Goodwill Gesture Goes Awry

Claremont (AP): Seven rooms were damaged in a Harvey Mudd College dormitory Tuesday evening after a misguided attempt to cheer up sleep-deprived students. “We were approached by a group of three penguins who wanted to sing Christmas carols,” explained a witness. “They promised that it would help us study.” Instead, the raucous squawking of the untrained and untalented birds quickly led to a violent dispute between supporters and detractors. One student attempted to encase the singers in foam rubber, but a second set fire to the material in hopes of freeing the animals. The resulting explosion unleashed a conflagration that spread to North Dorm, where there was extensive damage. However, losses in North were estimated at only $35.47, due to the advanced age of the furniture there.
The David Huffman Story!

Huffman coding is one of the fundamental ideas that people in computer science and data communications are using all the time - Donald Knuth
(“s”, ("p", ("a", "m")))

Recursive definition of a binary tree...
A binary tree is:
1. Just a symbol (i.e. a “leaf”) or
2. A left subtree and a right subtree

("a", "m")

Shouldn’t that be the list
["a", "m"]?
You Try It!

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>0.40</td>
</tr>
<tr>
<td>a</td>
<td>0.20</td>
</tr>
<tr>
<td>r</td>
<td>0.15</td>
</tr>
<tr>
<td>v</td>
<td>0.15</td>
</tr>
<tr>
<td>e</td>
<td>0.06</td>
</tr>
<tr>
<td>y</td>
<td>0.04</td>
</tr>
</tbody>
</table>
Rocket Science!

```python
>>> fuelNeeded = 42.0/1000
>>> tank1 = 36.0/1000
>>> tank2 = 6.0/1000
>>> tank1 + tank2 >= fuelNeeded
```

True? False? Maybe? DEMO!
>>> from Rational import *
>>> fuelNeeded = Rational(42, 1000)
>>> tank1 = Rational(36, 1000)
>>> tank2 = Rational(6, 1000)
>>> tank1 + tank2 >= fuelNeeded
True

That would be so SWEET!

The Rational factory!
class Rational(object):
    def __init__(self, n, d):
        if d == 0:
            print("Invalid denominator!")
            sys.exit(1)  # import sys for this to work (ugly!)
        else:
            self.numerator = n
            self.denominator = d

>>> from Rational import *
>>> myNum1 = Rational(1, 3)
>>> myNum2 = Rational(2, 6)
>>> myNum1.numerator
numer = 1
denominator = 3

>>> myNum1.denominator
?  

>>> myNum2.numerator
numer = 2
denominator = 6

In a file called Rational.py

Why is this code so selfish?

Nothing is returned here!
from exceptions import ValueError

class Rational(object):
    def __init__(self, n, d):
        if d == 0:
            raise ValueError("Invalid denominator!")
        else:
            self.numerator = n
            self.denominator = d

    def isZero(self):
        return self.numerator == 0

>>> myNum1 = Rational(1, 3)
>>> myNum2 = Rational(0, 6)
>>> myNum1.isZero()
True
>>> myNum2.isZero()
True

This is so class-y!
Thinking Rationally

class Rational(object):
    def __init__(self, n, d):
        if d == 0:
            raise ValueError("Invalid denominator!")
        else:
            self.numerator = n
            self.denominator = d

    def isZero(self):
        return self.numerator == 0

>>> myNum1 = Rational(1, 3)
>>> myNum2 = myNum1
>>> myNum2.numerator = 42  # CHEATING! myNum1
>> myNum1
<Rational instance at 0xdb3918>
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def isZero(self):
        return self.numerator == 0

    def __str__(self):
        return str(self.numerator) + "/" + str(self.denominator)

>>> myNum = Rational(1, 3)
>>> myNum.__str__()
'1/3'
>>> myNum
<__main__.Rational object at 0x2b513566b7d0>
>>> print(myNum)
'1/3'
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def isZero(self):
        return self.numerator == 0

    def __repr__(self):
        return "Rational(\n            " + str(self.numerator) + ", " + str(self.denominator) + ")"

>>> myNum = Rational(1, 3)
>>> myNum._ _repr_ _()  
myNum    
Rational(1, 3)  
>>> myNum
Rational(1, 3)
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def isZero(self):
        return self.numerator == 0

