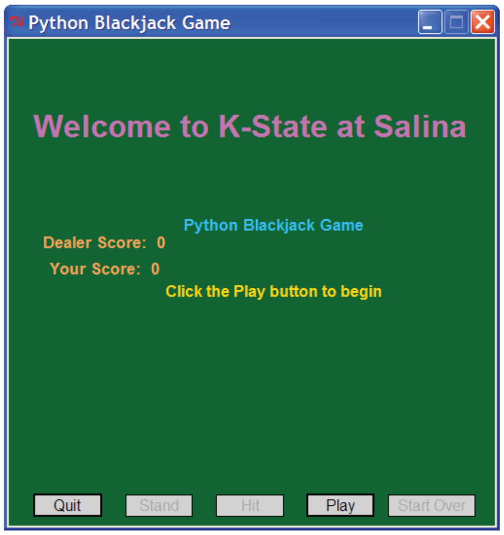
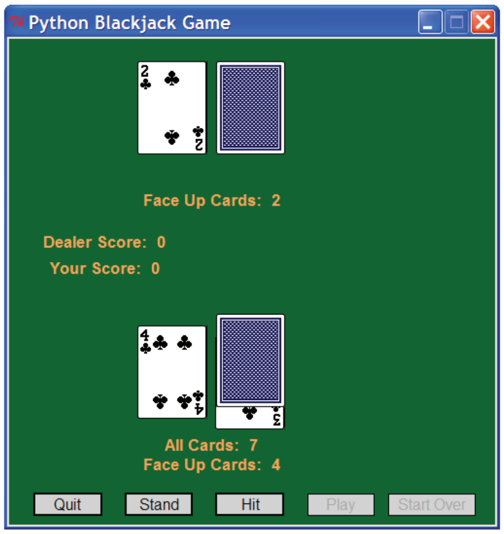
GRAPHICAL BLACKJACK GAME

The Python Programming class that I taught in the spring 2009 semester studied how to program graphical user interfaces. The final project of the class was to design an object oriented black jack card game with a graphical user interface. Described on the following pages is my implementation.

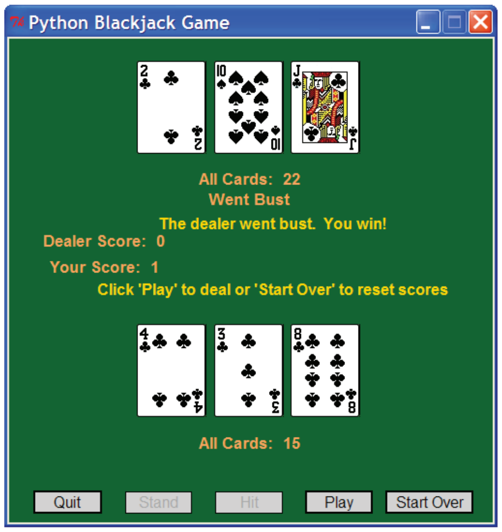
This class was taught as a lower level CMST elective. Thus it was an introductory programming class. The text book that we used was: Python Programming: An Introduction To Computer Science, by John Zelle. This book uses graphics programming as a means to introduce object oriented programming. Thus, the book provides an object oriented module called graphics, which has facilities for displaying basic graphics primitives such as circles, squares, lines and also has a method to catch mouse click events returning the x, y coordinates of where the mouse was clicked. Based on this module, a basic button class is developed. This graphics module is an object oriented wrapper around Python's standard tkinter module. The graphics module is in one sense an improvement on tkinter because it is object oriented, but much of the functionality of tkinter is removed, including the main loop that generates function backs based on event, such as mouse clicks and keyboard data entry. Thus, the first order of business to seriously use this module was to write a class to provide event driven method call backs. I wrote this class and used it to teach the basic concepts of asynchronous event driven programming. The journey from having a very basics graphics module to having an event driven programming framework to a final working black jack game was excellent from a teaching perspective. From talking to the students, I'm quite certain that a number of students advanced tremendously as programmers from this assignment. However, from just a pure programming perspective, the graphics framework that we used was fairly poor. Starting with a mature graphics framework, such as wxPython, as I did with the Multi-party Chat Client and Server and Graphical Weather Forecast Viewer applications is a much preferred way to write a graphical application. However, my prime objective here was not to write a black jack application, but to teach object oriented programming, and that most certainly was accomplished.



