Class Hierarchies

1. Points and Lines
   - points in the plane
   - extending the class `Point`
   - representing lines in the plane
   - visualizing lines

2. Parabolas
   - defining the class Parabola
   - extending the Parabola class for visualization

MCS 260 Lecture 33
Introduction to Computer Science
Jan Verschelde, 4 April 2016

Class Hierarchies

1. Points and Lines
   - points in the plane
   - extending the class `Point`
   - representing lines in the plane
   - visualizing lines
2. Parabolas
   - defining the class Parabola
   - extending the Parabola class for visualization

storing coordinates of points

```
class Point(object):
    """
    Stores a point in the plane.
    """
    def __init__(self, x=0, y=0):
        """
        Defines the coordinates.
        """
        self.xpt = x
        self.ypt = y
```

to print points

For a point `p`, if we do `print p`
then we want to see `(p.xpt, p.ypt)`:

```
def __str__(self):
    """
    Returns the string representation.
    """
    return '(' + str(self.xpt) \
               + ', ' \
               + str(self.ypt) + ')'
```

a test program

In the file with the class definition:

```
def main():
    """
    Instantiates two points.
    """
    print('instantiating two points ...')
    first = Point()
    print('p =', first)
    second = Point(3, 4)
    print('q =', second)

if __name__ == "__main__":
    main()
```

use `class_point.py` as a script or a module

The last two lines of `class_point.py` are

```
if __name__ == "__main__":
    main()
```

which allows

- to run the script `class_point.py`, at the prompt `$`:

  ```
  $ python class_point.py
  ```

  which will execute the function `main()`.

- or in an interactive Python shell do

  ```
  >>> from class_point import Point
  >>> p = Point(3, 4)
  >>> print p
  (3, 4)
  ```

Class Hierarchies

1. Points and Lines
   - points in the plane
   - extending the class `Point`
   - representing lines in the plane
   - visualizing lines
2. Parabolas
   - defining the class Parabola
   - extending the Parabola class for visualization

drawing points

Suppose we want to draw a point on canvas.

As before, with Tkinter, we could create a GUI class with a method to draw a point on canvas.

Instead, extend the class `Point` with

- a data attribute: a canvas; and
- a method: to draw the point on canvas.

Benefit: focus on the object that matters: the point.

inheriting from `Point`

```
from Tkinter import Tk, Canvas
from random import randint
from class_point import Point

class ShowPoint(Point):
    """
    Extends the class Point
    with a draw method on a Tkinter Canvas.
    """
    def __init__(self, cnv, x=0, y=0):
        """
        Defines the coordinates (x, y)
        and stores the canvas cnv.
        """
        Point.__init__(self, x, y)
        self.canvas = cnv
```

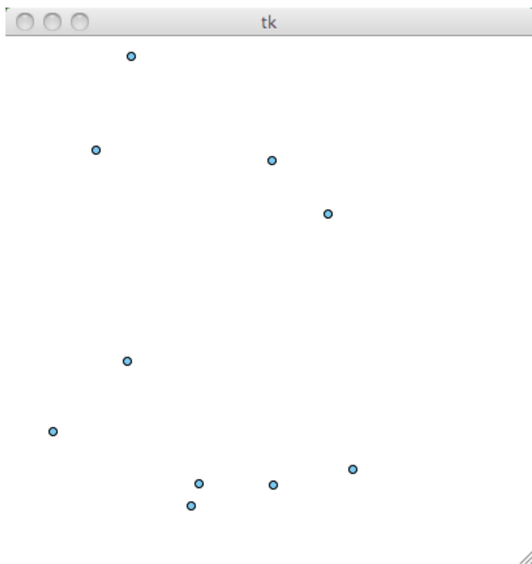
the method draw

```
def draw(self):
    """
    Draws the point on canvas.
    """
    (xpt, ypt) = (self.xpt, self.ypt)
    self.canvas.create_oval(xpt-3, ypt-3, \
        xpt+3, ypt+3, fill='SkyBlue2')
```

the main program

```
def main():
    """
    Shows 10 random points on canvas.
    """
    top = Tk()
    dim = 400
    cnv = Canvas(top, width=dim, height=dim)
    cnv.pack()
    points = []
    for _ in range(10):
        xrd = randint(6, dim-6)
        yrd = randint(6, dim-6)
        points.append(ShowPoint(cnv, xrd, yrd))
    for point in points:
        point.draw()
    top.mainloop()
```

ten random points



Class Hierarchies

1. Points and Lines
   - points in the plane
   - extending the class `Point`
   - representing lines in the plane
   - visualizing lines

2. Parabolas
   - defining the class Parabola
   - extending the Parabola class for visualization

lines in the plane

What is a line?

- $y = ax + b$ is a linear function.
  But what with vertical lines?
- $ax + by + c = 0$ is more general.
  But the algebraic expression is not suited for geometry.

In a geometric representation, every line has

- a basis point, given by (x, y); and
- a direction, given by an angle θ.

