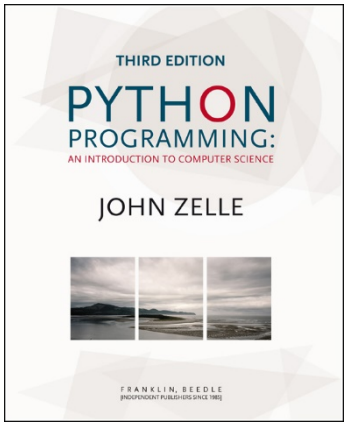
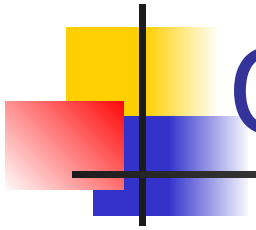
Python Programming:
An Introduction To
Computer Science

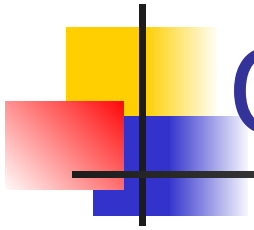


Chapter 12
Object-Oriented Design



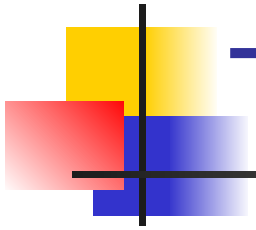
Objectives

- To understand the process of object-oriented design.
- To be able to read and understand object-oriented programs.
- To understand the concepts of encapsulation, polymorphism and inheritance as they pertain to object-oriented design and programming.



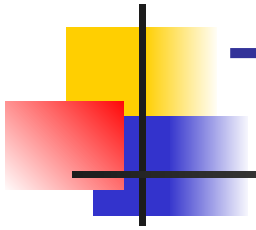
Objectives

- To be able to design moderately complex software using object-oriented design.



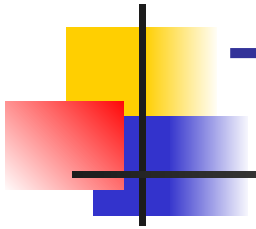
The Process of OOD

- Most modern computer applications are designed using a data-centered view of computing called object-oriented design (OOD).
- The essence of OOD is describing a system in terms of magical black boxes and their interfaces.



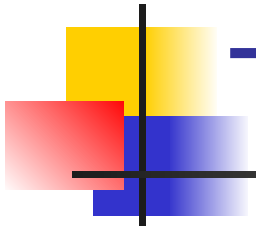
The Process of OOD

- Each component provides a service or set of services through its interface.
- Other components are users or clients of the services.
- A client only needs to understand the interface of a service – implementation details are not important; they may be changed and shouldn't affect the client at all!



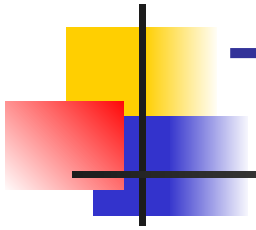
The Process of OOD

- The component providing the service shouldn't have to consider how the service is used – it just needs to provide the service "as advertised" via the interface.
- This separation of concerns makes the design of complex systems possible.



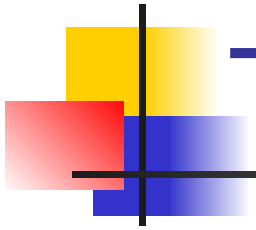
The Process of OOD

- In top-down design, functions serve the role of the black box.
  - Client programs can use the functions as long as it understands **what** the function does.
  - **How** the function accomplishes its task is encapsulated within the function.



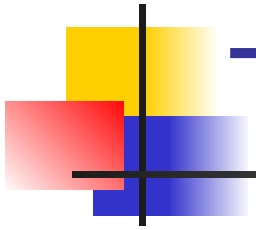
The Process of OOD

- In OOD, the black boxes are objects.
- The magic behind the objects is in the class definitions. Once a class definition is written, we can ignore how the class works and rely on the external interface, its methods.
- You've seen this when using the graphics library – you were able to draw a circle without having to know all the nitty-gritty details encapsulated in class definitions for `GraphWin` and `Circle`.



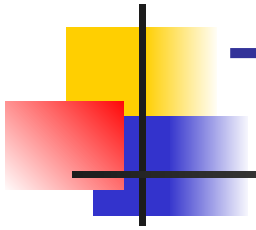
The Process of OOD

- Breaking a large problem into a set of cooperating classes reduces the complexity that must be considered to understand any given part of the program. Each class stands on its own!
- OOD is the process of finding and defining a useful set of classes for a given problem.
- Like design, it's part art and part science. The more you design, the better you'll get.



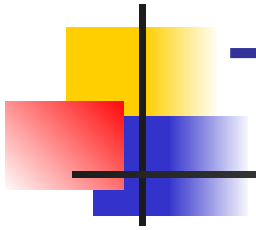
The Process of OOD

- Here are some guidelines for OOD:
  1. Look for object candidates
     - The goal is to define a set of objects that will be helpful in solving the problem.
     - Start with a careful consideration of the problem statement – objects are usually described by nouns. Which nouns in your problem statement would be represented in your program? Which have interesting behavior or properties?
     - Things that can be represented as primitive data types (numbers or strings) are probably not important object candidates.
     - Things to look for: a grouping of related data items (e.g., point coordinates, employee data)



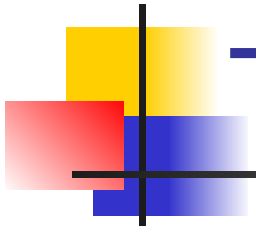
The Process of OOD

2. Identify instance variables
   - Once you think of some possible objects, think of the kinds of information each object will need to do its job.
   - Some object attributes will have primitive data types, while others may be complex types that suggest other useful objects/classes.
   - Strive to find good "home" classes for all the data in your program.



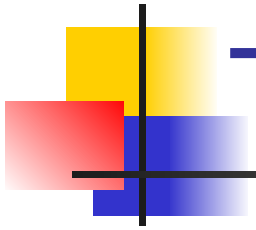
The Process of OOD

3. Think about interfaces
   - What operations would be required for objects of that class to be useful?
   - Consider the verbs in the problem statement.
   - Verbs describe actions.
   - List the methods that the class will require.
   - Remember – all of the manipulation of the object's data should be done through the methods you provide.



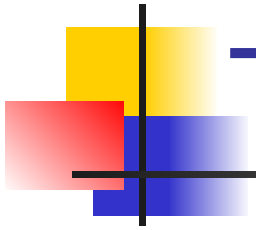
The Process of OOD

4. Refine the nontrivial methods
   - Some methods will probably look like they can be accomplished in a few lines of code, while others may take more programming effort.
   - Use top-down design and stepwise refinement to flesh out the details of the more difficult methods.
   - As you're programming, you may discover that some new interactions with other classes are needed, and you may need to add new methods to other classes.
   - Sometimes you may discover a need for a brand-new kind of object that calls for the definition of another class.



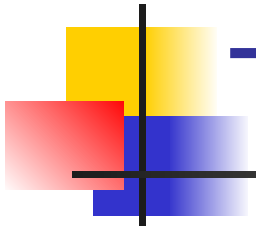
The Process of OOD

5. Design iteratively
   - It's not unusual to bounce back and forth between designing new classes and adding methods to existing classes.
   - Work on whatever is demanding your attention.
   - No one designs a program top to bottom in a linear, systematic fashion. Make progress wherever progress needs to be made.



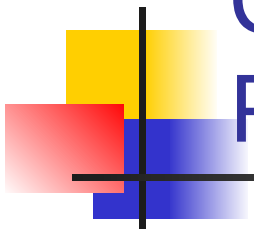
The Process of OOD

6. Try out alternatives
   - Don't be afraid to scrap an approach that doesn't seem to be working, or to follow an idea and see where it leads. Good design involves a lot of trial and error!
   - When you look at the programs of others, you are looking at finished work, not the process used to get there.
   - Well-designed programs are probably not the result of a first try. As Fred Brooks said, "Plan to throw one away."



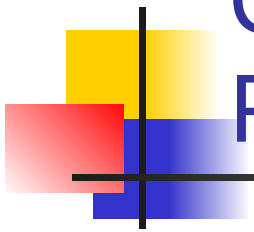
The Process of OOD

7. Keep it simple
   - At each step in the design, try to find the simplest approach that will solve the problem.
   - Don't design in extra complexity until it is clear that a more complex approach is needed.



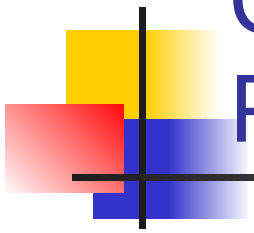
Case Study: Racquetball Simulation

- You may want to review our top-down design of the racquetball simulation from Chapter 9.
- We want to simulate multiple games of racquetball where the ability of the two opponents is represented by the probability that they win a point when they are serving.



