Fundamentals of Programming I

More Data Modeling

Objects, Classes, and Methods

- Every data value in Python is an *object*
- Every object is an instance of a *class*
- Built in classes include **`int`**, **`float`**, **`str`**, **`tuple`**, **`list`**, **`dict`**
- A class includes operations (*methods*) for manipulating objects of that class (**`append`**, **`pop`**, **`sort`**, **`find`**, etc.)
- Operators (**`==`**, **`[]`**, **`in`**, **`+`**, etc.) are "syntactic sugar" for methods

What Do Objects and Classes Do for Us?

- An object bundles together data and operations on those data
- A computational object can model practically any object in the real (natural or artificial) world
- Some classes come with a programming language
- Any others must be defined by the programmer

Programmer-Defined Classes

- The `EasyFrame` class is used to create GUI windows that are easy to set up
- The `Image` class is used to load, process, and save images
- Like the built-in classes, these classes include operations to run with their instances

Other Examples

- A **`Student`** class represents information about a student and her test scores
- A **`Rational`** class represents rational numbers and their operations
- A **`Die`** class represents dice used in games
- **`SavingsAccount`**, **`CheckingAccount`**, **`Bank`**, and **`ATM`** are used to model a banking system
- **`Proton`**, **`Neutron`**, **`Electron`**, and **`Positron`** model subatomic particles in nuclear physics

The **`Die`** Class:
Its Interface and Use

Interface

```
die.py              # The module for the Die class

Die()        # Returns a new Die object

roll()              # Resets the die's value

getValue()          # Returns the die's value
```

The **Die** Class:
Its Interface and Use

Interface

```
die.py              # The module for the Die class
Die()        # Returns a new Die object
roll()              # Resets the die's value
getValue()          # Returns the die's value
```

Use

```
from die import Die
d = Die()            # Create a new Die object
d.roll()             # Roll it
print(d.getValue())  # Display its value
help(Die)            # Look up the documentation
```

Accessing Data in an Object

```
>>> die = Die()

>>> die.getValue()
1

>>> print(die.getValue())
1
```

An object’s data can be viewed or accessed by using its *accessor methods*

String Representation

```
>>> die = Die()

>>> str(die)             # Same as die.__str__()
'1'

>>> print(str(die))
1
```

Each class can include a string conversion method named **`__str__`**

This method is automatically called when the **`str`** function is called with the object as a parameter

String Representation

```
>>> die = Die()

>>> str(die)             # Same as die.__str__()
'1'

>>> print(str(die))
1

>>> print(die)               # Better still
1
```

Each class can include a string conversion method named **`__str__`**

`print` runs **`str`** if it’s given an object to print - way cool!

The __str__ Method

```
class Die(object):
    """This class represents a savings account."""

    def __init__(self):
        self.value = 1

    def __str__(self):
        return str(self.value)
```

As a rule of thumb, you should include an __str__ method in each new class that you define

Other Common Methods

Python allows you to define other methods that are automatically called when objects are used with certain functions or operators

Function or Operator	Method Called
`len(obj)`	`obj.__len__()`
`obj1 in obj2`	`obj2.__contains__(obj1)`
`obj1 + obj2`	`obj1.__add__(obj2)`
`obj1 < obj2`	`obj1.__lt__(obj2)`
`obj1 > obj2`	`obj1.__gt__(obj2)`
`obj[index]`	`obj.__getitem__(index)`
`obj1[index] = obj2`	`obj1.__setitem__(index, obj2)`

Example: Rational Numbers

```
from rational import Rational
oneHalf = Rational(2, 4)
oneThird = Rational(1, 3)
print(oneHalf)                      # Prints 1/2
print(oneThird < oneHalf)           # Prints True
theSum = oneHalf + oneThird
print(theSum)                       # Prints 5/6
```

Example: Rational Numbers

```
class Rational(object):
    """This class represents a rational number."""

    def __init__(self, numerator = 1, denominator = 1):
        self.numer = numerator
        self.denom = denominator
        self.reduce()

    def __str__(self):
        return str(self.numer) + "/" + str(self.denom)
```

The `__init__` and `__str__` methods should always be defined first; then you can test the class to verify that its objects are appropriately instantiated.

