The end is near!
True Story
(apropos Markov Text Generation!)

Dan Aguayo
Max Krohn
Jeremy Stribling
Rooter: A Methodology for the Typical Unification of Access Points and Redundancy

Jeremy Stribling, Daniel Aguayo and Maxwell Krohn

Abstract

Many physicists would agree that, had it not been for congestion control, the evaluation of web browsers might never have occurred. In fact, few hackers worldwide would disagree with the essential unification of voice-over-IP and public-private key pair. In order to solve this riddle, we confirm that SMPs can be made stochastic, cacheable, and interposable.

I. Introduction

Many scholars would agree that, had it not been for active networks, the simulation of Lamport clocks might never have occurred. The notion that end-users synchronize with the

The rest of this paper is organized as follows. For starters, we motivate the need for fiber-optic cables. We place our work in context with the prior work in this area. To address this obstacle, we disprove that even though the much-touted autonomous algorithm for the construction of digital-to-analog converters by Jones [10] is NP-complete, object-oriented languages can be made signed, decentralized, and signed. Along these same lines, to accomplish this mission, we concentrate our efforts on showing that the famous ubiquitous algorithm for the exploration of robots by Sato et al. runs in \( \Omega((n + \log n)) \) time [22]. In the end, we conclude.

II. Architecture

Check out… http://pdos.csail.mit.edu/scigen/#about
Learning Goals

• Describe inheritance and its benefits
Inheritance (inheriting)

class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __repr__(self):
        return self.first_name + " " + self.last_name

    def is_asleep(self, time):
        return 0 <= time <= 7  # using 24hr time format

>>> jessica = Person("Jessica", "Wu")
>>> jessica
Jessica Wu
>>> jessica.is_asleep(2)
True
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __repr__(self):
        return self.first_name + ' ' + self.last_name

    def is_asleep(self, time):
        return 0 <= time <= 7

class Student(Person):
    def __init__(self, first_name, last_name, age):
        Person.__init__(self, first_name, last_name)  # call base class constructor
        self.age = age

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

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

>>> s = Student("Sue", "Persmart", 18)
>>> s
Sue Persmart, 18 years old
>>> s.is_asleep(2)
False

Inheritance!
Read as: Student “is a” Person
Person is the base class.
Student is the derived class.
Student inherits all attributes and methods from Person.

Function Overriding!
Sleeping until 11 AM!?
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __repr__(self):
        return self.first_name + " " + self.last_name

    def is_asleep(self, time):
        return 0 <= time <= 7

class Student(Person):
    def __init__(self, first_name, last_name, age):
        Person.__init__(self, first_name, last_name)
        self.age = age

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

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

class Mudder(Student):
    def __init__(self, first_name, last_name, age, dorm):
        Student.__init__(self, first_name, last_name, age)
        self.dorm = dorm

    def is_asleep(self, time):
        return False

>>> wally = Mudder("wally", "wart", 42, "west")

>>> wally

>>> wally.is_asleep(10)

Get some sleep!!!
Objects are self-ish

class Person:
...

```python
def rename(self, new_first_name, new_last_name):
    self.first_name = new_first_name
    self.last_name = new_last_name
```

```python
def rename(self, new_first_name, new_last_name):
    self = Person(new_first_name, new_last_name)
```

```python
def rename(self, new_first_name, new_last_name):
    self = Person(new_first_name, new_last_name)
```
>>> p = polite()
Thank you for using polite lists!
>>> p.append(42)
>>> p.append(50)
>>> p[0] += 1
>>> p
This polite list contains [43, 50]
>>> p[3] = 100
Pardon me, but your index is out of bounds
>>> p[3]
Pardon me, but your index is out of bounds
```python
class polite(list):
    def __init__(self):
        print("Thank you for using polite lists!")

    def __repr__(self):
        return "This polite list contains " + list.__repr__(self)

    def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)

    def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)
```

We are inheriting from the `list` class!
class polite(list):
    def __init__(self):
        print("Thank you for using polite lists!")
    def __repr__(self):
        return "This polite list contains " + list.__repr__(self)
    def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)
    def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)

What does this do?
def __add__(self, other):
    output = polite()
    for x in self:
        output.append(x)
    if isinstance(other, int):
        output.append(other)
    elif isinstance(other, list):
        output.extend(other)
    else:
        print("Pardon me, but your other is confusing")
    return output

