Suggested answers: These are not the only answers accepted, but should give a clear idea of what we had in mind. Also, there is a lot of explanatory verbiage and alternative answers given for each problem which you were NOT required to give.

1. There are many, many ways to program this GUI. One could have the pull function just randomly reconfigure the text of three Labels and then use .cget to recover the values for counting up $ for payoff. And so on... The easiest way seemed to me to keep a separate array for the symbols and their associated Labels and reconfig to change the Labels. This gives the following script:

```python
from random import *
from Tkinter import *
root=Tk()
#set up the payoff display
payoff=Label(root,text="You get 0 coins back")
payoff.grid(row=0, column=0, columnspan=3)
#then the displays for the random pattern symbols
symbols=[] #symbols go here
displays=[] #labels for their display go here
for i in range(3):
    symbols=symbols+["-"] #initially set them all -
    displays=displays+[Label(root,text=symbols[i])]
displays[i].grid(row=1,column=i)
#finally the PULL button and its call back function
def play():
    for i in range(3):
        if uniform(0,1)<0.5: #random 50-50 test
            symbols[i]="$"
        else:
            symbols[i]="-"
        displays[i].config(text=symbols[i])
    #now we must count the $ and set the payoff message
    count=0
    for i in range(3):
        if symbols[i]=="$":
            count=count+1
    coins_back=["0 coins","0 coins","1 coin","3 coins"] #payback list for count
    payoff.config(text="You get "+coins_back[count]+" back.")
    return
pull=Button(root,text="PULL",command=play)
pull.grid(row=2,column=0,columnspan=3)
root.mainloop()
```

2. The Python ord function returns the integer,decimal ASCII code of a character (string of length 1) given as argument. The chr function is its inverse. The key point for you to know in this question is that the ASCII codes of the digit and letter symbols are laid out in the normal sequence 0 to 9 and a to z.

2a. ord(“4”)==52, so ord(“9”) is 5 greater = 57. Then ord(“0”) is 4 less = 48 and chr(57) is, of course, “9”. The expression chr(ord(“3”)+ord("5")-ord("0")) has the value “8”. This is a clever
way to do single digit addition of digit symbols (Of course, str(int("3") + int("5")) is a better way and works for any number of digits).

2b. The decimal codes of “1”, “2” and “3” are 49, 50 and 51, with the hexadecimal equivalents 31, 32 and 33 (31 hex is 3*16+1=49 decimal).

2c. “g” is the 7th letter of the alphabet and “b” is the 2nd, so ord(“g”) - ord(“b”) has the value 5. The next expression is a clever way to convert from lower to upper case if you have forgotten the difference in the ASCII codes for the cases, chr(ord(“d”)+ord(“X”)-ord(“x”)) is “D”.

3. I found it convenient to make a sub-function to test if a single word fits the pattern. Then we just have a loop over the word list to find the matches. The point is that we want to break out of the character testing loop as soon as we find a mismatch. The same thing could be done in a single function by using a logical “flag” which is negated when a mismatch is found. My solution:

```python
def fits(pattern, word):
    # this gives true or false for the pattern match of a single word
    if len(pattern) != len(word):
        return 0  # wrong length
    for i in range(len(word)):
        if pattern[i] != "-" and pattern[i] != word[i]:
            return 0  # required letter position doesn't match
    return 1  # all was OK

def match(pattern, words):
    # given the sub-function fits, the overall function is trivial
    matchlist = []
    for word in words:
        if fits(pattern, word):
            matchlist = matchlist + [word]
    return matchlist

Here is an alternative version in a single function:

```python
def match(pattern, words):
    matchlist = []  # build up list in matchlist
    for word in words:
        # for every word in given list
        matches = 0  # as default, assume it doesn't match
        if len(word) == len(pattern):  # are they the same length?
            matches = 1  # then assume they will match until proven otherwise
            for i in range(len(word)):
                if pattern[i] != "-" and pattern[i] != word[i]:  # if the position
                    matches = 0  # isn't "don't care" and they not equal, no match
                    if matches:  # after all that, is matches true?
                        matchlist = matchlist + [word]  # add the word to the list
        return matchlist
```
4. Fitting the errors into the proper debugging categories can involve subtle understanding, so we gave most credit for finding the errors and took off less for putting them into the wrong category. By the way, this is a classic selection sort. We didn’t ask you to say that, but recognizing it might have made finding the errors easier.

4a. These produce errors in the translation to machine language and involve syntax errors and errors of format, usually mixed indentation in a single command block. In this messed up function there is a missing right parenthesis in the if command. There is mixed indentation in a block following the if. The command next=next+1 must line up with “shortest=next” or with the if. The latter is logically correct, so it is really an error in both categories a and c.

4b. These are called “run-time” errors (we would say “execute-time” to avoid confusion with “running” (translating) a module) and are the ones we deal with by the try/except commands. They range from incompatible types (adding strings and ints) through index out of bounds to undefined names. In this function “length” is an undefined function name. The programmer clearly meant the built-in function “len”. The name “top” is used before being defined. The function should begin with the command, top=0.

4c. These are the hardest to find. They produce neither translate-time nor execute-time error messages, but give incorrect results with no warning. Careful comparison of generated and expected results is required to find them. In this function the attempt to switch the values in position top and position shortest destroys the value in a[top] before it can be moved. One needs to save it with a temporary name, say temp=a[top] and then set a[shortest]=temp after a[top]=a[shortest]. Also the indentation of the return is incorrect, although syntactically OK. It should be aligned under the first while command rather than the second. As it stands, the function will return after only finding the first “shortest” word.

5. These are all pretty straight-forward:

5a. The class definition (8 points were given for it):

```python
class card:
    def __init__(self,suit,name):
        self.suit=suit
        self.name=name
    return
    def phrase(self):
        return "The %s of %s" % (self.name,self.suit)
    def rank(self):
        ranks=["2","3","4","5","6","7","8","9","10","J","Q","K","A"]
        irank=0
        while irank<len(ranks):
            if self.name==ranks[irank]:
                return irank
            irank=irank+1
        return 0  #shouldn't ever get here
```
5b. Making a deck (5 points were given for it):

def new_deck():
    #this is a simple nested for loop over the suit and name lists
    deck=[]
    for suit in ["spades","hearts","diamonds","clubs"]:
        for name in ["2","3","4","5","6","7","8","9","10","J","Q","K","A"]:  
            deck=deck+[card(suit,name)]
    return deck

5c. (7 points were given for it) This involves setting an empty list, say hand=[] and concatenating cards onto it which are randomly selected, using the choice function in the random module, from the deck. The only problem is to make sure that you don’t select a card that has already been added to the list (hand). There are two ways to go with this. One is to make a trial selection and repeat it until you have a card that is not already in hand. This just requires a while loop such as

from random import *  #always needed
hand=[]
for i in range(5):
    try=choice(deck)
    while try in hand:
        try=choice(deck)  #retry selection until not in hand
    hand=hand+[try]
return hand

This is fairly elegant, but a more obvious solution is to select a random position, add it to hand and then slice it out of deck so that it can’t be used again. Here is the function with that algorithm:

from random import *  #always needed
def deal(deck):
    #this is called sampling without replacement
    hand=[]
    for ihand in range(5):  
        icard=choice(range(len(deck)))  #pick random card, randint would work
        hand=hand+[deck[icard]]  #put it in the hand
        deck=deck[:icard]+deck[icard+1:]  #slice it out of the deck
    return hand

5d. (5 points were given for it) This is pretty trivial, but I figured everyone might be running out of steam here and wanted something easy.

def order(card1,card2):
    if card1.rank()>card2.rank():  
        return card2,card1  #or we accepted reverse order too
    else:
        return card1,card2

#have a good summer!