Addition of Rational Numbers

```
class Rational(object):
    """This class represents a rational number."""

    def __add__(self, other):
        """Returns the sum of self and other."""
        numerSum = self.numer * other.denom + \
                   other.numer * self.denom
        denomSum = self.denom * other.denom
        return Rational(numerSum, denomSum)
```

$$n_{sum} / d_{sum} = (n_1 * d_1 + n_2 * d_1) / (d_1 * d_2)$$

Comparison of Rational Numbers

```
class Rational(object):
    """This class represents a rational number."""

    def __lt__(self, other):
        """Returns True if self < other or False otw."""
        extremes = self.numer * other.denom
        means = other.numer * self.denom
        return means < extremes
```

Operator	Method	Implementation
r1 < r2	__lt__	means < extremes
r1 > r2	__gt__	means > extremes
r1 == r2	__eq__	means == extremes
r1 <= r2	__le__	means <= extremes
r1 >= r2	__ge__	means >= extremes

Printing in the Shell

```
>>> oneHalf = Rational(1, 2)

>>> print(oneHalf)
1/2

>>> oneHalf
<__main__.Rational object at 0x106166f28>
```

Printing in the Shell

```
>>> oneHalf = Rational(1, 2)

>>> print(oneHalf)
1/2

>>> oneHalf
<__main__.Rational object at 0x106166f28>
```

```
def __repr__(self):
    """Returns a string for shell printing."""
    return str(self)
```

Python runs the **`__repr__`** method to obtain the print reprensentation of an object in the IDLE shell.

Modeling Card Games

- Used in card games, such as *War*, *Blackjack*, *Poker*, and *Crazy Eights*

- A standard deck consists of 52 cards

- Each card has a rank (a number from 1-13, where Ace is 1 and face cards are 11-13)

- Each card has a suit (Spades, Hearts, Diamonds, Clubs)

A Card: State and Behavior

- State:
  - a rank
  - a suit

- Behavior: examine the values of the state variables, but don’t ever modify them

Using the **Card** Class

```
>>> Card card = Card(3, 'Spades')

>>> print(str(card.rank) + ' of ' + card.suit)
3 of Spades

>>> print(card)
3 of Spades
```

A **Card** object contains two data values, and these are never changed

Thus, there is no need for accessor or mutator methods

Just reference the card’s variables to get their values

Defining the **Card** Class

```
class Card(object):

    SUITS = ('Spades', 'Hearts', 'Diamonds', 'Clubs')
    RANKS = tuple(range(1, 14))

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

The **Card** class can specify the ranges of ranks and suits as *class variables*

Class Variables

- An *instance variable* refers to storage owned by a single instance

- A *class variable* refers to storage owned by the class, and thus available to all of its instances

- For example, each card owns a separate rank and suit, so theyshould be a instance variables

- But all cards have the same lists of ranks and suits, so they should be class variables

Create and Print all 52 Cards

```
for suit in Card.SUITS:
    for rank in Card.RANKS:
        card = Card(rank, suit)
        print(card)
```

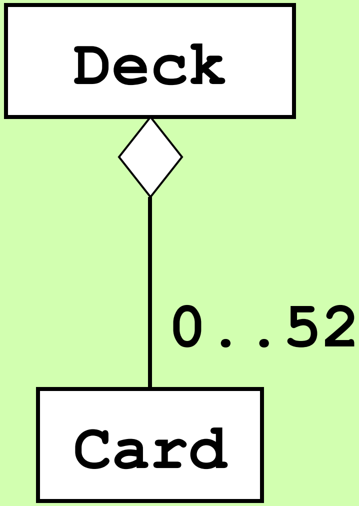
`print` automatically runs **`str`** to obtain an object’s string representation

String Representation

```
class Card(object):
    …
    def __str__(self):
        """Returns the string representation of a card."""
        if self.rank == 1:
            rank = 'Ace'
        elif self.rank == 11:
            rank = 'Jack'
        elif self.rank == 12:
            rank = 'Queen'
        elif self.rank == 13:
            rank = 'King'
        else:
            rank = self.rank
        return str(rank) + ' of ' + self.suit
```

A Deck: State and Behavior

- A deck initially contains 52 cards
- The user can
  - shuffle the deck
  - deal a single card from its top
  - check to see if there are more cards to deal



The **Deck** Interface and Its Use

Interface