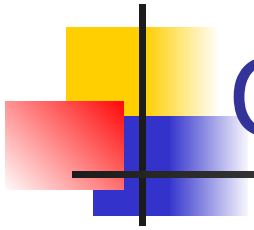
Case Study: Racquetball Simulation

- Inputs:
  - Probability for player A
  - Probability for player B
  - The number of games to simulate
- Output:
  - A nicely formatted summary of the results



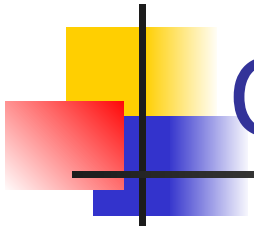
Case Study: Racquetball Simulation

- Previously, we ended a game when one of the players reached 15 points.
- This time, let's also consider shutouts. If one player gets to 7 points before the other player has scored a point, the game ends.
- The simulation should keep track of each players' wins and the number of wins that are shutouts.



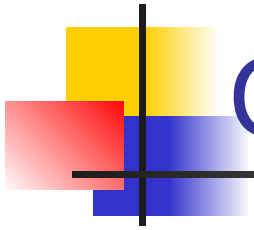
Candidate Objects and Methods

- Our first task – find a set of objects that could be useful in solving this problem.
- Problem statement – "Simulate a series of racquetball games between two players and record some statistics about the series of games."
- This suggests two things
  - Simulate a game
  - Keep track of some statistics



Candidate Objects and Methods

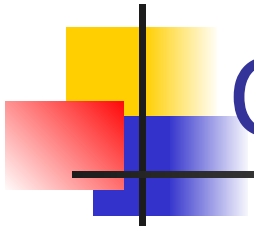
- First, let's simulate the game.
  - Use an object to represent a single game of racquetball.
  - This game will have to keep track of some information, namely, the skill levels of the two players.
  - Let's call this class `RBallGame`. Its constructor requires parameters for the probabilities of the two players.



Candidate Objects and Methods

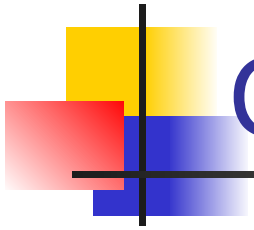
- What else do we need? We need to play the game.
- We can give the class a `play` method that simulates the game until it's over.
- We could then create and play a racquetball game with two lines of code!

```
theGame = RBallGame(probA, probB)
theGame.play()
```



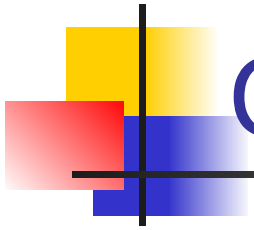
Candidate Objects and Methods

- To play several games, we just need to put a loop around this code.
- We'll need at least four counts to print the results of our simulation: wins for A, wins for B, shutouts for A, and shutouts for B
- We could also include the number of games played, but we can calculate this from the counts above.



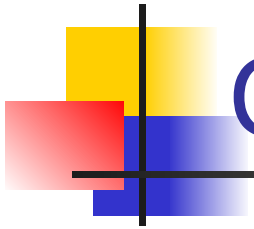
Candidate Objects and Methods

- These four related pieces of information could be grouped into a single object, which could be an instance of the class `SimStats`.
- A `SimStats` object will keep track of all the information about a series of games.



Candidate Objects and Methods

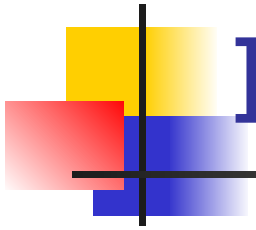
- What operations would be useful on these statistics?
  - The constructor should initialize the counts to 0.
  - We need a way to update these counts while the games are simulated. How can we do this?
  - The easiest approach would be to send the entire game object to the method and let it extract the appropriate information.
  - Once the games are done, we need a method to print out the results – `printReport`.



Candidate Objects and Methods

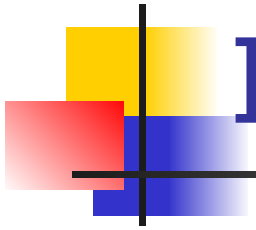
```
def main():
    printIntro()
    probA, probB, n = getInputs()
    # Play the games
    stats = SimStats()
    for i in range(n):
        theGame = RBallGame(probA, probB)    # Create a new game
        theGame.play()                       # Play it
        stats.update(theGame)           # Get info about completed game
    # Print the results
    stats.printReport()
```

- The helper functions that print an introduction and get inputs should be easy. Let's work on the `SimStats` class!



Implementing `SimStats`

- The constructor for `SimStats` just needs to initialize the four counts to 0.
- ```
  class SimStats:
      def __init__(self):
          self.winA = 0
          self.winB = 0
          self.shutsA = 0
          self.shutsB = 0
  ```
  ```

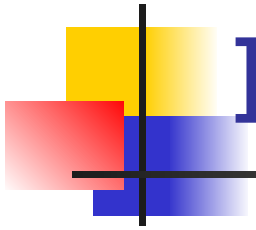


Implementing `SimStats`

- The update method takes a game as a parameter and updates the four counts appropriately. The heading will look like this:

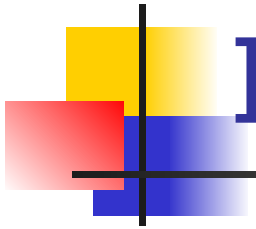
```
def update(self, aGame):
```

- We need to know the final score of the game, but we can't directly access that information since it is in instance variables of `aGame`.



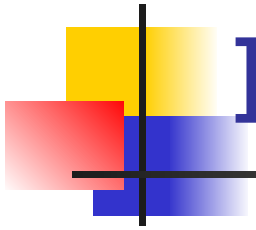
Implementing `SimStats`

- We need a new method in `RBallGame` that will report the final score.
- Let's call this new method `getScores`, and it will return the scores for player A and player B.
- Now the algorithm for `update` is straightforward.



Implementing SimStats

```
def update(self, aGame):
        a, b = aGame.getScores()
        if a > b:                                   # A won the game
            self.winsA = self.winsA + 1
            if b == 0:
                self.shutsA = self.shutsA + 1
        else:                                       # B won the game
            self.winsB = self.winsB + 1
            if a == 0:
                self.shutsB = self.shutsB + 1
```



Implementing `SimStats`

- The only thing left is a method to print out the results.
- The method `printReport` will generate a table showing the
  - wins
  - win percentage
  - shutouts
  - and shutout percentage for each player.

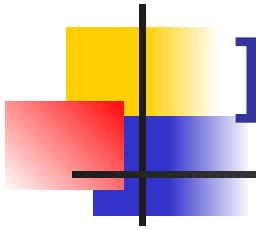
Implementing `SimStats`

- Here's sample output:

```
Summary of 500 games:

