Sleeping Coders

Assignment 2
CSSE1001/7030
Semester 2, 2019

Version 1.0.1
15 marks

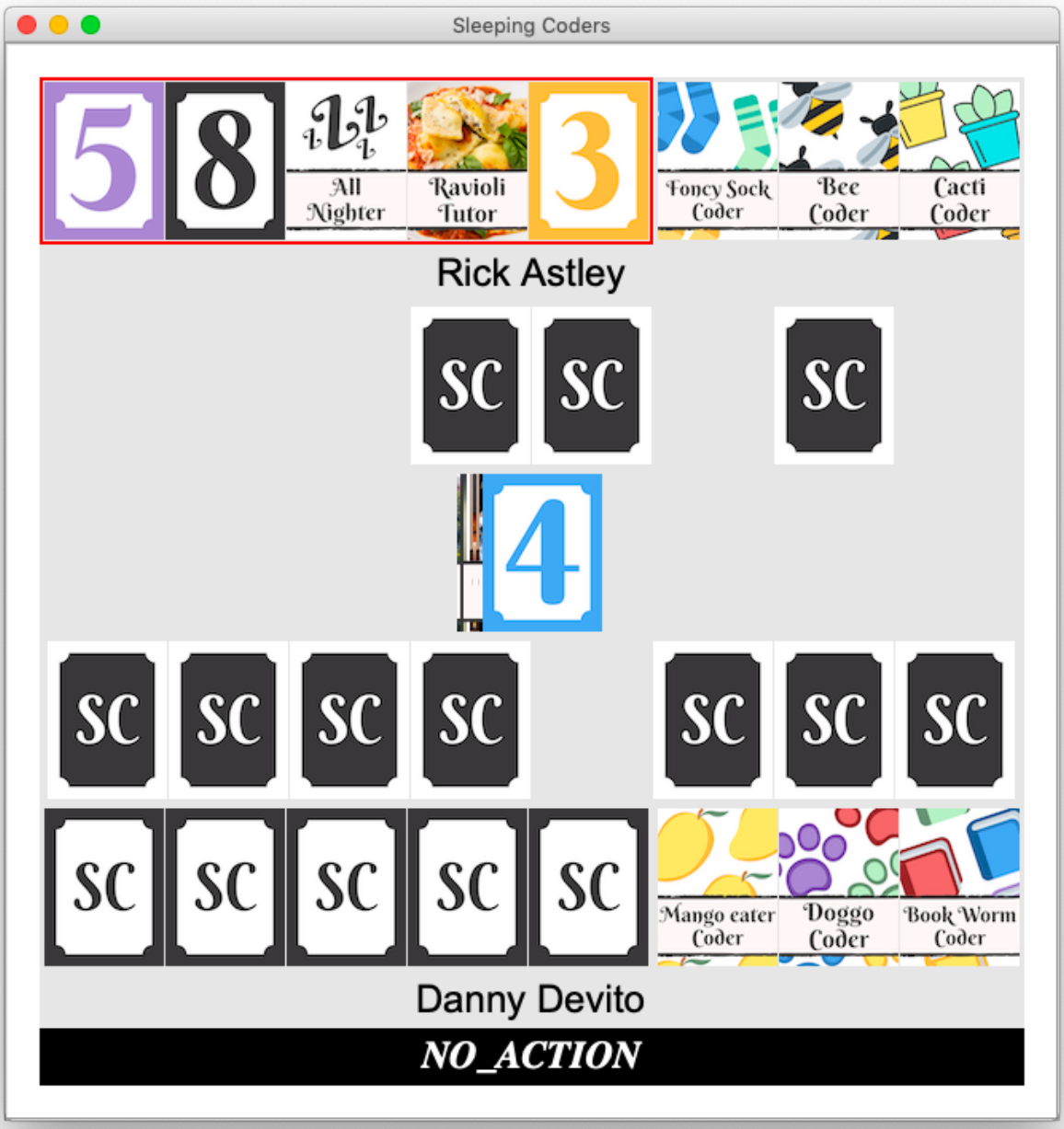
Due Friday 20th September, 2019, 20:30

Introduction

This assignment follows a programming pattern called MVC (the Model, View, Controller) pattern. You have been provided with the view and controller classes but you will be required to implement several modelling classes.

The modelling classes are based on the children's card game, Sleeping Queens. Each class you're required to implement has a specification that is outlined in this document. A specification is a description of each method in a class as well as their parameters and return values.

Once you have correctly implemented the modelling classes, you will have completed a digital version of Sleeping Coders.