```
Deck()

shuffle()

deal()       # Removes and returns the top card

isEmpty()

str(aDeck)
```

The **Deck** Interface and Its Use

Interface

```
Deck()

shuffle()

deal()      # Removes and returns the top card

isEmpty()
```

Use

```
from cards import Deck

deck = Deck()

deck.shuffle()

while not deck.isEmpty():
    card = deck.deal()
    print(card)
```

Application: The Game of *War*

- A simplified version with two players and a single war pile

- Each player has 2 piles of cards:
  - An unplayed pile
  - A winnings pile

Playing the Game of *War*

```
Deal 26 cards to each player’s unplayed pile

While both unplayed piles are not empty

    Each player moves the topmost card from the
    unplayed pile to the top of the game’s war pile

    If these cards do not have the same rank
        Move the cards from the game’s war pile to the
        winner’s winnings pile

The player with the largest winnings pile wins
```

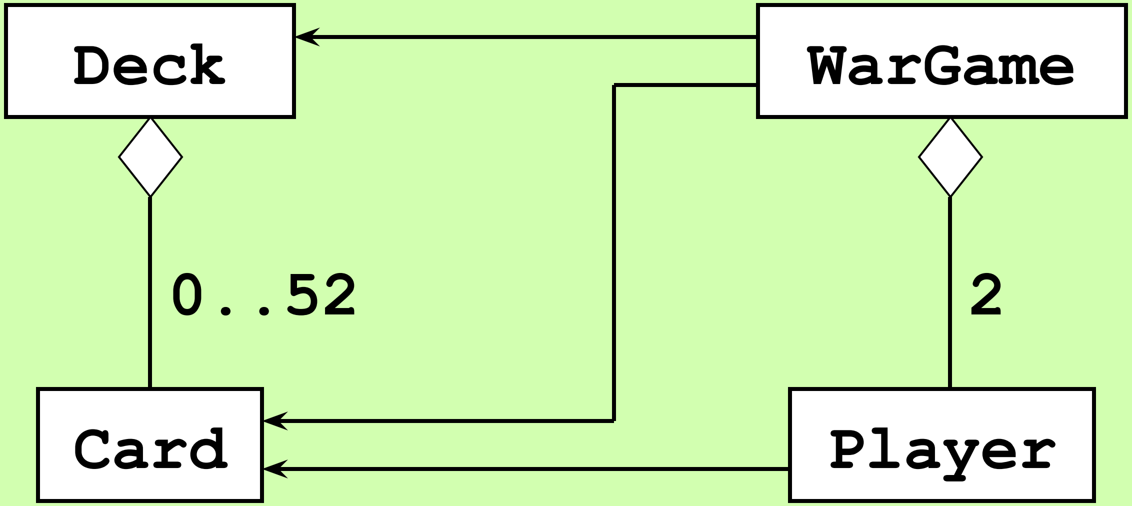
Additional Classes

- **`Player`** contains 2 piles of cards

- **`WarGame`** contains a deck, two players, and a single pile of cards; allows the user to step through the game

- On each step, the cards are drawn and the war pile is shifted if there is a winner

Classes and Relationships



cards.py: **`Card`**, **`Deck`**

wargame.py: **`Player`**, **`WarGame`**

The **WarGame** Interface

```
WarGame()    # Creates a deck and 2 players

deal()       # Deals 26 cards to each player

step()       # Draws the cards and shifts the piles

winner()     # Returns None if the game is not over,
             # or the results as a string otherwise

str(aGame)   # The current state of the game as a string
```

Playing the Game

```
WarGame()      # Creates a deck and 2 players

deal()         # Deals 26 cards to each player

step()         # Draws the cards and shifts the piles

winner()       # Returns None if the game is not over,
               # or the results as a string otherwise

str(aGame)     # The current state of the game as a string
```

```
def main():
    game = WarGame()
    game.deal()
    while not game.winner():
        game.step()
        print(game)
    print(game.winner())
```