            wins (% total)    shutouts (% wins)
-----------------------------------------------
Player A:    393   78.6%            72   18.3%
Player B:    107   21.4%             8    7.5%
```

- The headings are easy to handle, but printing the output in nice columns is harder. We also need to avoid division by 0 when calculating percentages.



Implementing `SimStats`

- Let's move printing the lines of the table into the method `printLine`.
- The `printLine` method will need the player label (A or B), number of wins and shutouts, and the total number of games (for calculating percentages).

Implementing `SimStats`

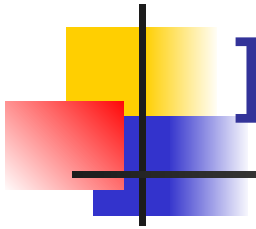
```
def printReport(self):
    # Print a nicely formatted report
    n = self.winsA + self.winsB
    print("Summary of", n , "games:\n")
    print("          wins (% total)   shutouts (% wins) ")
    print("---------------------------------------------")
    self.printLine("A", self.winsA, self.shutsA, n)
    self.printLine("B", self.winsB, self.shutsB, n)
```

- To finish the class, we will implement `printLine`. This method makes heavy use of string formatting.
- You may want to review string formatting in chapter 5.8.2

Implementing SimStats

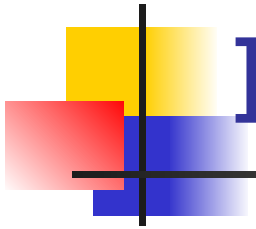
```
 def printLine(self, label, wins, shuts, n):
     template = "Player {0}:{1:5}  ({2:5.1%}) {3:11}   ({4})"
     if wins == 0:           # Avoid division by zero!
         shutStr = "----- "
     else:
         shutStr = "{0:4.1%}".format(float(shuts)/wins)
     print template.format(label, wins, float(wins)/n,\
shuts, shutStr)
```

- We define a template for the information that will appear in each line.
- The `if` ensures we don't divide by 0, and the template treats it as a string.



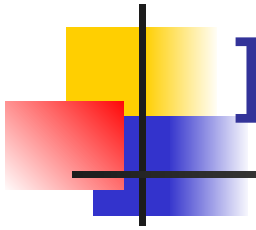
Implementing `RBallGame`

- This class needs a constructor that accepts two probabilities as parameters, a `play` method that plays the game, and a `getScores` method that reports the scores.



Implementing `RBallGame`

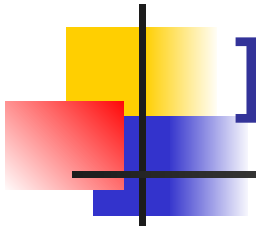
- What will a racquetball game need to know?
  - To play the game, we need to know
    - The probability for each player
    - The score for each player
    - Which player is serving
  - The probability and score are more related to a particular player, while the server is a property of the game between the two players.



Implementing `RBallGame`

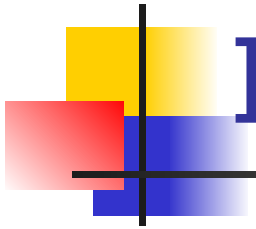
- So, a game needs to know who the players are
  - The players themselves could be objects that know their probability and score
- and which is serving.

- If the players are objects, then we need a class to define their behavior. Let's call it `Player`.



Implementing `RBallGame`

- The `Player` object will keep track of a player's probability and score.
- When a `Player` is initialized, the probability will be passed as a parameter. Its score will be set to 0.
- Let's develop `Player` as we work on `RBallGame`.



Implementing `RBallGame`

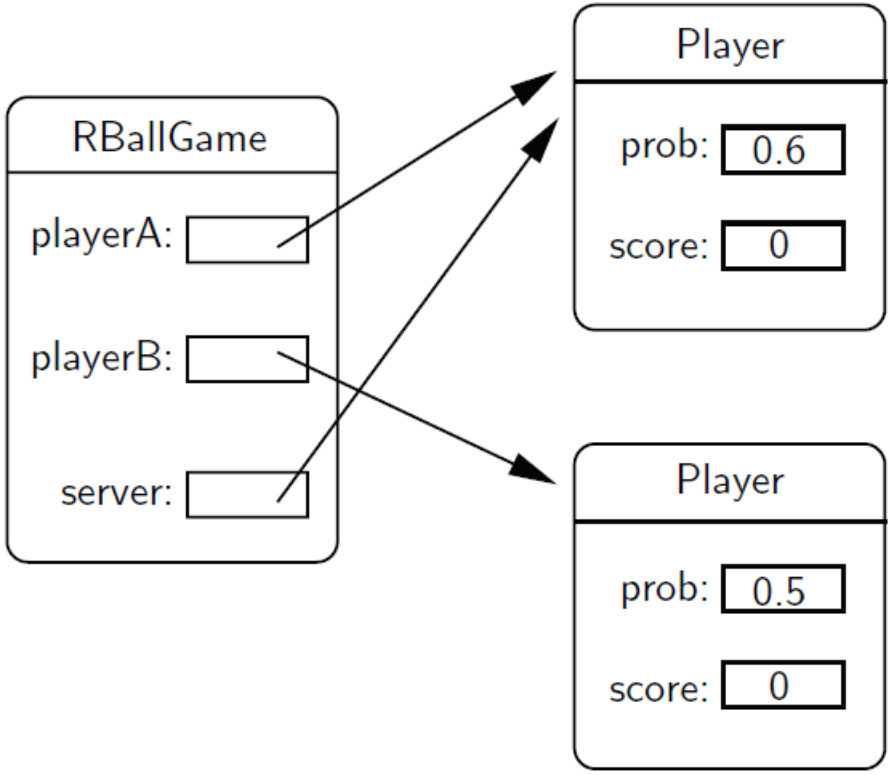
- The game will need instance variables for the two players, and another variable to keep track of which player has service.

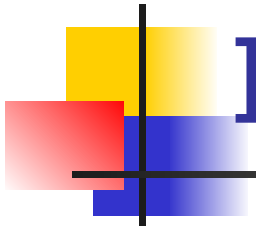
```
class RBallGame:
    def __init__(self, probA, probB):
        # Create a new game having players with the given probs.
        self.playerA = Player(probA)
        self.playerB = Player(probB)
        self.server = self.playerA  # Player A always serves first
```

Implementing `RBallGame`

- Suppose we create an instance of `RBallGame` like this:

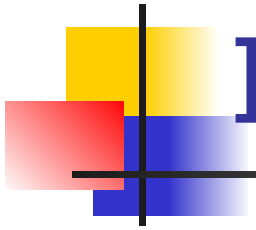
```
theGame = RBallGame(.6, .5)
```





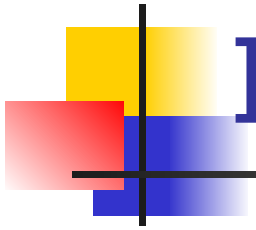
Implementing `RBallGame`

- Our next step is to code how to play the game!
- In chapter 9 we developed an algorithm that continues to serve rallies and awards points or changes service as appropriate until the game is over.
- Let's translate this algorithm into our object-based code!



Implementing `RBallGame`

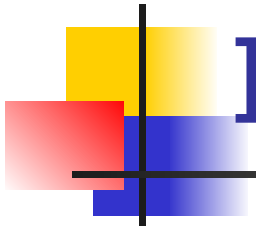
- Firstly, we need a loop that continues as long as the game is not over.
- The decision whether a game is over or not can only be done by looking at the game object itself.
- Let's assume we have an `isOver` method which can be used.



Implementing `RBallGame`

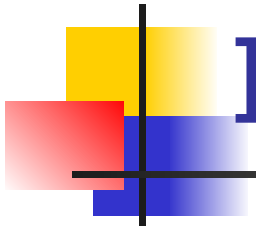
```
def play(self):
    # Play the game to completion
    while not self.isOver():
```

- Within the loop, the serving player needs to serve, and, based on the result, we decide what to do.
- This suggests that the `Player` objects should have a method that performs a serve.



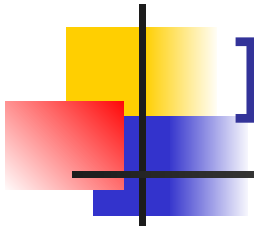
Implementing `RBallGame`

- Whether the serve is won or not depends on the probability stored within each player object, so, one can ask the server if the serve was won or lost!
- `if self.server.winsServe():`
- Based on the result, a point is awarded or service changes.
- To award a point, the player's score needs to be changed, which requires the player object to increment the score.



Implementing `RBallGame`

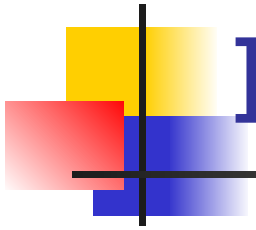
- Changing servers is done at the game level, since this information is kept in the `server` instance variable of `RBallGame`.
- Here's the completed `play` method:



Implementing RBallGame

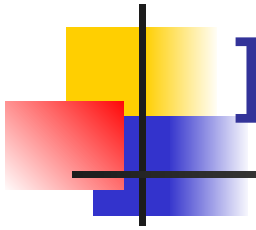
```
def play(self):
    # Play the game to completion
    while not self.isOver():
        if self.server.winsServe():
            self.server.incScore()
        else:
            self.changeServer()
```

- Remember, `self` is an `RBallGame`!
- While this algorithm is simple, we need two more methods (`isOver` and `changeServer`) in the `RBallGame` class and two more (`winsServe` and `incScore`) for the `Player` class.



Implementing `RBallGame`

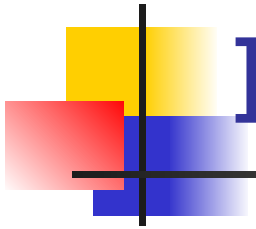
- Before working on these methods, let's go back and finish the other top-level method of the `RBallGame` class, `getScores`, which returns the scores of the two players.
- The `player` objects actually know the scores, so we need a method that asks a `player` to return its score.



Implementing `RBallGame`

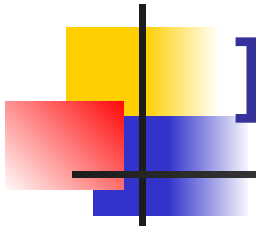
```
def getScores(self):
        # RETURNS the current scores of player A and player B
        return self.playerA.getScore(), self.playerB.getScore()
```

- This adds one more method to be implemented in the `Player` class! Don't forget it!!
- To finish the RBallGame class, all that is needed is to write the `isOver` and `changeServer` methods (left as an exercise).



Implementing `Player`

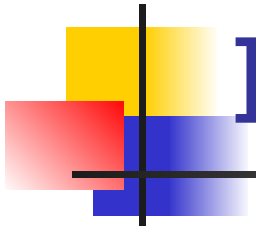
- While developing the `RBallGame` class, we discovered the need for a `Player` class that encapsulates the service probability and current score for a player.
- The `Player` class needs a suitable constructor and methods for `winsServe`, `incScore`, and `getScore`.



Implementing `Player`

- In the class constructor, we need to initialize the instance variables. The probability will be passed as a variable, and the score is set to 0.

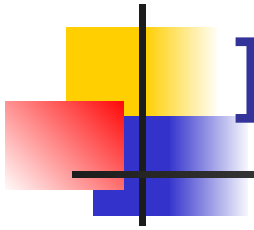
```
def __init__(self, prob):
    # Create a player with this probability
    self.prob = prob
    self.score = 0
```



Implementing `Player`

- To see if a player wins a serve, compare the probability of service win to a random number between 0 and 1.

```
def winsServe(self):
        # RETURNS true with probability self.prob
        return random() <= self.prob
```



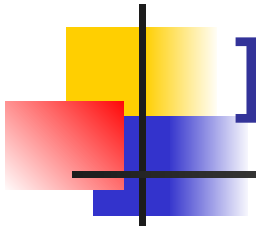
Implementing `Player`

- To give a player a point, we add one to the score.

```
def incScore(self):
    # Add a point to this player's score
    self.score = self.score + 1
```

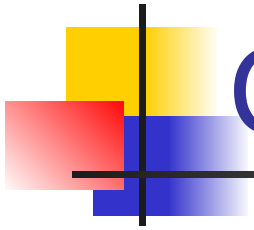
- The final method returns the value of the score.

