Today: Object Oriented Programs (OOPs)
A little reflection: the semester in perspective

• Basic elements of programming:
  • Functions
  • Loops
  • Conditionals
• Solving ‘hard’ problems with recursion
• A look under the hood: how recursion works at the level of the machine
Surgeon General’s Warning

• This lecture contains some syntactic details. Don’t memorize them! Concentrate on the big ideas and refer to these slides later for syntax details.
Rocket Science!

The CS 5 Gold/Black “textbook” is now linked from the HW 10 entry on the course website. Read Chapter 6: “OOPs! Object-Oriented Programming”

```python
>>> fuelNeeded = 42/1000
>>> tank1 = 36/1000
>>> tank2 = 6/1000
>>> tank1 + tank2 >= fuelNeeded
```

True? False? Maybe? What’s the right ants-er?
Wishful Thinking...

```python
>>> from Rational import *
>>> fuelNeeded = Rational(42, 1000)
>>> tank1 = Rational(36, 1000)
>>> tank2 = Rational(6, 1000)
>>> tank1 + tank2 >= fuelNeeded
True
```

We need an ant-i-dote for this problem!

The Rational factory!
Thinking Rationally

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

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

In a file called Rational.py

Notice that nothing is returned!

The “constructor”

Why is this code so selfish?

This “dot” notation is vaguely familiar!
class Rational:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

from Rational import *
myNum1 = Rational(1, 3)
myNum2 = Rational(1, 3)
myNum1 == myNum2

In a file called Rational.py

myNum1 →
numerator = 1
denominator = 3

myNum2 →
numerator = 1
denominator = 3

Rational numbers go back to the days of antiquity!
class Rational:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

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

>>> myNum1 = Rational(1, 3)
>>> myNum2 = Rational(0, 6)
>>> myNum1.isZero()
False
>>> myNum2.isZero()
True
class Rational:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

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

>>> myNum1 = Rational(1, 3)
>>> myNum2 = myNum1
>>> myNum2.numerator = 42
Thinking Rationally

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

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

>>> myNum = Rational(1, 3)
>>> myNum
<Rational instance at 0xdb3918>

__init__ally I thought this was weird, but now I like it!
class Rational:
    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)

>>> myNum = Rational(1, 3)
>>> myNum.__repr__()
class Rational:
    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)

>>> myNum = Rational(1, 3)
>>> myNum.__repr__() 
1/3
>>> myNum
1/3
class Rational:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

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

    def __repr__(self):
        return f"Numerator {str(self.numerator)} and Denominator {str(self.denominator)}"

>>> myNum = Rational(1, 3)
>>> myNum
Numerator 1 and Denominator 3
class Rational:
    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)

numerator = 1
denominator = 3
myNum1 = Rational(1, 3)
myNum2 = Rational(2, 6)
myNum1 == myNum2
False
Thinking Rationally

class Rational:
    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):

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

numerator = 2
denominator = 6

myNum1
numerator = 1
denominator = 3

myNum2
numerator = 2
denominator = 6

Working at cross purposes?

In your notes
class Rational:
    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

numerator = 1
denominator = 3
myNum1 = Rational(1, 3)
myNum2 = Rational(2, 6)

>>> myNum1.equals(myNum2)
True

>>> myNum2.equals(myNum2)
True
class Rational:
    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.__eq__(myNum2)
True
>>> myNum1 == myNum2
True
class Rational:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def __add__(self, other):
        numerator = 36
        denominator = 1000
        myNum1 = Rational(numerator, denominator)
        myNum2 = Rational(6, 1000)
        myNum3 = myNum1.__add__(myNum2)
        myNum3
        42/1000

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

myNum1
numerator = 36
denominator = 1000

myNum2
numerator = 6
denominator = 1000

Worksheet

What kind of thing is add returning?

