CSC 221: Introduction to Programming

Fall 2011

List comprehensions & objects

- building strings & lists
- comprehensions
- conditional comprehensions
- example: deck of cards
- object-oriented programming, classes & objects
- example: deck of cards

1

Review: building strings

we have seen numerous examples of building strings

```
def strReverse(str):
    copy = ""
    for ch in str:
        copy = ch + copy
    return copy
```

```
def stripNonLetters(str):
    copy = ""
    for ch in str:
        if ch.isalpha():
            copy += ch
    return copy
```

```
def caesar(word):
    copy = ""
    for ch in word:
        index = ALPHABET.find(ch)
        nextIndex = (index + 3) % 26
        copy += ALPHABET[nextIndex]
    return copy
```

```
def pigPhrase(phrase):
    words = phrase.split()
    pigPhrase = ""
    for nextWord in words:
        pigPhrase += pigLatin(nextWord) + " "
    return pigPhrase.strip()
```

2

Similarly: building lists

since lists & strings are both sequences, can similarly build lists

```
def squares(numSquares):
    nums = []
    for i in range(1, numSquares+1):
        nums.append(i*i)
    return nums
```

```
def evens(numEvens):
    nums = []
    for i in range(1, numEvens+1):
        nums.append(2*i)
    return nums
```

```
def pigList(words):
    pigs = []
    for nextWord in words:
        pigs.append(pigLatin(nextWord))
    return pigs
```

List comprehensions

Python has an alternative mechanism for building lists: comprehensions

- inspired by math notation for defining sets

  squares(N) = $\{i^2 \quad \text{for } 1 \le i \le N$

- in Python: `[EXPR_ON_X for X in LIST]`

```
def squares(numSquares):
    nums = []
    for i in range(1, numSquares+1):
        nums.append(i*i)
    return nums
```

```
def squares(numSquares):
    return [i*i for i in range(1, numSquares+1)]
```

More comprehensions…

```
def evens(numEvens):
    nums = []
    for i in range(1, numEvens+1):
        nums.append(2*i)
    return nums
```

```
def evens(numEvens):
    return [2*i for i in range(1, numEvens+1)]
```

```
def pigList(words):
    pigs = []
    for nextWord in words:
        pigs.append(pigLatin(nextWord))
    return pigs
```

```
def pigList(words):
    return [pigLatin(nextWord) for nextWord in words]
```

Even more…

```
def convertToInts(strList):
    numList = []
    for strVal in strList:
        numList.append(int(strVal))
    return numList

def convertToReals(strList):
    numList = []
    for strVal in strList:
        numList.append(float(strVal))
    return numList
```

```
def convertToInts(strList):
    return [int(strVal) for strVal in strList]

def convertToReals(strList):
    return [float(strVal) for strVal in strList]
```

Exercises:

use list comprehensions to build the following lists of numbers:

- a list of all odd numbers from 1 to N
- a list of all cubes from 1 to N
- a list of the first N powers of 2

given a list of words, use list comprehensions to build:

- a list containing all of the same words, but capitalized
- a list containing all of the same words, but each word reversed
- a list containing the lengths of all of the same words

7

Throwaway comprehensions

comprehensions can be useful as part of bigger tasks

```
def avgLength(words):
    sum = 0
    for w in words:
        sum += len(w)
    return float(sum)/len(words)
```

```
def avgLength(words):
    return float(sum([len(w) for w in words]))/len(words)
```

8

Another example

```
def pigPhrase(phrase):
    words = phrase.split()
    piggy = ""
    for w in words:
        piggy += pigLatin(w) + " "
    return piggy.strip()
```

```
def pigPhrase(phrase):
    words = phrase.split()
    pigWords = [pigLatin(w) for w in words]
    return ' '.join(pigWords)
```

note: the string method join appends the contents of a list into a single string, using the specified divider

Conditional comprehensions

comprehensions can include conditions

- in general: `[EXPR_ON_X for X in LIST if CONDITION]`

```
def longWords(words):
    longs = []
    for nextWord in words:
        if len(nextWord) >= 3:
            longs.append(nextWord)
    return longs
```

```
def longWords(words):
    return [nextWord for nextWord in words if len(nextWord) >= 3]
```