```
def getScore(self):
    # RETURN this player's current score
    return self.score
```



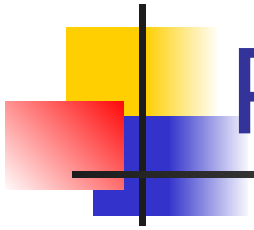
Implementing `Player`

- You may think it's silly to create a class with many one or two-line methods.
- This is quite common in well-modularized, object-oriented programs.
- If the pieces are so simple that their implementation is obvious, we have confidence that it must be right!



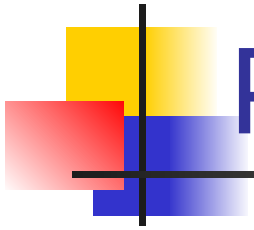
Case Study: Dice Poker

- Objects are very useful when designing graphical user interfaces.
- Let's look at a graphical application using some of the widgets developed in previous chapters.



Program Specification

- Our goal is to write a program that allows a user to play video poker using dice.
- The program will display a hand consisting of five dice.



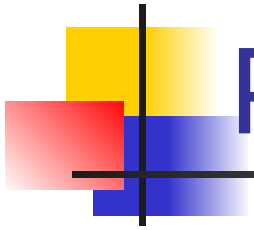
Program Specification

- The basic rules
  - The player starts with $100
  - Each round costs $10 to play. This amount is subtracted from the user's money at the start of the round.
  - The player initially rolls a completely random hand (all 5 dice are rolled).
  - The player gets two chances to enhance the hand by rerolling some or all of the dice.

Program Specification

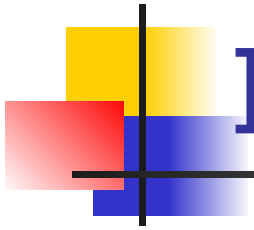
- At the end of the hand, the player's money is updated according to the following payout schedule:

Hand	Pay
Two Pairs	5
Three of a Kind	8
Full House (A Pair and a Three of a Kind)	12
Four of a Kind	15
Straight (1-5 or 2-6)	20
Five of a Kind	30



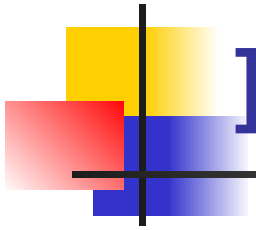
Program Specification

- Since we want a nice graphical interface, we will be interacting with our program through mouse clicks.
- The interface should have:
  - The current score (amount of money) is constantly displayed.
  - The program automatically terminates if the player goes broke.
  - The player may choose to quit at appropriate points during play.
  - The interface will present visual cues to indicate what is going on at any given moment and what the valid user responses are.



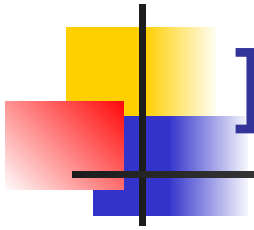
Identifying Candidate Objects

- The first step is to analyze the program description and identify some objects that will be useful in solving the problem.
- This game involves dice and money. Are they good object candidates?
- On their own, a single die and the money can be represented as numbers.



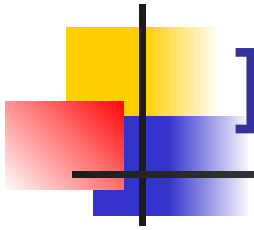
Identifying Candidate Objects

- However, the game uses five dice, and we need to be able to roll all or a selection of the dice, as well as analyze the collection to see what it scores.
- This can be encapsulated in a `Dice` class.



Identifying Candidate Objects

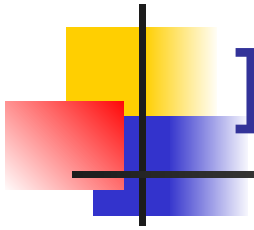
- Here are some obvious operations to implement:
  - Constructor – Create the initial collection
  - `rollAll` – Assign random values to each of the five dice
  - `roll` – Assign a random value to some subset of the dice, while maintaining the current value of the others.



Identifying Candidate Objects

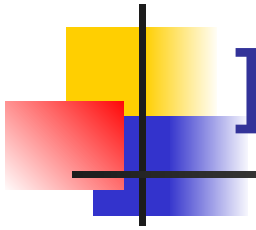
- `values` – Return the current values of the five dice
- `score` – Return the score for the dice

- The entire program can be thought of as an object. Let's call the class `PokerApp`.
- The `PokerApp` object will keep track of the current amount of money, the dice, the number of rolls, etc.
- `PokerApp` will use a method called `run` to start the game.



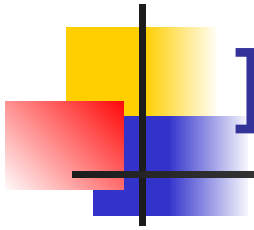
Identifying Candidate Objects

- Another component of the game is the user interface.
- A good way to break down the complexity of a more sophisticated problem is to separate the UI from the main program.
- This is often called the model-view approach, where the program implements some model and the interface is a view of the current state of the model.



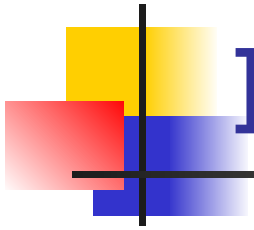
Identifying Candidate Objects

- We can encapsulate the decisions about the interface in a separate interface object.
- One advantage of this approach is that we can change the look and feel of the program by substituting a different interface object.
- Let's call our interface object `PokerInterface`.



Implementing the Model

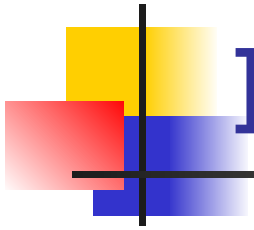
- The `Dice` class implements a collection of dice, which are just changing numbers.
- The obvious representation is a list of five `int`s. The constructor needs to create a list and assign some initial values.



Implementing the Model

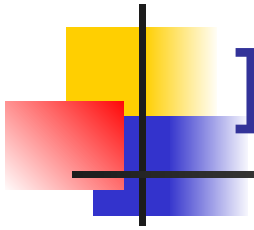
```
class Dice:
    def __init__(self):
        self.dice = [0]*5
        self.rollAll()
```

- This code first creates a list of five zeroes. Then they need to be set to random values.
- We need methods to roll selected dice and to roll all of the dice.



Implementing the Model

- Since rolling all dice is a special case of rolling selected dice, we can implement the former with the latter.
- We can specify which dice to roll by passing a list of indexes. For example, `roll([0,3,4])` will roll the dice in positions 0, 3, and 4.
- We can use a loop to go through the list, generating a new random value for each listed position.



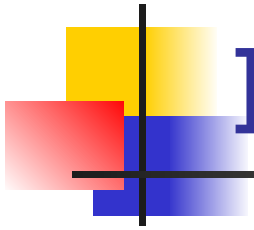
Implementing the Model

```
def roll(self, which)
    for pos in which:
        self.dice[pos] = randrange(1,7)
```

- We can use roll to implement `rollAll`...

```
def rollAll(self):
    self.roll(range(5))
```

- Here, `range(5)` is used to generate a list of all the indexes.

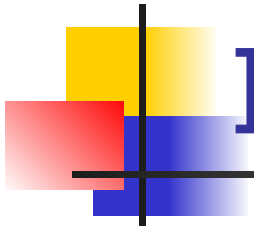


Implementing the Model

- The `values` function returns the values of the dice so they can be displayed.

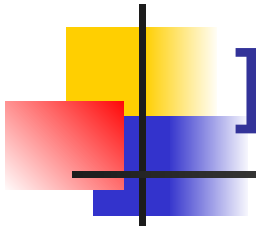
```
def values(self):
    return self.dice[:]
```

- Why did we create a copy of the dice list by slicing it? Defensive Programming!
- If a `Dice` client modifies the list it gets back from `values`, it will not affect the original copy stored in the `Dice` object.



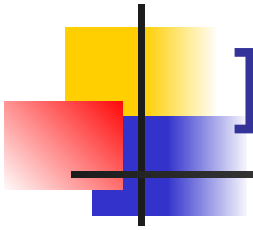
Implementing the Model

- The `score` method will determine the worth of the current dice.
- We need to examine the values and determine whether we have any of the patterns in the table.
- Let's return a string with what the hand is and an int that gives the payoff amount.



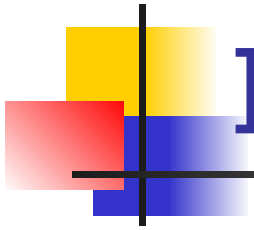
Implementing the Model

- We can think of this function as a multi-way decision, checking for each possible hand.
- The order that we do the check is important! A full house also contains a three of a kind, but the payout should be for a full house!



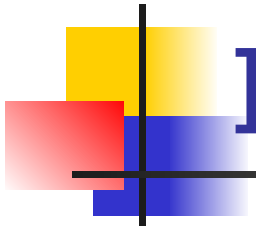
Implementing the Model

- One simple way to check the hand is to create a list of the counts of each value.
- `counts[i]` will be the number of times that `i` occurs in the roll.
- If the dice are `[3,2,5,2,3]`, then the count list will be `[0,0,2,2,0,1,0]`.
- `counts[0]` will always be 0 since dice go from 1 – 6.



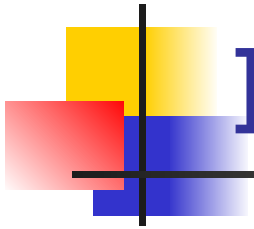
Implementing the Model

- With this approach, checking for a full house entails looking for a 3 and a 2 in `counts`.
- ```
  def score(self):
      # Create the counts list
      counts = [0] * 7
      for value in self.dice:
          counts[value] = counts[value] + 1
  ```
  ```