# The lazy way to do both str and repr
def __repr__(self):
    return str(self.numerator) + "/" + str(self.denominator)

>>> myNum1 = Rational(1, 3)
>>> myNum2 = Rational(2, 6)
>>> print(myNum2)
2/6
>>> myNum1 == myNum2
False
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def isZero(self):
        return self.numerator == 0

    def __repr__(self):
        return str(self.numerator) + "/" + str(self.denominator)

    def equals(self, other):
        return self.numerator * other.denominator ==
        self.denominator * other.numerator

myNum1 = Rational(1, 3)
myNum2 = Rational(2, 6)
myNum1.equals(myNum2)
True
myNum2.equals(myNum2)
True

numerator = 1
denominator = 3
numerator = 2
denominator = 6
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def isZero(self):
        return self.numerator == 0

    def __repr__(self):
        return str(self.numerator) + "/" + str(self.denominator)

    def __eq__(self, other):
        return self.numerator * other.denominator ==
        self.denominator * other.numerator

>>> myNum1 = Rational(1, 3)
>>> myNum2 = Rational(2, 6)
>>> myNum1 == myNum2
True
>>> myNum2 == myNum1
True

This is what I would really like!
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def add(self, other):
        Start by assuming that the denominators are the same, but then try to do the case that they may be different!

>>> myNum1 = Rational(36, 1000)
>>> myNum2 = Rational(6, 1000)
>>> myNum3 = myNum1.add(myNum2)
>>> myNum3
42000/1000000

What kind of thing is add returning?
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def __add__(self, other):
        newDenominator = self.denominator * other.denominator
        newNumerator = self.numerator * other.denominator + \
                       other.numerator * self.denominator
        return Rational(newNumerator, newDenominator)

>>> myNum1 = Rational(36, 1000)
>>> myNum2 = Rational(6, 1000)
>>> myNum3 = myNum1 + myNum2
>>> myNum3
42/1000

This is what I would really, really like!
Overloaded Operator Naming

+ __add__
- __sub__
* __mul__
/ __div__
// __floordiv__
% __mod__
** __pow__

+ __pos__
- __neg__
__abs__
__int__
__float__
__complex__
== __eq__
!= __ne__
<= __le__
>= __ge__
< __lt__
> __gt__

def __int__(self):
    return self.numerator//self.denominator

>>> myNum = Rational(9, 2)
>>> myNum.int()
Barf!
>>> int(myNum)
4

Very __int__ ेresting!
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def __add__(self, other):
        newNumerator =
        newDenominator =
        return Rational(newNumerator, newDenominator)

    def __eq__(self, other):
        return ???

    def __ge__(self, other):
        return ???

    def __repr__(self):
        return str(self.numerator) + "/" + str(self.denominator)

>>> from Rational import *
>>> fuelNeeded = Rational(42, 1000)
>>> tank1 = Rational(36, 1000)
>>> tank2 = Rational(6, 1000)
>>> tank1 + tank2 >= fuelNeeded
True
Putting It All Together

class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def __add__(self, other):
        newNumerator = self.numerator*other.denominator + self.denominator*other.numerator
        newDenominator = self.denominator*other.denominator
        return Rational(newNumerator, newDenominator)

    def __eq__(self, other):
        return self.numerator*other.denominator == self.denominator*other.numerator

    def __ge__(self, other):
        return self.numerator*other.denominator >= self.denominator*other.numerator

    def __repr__(self):
        return str(self.numerator) + "/" + str(self.denominator)

>>> from Rational import *
>>> fuelNeeded = Rational(42, 1000)
>>> tank1 = Rational(36, 1000)
>>> tank2 = Rational(6, 1000)
>>> tank1 + tank2 >= fuelNeeded
True
Rationals Are Now “First Class” Citizens!

>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 4)
>>> r3 = Rational(1, 8)
>>> L = [r1, r2, r3]
```python
>>> assigned = Date(1, 24, 2013)
>>> due = Date(2, 1, 2013)
>>> due - assigned
8

>> if due > assigned:
    print("Go watch a movie!"")
```
class Date(object):
    def __init__(self, m, d, y):
        self.month = m
        self.day = d
        self.year = y

>>> d = Date(1, 21, 1969)
Another Implementation...

class Date(object):
    def __init__(self, m, d, y):
        self.daysSince1900 = ...

>>> d = Date(1, 21, 1969)

Why would any sane person want to store the date as the number of days since January 1, 1900?
class Date(object):
    def __init__(self, m, d, y):
        self._daysSince1900 = ...

    def setDay(self, d):
        if d <= 0 or d > 31:
            ...
        else:
            self._daysSince1900 = ...