Then any point on the line is $(x + t\cos(\theta), y + t\sin(\theta))$.

the class Line

```
from math import cos, sin
from class_point import Point

class Line(Point):
    """
    A line is a base point and
    a direction angle.
    """
    def __init__(self, x=0, y=0, a=0):
        """
        Defines base point and angle.
        """
        Point.__init__(self, x, y)
        self.angle = a
```

implications of inheritance

```
>>> from class_point import Point
>>> from class_line import Line
>>> L = Line(3, 4, 1.23)
>>> isinstance(L, Line)
True
>>> isinstance(L, Point)
True
```

Because of the inheritance, the line `L`
is also an instance of the class `Point`.
We view a line as an extension of a point.

Different design: `Point` is an attribute in `Line`.

the string representation

We override the string representation for `Line`,
using the string representation defined in `Point`:

```
def __str__(self):
    """
    Returns the string representation.
    """
    strp = Point.__str__(self)
    stra = ', angle = ' + str(self.angle)
    return strp + stra
```

line as function

```
def __call__(self, argt):
    """
    Returns a new point on the line.
    """
    rxt = self.xpt + argt*cos(self.angle)
    ryt = self.ypt + argt*sin(self.angle)
    return Point(rxt, ryt)
```

This implements the geometric representation of a line:

- a basis point, given by (x, y); and
- a direction, given by an angle θ.

Then any point on the line is $(x + t\cos(\theta), y + t\sin(\theta))$.

testing the class

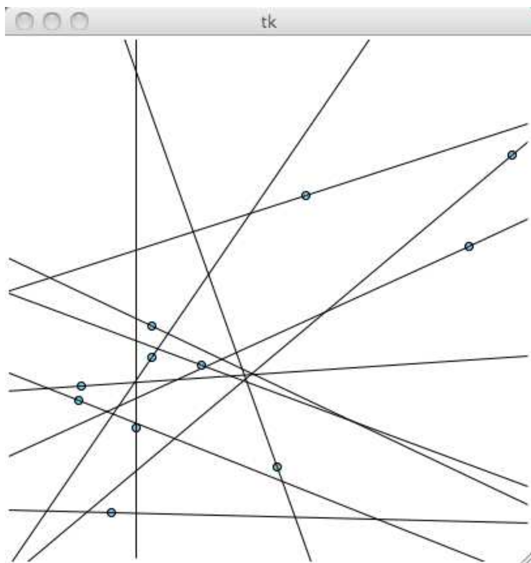
```
def main():
    """
    Defining a line.
    """
    print('defining a line ...')
    first = Line()
    print('the default line :', first)
    print('some point on it :', first(5))
    from math import pi
    second = Line(3, 4, pi/2)
    print('a vertical line :', second)
    print('   that passes through', second(0))
    apt = second(4)
    print('another point on it is', apt)

if __name__ == "__main__":
    main()
```

Class Hierarchies

1. Points and Lines
   - points in the plane
   - extending the class `Point`
   - representing lines in the plane
   - visualizing lines

2. Parabolas
   - defining the class Parabola
   - extending the Parabola class for visualization

eleven lines



extending the `Line` class

```
from class_line import Line
from class_showpoint import ShowPoint

class ShowLine(Line, ShowPoint):
    """
    Extends the Line class with a draw method.
    """
    def __init__(self, c, d, x=0, y=0, a=0):
        """
        Defines the line through the point
        (x, y), with angle a and stores the
        canvas c of dimension d.
        """
        Line.__init__(self, x, y, a)
        ShowPoint.__init__(self, c, x, y)
        self.dimension = d
```

multiple inheritance

- In the class definition:

  ```
  class ShowLine(Line, ShowPoint):
  ```

  The class `ShowLine` inherits from `Line` and `ShowPoint`.

- In the constructor:

  ```
  Line.__init__(self, x, y, a)
  ShowPoint.__init__(self, c, x, y)
  self.dimension = d
  ```

  we instantiate the superclasses `Line` and `ShowPoint`
  and add one extra data attribute, the dimension of the canvas.

Inheriting from `ShowPoint`,
the drawing of the base point of the line is already defined.

computing the range of the line

The extension of the class stores the canvas *and* the dimension of the canvas so we may compute the part of the line that fits on canvas.