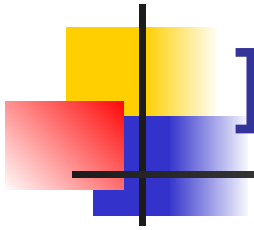
Implementing the Model

```
if 5 in counts:
    return "Five of a Kind", 30
elif 4 in counts:
    return "Four of a Kind", 15
elif (3 in counts) and (2 in counts):
    return "Full House", 12
elif (not (3 in counts)) and (not (2 in counts)) \
     and (counts[1]==0 or counts[6] == 0):
    return "Straight", 20
elif 3 in counts:
    return "Three of a Kind", 8
elif counts.count(2) == 2:
    return "Two Pairs", 5
else:
    return "Garbage", 0
```



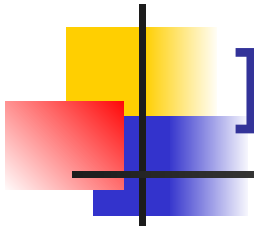
Implementing the Model

- Since we've already checked for 5, 4, and 3 of a kind, checking that there are no pairs -- `(not (2 in counts))` guarantees that the dice show five distinct values. If there is no 6, then the values must be 1-5, and if there is no 1, the values must be 2-6.



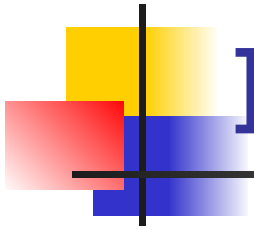
Implementing the Model

- Let's try it out!
-
```
>>> from dice import Dice
>>> d = Dice()
>>> d.values()
[2, 3, 2, 6, 3]
>>> d.score()
('Two Pairs', 5)
>>> d.roll([3])
>>> d.values()
 [2, 3, 2, 2, 3]
>>> d.score()
('Full House', 12)
```



Implementing the Model

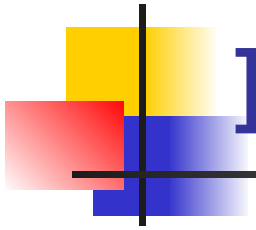
- We now are at the point where we can implement the poker game.
- We can use top-down design to flesh out the details and suggest which methods will need to be implemented in the `PokerInterface` class.
- Initially, `PokerApp` will need to keep track of the dice, the amount of money, and the interface. Let's initialize these values first.



Implementing the Model

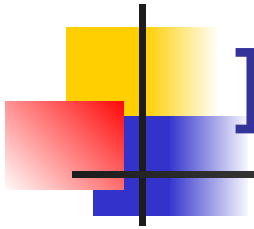
```
class PokerApp:
    def __init__(self):
        self.dice = Dice()
        self.money = 100
        self.interface = PokerInterface()
```

- To run the program, we create an instance of this class and call its `run` method.
- The program will loop, allowing the user to continue playing hands until they are either out of money or choose to quit.



Implementing the Model

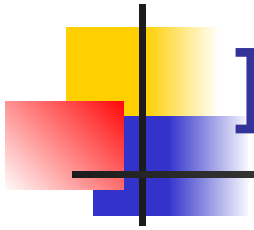
- Since it costs $10 to play a hand, we can continue as long as `self.money >= 10`.
- Determining whether the player wants to continue or not must come from the user interface.



Implementing the Model

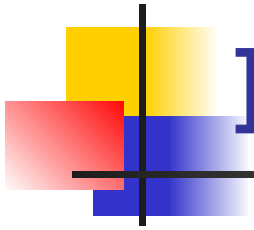
```
def run(self):
    while self.money >= 10 and self.interface.wantToPlay():
        self.playRound()
    self.interface.close()
```

- The `interface.close()` call at the bottom will let us do any necessary clean-up, such as printing a final message, closing graphics windows, etc.
- Now we'll focus on the `playRound` method.



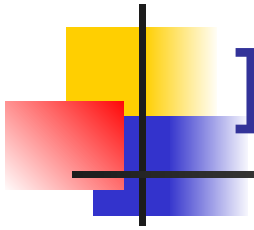
Implementing the Model

- Each round consists of a series of rolls. Based on the rolls, the player's score will be adjusted.
- ```
  def playRound(self):
      self.money = self.money - 10
      self.interface.setMoney(self.money)
      self.doRolls()
      result, score = self.dice.score()
      self.interface.showResult(result, score)
      self.money = self.money + score
      self.interface.setMoney(self.money)
  ```
  ```



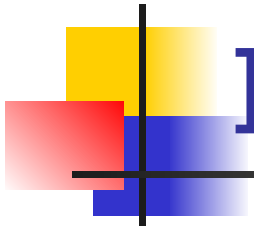
Implementing the Model

- When new information is to be presented to the user, the proper method from `interface` is invoked.
- The $10 fee to play is first deducted, and the interface is updated with the new amount of money remaining.
- The program processes a series of rolls (`doRolls`), displays the result, and updates the money.



Implementing the Model

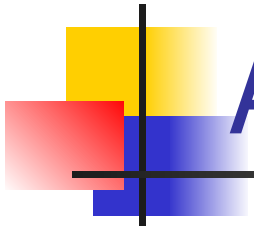
- Lastly, we need to implement the dice rolling process.
- Initially, all the dice are rolled.
- Then, we need a loop that continues rolling user-selected dice until either the user quits rolling or the limit of three rolls is reached.
- `rolls` keeps track of how many times the dice have been rolled.



Implementing the Model

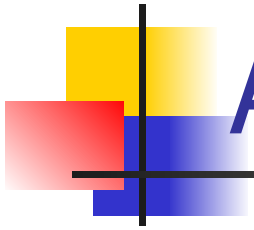
```
def doRolls(self):
    self.dice.rollAll()
    roll = 1
    self.interface.setDice(self.dice.values())
    toRoll = self.interface.chooseDice()
    while roll < 3 and toRoll != []:
        self.dice.roll(toRoll)
        roll = roll + 1
        self.interface.setDice(self.dice.values())
        if roll < 3:
            toRoll = self.interface.chooseDice()
```

- Whew! We've completed the basic functions of our interactive poker program.
- We can't test it yet because we don't have a user interface...



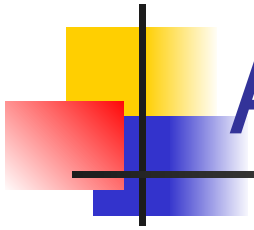
A Text-Based UI

- In the process of designing `PokerApp`, we also developed a specification for a generic `PokerInterface` class.
- The interface must support methods for displaying information –
  - `setMoney`
  - `setDice`
  - `showResult`



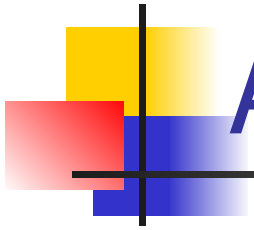
A Text-Based UI

- It also must have methods that allow input from the user –
  - `wantToPlay`
  - `chooseDice`
- These methods can be implemented in many different ways, producing programs that look quite different, even while the underlying model, `PokerApp`, remains the same.



A Text-Based UI

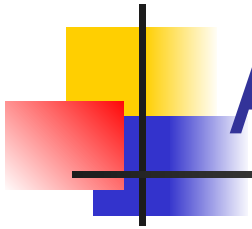
- Graphical interfaces are usually more complicated to build, so we might want to build a text-based interface first for testing and debugging purposes.
- We can tweak the `PokerApp` class so that the user interface is supplied as a parameter to the constructor.



A Text-Based UI

```
def __init__(self, interface):
    self.dice = Dice()
    self.money = 100
    self.interface = interface
```

- By setting the interface up as a parameter, we can easily use different interfaces with our poker program.
- Here's a bare-bones text-based interface:



A Text-Based UI

```
# textinter.py

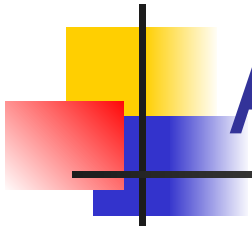
class TextInterface:
    def __init__(self):
        print("Welcome to video poker.")

    def setMoney(self, amt):
        print("You currently have ${0}.".format(amt))

    def setDice(self, values):
        print("Dice:", values)

    def wantToPlay(self):
        ans = input("Do you wish to try your luck? ")
        return ans[0] in "yY"

    def close(self):
        print("\nThanks for playing!")
```



A Text-Based UI

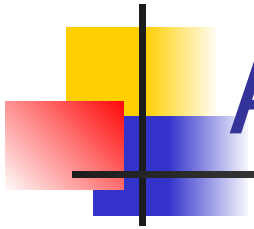
```
def showResult(self, msg, score):
    print("{0}. You win ${1}.".format(msg,score))

def chooseDice(self):
    return eval(input("Enter list of which to change ([] to stop) "))
```

- Using this interface, we can test our `PokerApp` program. Here's a complete program:

```
from pokerapp import PokerApp
from textinter import TextInterface

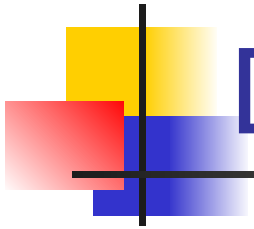
inter = TextInterface()
app = PokerApp(inter)
app.run()
```



A Text-Based UI

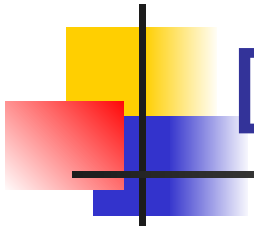
```
Welcome to video poker.
Do you wish to try your luck? y
You currently have $90.
Dice: [6, 4, 1, 1, 6]
Enter list of which to change ([] to stop) [1]
Dice: [6, 3, 1, 1, 6]
Enter list of which to change ([] to stop) [1]
Dice: [6, 4, 1, 1, 6]
Two Pairs. You win $5.
You currently have $95.
Do you wish to try your luck? y
You currently have $85.
Dice: [5, 1, 3, 6, 4]
Enter list of which to change ([] to stop) [1]
Dice: [5, 2, 3, 6, 4]
Enter list of which to change ([] to stop) []
Straight. You win $20.
You currently have $105.
Do you wish to try your luck? n

Thanks for playing!
```



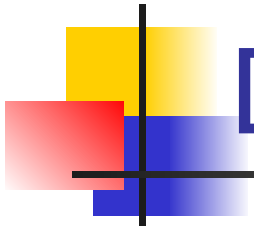
Developing a GUI

- Now that we've verified that our program works, we can start work on the GUI user interface.
- This new interface will support the various methods found in the text-based version, and will likely have additional helper methods.



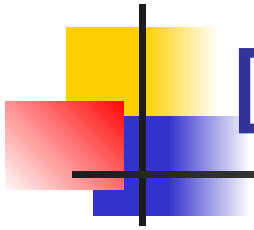
Developing a GUI

- Requirements
  - The faces of the dice and the current score will be continuously displayed.
  - The `setDice` and `setMoney` methods will be used to change these displays.
  - We have one output method, `showResult`. One way we can display this information is at the bottom of the window, in what is sometimes called a status bar.



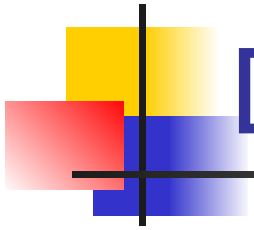
Developing a GUI

- We can use buttons to get information from the user.
- In `wantToPlay`, the user can choose between rolling the dice or quitting by selecting the "Roll Dice" or "Quit" buttons.
- To implement `chooseDice`, we could have a button to push for each die to be rolled. When done selecting the dice to roll, the "Roll Dice" button could be pushed.



Developing a GUI

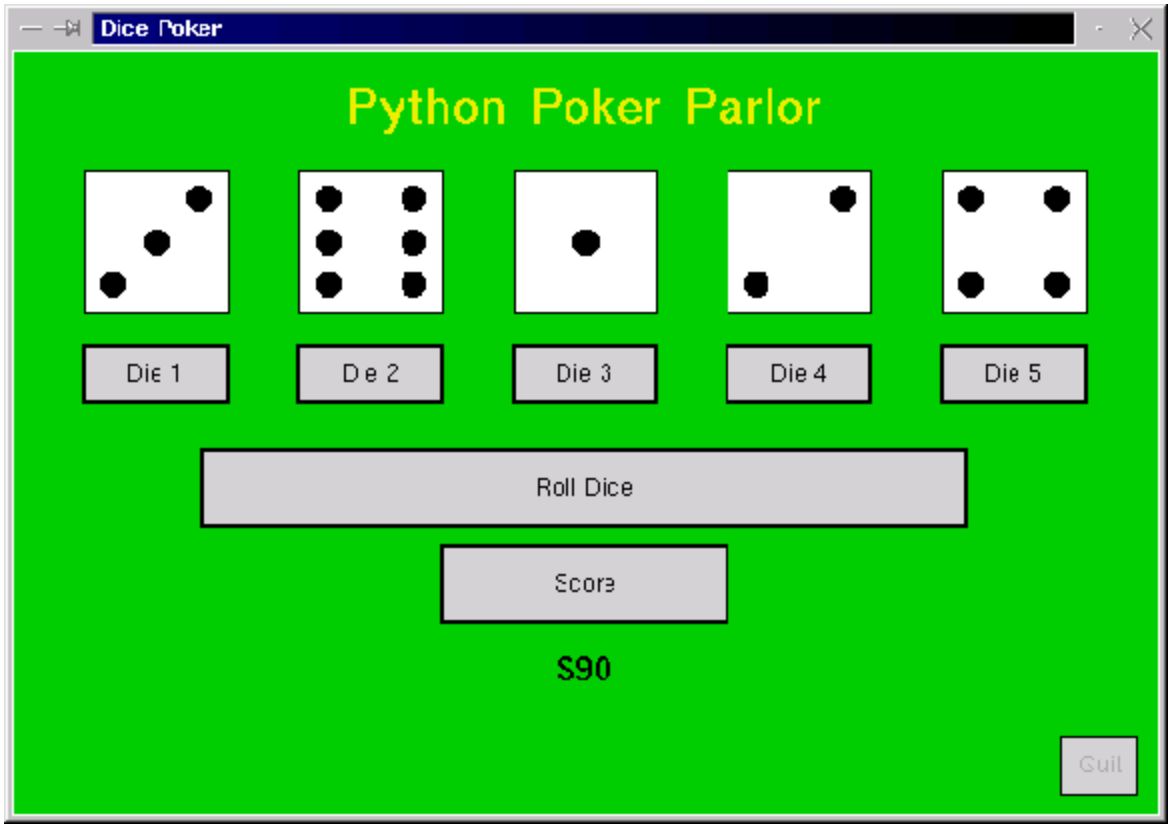
- We could allow the users to change their mind on which dice to choose by having the button be a toggle that selects/unselects a particular die.
- This enhancement suggests that we want a way to show which dice are currently selected. We could easily "gray out" the pips on dice selected for rolling.

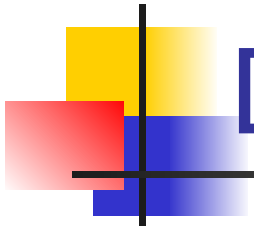


Developing a GUI

- We also need a way to indicate that we want to stop rolling and score the dice as they are. One way to do this could be by not having any selected dice and choosing "Roll Dice". A more intuitive solution would be to add a new button called "Score".
- Now that the functional aspects are decided, how should the GUI look?

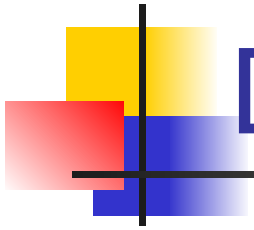
Developing a GUI





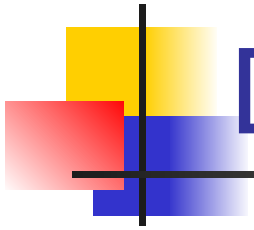
Developing a GUI

- Our GUI makes use of buttons and dice. We can reuse our `Button` and `DieView` classes from previous chapters!
- We'll use a list of `Button`s as we did in the calculator program in Chapter 11.
- The buttons of the poker interface will not be active all of the time. E.g., the dice buttons are only active when the user is choosing dice.



Developing a GUI

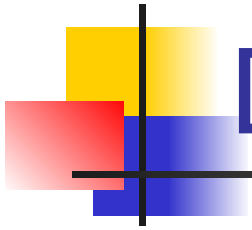
- When user input is required, the valid buttons for that interaction will be set active and the others set inactive, using a helper method called `choose`.
- The `choose` method takes a list of button labels as a parameter, activates them, and then waits for the user to click one of them.



Developing a GUI

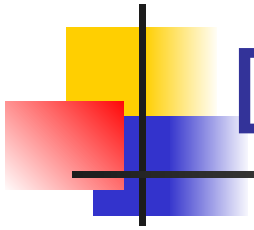
- The return value is the label of the button that was clicked.
- For example, if we are waiting for the user to choose either the “Roll Dice” or “Quit” button, we could use this code:

```
choice = self.choose(["Roll Dice", "Quit"])
if choice == ("Roll Dice"):
    …
```



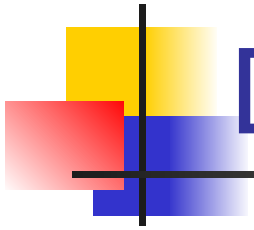
Developing a GUI