>>> L = polite()
Thank you for using polite lists!
>>> L.append(42)
>>> L.append(47)
>>> L
[42, 47]
>>> M = L + 5
>>> M
[42, 47, 5]
>>> N = M + [50]
>>> N
[42, 47, 5, 50]
class polite(list):
    def __init__(self):
        print("Thank you for using polite lists!")
    def __repr__(self):
        return "This polite list contains " + list.__repr__(self)
    def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)
    def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)

What does this do?
def __add__(self, other):
    output = polite()
    for x in self:
        output.append(x)
    if isinstance(other, int):
        output.append(other)
    elif isinstance(other, list):
        output.extend(other)
    else:
        print("Pardon me, but your other is confusing")
    return output

>>> L = polite()
Thank you for using polite lists!
>>> L.append(42)
>>> L.append(47)
>>> L
This polite list contains [42, 47]
>>> M = L + ...
This polite list contains [42, 47, 5]
>>> N = M + [50]
Thank you for using polite lists!
>>> N
This polite list contains [42, 47, 5, 50]
class polite(list):
    def __init__(self):
        print("Thank you for using polite lists!")
    def __repr__(self):
        return "This polite list contains " + list.__repr__(self)
    def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)
    def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)

Complete the function

def __mul__(self, other):

>>> L = polite()
Thank you for using polite lists!
>>> L.append(42)
>>> L.append(47)
>>> L.append(23)
>>> M = polite()
Thank you for using polite lists!
>>> M.append(5)
>>> M.append(6)
>>> N = L * M  # same as N = L.__mul__(M)
Thank you for using polite lists!
>>> N
This polite list contains [(42, 5), (42, 6), (47, 5), (47, 6), (23, 5), (23, 6)]
class polite(list):
    def __init__(self):
        print("Thank you for using polite lists!")
    def __repr__(self):
        return "This polite list contains " + list.__repr__(self)
    def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)
    def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)

Complete the function
def __mul__(self, other):
    output = polite()
    for x in self:
        for y in other:
            output.append((x, y))
    return output

>>> L = polite()
Thank you for using polite lists!
>>> L.append(42)
>>> L.append(47)
>>> L.append(23)
>>> M = polite()
Thank you for using polite lists!
>>> M.append(5)
>>> M.append(6)
>>> N = L * M  # same as N = L.__mul__(M)
Thank you for using polite lists!
>>> N
This polite list contains [(42, 5), (42, 6), (47, 5), (47, 6), (23, 5), (23, 6)]
class polite(list):
    def __init__(self):
        print("Thank you for using polite lists!")
    def __repr__(self):
        return "This list contains " + list.__repr__(self)
    def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)
    def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)

>>> L = polite([1, 2, 3, 4])
Thank you for using polite lists!
>>> L
This polite list contains [1, 2, 3, 4]
>>> 3 in L
True

What needs to be done to do this?
class polite(list):
    def __init__(self):
        print("Thank you for using polite lists!")
    def __repr__(self):
        return "This list contains " + list.__repr__(self)
    def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)
    def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)

What needs to be done to do this?

def __init__(self, lst):
    print("Thank you for using polite lists!")
    list.__init__(self, lst)

# nothing needs to be done
# polite is a list
# list knows in (__contains__)
Default Arguments

class Person:
    def __init__(self, ...):
    def __repr__(self):
    def is_asleep(self):

class Student(Person):
    def __init__(self,
        first_name,
        last_name,
        school = "Claremont Colleges",
        major = "undeclared"):
        self.first_name = first_name
        self.last_name = last_name
        self.college = school
        self.major = major

>>> wally = Student("Wally", "Wart", "Harvey Mudd")
>>> cecil = Student("Cecil", "Sagehen", "Pomona", "Biology and Mathematics")
>>> la_semeuse = Student("La", "Semeuse", school="Scripps")
>>> sammy = Student("Sammy", "the Owl", major="Engineering")
>>> elmo = Student("Elmo")

In my experience, arguments are usually default of deperson who started dem!
Learning Goals

• Describe inheritance of shapes
• Describe matrix operations on shapes
Past Pretty Pictures

