# Racquetball simulation with OOD

#1.py

# We'll put all the classes and the main program in this one file. But you now know that
# you can make separate modules

# It shows how to design with objects

from random import random

#-------------------------------------------
class Player:
    # A player keeps track of service probability and score

def __init__(self, prob):
    # create a player with given probability of winning
    self.prob = prob
    self.score = 0

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

def incScore(self):
    # increment score by 1
    self.score = self.score + 1

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

#-------------------------------------------
class RBallGame:
    # The RBallGame object is a game in progress. There are two players. The game
    # keeps track of who is now serving.

[Nov 17, 2015]

Week 13, Examples 1
def __init__(self, probA, probB):
    # create new game; give players these probabilities

    self.playerA = Player(probA)  # notice how this
    self.playerB = Player(probB)  # another object

    self.server = self.playerA  # we make player A
    always serve first

def play(self):
    # Play a complete game

    while (not self.isOver()):
        if self.server.winsServe():
            self.server.incScore()
        else:
            self.changeServer()

def isOver(self):
    # returns True if game is over (i.e., A or B has won)
    a, b = self.getScores()

    return (a == 15 or b == 15) or \  
           ((a == 7 and b == 0) or (b == 7 and a == 0))  # that "\" is
to show how to continue
    # lines, if necc.

def changeServer(self):
    # switch server
    if (self.server == self.playerA):
        self.server = self.playerB
    else:
        self.server = self.playerA

def getScores(self):
    # Return current scores for A and B
    return (self.playerA.getScore(), self.playerB.getScore())

#----------------------------------------------------------------------------------
---

class SimStats:

    # This class keeps track of statistics over multiple games; it tracks
    wins and shutouts for A & B
def __init__(self):
    self.winsA = 0
    self.winsB = 0
    self.shutsA = 0
    self.shutsB = 0

def update(self, aGame):
    # here we will pass the whole game
    # object as parameter aGame
    a, b = aGame.getScores()
    if (a > b):
        self.winsA = self.winsA + 1
        if (b == 0):
            self.shutsA = self.shutsA + 1
    else:
        self.winsB = self.winsB + 1
        if (a == 0):
            self.shutsB = self.shutsB + 1

def printReport(self):
    # print formatted report
    n = self.winsA + self.winsB
    print("Sumar of ",n," games: \n")
    print("                  wins  (% total)            shutouts  "/wins)   
    print("__{0:_<20}__{1:_<15}__{2:_<15}__")
    self.printLine("A", self.winsA, self.shutsA, n)
    self.printLine("B", self.winsB, self.shutsB, n)

def printLine(self, label, wins, shuts, n):
    template = "Player  {0}:{1:5}    ({2:5.1%})        {3:11} ({4})"
    if (wins == 0):
        shutStr = "-----"
    else:
        shutStr = "[0:4.1%].format(float(shuts)/wins)"
    print(template.format(label, wins, float(wins)/n, shuts, shutStr))

#______________________ classes done ___________________________________

def printIntro():

Computer Science Courses - http://courses.cs.purdue.edu/
This program simulates games of racquetball between players A and B.

Each player has a given input probability of winning when serving.

A always serves first.

def getInputs():
    # asks user for 3 simulation parameters
    a = eval(input("What is the probability that player A wins a serve? "))
    b = eval(input("What is the probability that player B wins a serve? "))
    n = eval(input("How many games do you want to simulate? "))
    return (a, b, n)

def main():
    printIntro()
    probA, probB, n = getInputs()
    # play the game
    stats = SimStats()
    for i in range(n):
        theGame = RBallGame(probA, probB)  # create a new game
        theGame.play()  # play the game
        stats.update(theGame)  # pass the game object to SimStats so it
        # can access the info it needs
    # print results
    stats.printReport()

main()

input("\nPress <Enter> to quit")
#2.py

''' What is a rational number?
A rational number has two integers: one a numerator, the other a denominator
e.g., 1/2, 1/3, 2/17 ..... 

What operations can you do on rational numbers? Well ....
-- all arithmetic operations because rational numbers are numbers, and also
-- comparison (are two rational numbers equal? is one larger than the other?)

Python has built-in integers, floats etc .... but has no built-in type for rational numbers.
So let us write a class for rational numbers. Call it Rational.''

