Introduction to Programming CS101

Fall 2011

Lecture #11

Advertisement: CS109

Did you like CS101 so far? Would you like to learn more about programming?

Starting next spring, the CS department is offering CS109 "Programming Practice". It is especially meant for students who did NOT learn programming in high school and who want to learn and practice more programming than we can teach in CS101.

If you consider majoring computer science but did not learn programming in high school, then CS109 is highly recommended, as the second year computer science courses assume you are familiar with the material taught in CS109.

Please visit the CS109 homepage, if you're interested:

`http://tclab.kaist.ac.kr/~otfried/cs109/`

Roadmap

Last week we learned

- Controlling hardware with software

Roadmap

Last week we learned

- Controlling hardware with software

This week we will learn

- Objects
  - Constructors
  - String conversion
- User interface programming

Blackjack

There are 52 cards. Each card has a face and a suit. The suits are clubs, spades, hearts, and diamonds. The faces are 2, 3, . . . , 10, Jack, Queen, King, and Ace. The value of a card is the number for a number card, 11 for an Ace, and 10 for Jack, Queen, and King.

Blackjack

There are 52 cards. Each card has a face and a suit. The suits are clubs, spades, hearts, and diamonds. The faces are 2, 3, . . . , 10, Jack, Queen, King, and Ace. The value of a card is the number for a number card, 11 for an Ace, and 10 for Jack, Queen, and King.

```
class Card(object):
  """A Blackjack card."""
  pass

card = Card()
card.face = "Ace"
card.suit = "Spades"
card.value = 11
```

Methods

We do not really need the `value` attribute, since the value can be computed from the `face` attribute. We add a `value()` method to our `Card` class:

```
class Card(object):
  """A Blackjack card."""
  def value(self):
    if type(self.face) == int:
      return self.face
    elif self.face == "Ace":
      return 11
    else:
      return 10
```

Methods

We do not really need the `value` attribute, since the value can be computed from the `face` attribute. We add a `value()` method to our `Card` class:

```
class Card(object):
  """A Blackjack card."""
  def value(self):
    if type(self.face) == int:
      return self.face
    elif self.face == "Ace":
      return 11
    else:
      return 10
```

method of `Card`

Methods

We do not really need the `value` attribute, since the value can be computed from the `face` attribute. We add a `value()` method to our `Card` class:

```
class Card(object):
  """A Blackjack card."""
  def value(self):
    if type(self.face) == int:
      return self.face
    elif self.face == "Ace":
      return 11
    else:
      return 10
```

`method of Card`

`self` refers to the object itself inside the method.

Methods

We can create and use `Card` objects:

```
>>> card = Card()
>>> card.face = "Ace"
>>> card.suit = "Spades"
>>> card_string(card)
'an Ace of Spades'
>>> card.value()
11
```

Methods

We can create and use `Card` objects:

```
>>> card = Card()
>>> card.face = "Ace"
>>> card.suit = "Spades"
>>> card_string(card)
'an Ace of Spades'
>>> card.value()
11
```

We need nicer syntax to create `Card` objects:
`Card(8, "Clubs")`.

And `card_string` should be a method of `Card`.

Constructors

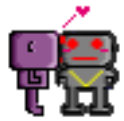
Objects can have a special method `__init__`, called a constructor. Whenever an object of this type is created, the constructor is called.

```
FACES = range(2,11) + ['Jack','Queen','King','Ace']
SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

class Card(object):
  """A Blackjack card."""

  def __init__(self, face, suit):
    assert face in FACES and suit in SUITS
    self.face = face
    self.suit = suit
```

Constructors

Now creating cards is elegant:

```
hand = [ Card("Ace", "Spades"),
         Card(8, "Diamonds"),
         Card("Jack", "Hearts"),
         Card(10, "Clubs") ]
```



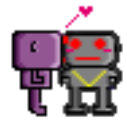
Constructors

Now creating cards is elegant:

```
hand = [ Card("Ace", "Spades"),
         Card(8, "Diamonds"),
         Card("Jack", "Hearts"),
         Card(10, "Clubs") ]
```

Let's change `card_string(card)` into `card.string()`:

```
  def string(self):
    article = "a "
    if self.face in [8, "Ace"]: article = "an "
    return (article + str(self.face) +
            " of " + self.suit)
```



Constructors

Now creating cards is elegant:

```
hand = [ Card("Ace", "Spades"),
         Card(8, "Diamonds"),
         Card("Jack", "Hearts"),
         Card(10, "Clubs") ]
```

Let's change `card_string(card)` into `card.string()`:

```
  def string(self):
    article = "a "
    if self.face in [8, "Ace"]: article = "an "
    return (article + str(self.face) +
            " of " + self.suit)

for card in hand:
  print card.string(), "has value", card.value()
```