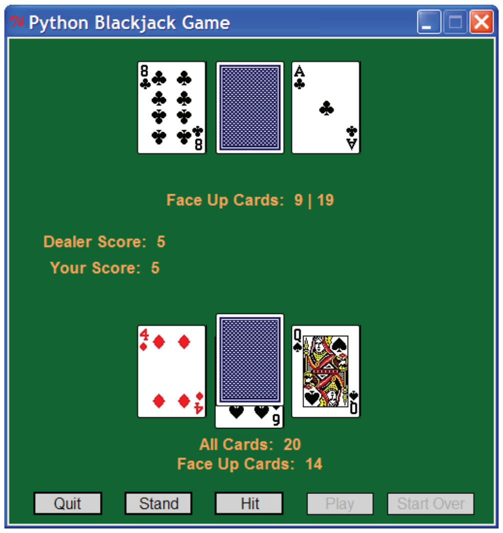
The splash screen.
The exact message can be easilty changed in the source code if desired.



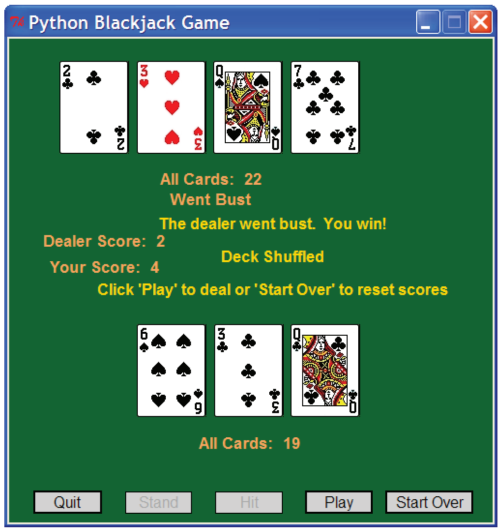
After the initial deal. Player cards are the bottom. Dealer cards are on the top.
The second card, with another card over it, is the face down card
that the dealer can not see.



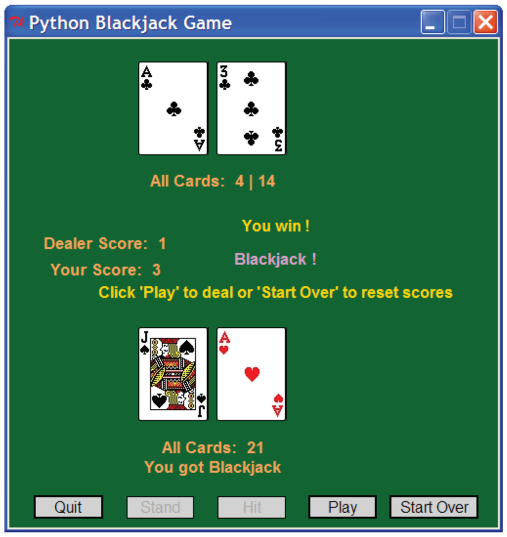
At the end of a hand. The number of games won by the dealer
and player are in the middle of the screen.



Ace cards require exceptions to how the cards are counted
since can count as either 1 or 11.



At the end of another game. The cards for each player stay mostly centered
regardless of how many cards are in the hand. Figuring out where to put
the cards was about the hardest par of the program.



Player got a blackjack.

DOCUMENTATION OF BLACKJACK GAME

Here is a little bit of documentation about the more important functions and classes from the blackjack program. This documentation is auto-generated from the Python source code and doc strings contained in the source code.

`blackjack.py`

A Blackjack game

class `blackjack.`**`Blackjack`***(win)*

Controller or coordinator of the game. All buttons belong to this class.

`go()`

This the entry and exit point to/from main. The MouseTrap engine exits to here, then we exit back to main

`hit()`

Button call back – give player another card and let dealer play then check if game over yet. Otherwise, back to the Hit or Stand choice.

`play()`

Button call back to begin a hand. It does some clean up from the previous hand and does the initial deal to the dealer and player. Unless the player or dealer got a Blackjack (21 from just the two cards in the initial deal), then after this function exits, players next pick to Hit or Stand

`quit()`

Button call back to exit the program. It just returns False so that the MouseTrap engine will exit

`stand()`

Button call back – player does not need another card. Let the dealer play now, which finishes the hand.

`startOver()`

Button click handler for reseting the game

`blackjack.`**`main()`**

The main() program that gets everything started. It basically just creates the graphics window, creates the Blackjack (controller) object and gets it started. Another external program that imports this module could easily do the same and start a Blackjack game in it's own graphics window.