# How will we use it? Here are some examples:
'''
>> oneHalf = Rational(1, 2)
>> oneSixth = Rational (1, 6)
>> print(oneHalf)
1/2
>> print(oneHalf + oneSixth)
2/3
>> oneHalf == oneSixth
False
>> oneHalf > oneSixth
True
'''

# We will start by using the built-in operators +, ==, < with objects of class Rational

''' STEPS:

1. Handle the internal representation of a rational number and also its string representation.

    constructor: wants num and denom. It uses 2 instance variables.

    Next the constructor must reduce the rational number to its lowest terms.

Example: 88/33. How to reduce it to its lowest terms?

    Find greatest common divisor (GCD) of numerator and denominator
11 is GCD of num and denom; so the lowest term is 8/3 (i.e., divide num and denom by the gcd) to do this are The functions we will write "reduce()" and "gcd()"

# How to write a Class to manipulate rational numbers

#File: rational.py

Resources to manipulate rational numbers

class Rational:
    """Represents a rational number.""
    def __init__(self, num, denom):
        """Creates a number with given numerator and denominator and then reduces it to lowest terms.""
        self.num = num
        self.denom = denom

        #__________________ Explaining how to reduce with internal helper methods

        # Now get help to reduce the rational number to lowest terms
        self._reduce()  # It's an internal helper function/method

def _reduce(self):
    """Helper to reduce number to lowest terms.""
    divisor = self._gcd(self.num, self.denom)

    self.num = self.num // divisor
    self.denom = self.denom // divisor

def _gcd(self, a, b):
    """Euclid's algorithm to find the greatest common divisor of two numbers.""
    (a, b) = (max(a, b), min(a, b))
while (b > 0):
    (a, b) = (b, a % b)
return (a)

#_________ we've done the reduce(); now for the vanilla class methods

def numerator(self):
    """ Returns numerator.""
    return self.num

def denominator(self):
    """ Returns denominator. ""
    return self.denom

def __str__(self):
    """ Returns the string representation of the number.""
    return (str(self.num) + "/" + str(self.denom))

#

#___ at this point the class Rational is ready; you can test rational objects and print them ______
#

# You'll see that we'll have trouble using the built in arithmetic operators like +, >, etc
# because those operator do not know how to work with our newly defined Rational
# object. They were written for built-in types "int" and "float" etc.

# Try the examples we showed at the very top of this file. + and > will not work.

...

Python has built-in data types such as "int" and "float" etc.

Each arithmetic operator corresponds to a special method name.

Want to know what these names are? Simply type "dir(int)", or "dir(str)" etc on a Python prompt.

Example:

>>> dir(int)
[ '__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
 '__delattr__', '__dir__',
 '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
 '__floordiv__', '__format__',
 '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
 '__index__',
 '__init__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
 '__mod__', '__mul__', ETC. ETC.
 ...# How do built-in operators like "+" etc. work on the built-in data types?

''' First take a look at the usual operators, and their method names:

<table>
<thead>
<tr>
<th>operator</th>
<th>method name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td><strong>add</strong></td>
</tr>
<tr>
<td>-</td>
<td><strong>sub</strong></td>
</tr>
<tr>
<td>*</td>
<td><strong>mul</strong></td>
</tr>
<tr>
<td>/</td>
<td><strong>div</strong></td>
</tr>
<tr>
<td>%</td>
<td><strong>mod</strong></td>
</tr>
</tbody>
</table>

...# How is "+" a method?

''' x + y: x is called "left-operand", + is called "operator", y is called "right-operand"

When you do an "x + y" on built-in objects x and y

Python does: x.__add__(y)

NOTE: the __add__() method is called on the object in the left-operand

the object in the right-operand becomes the second parameter of this method
(recall the first parameter is always "self")

So x + y is really short-hand for the code x.__add__(y)

...# How to OVERLOAD operators so we can get +, > to work with our Rational type___________

#3.py

$\text{code between lines with dollar symbols is same as in file}
What is a rational number?
A rational number has two integers: one a numerator, the other a denominator
e.g., 1/2, 1/3, 2/17 ..... 

What operations can you do on rational numbers?
-- well, all arithmetic operations because rational numbers are numbers, and
-- comparison (are two rational numbers equal? is one larger than the other?)

Python has built-in integers, floats etc .... but has no built-in type for rational numbers.
So let us write a class for rational numbers. Call it Rational.''