Alistair Dobke

Jennifer Phuong

Kevin Vigue

Alice Zhang
Millisoft “Shapes”
(Now in HW 11!)

```python
>>> r = Rectangle(100, 50, center=Vector(200, 0), color="blue")
>>> c = Circle(radius=30, color="red") # default center (0,0)
>>> r.rotate(45) # 45 degree ccw rotation about origin
>>> r.render() # draw it!
>>> c.render() # draw it!
```

This material is shared with CS 5 “Black”
from shapes import *

def demo():
    my_rect = Rectangle(130, 50, center=Vector(0, 50), color="red")
    eye0 = Circle(Vector(-80, 100), 15, color="blue")
    eye1 = Circle(Vector(-40, 100), 15, color="blue")
    eye2 = Circle(Vector(0, 100), 15, color="blue")
    eye3 = Circle(Vector(40, 100), 15, color="blue")
    eye4 = Circle(Vector(80, 100), 15, color="blue")
    parts = [my_rect, eye0, eye1, eye2, eye3, eye4]
    for part in parts:
        part.render()
```python
class Shape:
    def __init__(self):
        self.points = []  # list of Vectors!
    def render(self):  # code here...
    def erase(self):  # code here...
    def rotate(self, theta):  # code here...
    def scale(self, stretch):  # code here...
    def translate(self, shift):  # code here..

class Rectangle(Shape):

class Circle(Shape):

>>> r = Rectangle(100, 50, center=Vector(200, 0), color="blue")
>>> c = Circle(radius=30, color="red")  # default center (0,0)
>>> r.rotate(45)  # 45 degree ccw rotation about origin
>>> r.render()    # draw it!
>>> c.render()    # draw it!
```

I don’t plan to ever call the `Shape` factory to make a `Shape`. But I will call the `Rectangle` and `Circle` factories!

an “abstract” class

Inheriting from `Shape` means we don’t have to reinvent the wheel!
class Shape:
    def __init__(self):
        self.points = []  # list of Vectors!

    def render(self):
        code here...
    def erase(self):
        code here...
    def rotate(self, theta):
        code here...
    def scale(self, stretch):
        code here...
    def translate(self, shift):
        code here..

class Rectangle(Shape):
    def __init__(self, width, height,
                     center=Vector(0, 0), color="black"):
        SW = Vector(center.x - width/2.0, center.y - height/2.0)
        NW = Vector(center.x - width/2.0, center.y + height/2.0)
        NE = Vector(center.x + width/2.0, center.y + height/2.0)
        SE = Vector(center.x + width/2.0, center.y - height/2.0)
        self.points = [SW, NW, NE, SE]
        self.color = color

class Square(Rectangle):
    def __init__(self, width,
                     center=Vector(0, 0), color="black"):
class Shape:
    def __init__(self):
        self.points = []  # list of Vectors!
    def render(self):  code here...
    def erase(self):  code here...
    def rotate(self, theta):  code here...
    def scale(self, stretch):  code here...
    def translate(self, shift):  code here..

class Rectangle(Shape):
    def __init__(self, width, height,
                 center=Vector(0, 0), color="black"):
        SW = Vector(center.x - width/2.0, center.y - height/2.0)
        NW = Vector(center.x - width/2.0, center.y + height/2.0)
        NE = Vector(center.x + width/2.0, center.y + height/2.0)
        SE = Vector(center.x + width/2.0, center.y - height/2.0)
        self.points = [SW, NW, NE, SE]
        self.color = color

class Square(Rectangle):
    def __init__(self, width,
                 center=Vector(0, 0), color="black"):
        Rectangle.__init__(self, width, width, center, color)
import math
import turtle
from vector import *
from matrix import *

class Shape:
    def __init__(self):
        self.points = []  # list of Vectors!

    def render(self):
    def erase(self):
    def rotate(self, theta):
    def scale(self, stretch):
    def translate(self, shift):

class Rectangle(Shape):

class Square(Rectangle):

class Circle(Shape):

Specific shapes inherit from Shape!

How does render work?
First a few words about matrices…

Matrix-vector multiplication…

\[
\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}
\]

Matrix-Matrix multiplication…

\[
\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \cdot \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} = \begin{bmatrix} a_1 a_2 + b_1 c_2 & a_1 b_2 + b_1 d_2 \\ c_1 a_2 + d_1 c_2 & c_1 b_2 + d_1 d_2 \end{bmatrix}
\]
class Shape:
    def __init__(self):
        self.points = []  # list of Vectors!

    def render(self): code...
    def erase(self): code...
    def rotate(self, theta): code...
    def scale(self, stretch): code...
    def translate(self, shift): code...