String conversion

We can make conversion to strings even nicer: `str(card)` calls the special method `__str__`:

```
def __str__(self):
  article = "a "
  if self.face in [8, "Ace"]: article = "an "
  return (article + str(self.face) +
          " of " + self.suit)
```

String conversion

We can make conversion to strings even nicer: `str(card)` calls the special method `__str__`:

```
  def __str__(self):
    article = "a "
    if self.face in [8, "Ace"]: article = "an "
    return (article + str(self.face) +
            " of " + self.suit)
```

Now we can write:

```
for card in hand:
  print card, "has value", card.value()
```

String conversion

We can make conversion to strings even nicer: `str(card)` calls the special method `__str__`:

```
def __str__(self):
  article = "a "
  if self.face in [8, "Ace"]: article = "an "
  return (article + str(self.face) +
          " of " + self.suit)
```

Now we can write:

```
for card in hand:
  print card, "has value", card.value()
```

`print` automatically converts its arguments to `str`

More chicken!

Let's improve our `Chicken` object by adding a constructor, `move` and `jump` methods.

More chicken!

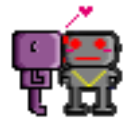
Let's improve our `Chicken` object by adding a constructor, `move` and `jump` methods.

```
class Chicken(object):
  """Graphic representation of a chicken."""
  def __init__(self, hen = False):
    layer = Layer()
    # make all the parts
    self.layer = layer
    self.body = body
    self.wing = wing
    self.eye = eye

  def move(self, dx, dy):
    self.layer.move(dx, dy)
```

A Deck object

Let's create another object that represents a shuffled deck of 52 cards. We only need one method: drawing a card from the top of the deck:

```
class Deck(object):
  """A deck of cards."""
  def __init__(self):
    "Create a deck of 52 cards and shuffle them."
    self.cards = []
    for suit in SUITS:
      for face in FACES:
        self.cards.append(Card(face, suit))
    random.shuffle(self.cards)

  def draw(self):
    """Draw the top card from the deck."""
    return self.cards.pop()
```



A Deck object

```
num_players = 3
num_cards = 5
deck = Deck()
hands = []
for j in range(num_players):
  hands.append([])

for i in range(num_cards):
  for j in range(num_players):
    card = deck.draw()
    hands[j].append(card)
    print "Player", j+1, "draws", card

for j in range(num_players):
  print ("Player %d's hand (value %d):" %
         (j+1, hand_value(hands[j])))
  for card in hands[j]:
    print " ", card
```

A list of lists (one for each player)

Blackjack

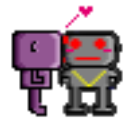
Time to play Blackjack:

```
You are dealt a 6 of Hearts
Dealer is dealt a hidden card
You are dealt a 3 of Spades
Dealer is dealt a 9 of Hearts
Your total is 9
Would you like another card? (y/n) y
You are dealt an Ace of Clubs
Your total is 20
Would you like another card? (y/n) n
The dealer's hidden card was a 10 of Spades
The dealer's total is 19
Your total is 20
The dealer's total is 19
You win!
```

Equality

The comparison operators ==, !=, < etc. do not work automatically for objects:

```
>>> Card(8, "Diamonds") == Card(8, "Diamonds")
False
>>> Card(8, "Diamonds") == Card(9, "Diamonds")
False
```