# How will we use it? Here are some examples:
''
>> oneHalf = Rational(1, 2)
>> oneSixth = Rational (1, 6)
>> print(oneHalf)
1/2
>> print(oneHalf + oneSixth)
2/3
>> oneHalf == oneSixth
False
>> oneHalf > oneSixth
True
''

# We will start by using the built-in operators +, ==, < with objects of class Rational

'' STEPS:
1. Handle the internal representation of a rational number and also its string representation.
   constructor: wants num and denom. It uses 2 instance variables.
   Next the constructor must reduce the rational number to its lowest terms.
   Example: 88/33. How to reduce it to its lowest terms?
   (GCD) of numerator and denominator
   Find greatest common divisor
   11 is GCD of num and denom;
so the lowest term is \(\frac{8}{3}\) (i.e., divide num and denom by the gcd)

The functions we will write to do this are "reduce()" and "gcd()"

...

# How to write a Class to manipulate rational numbers
...

# File: rational.py
Resources to manipulate rational numbers
...

class Rational:
    """Represents a rational number.""

    def __init__(self, num, denom):
        """Creates a number with given numerator and denominator and then reduces it to lowest terms.""
        self.num = num
        self.denom = denom

#_________________ Explaining how to reduce with internal helper methods
_________________

    # Now get help to reduce the rational number to lowest terms
    self._reduce()  # It's an internal helper function/method

    def _reduce(self):
        """ Helper to reduce number to lowst terms.""
        divisor = self._gcd(self.num, self.denom)

        self.num = self.num // divisor
        self.denom = self.denom // divisor

    def _gcd(self, a, b):
        """ Euclid's algorithm to find the greatest common divisor of two numbers.""
        (a, b) = (max(a, b), min(a, b))
while (b > 0):
    (a, b) = (b, a % b)
return (a)

#_________ we've done the reduce(); now for the vanilla class methods

def numerator(self):
    """ Returns numerator.""
    return self.num

def denominator(self):
    """ Returns denominator. ""
    return self.denom

def __str__(self):
    """ Returns the string representation of the number.""
    return (str(self.num) + "/" + str(self.denom))

#__________________________________________________________
#___ at this point the class Rational is ready; you can test rational
#___ objects and print them _______
#
#
# You'll see that we'll have trouble using the built in arithmetic
# operators like +, >, etc
# because those operator do not know how to work with our newly defined
# Rational
# object. They were written for built-in types "int" and "float" etc.
#
# Try the examples we showed at the very top of this file. + and > will
# not work.

#$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
#
# We want to use '+' for our Rational objects. But Python complains that
# '+' is not defined
# for our Rational class. So we must "extend" the definition of '+' to
# handle our new
# objects. This is called OVERLOADING.
# We simply define a new method using the appropriate method name. For "+" we must use
# the name __add__(). But we also need to get the MEANING of the operation right, as we
# show in the table below


Type of operation                                      What it means
(please check these meanings!)                          ------------------------                     --------------------
addition                                               a/b + c/d =   (ad + bc)/bd
subtraction                                            a/b - c/d =   (ad - bc)/bd
multiplication                                         a/b * c/d =    ac / bd
division                                               a/b / c/d =    ad / bc


# So when you redefine (overload) the __add__() method, you will give it this new meaning
# which makes perfect sense for two rational operands

def __add__(self, other):
    # other refers to the "other" operand, not the one on which
    # the __add__() method
    # is called

    """Returns the sum of the two rational numbers self and other.""
    # self is the left-operand, and other is the right-operand

    newNum = self.num*other.denom + other.num*self.denom       # new numerator

    newDenom = self.denom * other.denom                        # new denominator

    return ( Rational(newNum, newDenom) )

#NOTE: When you test "+" make sure to say

    # print( rational1 + rational2 ) because we need the result to come via the print.
# Homework: implement subtraction, multiplication, division via overloading, as we did with add

#______ How to overload comparison methods ________________________________

...

We can compare integers and floats using: ==, <, >, >=, etc etc. For example, when Python encounters ">" it invokes the corresponding method in the float or int class, depending on which type it sees as your operands. These methods have two operands, and it works just as it did with "+

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
<td><strong>eq</strong></td>
</tr>
<tr>
<td>!=</td>
<td>not equals</td>
<td><strong>neq</strong></td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td><strong>lt</strong></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td><strong>le</strong></td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td><strong>gt</strong></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td><strong>ge</strong></td>
</tr>
</tbody>
</table>

Just as with "+", we must use the correct logic inside the method to get the right result for the Rational operands. But we have a small advantage here. Once we have written the logic for "==" and "<" we can define the rest in terms of these two.