```
def choose(self, choices):
    buttons = self.buttons
     # activate choice buttons, deactivate others
    for b in buttons:
        if b.getLabel() in choices:
            b.activate()
        else:
            b.deactivate()
    # get mouse clicks until an active button is clicked
    while True:
        p = self.win.getMouse()
        for b in buttons:
            if b.clicked(p):
                return b.getLabel() # function exit here
```



Developing a GUI

- The `DieView` class will be basically the same as we used before, but we want to add a new feature – the ability to change the color of a die to indicate when it is selected for rerolling.
- The `DieView` constructor draws a square and seven circles to represent where the pips appear. `setValue` turns on the appropriate pips for a given value.

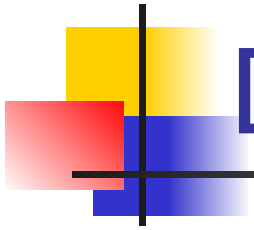


Developing a GUI

- Here's the `setValue` method as it was:

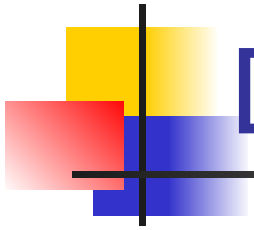
```
def setValue(self, value):
    # Turn all the pips off
    for pip in self.pips:
        pip.setFill(self.background)

    # Turn the appropriate pips back on
    for i in self.onTable[value]:
        self.pips[i].setFill(self.foreground)
```



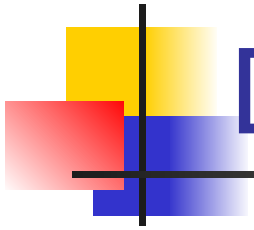
Developing a GUI

- We need to modify the `DieView` class by adding a `setColor` method to change the color used for drawing the pips.
- In `setValue`, the color of the pips is determined by the value of the instance variable `foreground`.



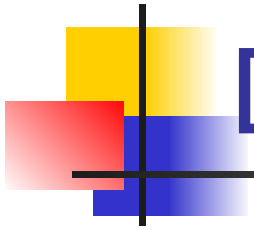
Developing a GUI

- The algorithm for `setColor` seems straightforward.
  - Change `foreground` to the new color
  - Redraw the current value of the die
- The second step is similar to `setValue`, but `setValue` requires the value to be sent as a parameter, and `dieView` doesn't store this value anywhere. Once the pips have been turned on the value is discarded!



Developing a GUI

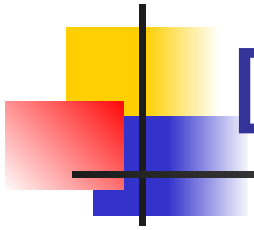
- To implement `setColor`, we tweak `setValue` so that it remembers the current value:
  `self.value = value`
- This line stores the value parameter in an instance variable called `value`.
- With the modification to `setValue`, `setColor` is a breeze.



Developing a GUI

```
def setColor(self, color):
    self.foreground = color
    self.setValue(self.value)
```

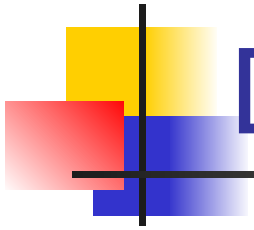
- Notice how the last line calls `setValue` to draw the die, passing along the value from the last time `setValue` was called.
- Now that the widgets are under control, we can implement the poker GUI! The constructor will create all the widgets and set up the interface for later interactions.



Developing a GUI

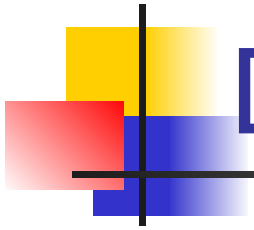
```
class GraphicsInterface:

    def __init__(self):
        self.win = GraphWin("Dice Poker", 600, 400)
        self.win.setBackground("green3")
        banner = Text(Point(300,30), "Python  Poker  Parlor")
        banner.setSize(24)
        banner.setFill("yellow2")
        banner.setStyle("bold")
        banner.draw(self.win)
        self.msg = Text(Point(300,380), "Welcome to the dice table.")
        self.msg.setSize(18)
        self.msg.draw(self.win)
```



Developing a GUI

```
self.createDice(Point(300,100), 75)
self.buttons = []
self.addDiceButtons(Point(300,170), 75, 30)
b = Button(self.win, Point(300, 230), 400, 40, "Roll Dice")
self.buttons.append(b)
b = Button(self.win, Point(300, 280), 150, 40, "Score")
self.buttons.append(b)
b = Button(self.win, Point(570,375), 40, 30, "Quit")
self.buttons.append(b)
self.money = Text(Point(300,325), "$100")
self.money.setSize(18)
self.money.draw(self.win)
```

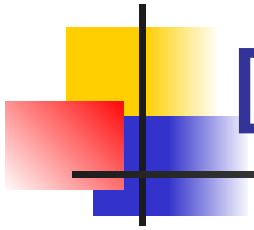


Developing a GUI

- Did you notice that the creation of the dice and their associated buttons were moved into a couple of helper methods?

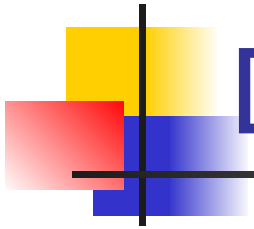
```
def createDice(self, center, size):
    center.move(-3*size,0)
    self.dice = []
    for i in range(5):
        view = ColorDieView(self.win, center, size)
        self.dice.append(view)
        center.move(1.5*size,0)

def addDiceButtons(self, center, width, height):
    center.move(-3*width, 0)
    for i in range(1,6):
        label = "Die %d" % (i)
        b = Button(self.win, center, width, height, label)
        self.buttons.append(b)
        center.move(1.5*width, 0)
```



Developing a GUI

- `center` is a `Point` variable used to calculate the positions of the widgets.
- The methods `setMoney` and `showResult` display text in an interface window. Since the constructor created and positioned the `Text` objects, all we have to do is call `setText`!
- Similarly, the output method `setDice` calls the `setValue` method of the appropriate `DieView` objects in `dice`.

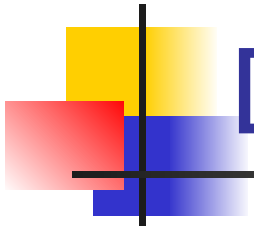


Developing a GUI

```
def setMoney(self, amt):
    self.money.setText("${0}".format(amt))

def showResult(self, msg, score):
    if score > 0:
        text = "{0}! You win ${1}".format(msg, score)
    else:
        text = "You rolled {0}".format(msg)
    self.msg.setText(text)

def setDice(self, values):
    for i in range(5):
        self.dice[i].setValue(values[i])
```

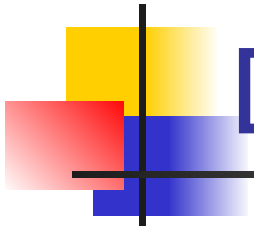


Developing a GUI

- The `wantToPlay` method will wait for the user to click either "Roll Dice" or "Quit". The `chooser` helper method can be used.

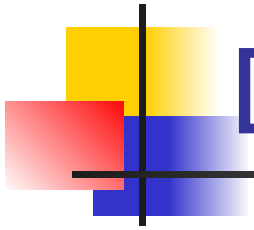
```
def wantToPlay(self):
    ans = self.choose(["Roll Dice", "Quit"])
    self.msg.setText("")
    return ans == "Roll Dice"
```

- After the user clicks a button, setting `msg` to `""` clears out any messages.



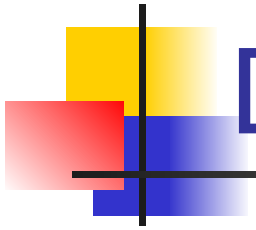
Developing a GUI

- The `chooseDice` method is a little more complicated – it will return a list of the indexes of the dice the user wishes to roll.
- In our GUI, the user chooses dice by clicking on the corresponding button.
- We need to maintain a list of selected buttons.



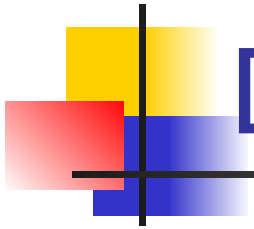
Developing a GUI

- Each time a button is clicked, that die is either chosen (its index appended to the list) or unchosen (its index removed from the list).
- The color of the corresponding `dieView` will then reflect the current status of the dice.



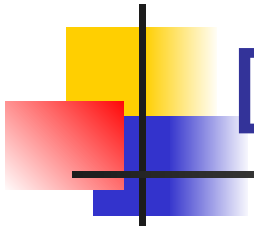
Developing a GUI

- If the roll button is clicked, the method returns the list of currently chosen indexes.
- If the score button is clicked, the function returns an empty list to signal that the player is done rolling.



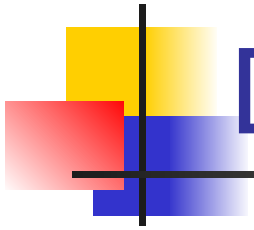
Developing a GUI

```
def chooseDice(self):
    # choices is a list of the indexes of the selected dice
    choices = []                    # No dice chosen yet
    while True:
        # Wait for user to click a valid button
        b = self.choose(["Die 1", "Die 2", "Die 3", "Die 4", "Die 5",
                         "Roll Dice", "Score"])
        if b[0] == "D":             # User clicked a die button
            i = eval(b[4]) - 1      # Translate label to die index
            if i in choices:        # Currently selected, unselect it
                choices.remove(i)
                self.dice[i].setColor("black")
            else:                   # Currently unselected, select it
                choices.append(i)
                self.dice[i].setColor("gray")
        else:                       # User clicked Roll or Score
            for d in self.dice:     # Revert appearance of all dice
                d.setColor("black")
            if b == "Score":        # Score clicked, ignore choices
                 return []
             elif choices != []:     # Don't accept Roll unless some
                  return choices      # dice are actually selected
```



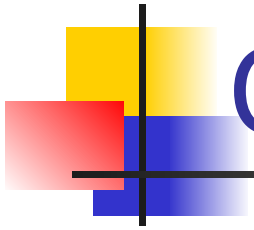
Developing a GUI

- The only missing piece of our interface class is the `close` method.
- To close the graphical version, we just need to close the graphics window.
- ```
  def close(self):
      self.win.close()
  ```
  ```



