    print('Generated deck of cards for the table')
    return cards\
```

Function that generates the deck, instead of writing down 50 cards, we use iteration to generate the cards for use

```
if __name__ == '__main__':
    table = PokerTable([Player("Jack"), Player("John"), Player("Peter")])
    table.texas_holdem()
```

And finally this is how we start the "game"

2.4 Exploding Kittens

Here's a bit more advanced game using pyCardDeck. This code itself is not the full game, but should showcase how the library is meant to be used. If you find anything in here impractical or not clean, easy and nice, please file an issue!

```
import pyCardDeck
from pyCardDeck.cards import BaseCard
from random import randrange


class Player:

    def __init__(self):
        self.hand = []
```

(continues on next page)

(continued from previous page)

```
    def turn(self):
        pass

    def skip(self):
        pass

    def take_turn_twice(self):
        self.turn()
        self.turn()

    def nope_prompt(self) -> bool:
        for card in self.hand:
            if card.name == "Nope":
                if input("Do you want to use your Nope card?").lower().startswith("y
↪"):
                    return True
                else:
                    return False
        return False

    def insert_explode(self) -> int:
        position = int(input("At which position from top do you want to insert␣
↪Exploding Kitten back into the deck?"))
        return position


class KittenCard(BaseCard):

    def __init__(self, name: str, targetable: bool = False, selfcast: bool = False):
        super().__init__(name)
        self.selfcast = selfcast
        self.targetable = targetable

    def effect(self, player: Player, target: Player):
        pass


class ExplodeCard(KittenCard):

    def __init__(self, name: str = "Exploding Kitten"):
        super().__init__(name)


class DefuseCard(KittenCard):

    def __init__(self, deck: pyCardDeck.deck, name: str = "Defuse"):
        super().__init__(name, selfcast=True)
        self.deck = deck

    def effect(self, player: Player, target: Player):
        position = player.insert_explode()
        self.deck.add_single(ExplodeCard(), position=position)


class TacocatCard(KittenCard):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, name: str = "Tacocat"):
        super().__init__(name)


class OverweightCard(KittenCard):

    def __init__(self, name: str = "Overweight Bikini Cat"):
        super().__init__(name)


class ShuffleCard(KittenCard):

    def __init__(self, deck: pyCardDeck.Deck, name: str = "Shuffle"):
        super().__init__(name)
        self.deck = deck

    def effect(self, player: Player, target: Player):
        self.deck.shuffle()


class AttackCard(KittenCard):

    def __init__(self, name: str = "Attack"):
        super().__init__(name, selfcast=True, targetable=True)

    def effect(self, player: Player, target: Player):
        player.skip()
        target.take_turn_twice()


class SeeTheFuture(KittenCard):

    def __init__(self, deck: pyCardDeck.Deck, name: str = "See The Future"):
        super().__init__(name)
        self.deck = deck

    def effect(self, player: Player, target: Player):
        self.deck.show_top(3)


class NopeCard(KittenCard):

    def __init__(self, name: str = "Nope"):
        super().__init__(name)


class SkipCard(KittenCard):

    def __init__(self, name: str = "Skip"):
        super().__init__(name, selfcast=True)

    def effect(self, player: Player, target: Player):
        player.skip()


class FavorCard(KittenCard):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, name: str = "Favor"):
        super().__init__(name, targetable=True, selfcast=True)

    def effect(self, player: Player, target: Player):
        random_target_card = target.hand.pop(randrange(target.hand))
        player.hand.append(random_target_card)


class Game:

    def __init__(self, players: list):
        self.deck = pyCardDeck.Deck()
        self.players = players
        self.prepare_cards()
        self.deal_to_players()
        self.add_defuses()
        self.add_explodes()
        while len(self.players) > 1:
            self.play()

    def play(self):
        pass

    def turn(self):
        pass

    def prepare_cards(self):
        print("Preparing deck from which to deal to players")
        self.deck.add_many(construct_deck(self))

    def deal_to_players(self):
        print("Dealing cards to players")
        for _ in range(4):
            for player in self.players:
                player.hand.append(self.deck.draw())

    def ask_for_nope(self):
        noped = False
        for player in self.players:
            noped = player.nope_prompt()
        return noped

    def add_explodes(self):
        print("Adding explodes to the deck")
        self.deck.add_many([ExplodeCard() for _ in range(len(self.players) - 1)])

    def add_defuses(self):
        print("Adding defuses to the deck")
        self.deck.add_many([DefuseCard(self.deck) for _ in range(6 - len(self.
↪players))])

    def play_card(self, card: KittenCard, player: Player = None, target: Player =␣
↪None):
        if card.selfcast and player is None:
            raise Exception("You must pass a player who owns the card!")
        elif card.targetable and target is None:
            raise Exception("You must pass a target!")
```

(continues on next page)

(continued from previous page)

```
        elif not self.ask_for_nope():
            card.effect(player, target)
        else:
            print("Card was noped :(")



def construct_deck(game: Game):
    card_list = [
        TacocatCard(),
        TacocatCard(),
        TacocatCard(),
        TacocatCard(),
        OverweightCard(),
        OverweightCard(),
        OverweightCard(),
        OverweightCard(),
        ShuffleCard(game.deck),
        ShuffleCard(game.deck),
        ShuffleCard(game.deck),
        ShuffleCard(game.deck),
        AttackCard(),
        AttackCard(),
        AttackCard(),
        AttackCard(),
        SeeTheFuture(game.deck),
        SeeTheFuture(game.deck),
        SeeTheFuture(game.deck),
        SeeTheFuture(game.deck),
        SeeTheFuture(game.deck),
        NopeCard(),
        NopeCard(),
        NopeCard(),
        NopeCard(),
        NopeCard(),
        SkipCard(),
        SkipCard(),
        SkipCard(),
        SkipCard(),
        FavorCard(),
        FavorCard(),
        FavorCard(),
        FavorCard(),
    ]
    return card_list
```

CHAPTER 3

Indices and tables

- genindex
- modindex

Python Module Index

p

Index

Symbols

A

B

C

D

E

J

L

N

O

P

R

S

U

Y