First, let us rotate a single point about the origin. We will extend in homework...

\[
R = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}
\]

\[
\begin{bmatrix}
x' \\
y'
\end{bmatrix} = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix}
x \\
y
\end{bmatrix}
\]
A Matrix Class

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return ...
    def magnitude(self):
        blah, blah, blah
        return ...
    def normalize(self):
        blah, blah, blah
    def __neg__(self):
        return Vector(blah, blah)
    def __add__(self, other):
        blah, blah, blah
        return Vector(blah, blah)
    def __sub__(self, other):
        blah, blah, blah
        return Vector(blah, blah)

>>> M = Matrix(1, 2, 3, 4)
>>> M
1 2
3 4
>>> M.get(0, 1)
2
>>> v = Vector(1, 20)
>>> v
(1, 20)
>>> w = M * v  # w = M.__mul__(v)
>>> w
???
```

I thought that Linear Algebra was the Matrix Class!
Matrix Class

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return ...
    def magnitude(self):
        blah, blah, blah
    def normalize(self):
        blah, blah, blah
    def __neg__(self):
        return Vector(blah, blah)
    def __add__(self, other):
        blah, blah, blah
    def __sub__(self, other):
        blah, blah, blah

def Matrix:
    """2x2 Matrix class""
    def __init__(self, a11=0, a12=0, a21=0, a22=0):
        self.array = [[a11, a12], [a21, a22]]
    def __repr__(self):
        blah, blah, blah
    def get(self, row, column):
        return self.array[row][col]
    def set(self, row, column, value):
        self.array[row][col] = value
    def __add__(self, other):
        return ...
    def __mul__(self, other):
        return ...

>>> M = Matrix(1, 2, 3, 4)
>>> N = Matrix(1, 1, 2, 2)
>>> v = Vector(1, 3)
>>> P = M * N  # M.__mul__(N)
>>> q = M * v  # M.__mul__(v)
```

Notice that we store the matrix in a “funny” way. Why?!
def __mul__(self, other):
    """If other is a Matrix, returns a Matrix X*Y. If other is a Vector, returns a Vector X*y."""

    if isinstance(other, Matrix):
        result = Matrix()
        for row in range(0, 2):
            for col in range(0, 2):
                # compute result matrix in the given row and col
                entry = 0
                for i in range(0, 2):
                    entry += ______________________________
                result.set(row, col, entry)
        return result

    elif isinstance(other, Vector):
        x = ____________________________________
        y = ____________________________________
        return ______________

    else:
        print("Can't multiply a matrix by a ",
              other.__class__.__name__, "!!!") # !!! is a nice touch

Fill this in in your notes

Get the matrix element at row 0 and column 0 with self.get(0, 0)
Get the x-coordinate of a Vector v with v.x
def __mul__(self, other):
    """If other is a Matrix, returns a Matrix X*Y.
    If other is a Vector, returns a Vector X*y.""

    if isinstance(other, Matrix):
        result = Matrix()
        for row in range(0, 2):
            for col in range(0, 2):
                # compute result matrix in the given row and col
                entry = 0
                for i in range(0, 2):
                    entry += self.get(row, i) * other.get(i, col)
                result.set(row, col, entry)
        return result

    elif isinstance(other, Vector):
        x = self.get(0, 0) * other.x + self.get(0, 1) * other.y
        y = self.get(1, 0) * other.x + self.get(1, 1) * other.y
        return Vector(x, y)

    else:
        print("Can't multiply a matrix by a ",
              other.__class__.__name__, ", "!!!")  # !!! is a nice touch