Equality

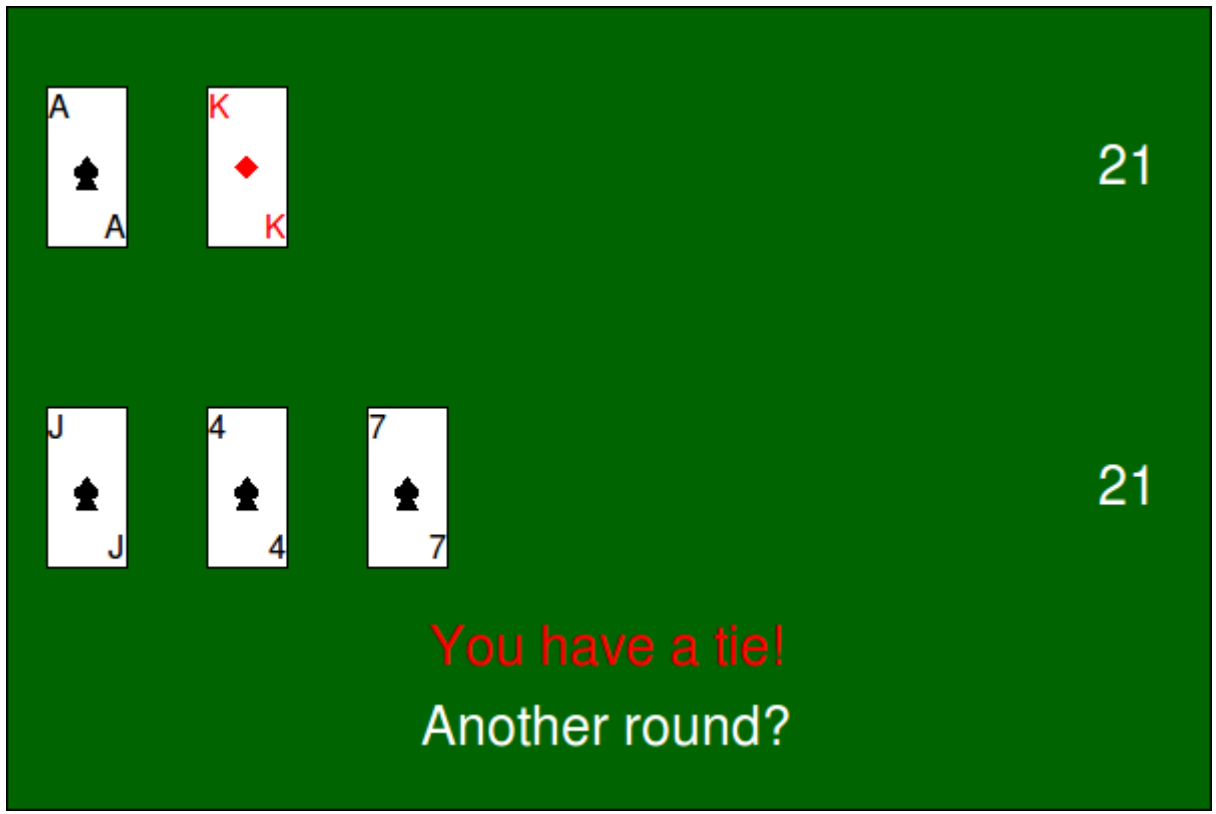
The comparison operators ==, !=, < etc. do not work automatically for objects:

```
>>> Card(8, "Diamonds") == Card(8, "Diamonds")
False
>>> Card(8, "Diamonds") == Card(9, "Diamonds")
False
```

We can define equality through the special method __eq__:

```
  def __eq__(self, rhs):
    return (self.face == rhs.face and
            self.suit == rhs.suit)
  def __ne__(self, rhs):
    return not self == rhs
```