Example of how to do "<"

-----------------------------

1/6 < 2/3 ? How can we tell?

Simply multiply across and compare: num1*denom2 < num2*denom1

Is: 1*3 < 6*2 ? If yes, then it's True. If no, then it's False.

... 

```python
def __lt__(self, other):
    """ Compares two rational numbers self and other, using <. """
    left = self.num*other.denom
```
Whenever objects are comparable, it is a good idea to include all the comparison methods as we have started to do. In this way, methods like "sort()" will be able to work with these new objects.

# How to implement the "==" method?

''' Equality is a bit different from the other comparisons. Why?

Many objects cannot be compared using <, <=, >, >= etc. For example consider the points (1,2) and (2,1) in the x-y plane.

How can you ask if (1,2) < (2,1) ? They cannot be ordered, and so cannot be compared.

BUT, ANY TWO OBJECTS can be compared for equality or inequality.

We cannot ask "oneHalf < "Hi There!" because they cannot be compared. SEMANTIC ERROR!

OneHalf is our Rational object. "Hi There!" is a Python String.

BUT we can ask "oneHalf != "Hi There!" (True, they are not the same)

When Python sees "==" or "!=" operators it picks out the "__eq()__" method

# This is how we can test equality with our Rational objects

def __eq__(self, other):  # always two operands are involved
    ''' Tests self and other for equality. '''
    if self is other:  # are the objects identical?
        return True  # Python's "is" operator returns True if self and other refer to
                       # EXACTLY the SAME OBJECT.
elif type(self) != type(other):  # do both object types not match?
    return False

else:  # they match; check if all their variables are equal
    return((self.num == other.num) and \
            (self.denom == other.denom))

# Now you can implement <=, >, >=, != in terms of the above < and == methods.

#_________ Implement a Savings Account using Classes__________________

# We'll use this (in the example following this one) to explain INHERITANCE/subclasses

#4.py

... Simple savings account --- needs owner's name, PIN, and balance.

method what it does
----------
----------
a = SavingsAccount(name,pin,balance = 0.0) returns a new acct with given name, pin, bal
a.deposit(amount) adds amount to balance
a.withdraw(amount) remove amount from balance
a.getbalance() return balance
a.getName() return name
a.getPin() return PIN number
a.computeInterest() compute interest on acct; deposit it
```python
__str__(a)
same as str(a); returns string representation
of the account

# You know what an instance variable is. What is a Class variable?

When you compute interest, you use the bank's "rate of return". Since ALL
accounts use the
same rate, you want a variable that ALL objects instantiated from this class
can see. It is not
private to an object. For clarity, we will use uppercase letters for Class
variables.

class SavingsAccount:
    """ This class represents a savings acct. with owner's name, PIN number
    and balance. """

    RATE = 0.02
    # This is the class variable; common to whole
    # Class.

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    def __str__(self):
        result = "Name:         " + self.name + "\n"
        result += "PIN:             " + self.pin + "\n"
        result += "Balance:      " + str(self.balance)
        return result

    def getBalance(self):
        return self.balance

    def getName(self):
        return self.name

    def getPin(self):
        return self.pin

    def deposit(self, amount):
        """Deposits the given amount and returns the new balance."""
```
self.balance += amount  # self.balance = self.balance + amount
return self.balance

def withdraw(self, amount):
    """Withdraws the given amount. Returns None if successful; error message if unsuccessful.""
    if (amount < 0):
        return "Amount must be >= 0"
    elif self.balance < amount:
        return "Insufficient funds"
    else:
        self.balance -= amount  # self.balance = self.balance - amount
        return None

def computeInterest(self):
    """Computes, deposits and returns the interest.""
    Interest = self.balance * SavingsAccount.RATE  # see how you access the class variable!!
    self.deposit(Interest)
    return Interest

    # General Rule: Use class variables only for symbolic constants or to keep data that is common to all objects in the class. For data that is owned by individual objects you must use instance variables instead.

#_____ Let's now make a simple Bank________________________________

''' Bank allows users to add new accounts, remove accounts, get existing accounts, and compute interest on all accounts.