Another example

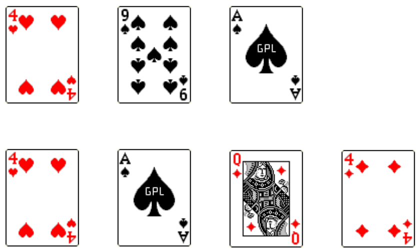
```
def isPrime(num):
    if num < 2:
        return False
    elif num == 2:
        return True
    else:
        for i in range(2, (num/2)+1):
            if num % i == 0:
                return False
        return True

def primesInRange1(low, high):
    primes = []
    for i in range(low, high+1):
        if isPrime(i):
            primes.append(i)
    return primes
```

```
def primesInRange(low, high):
    return [i for i in range(low, high+1) if isPrime(i)]
```

HW6 application

Skip-3 Solitaire:

- cards are dealt one at a time from a standard deck and placed in a single row
- if the rank or suit of a card matches the rank or suit either 1 or 3 cards to its left, then that card (and any cards beneath it) can be moved on top
- goal is to have the fewest piles when the deck is exhausted



Modeling cards

there are many ways we could represent individual cards

- `card = "Ace of Hearts"`

- `rank = "Ace"`
- `suit = "Hearts"`

to keep things simple (and consistent), we will use a 2-char string:

- ranks are from `"23456789TJQKA"`
- suits are from `"CDHS"`

- `card1 = "4C"`
- `card2 = "AS"`
- `card3 = "TH"`

13

Modeling a deck of cards

we can represent a deck of cards as a list

```
deck = ["2C", "3C", "4C", … ]
```

extremely tedious to build!

better solution using nested loops:

```
def getDeck():
    cards = []
    for rank in "23456789TJQKA":
        for suit in "CDHS":
            cards.append(rank+suit)
    return cards
```

even better solution using a comprehension:

```
def getDeck():
    return [rank+suit for rank in "23456789TJQKA" \
                      for suit in "CDHS"]
```

14

Modeling a deck of cards (cont.)

```
def getDeck():
    """Returns a list containing 52 cards."""
    return [rank+suit for rank in "23456789TJQKA" \
                      for suit in "CDHS"]

from random import shuffle
def shuffleDeck(deckList):
    """Randomly shuffles the order of cards in deckList."""
    shuffle(deckList)

def dealCard(deckList):
    """Removes the top/last card in deckList & returns it."""
    if len(deckList) > 0:
        return deckList.pop()
    else:
        return ""

def addCard(deckList, card):
    """Adds card to the bottom/front of
    deckList.insert(0, card)

def numCardsLeft(deckList):
    """Returns the number of cards left in deckList."""
    return len(deckList)
```

```
>>> deck = getDeck()
>>> shuffleDeck(deck)
>>> c1 = dealCard(deck)
>>> c2 = dealCard(deck)
>>> print c1, c2
5C 2D
>>> numCardsLeft(deck)
50
```

15

Problems…

this approach is far from ideal

- since the deck is represented as a list, any list methods can be applied

```
>>> deck = getDeck()
>>> c1 = deck[5]
>>> print c1
3D
>>> deck.remove('QH')
>>> deck.insert(13, '??')
>>> deck = deck[:10]
>>> deck
['2C', '2D', '2H', '2S', '3C', '3D', '3H', '3S', '4C', '4D']
```

- we would like to limit the actions on the list to those that make sense for a deck
- e.g., should only be able to deal from the top
  should not be able to insert cards in the middle

16

Object-oriented approach

the dominant approach to (large-scale) software development today is *object-oriented programming (OOP)*

- in order to solve a problem, you identify the objects involved in the real-world solution and model them in software
- the software model should package all of the data & allowed operations (methods) for that type of object

in OOP, a *class* is a new type definition that encapsulates data & methods

e.g., a DeckOfCards class would include

- data in the form of a list of cards (2-char strings)
- methods such as dealCard, addCard, numCards, …

once a class is defined, can create and manipulates *objects* of that class

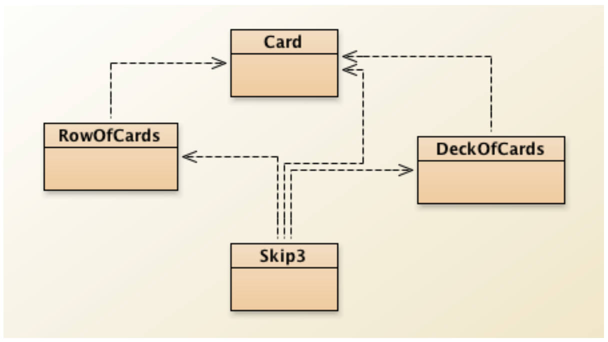
```
e.g., str = "foo"          num = [4, 2, 7]          deck = DeckOfCards()
      foo.upper()          nums.sort()              deck.shuffle()
```