>>> d = Date(1, 21, 1969)
>>> d.setDay(28)    # SETTER
>>> x = d.getDay()  # GETTER
>>> x = d.getDay()  # GETTER
Date “Abstraction”

Date

__init__(self, month, day, year)
setDay(self, day)
setMonth(self, month)
setYear(self, year)
getDay(self)
getMonth(self)
getYear(self)
==, >, <, >=, <=, +, -
The Advantage of Abstraction

import turtle
import math
import Date

turtle.forward(100)
print(math.cos(math.pi))
today = Date.Date(11, 9, 2011)
class Point(object):
    def __init__(self, inputX, inputY)
        self.x = inputX
        self.y = inputY
    def __str__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
    def __repr__(self):
        return "Point(" + str(self.x) + "," + str(self.y) + ")"
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

>>> P1 = Point(1.0, 2.0)
>>> P2 = Point(1.0, 2.0)
>>> P1
???
>>> P1 == P2
???
**repr vs. str**

class Silly(object):
    # No __init__ needed!
    def __str__(self):
        return "silly indeed"
    def __repr__(self):
        return "Silly()"

>>> s = Silly()
>>> s
Silly()
>>> print(s)
silly indeed
class Point(object):
    def __init__(self, inputX, inputY):
        self.x = inputX
        self.y = inputY
    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

class Line(object):
    def __init__(self, Point1, Point2):
        self.Point1 = Point1
        self.Point2 = Point2
        self.slope = (Point1.y - Point2.y) / (Point1.x - Point2.x)  # BUG!
        self.yIntercept = Point1.y - Point1.x*(Point2.y - Point1.y)/(Point2.x - Point1.x)
    def __repr__(self):
        return "y = " + str(self.slope) + " x + " + str(self.yIntercept)
    def __eq__(self, other):
        return self.slope == other.slope and self.yIntercept == other.yIntercept

>>> P1 = Point(1.0, 2.0)
>>> P2 = Point(2.0, 3.0)
>>> L1 = Line(P1, P2)
>>> L1
y = 1.0 x + 1.0
>>> P3 = Point(3.0, 4.0)
>>> P4 = Point(42.0, 43.0)
>>> L2 = Line(P3, P4)
>>> L1 == L2
True
Can you think of another way of writing this line?
class Person(object):
    def __init__(self, first, last):
        self.firstName = first
        self.lastName = last

    def asleep(self, time):
        return time > 20 or time < 4 # MILITARY TIME IN HOURS

    def __repr__(self):
        return self.firstName + " " + self.lastName

>>> julie = Person("Julie", "Medero")
>>> julie
Julie Medero
>>> julie(2)
True
class Person(object):
    def __init__(self, first, last):
        self.firstName = first
        self.lastName = last

    def asleep(self, time):
        return time > 20 or time < 4  # MILITARY TIME IN HOURS

    def __repr__(self):
        return self.firstName + " " + self.lastName

class Student(Person):
    def __init__(self, first, last, age):
        super(Student, self).__init__(first, last)
        self.age = age

    def asleep(self, time):
        return 3 <= time <= 11

    def __repr__(self):
        return Person.__repr__(self) + ", " + str(self.age) + " years old"

>>> s = Student("Sue", "Persmart", 18)
>>> s
Sue Persmart, 18 years old
>>> s.asleep(2)
False
class Person(object):
    def __init__(self, first, last):
        self.firstName = first
        self.lastName = last

    def asleep(self, time):
        return time > 20 or time < 4 # MILITARY TIME IN HOURS

    def __repr__(self):
        return self.firstName + " " + self.lastName

class Student(Person):
    def __init__(self, first, last, age):
        super(Student, self).__init__(first, last)
        self.age = age

    def asleep(self, time):
        return 3 <= time <= 11

    def __repr__(self):
        return Person.__repr__(self) + ", " + str(self.age) + " years old"

class Mudder(Student):
    def __init__(self, first, last, age, dorm):
        super(Mudder, self).__init__(first, last, age)
        self.dorm = dorm

    def asleep(self, time):
        return False

wally = Mudder("wally", "wart", 42, "west")
wally
wally.asleep()
Default Arguments

In my experience, arguments are usually default of deperson who started them!

class Student(object):
    def __init__(self, firstName, lastName,
                  school = "HMC", major = "undeclared")

>>> where = Student("Carmen", "Sandiego")
>>> stu = Student("Stu", "Dious", "PIT")
>>> anna = Student("Anna", "Litik", major = "Physics")
>>> elmo = Student("Elmo")
>>> bigBird = Student("Big", "Bird", firstName = "Tweety")
>>> bart = Student(school="PIT", "Bart", "Simpson")
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

def magnitude(self):
    blah, blah, blah
    return ...

def normalize(self):
    mag = self.magnitude()
    newVector = Vector(
        self.x/mag, self.y/mag)
    self = newVector

Python Forbids Personality Transplants!