Start by assuming that the denominators are the same, but then try to do the case that they may be different!
Thinking Rationally

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

def __add__(self, other):
    num = self.numerator + other.numerator
    den = self.denominator  # assuming same denominators!
    sum = Rational(num, den)
    return sum

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

Worksheet

numerator = 36
denominator = 1000

numerator = 6
denominator = 1000
class Rational:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def __add__(self, other):
        num = self.numerator + other.numerator
        den = self.denominator # assuming same denominators!
        return Rational(num, den)

numerator = 36
denominator = 1000
myNum1 = Rational(36, 1000)
myNum2 = Rational(6, 1000)
myNum3 = myNum1.__add__(myNum2)
myNum3
42/1000

myNum1 + myNum2
Overloaded Operator Naming

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

+  __pos__
-  __neg__
  __abs__
  __int__
  __float__
  __complex__

==  __eq__
!=  __ne__
<=  __le__
>=  __ge__
<  __lt__
>  __gt__

That’s the ant-ire list!
Putting it all Together

class Rational:
    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 *  # (necessary?)

>>> fuelNeeded = Rational(42, 1000)

>>> tank1 = Rational(36, 1000)

>>> tank2 = Rational(6, 1000)

>>> tank1 + tank2 >= fuelNeeded
True
Rationals are now “first class” citizens

from Rational import *

def initely():
    r1 = Rational(1, 2)
    r2 = Rational(21, 42)
    r3 = Rational(1, 42)
    myList = [r1, r2, r3]
    r4 = Rational(0, 1)
    for r in myList:
        r4 = r4 + r
    return r

This is beyond awesome!

That’s cooler than Ant-arctica!
What’s the Point?

Without this example, the lecture would be Point-less!

```python
class Point:
    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):
```

```python
>>> P1 = Point(1.0, 2.0)
>>> P2 = Point(1.0, 2.0)
>>> P1
???
>>> P1 == P2
True
```
class Point:
    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

>>> P1 = Point(1.0, 2.0)
>>> P2 = Point(1.0, 2.0)
>>> P1
???
>>> P1 == P2
True

Without this example, the lecture would be Point-less!
Thinking Linearly

class Point:
    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:
    def __init__(self, Point1, Point2):
        self.Point1 = Point1
        self.Point2 = Point2
        self.slope = (Point1.y - Point2.y) / (Point1.x - Point2.x)
        self.y_intercept = Point1.y - Point1.x*(Point2.y - Point1.y)/(Point2.x - Point1.x)
    def __repr__(self):
        return "y = " + str(self.slope) + "x + " + str(self.y_intercept)
    def __eq__(self, other):
        return self.slope == other.slope and self.y_intercept == other.y_intercept

>>> 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
Thinking Linearly

class Point:
    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:
    def __init__(self, Point1, Point2):
        self.Point1 = Point1
        self.Point2 = Point2
        self.slope = (Point1.y - Point2.y) / (Point1.x - Point2.x)
        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
```python
>>> from Point import *
>>> p1 = Point(0, 1)
>>> p2 = Point(1, 2)
>>> L1 = Line(p1, p2)
>>> p3 = Point(2, 0)
>>> p4 = Point(0, 2)
>>> L2 = Line(p3, p4)
>>> L1.parallel(L2)
False
>>> L1.intersection(L2)
(0.5, 1.5)
```

```python
class Point:
    def __init__(self, InputX, InputY):
        self.x = 1.0*InputX
        self.y = 1.0*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:
    def __init__(self, Point1, Point2):
        self.Point1 = Point1
        self.Point2 = Point2
        self.slope = (Point1.y - Point2.y) / (Point1.x - Point2.x)
        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
    def parallel(self, other):
        
    def intersection(self, other):
        
```
```python
>>> from Point import *
>>> p1 = Point(0, 1)
>>> p2 = Point(1, 2)
>>> L1 = Line(p1, p2)
>>> p3 = Point(2, 0)
>>> p4 = Point(0, 2)
>>> L2 = Line(p3, p4)
>>> L1.parallel(L2)
False
>>> L1.intersection(L2)
(0.5, 1.5)
```

```python
class Point:
    def __init__(self, InputX, InputY):
        self.x = 1.0*InputX
        self.y = 1.0*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:
    def __init__(self, Point1, Point2):
        self.Point1 = Point1
        self.Point2 = Point2
        self.slope = (Point1.y - Point2.y) / (Point1.x - Point2.x)
        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

    def parallel(self, other):
        return self.slope == other.slope

    def intersection(self, other):
        if self.parallel(other): return None
        else:
            x = (self.yintercept - other.yintercept)/(other.slope - self.slope)
            y = self.slope * x + self.yintercept
        return Point(x, y)
```
Vector, Victor!