Bank method what it does
------------------ --------
b = Bank() returns a Bank object
b.add(account)  adds given account to bank

b.remove(pin)  remove acct with this pin from bank; return acct.
    if pin does not exist, return None

b.get(pin)  if pin exists, return this acct; else return None

b.computeInterest()  compute interest on each acct; deposit it in that acct and return total interest.

__str__(b)  same as str(b). Returns a string represtentation of the bank (i.e., all the accounts)

#We'll use a dictionary; we won't have account ordering, to keep things simple.

class Bank:
    def __init__(self):
        self.accounts = { }

    def __str__(self):
        """ Returns the string representation of the entire bank.""
        return "\n".join(map(str, self.accounts.values( )))

    def add(self, account):
        """ Inserts an account using its PIN as a key.""
        self.accounts[account.getPin()] = account

    def remove(self, pin):
        return self.accounts.pop(pin, None)  # observe use of None

    def get(self, pin):
        return self.accounts.get(pin, None)  # observe use of None

    def computeInterest(self):
        """ Computes the interest for each account and returns the total.""
        total = 0.0
for account in self.accounts.values():
    total = total + account.computeInterest()

    return (total)

#________ Structuring classes with Inheritance and Polymorphism

#5.py

''' Techniques that OOD uses:

1. Data Encapsulation: Restrict how external users can manipulate an object's state by forcing them to use a set of method calls.

2. Inheritance: Allow a class to automatically reuse and extend the code of similar but more general classes.

3. Polymorphism: Allow several different classes to use the same general method names

#main()
Note: Python's syntax does not enforce data encapsulation. It is up to you as a programmer.

But inheritance and polymorphism are built into Python syntax

```python
# What is an example of "inheritance" in OOD?

_____ start figure

'''
Look at objects around us and see this structure:

physical object
  I
  I

living thing

inanimate object
  I
  I

(Wiki repeatedly messes up the rest of this tree structure because of how it handles spaces. So I'll explain the rest in text.)

On the extreme left branch write "Mammals" with a branch under it to its child "Cat", since Cats are Mammals.

On the second branch from the left, write "Insect" with a branch under it to its child "Ant", since Ants are Insects.

(remember how the __add__( ) method was used by different classes?)```
On the third branch from the left write "Stone" since a Stone is an inanimate object.

On the extreme right branch write "Asteroid" since an Asteroid is another type of inanimate object with properties different from stone, but has everything from its parent "Inanimate".

end figure

_______

At the top (physical object) is a class that has features common to everything below. Every physical object has a mass, for example.

Classes below have the features from above plus some additional features. For example, living thing has a mass and can grow and die.

Each class INHERITS attributes and behaviors from its ancestors. It extends these with additional attributes and behaviors.

In Python, the top-most class is the built-in OBJECT class. This is the most general.

```
# Let's use our savings account example to see how Inheritance works.

''' We'll add RESTRICTED savings accounts to the previous type. These new accounts look the same as the previous accounts, but they allow ONLY THREE withdrawals per month. The fourth withdrawal gives an error message. A counter is reset for each month.

In 6.py we will copy all of 4.py's contents, and then add the code for the restricted account subclass. It will inherit many things from its parent class SavingsAccount, and then extend the parent class with a restriction.

__Implement a Restricted Savings Account using previous Savings Account class______

# We'll use this to explain INHERITANCE/subclasses. Go to the tail end of the file to
# see the subclass

#6.py

...  

Simple savings account --- needs owner's name, PIN, and balance.

<table>
<thead>
<tr>
<th>method</th>
<th>what it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = SavingsAccount(name,pin,balance = 0.0)</td>
<td>returns a new acct with given name, pin, bal</td>
</tr>
<tr>
<td>a.deposit(amount)</td>
<td>adds amount to balance</td>
</tr>
<tr>
<td>a.withdraw(amount)</td>
<td>remove amount from balance</td>
</tr>
<tr>
<td>a.getbalance( )</td>
<td>return balance</td>
</tr>
<tr>
<td>a.getName( )</td>
<td>return name</td>
</tr>
<tr>
<td>a.getPin( )</td>
<td>return PIN number</td>
</tr>
<tr>
<td>a.computeInterest( )</td>
<td>compute interest on acct; deposit it</td>
</tr>
<tr>
<td><strong>str</strong>(a)</td>
<td>same as str(a); returns string representation</td>
</tr>
</tbody>
</table>

...  

