Objectives

- Helper methods
- __cmp__ method
- Group Work: Designing Classes

Helper Methods

- Sometimes, you may need helper methods that are part of the class but are not meant to be part of the class's API
  - Make your code easier but others outside the class shouldn't use
- Convention: method name begins with "_"

Example Helper Methods

- Only loosely enforces that other can't use
  - Doesn't show up in `help`
  - Does show up in `dir`

Helper Method:

```
def _isFaceCard(self):
    if self.rank > 10 and self.rank < 14:
        return True
    return False
```

In use:

```
def rummyValue(self):
    if self._isFaceCard():
        return 10
    elif self.rank == 14:
        return 15
    else:
        return 5
```

card4.py

Comparing Objects of the Same Type

- Special `__cmp__` method
  - Header: `_cmp__(self, other)`
    - `other` is another object of the *same type*
  - Returns
    - Negative integer if self < other
    - 0 if self==other
    - Positive integer if self > other
- Similar to implementing `Comparable` interface in Java
- If no `__cmp__` method, defaults to comparing memory addresses of objects

card3_nocmp.py

Comparing Objects of the Same Type

- What uses the __cmp__ method?
  - Comparison operators: <,>,==, etc.
  - List's sort() method
    - Works best if list contains all the same type of objects

Another Example of __cmp__

Comparing by suit then rank; order is 2 Clubs, 3 Clubs, ..., Ace Clubs, 2 Diamonds, 3 Diamonds, ...

```
def __cmp__(self, other):
    if self.getSuit() < other.getSuit():
        return -1
    if self.getSuit() > other.getSuit():
        return 1
    # same suit; differentiate by rank
    if self.getRank() < other.getRank():
        return -1
    if self.getRank() > other.getRank():
        return 1
    return 0
```

card5.py

Summary: Designing Classes

- What does the object/class represent?
- How to model/represent the class's *data*?
  - Instance variable
  - Data type
- What *functionality* should objects of the class have?
  - How will others want to use the class?
  - Put into methods for others to call (API)
- In general, the **nouns** in a problem are the classes/objects, **verbs** are the methods

Benefits of Classes

- Package/group related data into one object
- Reusing code
  - E.g., Don't need to check if user put in valid key
- Provide interface, can change underlying implementation without affecting calling code

Changing Implementations

- Same API, different implementations

```
def __init__(self, rank, suit):
    self.rank = rank
    self.suit = suit

def getRank(self):
    return self.rank

def getSuit(self):
    return self.suit
```

Tradeoff: Saving information (memory); Computing information

```
def __init__(self, rank, suit):
    self.cardid=rank
    if suit == “clubs”:
        self.cardid += 13
    elif suit == “hearts”:
        self.cardid += 26
    elif suit == “diamonds”:
        self.cardid += 39

def getRank(self):
    return (self.cardid-2) % 13 + 2

def getSuit(self):
    suits = ["spades", "clubs", "hearts", "diamonds"]
   whichsuit = (self.cardid-2)/13
   return suits[whichsuit]
```

card_byid.py

Considerations for Using Classes

- Only use class if you're using most of its functionality/information
  - Don't use Counter for validating if a number is within the valid range
    - Because not using the wrapping/current value
- Since don't know implementation, may inadvertently duplicate code
  - Redo something done by class
  - Could have efficiency penalties
  - But time saved reusing code is usually worth it

Top-Down Design

- Break down larger problems into pieces that you can solve
  - Smaller pieces: classes, methods, functions
  - Implement smallest pieces and build up
- We've been doing this most of the semester
  - Typically, program was 1) read input, 2) process input, 3) print result
    - Started putting Step 2 into >= 1 functions
    - Steps 1 and 3 were sometimes a function
  - Now: on larger scale

Design a Music Manager

- Reads your music library from a file
- Displays the songs in your music library
- Stores your music library in a file
- Allows you to add songs to your library from a file
- Keeps track of the total length of your music library
- Allows you to sort the songs in your library
- Provides a user interface to do these things

Designing a Music Manager

- Break down into pieces
- What classes do we need?
  - What data needed to model those classes?
  - What functionality do each of those classes need?
- What does our driver program (user interface) do?
- How should we implement those classes/program?

Designs

- For each of your classes
  - Data
  - API

Group 1: Greg, Dave, Joe, Colin
Group 2: Alex, Nay, Julie, Vasil
Group 3: Ty, Clay, Arturo
Group 4: Joa, Lucy, Stuart

Music Manager Classes/Driver Data

- MusicLibrary
  - Songs
  - Total length
  - Filename
- Song
  - Title
  - Artist name
  - Album name
  - Length
- PlayTime
  - Days, hours,
  - Minutes, seconds
- Driver (UI)
  - Music library

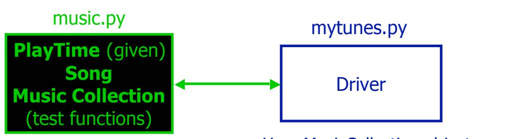
What are the data types for each class's data?

MM Classes/Driver Functionality

- MusicLibrary
  - Getters (accessors)
  - String rep
  - Reading library from file
  - Saving library to file
  - Adding albumclear
  - Sorting
- Song
  - Getters
  - String rep
  - Comparator
  - Writing to a file
- PlayTime (given)
  - Getters, String rep
  - Adding play time
- Driver
  - Getting user input to
    - Read library, album files
    - Store library to file
    - Sort songs
    - View songs
  - Summary: call appropriate methods on classes to do above

Lab 10 Design

- 2 files: music.py and mytunes.py



- Uses MusicCollection object
- Gets command-line argument
- Handles UI
- Calls methods on the MC object

Problem: Album Music Files

- Given an album file that has the format
  - <Artist name>
  - <Album name>
  - <number of songs>
  - <Song name 1>
  - <Song length 1>
  - …
  - <Song name n>
  - <Song length n>
- Write algorithm to create Song objects to represent each song

Length has the format min:seconds

Problem: Library Music Files

- Given a library file that has the format
  - <number of songs>
  - <Song artist 1>
  - <Song album 1>
  - <Song name 1>
  - <Song length 1>
  - …
  - <Song artist n>
  - <Song album n>
  - <Song name n>
  - <Song length n>
- Create a MusicLibrary object

UI Specification

- Checks if user entered a command-line argument
  - Default library: libraries/mytunes.library
- Read library from file
- Repeatedly gets selected options from the user, until quits
- Repeatedly prompts for new selection if invalid option
- Executes the appropriate code for the selection
- Stops when user quits
- Stores the library into the file

Demonstrate program
Write pseudocode

UI Pseudocode

```
Use default library if only one command-line argument
Read library from file
while True:
        display menu options
        prompt for selection
        while invalid option
                print error message
                prompt for selection
        break if selected quit
        otherwise, do selected option
Store library to designated file
```

Implementation Plan

- Review PlayTime class
  - How will you create a PlayTime object?
  - How will you use it?
- Implement Song class
  - Test (write test functions, e.g., testSong())
- Implement MusicCollection class
  - Example runs in lab write up
  - Note: in general, methods for classes will not prompt for input (Use input parameters)
  - Test
- Implement driver program

Plan for Implementing a Class

- Write the constructor and string representation/print methods first
- Write function to test them
- While more methods to implement …
  - Write method
  - Test

- See counter.py and card.py for example test functions

Broader Issue

- One Laptop Per Child Project
  - Main story on CS111 page
  - Blog entry has a lot of other interesting links, videos