`bkJplayers.py`

The participants (Dealer, Player) of a Blackjack game

class `bkJplayers.`**`Contestant`***(win, deck, xmin, ymin, xmax, ymax)*

Common stuff between player and dealer. Don't create an instance of this class - that's what player and dealer are for. They inherit form this class, which on exists to cut down on code duplication.

`busted()`

Just return if contestant went bust

`clear()`

Clear the stuff from last hand and get ready for next

`finish()`

Not really a turn, just flip the downcard and report final score.

`getScore()`

Pick out the best score and return it

`hit()`

Called from player.hit and from dealer.

`standing()`

Just return if contestant is standing - satisfied with cards in hand

`upScore()`

Pick out the best score of the only the face up cards and return it

`wentBust()`

Set the bust status to true and show a message saying went bust

class `bkJplayers.`**`Dealer`***(win, deck, xmin, ymin, xmax, ymax)*

Functions specific to the dealer (different than player). The init function from Contestant does what's needed for this class.

`deal()`

The initial deal to dealer - one face up, one face down

`play()`

Dealer finishes the hand after player is finished

`setStand()`

Set the stand status to true and display a message that the dealer stands.

`turn()`

Dealer takes a to turn see what they come up with

class `bkJplayers.`**`Player`***(win, deck, xmin, ymin, xmax, ymax)*

Functions specific to the player (different than dealer) The init function from Contestant does what's needed for this class.

`deal()`

The initial deal to player - one card face up, one down

`bkJhand.py`

Each player's hand of cards for a Blackjack game

class `bkJhand.`**`Twohands`***(win, xmin, xmax, ycenter, yscore)*

One hand for all cards and another hand for just the face-up cards. The later is just for calculating the points of the face up cards, which your opponent can see. This class is a wrapper around the Hand class so that the player classes only have to talk to one Hand class.

In hindsight, this may not have been the best approach. I could have changed the Hand class to make note of the value of the face down card and kept two scores and two scoreBoxes.

`clear()`

Reset the Hand and remove displayed cards for a new hand

`dealdown(card)`

Deal a card Face down

`dealup(card)`

Deal a card Face up

`flip2nd()`

Flip the 2nd, face down, card over. Remove the face up score and show the score for all cards.

`score()`

Return the list of scores

`scoreBoxAMesg(mesg)`

Display a message in the all card score box

`scoreBoxBMesg(mesg)`

Display a message in the face up card score box

`setDealer()`

Dealer calls this, just so we know this is the dealer's hand. It sets a Boolean variable Only difference is how the face down card is shown.

`showScore()`

Display the score for all cards

`upScore()`

Return the list of scores

`upShowScore()`

Display the score for the face up cards

class `bkJhand.`**`Hand`***(win, xmin, xmax, ycenter, yscore, label)*

Manage the set of card in the hand. Keeps a list of the cards, shows them on the screen and counts the points. Also has functions to display the points for the hand, but the player and dealer determine when to do that.

`clear()`

Clear the screen and the hand of cards in prep for next hand

`dealdown(card)`

Add a dealt card face down

`dealup(card)`

Add a dealt card face up

`flip2nd()`

The second card is first dealt face down. At the last turn of each dealer and player, it is turned up.

`hideScoreBox()`

Clear the score box message area

`score()`

Return the set of current non-bust scores. If best is 21, return it as the only item in the score list.

`scoreBoxMesg(mesg)`

Display some special message in the score box

`showScore()`

Display the current score for the hand

`cards.py`

Card and Deck classes for games played with playing cards - Blackjack being the first...

class `cards.`**`Card`***((rank, suit))*

Playing card for games such as Blackjack - this is the non-drawn card, DrawnCard class extends this class to add drawing the card on the screen.

Note: Deck's deal method returns a tuple of form (rank, suit), and Card's __init__ method takes the same tuple to create a deck, so they work nicely together, but may require special care if not used together. This makes it easier to create the type of card needed.

`BJValue()`

Returns how may points the card is worth in Blackjack. Returns Ace as 1 point, so counting it as either 1 or 11 point must be done some where else

`draw(center)`

Just a stub - it does nothing

`drawDown(center)`

Just a stub - it does nothing

`flip()`

Just a stub - it does nothing

`getImageDown()`

Just a stub - it does nothing

`getImageName()`

Just a stub - it does nothing

`getRank()`

Returns in range 1 (Ace) to 13 (King) for card

`getSuit()`

Returns suit (one of 'd', 'c', 'h', 's') for diamonds, clubs, hearts or spades

`moveTo(new)`

Just a stub - it does nothing

`undraw()`

Just a stub - it does nothing

class `cards.`**`DrawnCard`***(win, cardTuple)*

Playing card for games such as Blackjack - this is for a card object that we want to display in a graphics window. It extends the generic card.