Basic Running of the Game

Gameplay

Sleeping Coders (Coders for short) is a card game that is based on the popular children's card game, Sleeping Queens. The aim of the game is to wake as many sleeping coders as possible. A sleeping coder is woken by playing a tutor card.

At the start of the game there are 16 coders placed face down in the center of the board, a face down coder card means that coder is sleeping. Each player is dealt five cards randomly.

Players will take turns playing a card from their deck; number cards have no action and are meant to be disposed of while other cards have actions attributed to their playing.

A tutor card can be played at any time. When a tutor card is played the player who played it can pick up one of the coder cards from the center of the board.

Playing a keyboard kidnapper card will allow the player to take a coder card from another player.

Playing an all-nighter card will allow the player to put another player's coder back to sleep (add it back to the center of the board).

The aim of the game is to collect coders by waking them up. The first player to collect 4 coders wins the game.

Getting Started

Files

a2.py - file where you write your submission

a2_support.py - contains supporting code and part of the controller

gui.py - contains the view

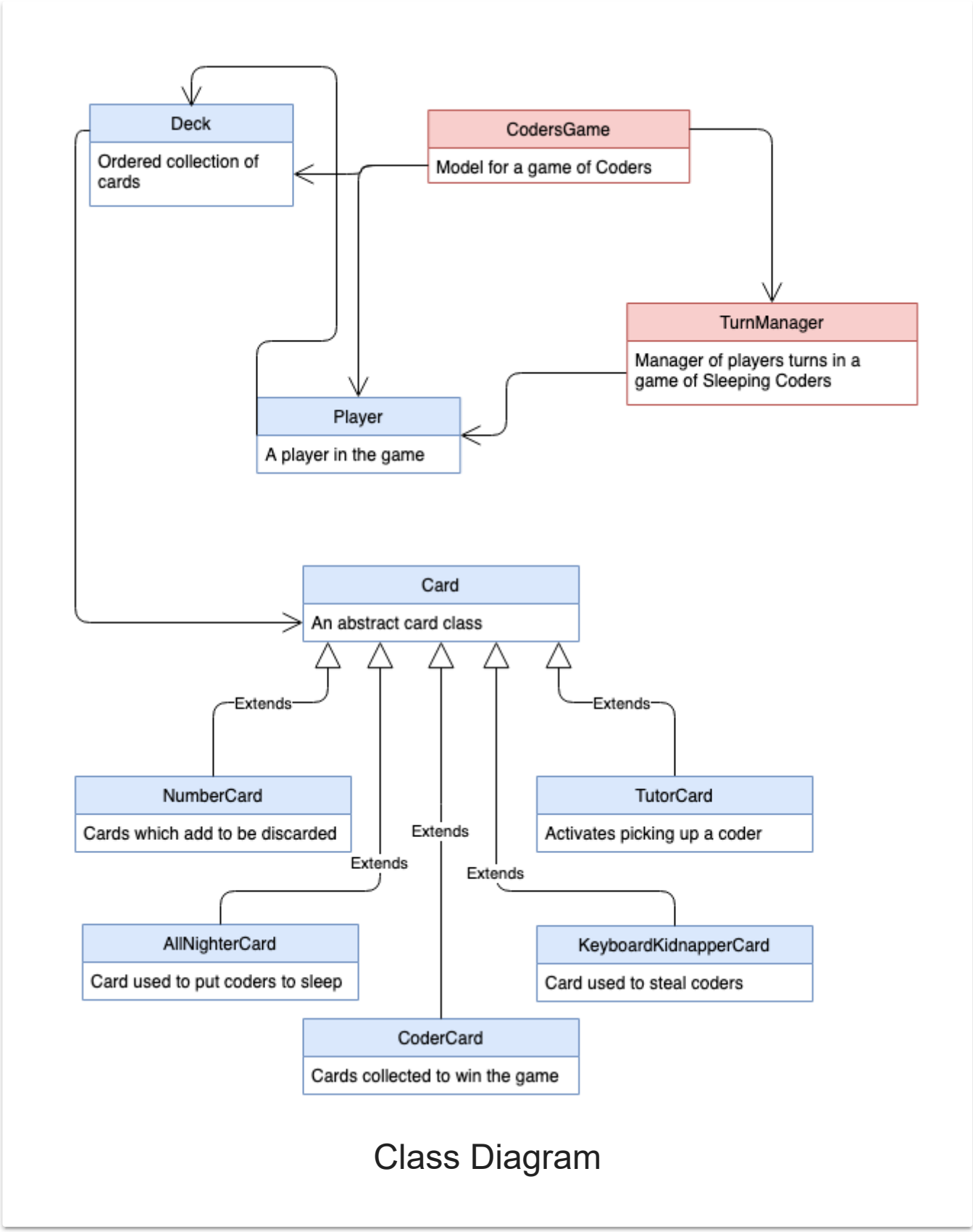
players.txt - a list of random player names

images/ - a directory of card images

Classes

In the diagram below, an arrow indicates that one class stores an instance of another class. An arrow with a head indicates that one class is a subclass of another class. Red classes are classes that have been provided already.