Fill this in in your notes

Get the matrix element at row 0 and column 0 with self.get(0, 0)
Get the x-coordinate of a Vector v with v.x
import math  # has math.cos(angle) and math.sin(angle)

class Shape:
    def __init__(self):
        self.points = []
    def render(self):
        blah blah blah
    def rotate(self, theta):
        """Rotate shape by theta degrees""
        # Python's trig functions expect input in radians,
        # so this function converts from degrees into radians.
        theta = math.radians(theta)

class Rectangle(Shape):
    def __init__(self, width, height, center=Vector(0, 0), color="black"):
        SW = Vector(center.x - width/2.0, center.y - height/2.0)
        NW = Vector(center.x - width/2.0, center.y + height/2.0)
        NE = Vector(center.x + width/2.0, center.y + height/2.0)
        SE = Vector(center.x + width/2.0, center.y - height/2.0)
        self.points = [SW, NW, NE, SE]
        self.color = color

class Square(Rectangle):
    def __init__(self, width, center=Vector(0, 0), color="black"):
        Rectangle.__init__(self, width, width, center, color)
import math  # has math.cos(angle) and math.sin(angle)

class Shape:
    def __init__(self):
        self.points = []
    def render(self):
        blah blah blah
    def rotate(self, theta):
        """Rotate shape by theta degrees""

        # Python's trig functions expect input in radians,
        # so this function converts from degrees into radians.
        theta = math.radians(theta)
        rotation_matrix = Matrix(
            math.cos(theta), -math.sin(theta),
            math.sin(theta), math.cos(theta))
        
        new_points = []
        for vector in self.points:
            new_vector = rotation_matrix * vector
            new_points.append(new_vector)
        self.points = new_points

class Rectangle(Shape):
    def __init__(self, width, height, center=Vector(0, 0), color="black"):  
        SW = Vector(center.x - width/2.0, center.y - height/2.0)
        NW = Vector(center.x - width/2.0, center.y + height/2.0)
        NE = Vector(center.x + width/2.0, center.y + height/2.0)
        SE = Vector(center.x + width/2.0, center.y - height/2.0)
        self.points = [SW, NW, NE, SE]
        self.color = color

class Square(Rectangle):
    def __init__(self, width, center=Vector(0, 0), color="black"):  
        Rectangle.__init__(self, width, width, center, color)
What does this do?

def demo1():
    r = Square(50, Vector(0, 100), color="blue")
    for i in range(6):
        for j in range(6):
            r.render()
            r.rotate(60, r.center)
            r.rotate(60, Vector(0, 0))
            rotate about this vector

Demo shapes_demo.py!
class Shape:
    def __init__(self):  
        self.points = []  # list of Vectors!

    def render(self):     code...
    def erase(self):      code...
    def rotate(self, theta):  code...

    def scale(self, stretch):   code...
    def translate(self, shift):   code...
Circles...

class Shape:
    def __init__(self):
        self.points = []
    def render(self):
        code here...
    def rotate(self, theta):
        code here...

class Circle(Shape):
    def __init__(self, center=Vector(0,0), radius=10, color="black"):
        self.center = center
        self.radius = radius
        self.color = color

The turtle draws circles using
turtle.circle(radius)

Should we still use the Shape class render, rotate, translate, etc?
def demo2():
    r = Square(40, Vector(0, 100), color="blue")
    for i in range(6):
        for j in range(6):
            r.render()
            r.rotate(60, r.center)
            r.scale(1.1)
            r.rotate(60)
    c = Circle(Vector(0, 250), 50, color="red")
    for i in range(6):
        c.render()
        c.rotate(60)
        c.scale(0.8)
Getting in **Shape**

- Translate
- Rotate about an arbitrary point
- Scale
- Add another shape of your choice (inheritance!)
- Compound shapes!
- Flip about an arbitrary line (defined by two Vectors) [EXTRA CREDIT!]
def demo3():
    radius = 30
    side_length = 10
    group = Compound()
    for angle in range(0, 360, 30):
        angle = math.radians(angle)  # convert from degrees to radians
        s = Square(side_length, Vector(radius * math.cos(angle), \
                   radius * math.sin(angle)))
        group.append(s)
    group.translate(Vector(-200, -200))
    for do_it in range(5):
        group.render()
        group.translate(Vector(30, 30))

Demo shapes_demo.py!
class Compound(Shape):
    def __init__(self, center=Vector(0, 0), shape_list=[]):
        self.shapes = shape_list

    def render(self):

    def translate(self, shift):

    def rotate(self, theta, about=Vector(0, 0)):

    def scale(self, stretch):

    def append(self, shape):

    def __add__(self, other):

Compound “is a” Shape (inheritance) and “has” Shapes (composition).
class Person:
    def __init__(self, ...):
        blah, blah, blah

class Infantryman(Person):
    def __init__(self, ...):
        blah, blah, blah

class Kangaroo(Infantryman):
    def __init__(self, ...):
        blah, blah, blah