`draw(center)`

Show the image of the card face up

`drawDown(center)`

Show the image of the card face down

`flip()`

Flip the card over. Switch the image between face up and face down

`getImageDown()`

Returns file name for face down image of the card

`getImageName()`

Returns file name for face up image of the card

`moveTo(new)`

Move the card image to a new location. new is a Point object

`undraw()`

Remove the image of the card from graphics window

class `cards.`**`CoveredCard`***(win, cardTuple)*

This is a specialized card for games such as Blackjack. Like the DrawnCard, it draws an image of the card, but in the case when card is supposed to be face down, instead of displaying the image for the back of the card, it first shows the card face up, and then covers most of the card with a face down image. This is to communicate to the user that to their opponents, this is a face down card, but it lets them see enough of the card to know what it is. It extends the DrawnCard Class. While self.face = True (it is face up) it behaves the same as DrawnCard class.

`drawDown(center)`

Show the image of the card covered by a face down card

`flip()`

Flip the card over. Switch the image between face up and face down

`moveTo(new)`

Move the card image to a new location. new is a Point object

`undraw()`

Remove the image of the card from graphics window

class `cards.`**`Deck`**

A deck of 52 playing cards

`cardsLeft()`

Returns how many undealt cards are left in the deck

`deal()`

Deal one card from the deck. Returns a card object.

`shuffle()`

Just what the function name says

`cards.`**`unitTest()`**

Some unit testing code for deck and card classes: Card, DrawnCard and Covered Card

`textboxes.py`

Part of the Blackjack game. Displays the scoreboard and other messages.

class `textboxes.`**`ScoreBox`***(win, center, begin=None)*

Just a Text area to keep the score in

`setScore(score=None)`

Show the score with the begin message, call with no parameters to remove the score from the screen

class `textboxes.`**`MessageBox`***(win, center)*

Just a Text area to display a message in

`setColor(color)`

Change the text color

`setMsg(msg=None)`

Show a message, call with no parameters to remove the score from the screen

`guiengine.py`

A simple mouse click event catching engine with button call backs and a simple Button widget.

class `guiengine.`**`MouseTrap`***(win)*

A class to catch mouse click events on buttons and run the appropriate call back function. This provides a framework for asychronous programming using the event catching and call back model.

Buttons are registered with the trap engine so that they are watched for a mouse click over the button.

Important notes about call back functions:

1) If the class extends the Button class and has a function named run(), then it will be the call back. That implies that the class has only one button. Probably more useful is to use the base Button class or an extension of it and use the Button classes setRun() function to specifiy a function of choice for the call back. (See the Button class for more on info.)

2) Rather using a parameter to init or other special function to specify which Button is the Quit button, the mechanism used is that a call back function can cause the MousTrap engine to exit (and presumably, the rest of the program) by having the function return a Boolean False value. Thus any function that wants the program to continue watching for mouse clicks needs to return True. This can allow more than one exit path for error handling and avoids needing to register a button as a Quit button. But don't forget in coding the call back function to return the desired value of True or False.

`registerButton(button)`

Register the button class, each button needs to have a run() method and a clicked() boolean method (part of the base Button class).

`run()`

Run the event catcher waiting for mouse clicks.

class `guiengine.`**`Button`***(win, center, width, height, label)*

A button is a labeled rectangle in a window. It is activated or deactivated with the activate() and deactivate() methods. The clicked(p) method returns true if the button is active and p is inside it.

`activate()`

Sets this button to ‘active’.

`clicked(p)`

Returns true if button active and p is inside

`deactivate()`

Sets this button to ‘inactive’.

`getLabel()`

Returns the label string of this button.

`run()`

The default event handler. It either runs the handler function set in setRun() or it raises an exception.

`setRun(function)`

set a function to be the mouse click event handler

`setUp(win, center, width, height, label)`

set most of the Button data - not in init to make easier for child class methods inheriting from Button. If called from child class with own run(), set self.runDef

`graphics.py`

Simple object oriented graphics library for Introduction to Programming using Python by John Zelle, Ph.D.