# You know what an instance variable is. What is a Class variable?  

...  

When you compute interest, you use the bank's "rate of return". Since ALL accounts use the same rate, you want a variable that ALL objects instantiated from this class can see. It is not private to an object. For clarity, we will use uppercase letters for Class variables.
class SavingsAccount:
    """ This class represents a savings acct. with owner's name, PIN number and balance. """
    RATE = 0.02  # This is the class variable; common to whole class.

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    def __str__(self):
        result = "Name:         " + self.name + "\n"
        result += "PIN:              " + self.pin + "\n"
        result += "Balance:      " + str(self.balance)
        return result

    def getBalance(self):
        return self.balance

    def getName(self):
        return self.name

    def getPin(self):
        return self.pin

    def deposit(self, amount):
        """Deposits the given amount and returns the new balance."
        """
        self.balance += amount  # self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        """Withdraws the given amount. Returns None if successful; error message if unsuccessful."
        """
        if (amount < 0):
            return "Amount must be >= 0"
        elif self.balance < amount:
            return "Insufficient funds"
        else:
            self.balance -= amount  # self.balance = self.balance - amount
            return None

    def computeInterest(self):
        """Computes, deposits and returns the interest."
        """
        Interest = self.balance * SavingsAccount.RATE  # see how you
access class variable.

# class variables are visible both inside
# the class definition and to external
# users of the class

```python
    self.deposit(interest)
    return interest
```

# General Rule: Use class variables only for symbolic constants or to keep data that is common to all objects in the class. For data that is owned by individual objects you must use instance variables instead.

#_____ Let's now make a simple Bank

```python
''' Bank allows users to add new accounts, remove accounts, get existing accounts, and compute interest on all accounts.

Bank method
--------------
-----------
b = Bank( ) returns a Bank object
b.add(account) adds given account to bank
b.remove(pin) remove acct with this pin from bank; return acct.
    if pin does not exist, return None
b.get(pin) if pin exists, return this acct; else return None
b.computeInterest( ) compute interest on each acct; deposit it in that acct and return total interest.
__str__(b) same as str(b). Returns a string representation of
```
the bank (i.e., all the accounts)

# We'll use a dictionary; we won't have account ordering, to keep things simple.

class Bank:

    def __init__(self):
        self.accounts = {}

    def __str__(self):
        """ Returns the string representation of the entire bank."""
        return "\n".join(map(str, self.accounts.values()))

    def add(self, account):
        """ Inserts an account using its PIN as a key."""
        self.accounts[account.getPin()] = account

    def remove(self, pin):
        return self.accounts.pop(pin, None)  # observe use of None

    def get(self, pin):
        return self.accounts.get(pin, None)  # observe use of None

    def computeInterest(self):
        """ Computes the interest for each account and returns the total."""

        total = 0.0

        for account in self.accounts.values():
            total = total + account.computeInterest()

        return total

# Here is the restricted savings account class

class RestrictedSavingsAccount(SavingsAccount):  # Observe how we subclass it
    """This class represents a restricted savings account."""

    MAX_WITHDRAWALS = 3

    def __init__(self, name, pin, balance = 0.0):
        """ Same attributes as SavingsAccount but with a counter for withdrawals."""

        SavingsAccount.__init__(self, name, pin, balance)
self.counter = 0

def withdraw(self, amount):
    """Restricts number of withdrawals to MAX_WITHDRAWALS."""
    if self.counter == RestrictedSavingsAccount.MAX_WITHDRAWALS:
        return "No more withdrawals this month"
    else:
        message = SavingsAccount.withdraw(self, amount)
        if message == None:
            self.counter = self.counter + 1
            return message

def resetCounter(self):
    self.counter = 0

def main():
    account = RestrictedSavingsAccount("Barney","1003",500.00)
    print(account)
    account.getBalance( )
    for count in range(3):
        print(account.withdraw(100))
    print(account.withdraw(50))
    account.resetCounter( )
    print(account.withdraw(50))
main()}
From:
http://courses.cs.purdue.edu/ - Computer Science Courses

Permanent link:
http://courses.cs.purdue.edu/cs17700:fall15:week13_examples1

Last update: 2015/11/16 13:45