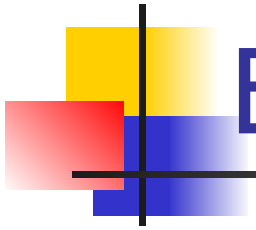
Developing a GUI

- Lastly, we need a few lines to get the graphical poker playing program started! We use `GraphicsInterface` in place of `TextInterface`.
- ```
  inter = GraphicsInterface()
  app = PokerApp(inter)
  app.run()
  ```
  ```



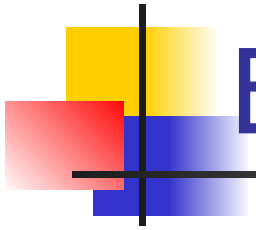
OO Concepts

- The OO approach helps us to produce complex software that is more reliable and cost-effective.
- OO is comprised of three principles:
  - Encapsulation
  - Polymorphism
  - Inheritance



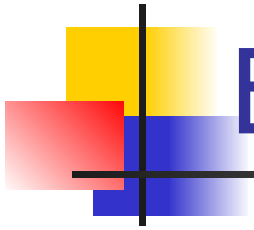
Encapsulation

- As you'll recall, objects know stuff and do stuff, combining data and operations.
- This packaging of data with a set of operations that can be performed on the data is called encapsulation.
- Encapsulation provides a convenient way to compose complex problems that corresponds to our intuitive view of how the world works.



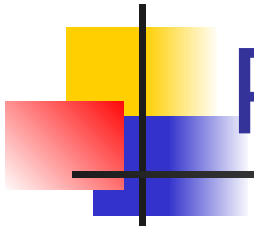
Encapsulation

- From a design standpoint, encapsulation separates the concerns of “what” vs. “how”. The implementation of an object is independent of its use.
- The implementation can change, but as long as the interface is preserved, the object will not break.
- Encapsulation allows us to isolate major design decisions, especially ones subject to change.



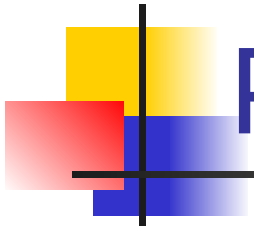
Encapsulation

- Another advantage is that it promotes code reuse. It allows us to package up general components that can be used from one program to the next.
- The `DieView` and `Button` classes are good examples of this.
- Encapsulation alone makes a system object-based. To be object-oriented, we must also have the properties of polymorphism and inheritance.



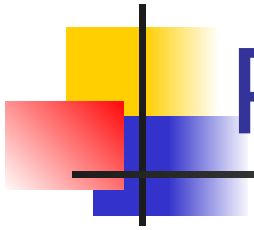
Polymorphism

- Literally, polymorphism means “many forms.”
- When used in object-oriented literature, this refers to the fact that what an object does in response to a message (a method call) depends on the type or class of the object.



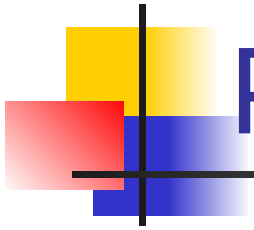
Polymorphism

- Our poker program illustrated one aspect of this by the `PokerApp` class being used with both `TextInterface` and `GraphicsInterface`.
- When `PokerApp` called the `showDice` method, the `TextInterface` showed the dice one way and the `GraphicsInterface` did it another way.



Polymorphism

- With polymorphism, a given line in a program may invoke a completely different method from one moment to the next.
- Suppose you had a list of graphics objects to draw on the screen – a mixture of `Circle`, `Rectangle`, `Polygon`, etc.

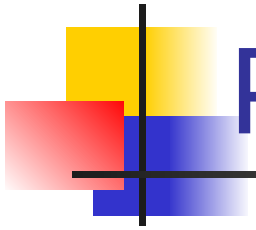


Polymorphism

- You could draw all the items with this simple code:

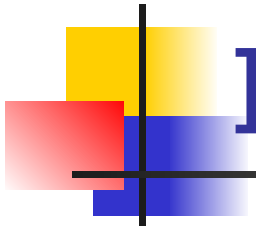
```
for obj in objects:
    obj.draw(win)
```

- What operation does this loop really execute?
- When `obj` is a circle, it executes the draw method from the circle class, etc.



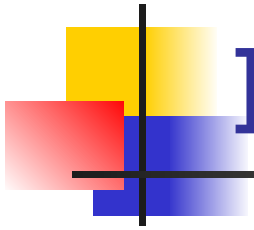
Polymorphism

- Polymorphism gives object-oriented systems the flexibility for each object to perform an action just the way that it should be performed for that object.



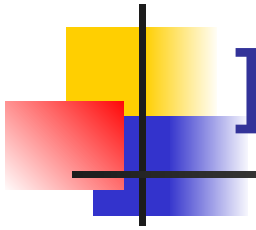
Inheritance

- The idea behind inheritance is that a new class can be defined to borrow behavior from another class.
- The new class (the one doing the borrowing) is called a subclass, and the other (the one being borrowed from) is called a superclass.
- This is an idea our examples have not included.



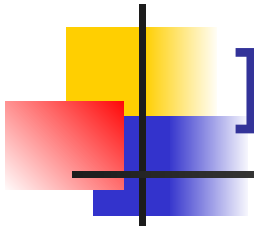
Inheritance

- Say we're building an employee management system.
- We might have a class called `Employee` that contains general information common to all employees. There might be a method called `homeAddress` that returns an employee's home address.



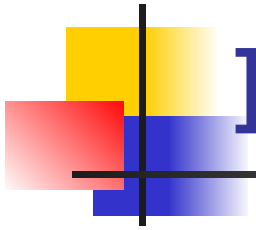
Inheritance

- Within the class of employees, we might distinguish between salaried and hourly employees with `SalariedEmployee` and `HourlyEmployee`, respectively.
- Each of these two classes would be a subclass of `Employee`, and would share the `homeAddress` method.



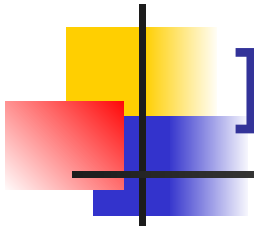
Inheritance

- Each subclass could have its own `monthlyPay` function, since pay is computed differently for each class of employee.
- Inheritance has two benefits:
  1. We can structure the classes of a system to avoid duplication of operations, e.g. there is one `homeAddress` method for `HourlyEmployee` and `SalariedEmployee`.
  2. New classes can be based on existing classes, promoting code reuse.



Inheritance

- We could have used inheritance to build the `DieView` class.
- Our first `DieView` class did not provide a way to change the appearance of the die.
- Rather than modifying the original class definition, we could have left the original alone and created a new subclass called `ColorDieView`.



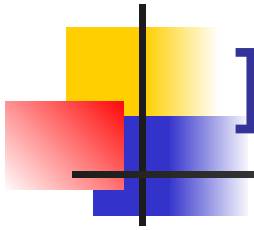
Inheritance

- A `ColorDieView` is just like `DieView`, except it has an additional method!

```
class ColorDieView(DieView):

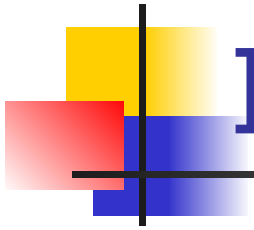
    def setValue(self, value):
        self.value = value
        DieView.setValue(self, value)

    def setColor(self, color):
        self.foreground = color
        self.setValue(self.value)
```



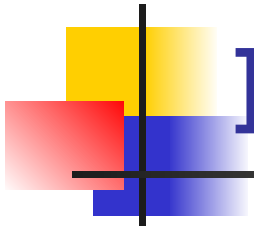
Inheritance

- The first line (`class ColorDieView(DieView):`) says that we are defining a new class `ColorDieView` that is based on (i.e. is a subclass of) `DieView`.
- Inside the new class we define two methods.
- The second method, `setColor`, adds the new operation. To make it work, `setValue` also needed to be slightly modified.



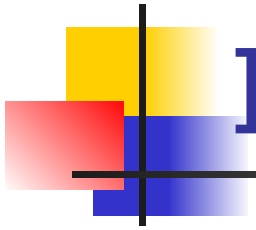
Inheritance

- The `setValue` method in `ColorDieView` redefines or overrides the definition of `setValue` that was provided in the `DieView` class.
- The `setValue` method in the new class first stores the value and then relies on the `setValue` method of the superclass `DieView` to actually draw the pips.



Inheritance

- The normal approach to set the value, `self.setValue(value)`, would refer to the `setValue` method of the `ColorDieView` class, since `self` is an instance of `ColorDieView`.
- To call the superclass's `setValue` method, it's necessary to put the class name where the object would normally go: `DieView.setValue(self,value)`



Inheritance

- `DieView.setValue(self,value)`
- The actual object to which the method is applied is sent as the first parameter.