17

OO view of Skip-3

can identify the objects involved in a Skip-3 game

- Card: models a playing card, with rank & suit
- DeckOfCards: models a deck, with methods shuffle, dealCard, numCards, …
- RowOfCards: models a row, with methods addAtEnd, moveCard, numCards, …



using these objects, can build an interactive game

18

cards.py

- each method has an implicit parameter: `self`
- `__init__` and `__str__` are special purpose methods
- `self.cards` is a list that belongs to this object

MORE DETAILS ABOUT CLASS DEFINITIONS LATER

```
from random import randint, shuffle

class DeckOfCards:
  def __init__(self):
      """Initializes a new deck of cards."""
      ranks = "23456789TJQKA"
      suits = "CDHS"
      self.cards = [r+s for r in ranks for s in suits]

  def shuffle(self):
      """Shuffles the cards into a random order."""
      shuffle(self.cards)

  def dealCard(self):
      """Removes the top/last card in the deck & returns it."""
      if self.numCards() > 0:
        return self.cards.pop()
      else:
        return ""

  def addCard(self, card):
      """Adds the specified card to the bottom/front of the deck."""
      self.cards.insert(0, card)

  def numCards(self):
      """Returns the number of cards remaining in the deck."""
      return len(self.cards)

  def __str__(self):
      """Returns the string representation of the deck."""
      return ' '.join(self.cards)
```

DeckOfCards example

```
>>> deck = DeckOfCards()
>>> deck.shuffle()
>>> print deck
JS 2D 4D TD KS 7H 9H 6D AD KH 6H AH 7S KC 2H 7D TC 9S 2C
3H 2S QD JD 8H 3C 4C TH 5C 3D 7C 6S 4H AC QS KD 3S QC JH
AS 4S 8C 5H 9D 9C 6C 5S JC 8D 8S 5D TS QH
>>> while deck.numCards() > 0:
        print deck.dealCard()


QH
TS
5D
8S
8D
JC
5S
6C
9C
9D
5H
```

HW 6: RowOfCards class

similarly, we could define a class that represents a row of cards

- you will complete the class definition for the first part of HW 6

```
class RowOfCards:
  def __init__(self):
    """Initializes an empty row (list) of cards."""
    self.cards = []

  def addAtEnd(self, card):
    """Adds the specified card to the end of the row (list)."""
    ### TO BE COMPLETED IN PART 1

  def moveCard(self, card, numSpots):
    """Moves the specified card numSpots to the left."""
    ### TO BE COMPLETED IN PART 1

  def numCards(self):
    """Returns the number of cards in the row (list)."""
    ### TO BE COMPLETED IN PART 1

  def __str__(self):
    """Returns the string representation of the deck (list)."""
    ### TO BE COMPLETED IN PART 1
```

21

HW 6: skip3 game

in PART 2, extend the basic framework

- add help command
- display # of cards in row at "quit"
- add error checking to handle illegal & malformed commands

```
from cards import DeckOfCards, RowOfCards

def skip3():
  """Basic framework for playing Skip3 Solitaire."""
  print "Welcome to Skip3 Solitaire."

  deck = DeckOfCards()
  deck.shuffle()
  row = RowOfCards()

  response = ""
  while response != "q":
    print row+"\n"

    response = raw_input("Action? ")
    firstLetter = response[0].lower()
    responseWords = response.split()

    if firstLetter == "d":
      row.addAtEnd(deck.dealCard())
    elif firstLetter == "m":
      row.moveCard(responseWords[1].upper(), 1)
    elif firstLetter == "s":
      row.moveCard(responseWords[1].upper(), 3)
    elif firstLetter == "q":
      print "Thanks for playing."
    else:
      print "Unknown command"
```

22