You are encouraged to modify the class structure by adding classes to reduce code duplication. This can include adding private helper methods.



Class Diagram

The type hints shown in the methods below are purely for your own understanding. Type hints do not need to be used in your assignment.

For all of the examples in the section below, you may assume that the following code has already been executed before each example.

```
>>> from a2 import *
>>> import a2_support
```

```
>>> peter = Player("Peter O'Shea")
>>> players = [peter, Player("Jason Storey"), Player("Mike Pham")]
>>> coders = [CoderCard("anna"), CoderCard("lochie")]
>>> deck = Deck([NumberCard(2), NumberCard(4), NumberCard(1)])
>>> game = a2_support.CodersGame(deck, coders, players)
```

It is advisable that you implement the `Card.play` and `Card.action` methods after implementing all of the classes (including Player and Deck)

Cards

A card represents a card in the game, which may be a coder card or a playable card held by the players in the game.

Card

The base card with methods for playing a card and performing a card action. Subclasses of Card should override these methods.

The following methods must be implemented:

play(self, player: Player, game: a2_support.CodersGame): Called when a player plays a card. Should remove the card from the player's hand and pickup a new card from the pickup pile in the game. Should also set the action that needs to be performed (see `CodersGame.set_action`). If no action needs to be performed, set the action to 'NO_ACTION'.

action(self, player: Player, game:a2_support.CodersGame, slot: int): Called when an action of a special card is performed. For tutor cards, this action is selecting a coder from the pile. For keyboard kidnapper and all-nighter cards, this is selecting a coder from another player's hand. For all other cards, no action needs to be performed. When playing a tutor card, the player refers to the current player. When playing a keyboard kidnapper card or all nighter card, the player refers to the player to which the coder belongs to.

__str__(self) -> str: Returns the string representation of this card, this should be 'Card()'.

__repr__(self) -> str: Same as __str__(self).

Each of the following classes should be a subclass of Card and should only alter methods required to implement the functionality of the card as described.

NumberCard

A card whose aim is to be disposed of by the player. A number card has an associated number value.

A number card should be constructed with **NumberCard(number)**.

When a number card is played, the game should move on to the next players turn.

get_number(self) -> int: Returns the number value of the card.

Examples

```
>>> card = NumberCard(3)
>>> card.get_number()
3
>>> str(card)
'NumberCard(3)'
>>> repr(card)
'NumberCard(3)'
```

CoderCard

A card which stores the name of a coder card.

A coder card should be constructed with **CoderCard(name)**.

get_name(self) -> str: Returns the card name.

Examples

```
>>> card = CoderCard("hanwei")
>>> card.get_name()
'hanwei'
>>> str(card)
'CoderCard(hanwei)'
>>> repr(card)
'CoderCard(hanwei)'
```

TutorCard

A card which stores the name of a tutor card.

A tutor card can be played by a player to pickup a coder card.

A tutor card should be constructed with **TutorCard(name)**.

When a tutor card is played, the game's action should be set to 'PICKUP_CODER'.

When a tutor card's action is performed, the selected card should be added to the player's deck, the position of the card in the sleeping coders' deck should be replaced with None, the action should be set back to 'NO_ACTION' and the game should move on to the next player.

get_name(self) -> str: Returns the card name

Examples

```
>>> card = TutorCard("luis")
>>> card.get_name()
'luis'
>>> str(card)
'TutorCard(luis)'
>>> repr(card)
'TutorCard(luis)'
```

KeyboardKidnapperCard

A card which, when played, allows the player to steal a coder card from another player.

When a keyboard kidnapper card is played the games action should be set to 'STEAL_CODER'.

When a keyboard kidnapper card's action is performed, the selected card should be added to the current player's deck and removed from its origin deck, the action should be set back to 'NO_ACTION' and the game should move on to the next player.

Examples

```
>>> card = KeyboardKidnapperCard()
>>> str(card)
'KeyboardKidnapperCard()'
>>> repr(card)
'KeyboardKidnapperCard()'
```

AllNighterCard

A card which, when played, allows the player to put a coder card from another player back to sleep.

When an all-nighter card is played, the game's action should be set to 'SLEEP_CODER'.