Blackjack with Graphics



Blackjack with Graphics

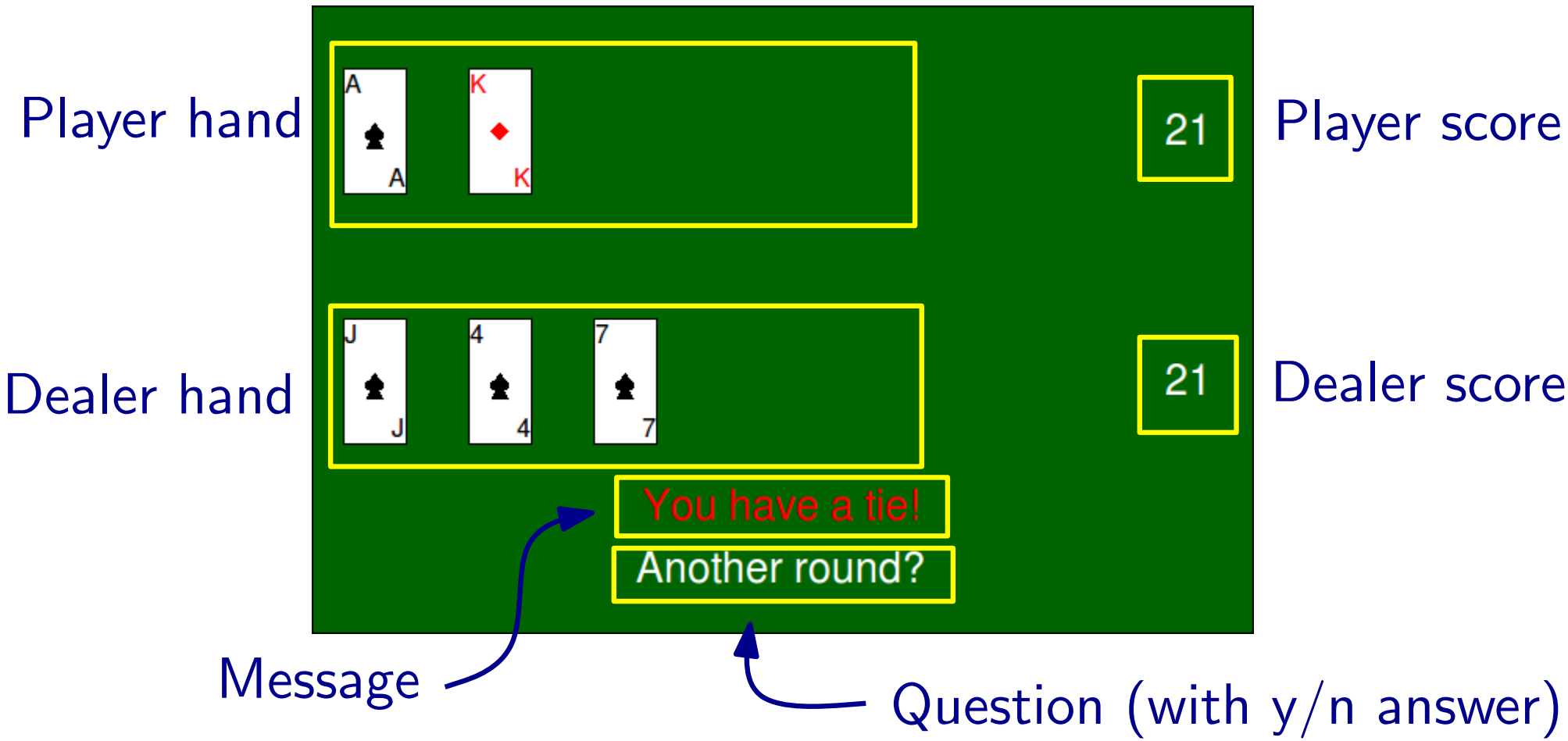


Table object

A `Table` represents the Blackjack table. It provides the following methods:

- `clear()` clear everything
- `close()` close window and end game
- `set_score(which, text)` where `which in [ 0, 1 ]`
- `set_message(text)`
- `ask(prompt)` waits for `y` or `n` and returns `True` or `False`

Table object

A `Table` represents the Blackjack table. It provides the following methods:

- `clear()` clear everything
- `close()` close window and end game
- `set_score(which, text)` where `which in [ 0, 1 ]`
- `set_message(text)`
- `ask(prompt)` waits for `y` or `n` and returns `True` or `False`

`Table` has two attributes `dealer` and `player`. These are `Hand` objects that represent the hand on the table.

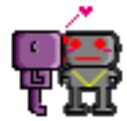


Table object

A `Table` represents the Blackjack table. It provides the following methods:

- `clear()` clear everything
- `close()` close window and end game
- `set_score(which, text)` where `which in [ 0, 1 ]`
- `set_message(text)`
- `ask(prompt)` waits for `y` or `n` and returns `True` or `False`

`Table` has two attributes `dealer` and `player`. These are `Hand` objects that represent the hand on the table.

Methods of `Hand` objects:

- `clear()`
- `add(card, hidden = False)`
- `show()` shows all hidden cards
- `value()` return value of hand

Waiting for user interaction

The `Table.ask(prompt)` method must wait for the user to press a key:

```
def ask(self, prompt):
  self.question.setMessage(prompt)
  while True:
    e = self.canvas.wait()
    d = e.getDescription()
    if d == "canvas close":
      sys.exit(1)
    if d == "keyboard":
      key = e.getKey()
      if key == 'y':
        return True
      if key == 'n':
        return False
```

Waiting for user interaction

The `Table.ask(prompt)` method must wait for the user to press a key:

```
def ask(self, prompt):
  self.question.setMessage(prompt)
  while True:
    e = self.canvas.wait()
    d = e.getDescription()
    if d == "canvas close":
      sys.exit(1)
    if d == "keyboard":
      key = e.getKey()
      if key == 'y':
        return True
      if key == 'n':
        return False
```

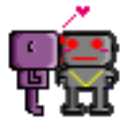
e is an event object

User interface programming

Programs with a graphical user interface (GUI) are structured around events. Most of the time, the program just waits for an event to happen.

Events are for instance:

- key presses
- window is minimized, maximized, or closed
- mouse is moved
- mouse button is pressed
- cursor enters window or leaves window



User interface programming

Programs with a graphical user interface (GUI) are structured around events. Most of the time, the program just waits for an event to happen.
Events are for instance:

- key presses
- window is minimized, maximized, or closed
- mouse is moved
- mouse button is pressed
- cursor enters window or leaves window

Event-based programming means that a program doesn't have a sequential flow of control, but consists of functions that are called by events.