class Vector:
    def __init__(self, x, y):
    def __repr__(self):
    def __add__(self, other):
    def __sub__(self, other):
    def magnitude(self):
    def normalize(self):

In a file called Vector.py

>>> victor = Vector(1, 1)
>>> victor
(1, 1)
>>> roger = Vector(0, 2)
>>> roger
(0, 2)
>>> A = victor + roger
>>> A
(1, 3)
>>> victor.magnitude()
1.4142135
An Ant Class

```python
from Vector import *

class Ant:
    def __init__(self, pos):
        self.position = pos

    def moveTowards(self, other):
        AbesPosition = Vector(0, 0)
        Abe = Ant(AbesPosition)
        Bess = Ant(Vector(100, 0))
        Abe.moveTowards(Bess)
```

Hey, draw me a picture of Abe and Bess!

I’m feeling strong ant-ipathy for ant puns!

(100, 0)!? That’s practically in the Dutch Ant-illes!
Abe = Ant(Vector(0, 0))
Bess = Ant(Vector(0, 100))
Cziggy = Ant(Vector(100, 100))
Dizzy = Ant(Vector(100, 0))

while True:
    Abe.moveTowards(Bess)
    Bess.moveTowards(Cziggy)
    Cziggy.moveTowards(Dizzy)
    Dizzy.moveTowards(Abe)

Ugh! What if there were 1000 ants, or even some variable n number of ants!
from Rational import *

def initely():
    r1 = Rational(1, 2)
    r2 = Rational(21, 42)
    r3 = Rational(1, 4)
    myList = [r1, r2, r3]
    r4 = Rational(0, 1)
    for r in myList:
        r4 = r4 + r
    return r

This is beyond awesome!

That’s cooler than Ant-arctica!
Abe = Ant(Vector(0, 0))
Bess = Ant(Vector(0, 100))
Cziggy = Ant(Vector(100, 100))
Dizzy = Ant(Vector(100, 0))
Ants = [Abe, Bess, Cziggy, Dizzy]

while True:
    for i in range(len(Ants)):
Abe = Ant(Vector(0, 0))
Bess = Ant(Vector(0, 100))
Cziggy = Ant(Vector(100, 100))
Dizzy = Ant(Vector(100, 0))
Ants = [Abe, Bess, Cziggy, Dizzy]

while True:
    for i in range(len(Ants)):
        Ants[i].moveTowards(Ants[(i+1) % len(Ants)])
In Python, everything is a class!

```python
>>> x = 5
>>> y = 37
>>> x + y
42
```

```python
class int:
    def __init__(self, str):
```

```python
>>> x = int("5")
>>> y = int("37")
>>> x.__add__(37)
42
>>> x + y
42
```

```python
class int:
    def __init__(self, str):
```
In Python, everything is a class!

```python
class list:
    def __init__(self):
    def __repr__(self):
    def __append__(self, item):
    def __add__(self, other):
    def __getitem__(self, index):
    def __setitem__(self, index, value):

>>> x = []
>>> x.append(42)
>>> x
[42]
>>> x[0]
42
>>> x[0] = 67

>>> x = list()
>>> x.append(42)
>>> x.__repr__()
[42]
>>> x.__getitem__(0)
42
>>> x.__setitem__(0, 67)
```