When an all-nighter card's action is performed, the selected card should be added to the first empty slot in the coders' pile and removed from its origin deck, the action should be set back to 'NO_ACTION' and the game should move on to the next player.

Examples

```
>>> card = AllNighterCard()
>>> str(card)
'AllNighterCard()'
>>> repr(card)
'AllNighterCard()'
```

Deck

A collection of ordered cards. A Deck should have a constructor with a signature of `Deck(starting_cards=None)`
When starting_cards is None, i.e. constructed with Deck(), it should be created with an empty list.
When starting_cards is provided, i.e. constructed with Deck(starting_cards=cards), it should be created with that list.

get_cards(self) -> List[Card]: Returns a list of cards in the deck.

get_card(self, slot) -> Card: Return the card at the specified slot in a deck.

top(self) -> Card: Returns the card on the top of the deck, i.e. the last added.

remove_card(self, slot): Removes a card at the given slot in a deck.

get_amount(self) -> int: Returns the amount of cards in a deck.

shuffle(self): Shuffles the order of the cards in the deck.

pick(self, amount: int=1) -> List[Card]: Takes the first 'amount' of cards off the deck and returns them.

add_card(self, card: Card): Places a card on top of the deck.

add_cards(self, cards: List[Card]): Places a list of cards on top of the deck.

copy(self, other_deck: Deck): Copies all of the cards from the other_deck parameter into the current deck, extending the list of cards of the current deck.

__str__(self) -> str: Returns the string representation of the deck, for a deck which contains cardA, cardB and cardC, this should be 'Deck(cardA, cardB, cardC)'.

__repr__(self) -> str: Same as __str__(self).

Examples

```
>>> card = NumberCard(3)
>>> all_nighter = AllNighterCard()
>>> last_card = NumberCard(2)
>>> cards = [card, all_nighter, last_card]
>>> deck = Deck(cards)
>>> str(deck)
'Deck(NumberCard(3), AllNighterCard(), NumberCard(2))'
>>> repr(deck)
'Deck(NumberCard(3), AllNighterCard(), NumberCard(2))'
>>> deck.get_cards()
[NumberCard(3), AllNighterCard(), NumberCard(2)]
>>> deck.get_amount()
3
>>> deck.get_card(0)
NumberCard(3)
>>> deck.get_card(2)
NumberCard(2)
>>> deck.top()
NumberCard(2)
>>> new_card = AllNighterCard()
>>> deck.add_card(new_card)
>>> deck.add_cards([card, all_nighter, last_card])
>>> deck.get_amount()
7
>>> deck.get_cards()
[NumberCard(3), AllNighterCard(), NumberCard(2), AllNighterCard(),
NumberCard(3), AllNighterCard(), NumberCard(2)]
>>> deck.remove_card(1)
>>> deck.remove_card(4)
>>> deck.get_cards()
[NumberCard(3), NumberCard(2), AllNighterCard(), NumberCard(3),
NumberCard(2)]
>>> deck.pick()
[NumberCard(2)]
>>> deck.pick(2)
[NumberCard(3), AllNighterCard()]
```

```
>>> deck.add_cards([card, all_nighter, last_card])
>>> deck.get_cards()
[NumberCard(3), NumberCard(2), NumberCard(3), AllNighterCard(),
NumberCard(2)]
>>> deck.shuffle()
>>> deck.get_cards()
[NumberCard(2), NumberCard(2), AllNighterCard(), NumberCard(3),
NumberCard(3)]
>>> another_deck = Deck()
>>> another_deck.get_cards()
[]
>>> another_deck.copy(deck)
>>> another_deck.get_cards()
[NumberCard(2), NumberCard(2), AllNighterCard(), NumberCard(3),
NumberCard(3)]
```

Player

A player represents one of the players in the game.

The Player class should be initiated with Player(name) and implement the following methods:

get_name(self) -> str: Returns the name of the player.

get_hand(self) -> Deck: Returns the players deck of cards.

get_coders(self) -> Deck: Returns the players deck of collected coder cards.

has_won(self) -> bool: Returns True if and only if the player has 4 or more coders.

__str__(self) -> str: Returns the string representation of the player, for a player with name of 'Luis', hand deck of 'hand' and coder deck of 'coders', this should be 'Player(Luis, hand, coders)'. See the examples for more details.

__repr__(self) -> str: Same as __str__(self).

Examples

```
>>> player = Player("Lochie Deakin-Sharpe")
>>> player.get_name()
'Lochie Deakin-Sharpe'
>>> player.get_hand()
Deck()
```

```
>>> player.get_coders()
Deck()
>>> str(player)
'Player(Lochie Deakin-Sharpe, Deck(), Deck())'
>>> repr(player)
'Player(Lochie Deakin-Sharpe, Deck(), Deck())'
>>> player.get_hand().add_card(NumberCard(3))
>>> player.get_hand().add_card(KeyboardKidnapperCard())
>>> player.get_hand().add_card(AllNighterCard())
>>> player.get_hand()
Deck(NumberCard(3), KeyboardKidnapperCard(), AllNighterCard())
>>> player.has_won()
False
>>> player.get_coders().add_card(CoderCard("ashleigh"))
>>> player.get_coders().add_card(CoderCard("kt"))
>>> player.get_coders().add_card(CoderCard("sanni"))
>>> player.has_won()
False
>>> player.get_coders().add_card(CoderCard("kaleb"))
>>> player.has_won()
True
>>> str(player)
'Player(Lochie Deakin-Sharpe, Deck(NumberCard(3),
KeyboardKidnapperCard(), AllNighterCard()), Deck(CoderCard(ashleigh),
CoderCard(kt), CoderCard(sanni), CoderCard(kaleb)))'
>>> repr(player)
'Player(Lochie Deakin-Sharpe, Deck(NumberCard(3),
KeyboardKidnapperCard(), AllNighterCard()), Deck(CoderCard(ashleigh),
CoderCard(kt), CoderCard(sanni), CoderCard(kaleb)))'
```

Marking

The marks in this section add to a total of 30, which will be scaled down to the 15 marks available for this assignment. Marks will be awarded based on the formula below:

```
Total Mark = (functionality + style)/2
```

Functionality Assessment

Your assignment will be marked automatically using the test suite provided. Marks will be awarded based on how many tests pass.

Syntax errors which prevent your assignment from running will be removed from your assignment where reasonable, deducting one mark per syntax error for up to 3 syntax errors. What is considered a reasonable fix to your assignment is left to the judgement of the tutor who marks your assignment and their judgement is final.

Class	Marks
`Card`	4
`NumberCard`	3
`TutorCard`	1
`CoderCard`	1
`AllNighterCard`	3
`KeyboardKidnapperCard`	3
`Deck`	6
`Player`	4
Total	25

Code Style

The style of your assignment will be assessed by one of the tutors, and you will be marked on the broad categories listed below.

	Description	Marks
Readability	Code is readable. Appropriate and meaningful identifier names have been used. Simple and clear code structure. Repeated code has been avoided.	**2**
Simplicity	Code has been simplified where appropriate and is not overly convoluted.	**1**

	Description	Marks
Documentation	Documented clearly and concisely, without excessive or extraneous comments.	**2**
Total		**5**

Feedback Sessions

In week 10, you must attend the practical that you are signed onto on mySI-net where the tutor who assessed your assignment will provide you with feedback.

When you arrive at the feedback session please take a seat but don't sit in the row furthest from the door as this will be where tutors will be providing feedback and it will be need to be kept free for privacy.

Allow for being in attendance at the practical for the full 2 hours, when your name is called walk over to the tutor who called your name and they will provide you with feedback on your assignment. You can leave once you have discussed your assignment with a tutor.

Assignment Submission

Your assignment must be submitted via the assignment one submission link on Blackboard. You must submit a Python file, `a2.py` , containing your implementation of the assignment.

Late submission of the assignment will **not** be accepted. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. Multiple submissions are allowed, so ensure that you have submitted an almost complete version of the assignment *well* before the submission deadline. Your latest on-time, submission will be marked. Ensure that you submit the correct version of your assignment. An incorrect version that does not work **will** be marked as your final submission.

In the event of exceptional circumstances, you may submit a request for an extension. See the course profile for details of how to apply for an extension. Requests for extensions must be made **no later** than 48 hours prior to the submission deadline. The expectation is that with less than 48 hours before an assignment is due it should be substantially completed and submittable. Applications for extension, and any supporting documentation (e.g. medical certificate), must be submitted via my.UQ. You must retain the original documentation for a *minimum period* of six months to provide as verification should you be requested to do so.

Change Log

Version 1.0.2 - Sep 16

Assignment Sheet

- Explicitly included __str__ and __repr__ methods for Player.
- Fixed the typo when describing the starting_cards parameter.

Version 1.0.1 - Sep 2

Assignment Sheet

- Added various grammar improvements.
- Clarified the wording of Card class.
- Clarified the wording of the play and action method.
- Fixed quotations in code examples.
- Included import in code examples.
- Fixed the contradiction about Player class construction.
- Fixed the type of parameter for the Decks copy method.

Support Files

- Fixed the docstring for the `Game.pick_card` method so that it has the correct return type.