The base point is (x, y) and we want to compute the point $x + t\cos(\theta) = d$, where d is the right edge of the canvas.

$$\Rightarrow t = \frac{d - x}{\cos(\theta)}$$

One special case: a vertical line.

drawing a line

```
def draw(self):
    """
    Draws the line on canvas.
    """
    ShowPoint.draw(self)
    cnv = self.canvas
    (xpt, ypt) = (self.xpt, self.ypt)
    dmx = self.dimension - xpt
    csa = cos(self.angle)
    if csa + 1.0 != 1.0:
        pt1 = Line.__call__(self, dmx/csa)
        cnv.create_line(xpt, ypt, pt1.xpt, pt1.ypt)
        pt2 = Line.__call__(self, -xpt/csa)
        cnv.create_line(pt2.xpt, pt2.ypt, xpt, ypt)
    else: # vertical line
        cnv.create_line(xpt, 0, xpt, self.dimension)
```

the test program

```
def main():
    """
    Shows 11 lines on canvas.
    """
    top = Tk()
    dim = 400
    cnv = Canvas(top, width=dim, height=dim)
    cnv.pack()
    lines = [ShowLine(cnv, dim, 100, 300, pi/2)]
    for _ in range(10):
        (xrd, yrd) = (randint(6, dim-6), randint(6, dim-6))
        rnd = uniform(0, 2*pi)
        lines.append(ShowLine(cnv, dim, xrd, yrd, rnd))
    for line in lines:
        line.draw()
    top.mainloop()
```

Class Hierarchies

1. Points and Lines
   - points in the plane
   - extending the class `Point`
   - representing lines in the plane
   - visualizing lines

2. Parabolas
   - defining the class Parabola
   - extending the Parabola class for visualization

representing a parabola

The expression $y = ax^2 + bx + c$ is a quadratic which defines points (x, y) on a parabola.

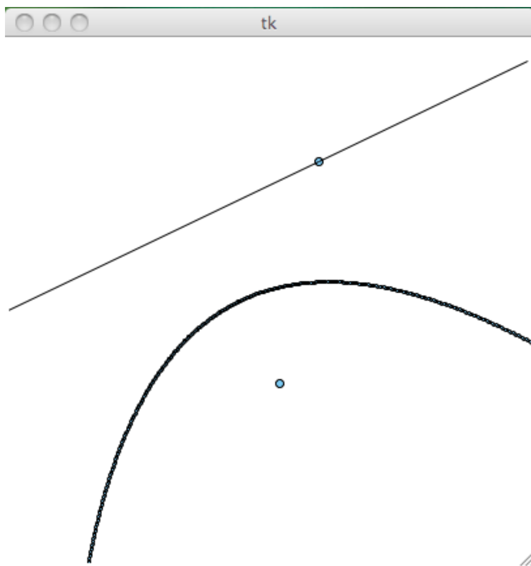
Problem: $y^2 = x$ is also a parabola.

Geometric definition: a parabola is the set of points

- at equal distance from a point: the *focus*
- at equal distance from a line: the *directrix*.

Thus we define a parabola by a line and a point.

focus and directrix



the class Parabola

```
from class_point import Point
from class_line import Line

class Parabola(Line):
    """
    A parabola is defined by a line,
    its directrix and a point, its focus.
    """
    def __init__(self, px, py, agl, fx, fy):
        """
        Defines the line at (px, py)
        and angle agl and focus (fx, fy).
        """
        Line.__init__(self, px, py, agl)
        self.focus = Point(fx, fy)
```

is instance versus *has a*

```
class Parabola(Line):
    def __init__(self, px, py, agl, fx, fy):
        Line.__init__(self, px, py, agl)
        self.focus = Point(fx, fy)
```

An instance of a Parabola

- is an instance of a Line,
- has an instance of a Point as data attribute.

the string representation

```
def __str__(self):
    """
    Returns the string representation.
    """
    strfocus = str(self.focus)
    line = Line.__str__(self)
    result = 'focus : ' + strfocus
    result += ', directrix : ' + line
    return result
```

computing points on a parabola

We compute points on the parabola, running along the directrix $(c_x, c_y) = (x + t\cos(\theta), y + t\sin(\theta))$.

Let the focus have coordinates (f_x, f_y).

The point (α, β) on the parabola at t satisfies

$$(\alpha - c_x)^2 + (\beta - c_y)^2 = (\alpha - f_x)^2 + (\beta - f_y)^2$$

and the line spanned by (α, β) and (c_x, c_y) is orthogonal to the line spanned by (x, y) and (c_x, c_y).

This leads to a linear system in α and β.

deriving the linear system in α and β

Expanding both sides of

$$(\alpha - c_x)^2 + (\beta - c_y)^2 = (\alpha - f_x)^2 + (\beta - f_y)^2$$

leads to

$$\begin{aligned}(\alpha - c_x)^2 + (\beta - c_y)^2 &= \alpha^2 - 2\alpha c_x + c_x^2 + \beta^2 - 2\beta c_y + c_y^2 \\ (\alpha - f_x)^2 + (\beta - f_y)^2 &= \alpha^2 - 2\alpha f_x + f_x^2 + \beta^2 - 2\beta f_y + f_y^2.\end{aligned}$$

Subtracting the equations eliminates α^2 and β^2:

$$2(f_x - c_x)\alpha + 2(f_y - c_y)\beta = f_x^2 + f_y^2 - c_x^2 - c_y^2.$$

The other linear equation: $(\alpha - c_x, \beta - c_y) \perp (x - c_x, y - c_y)$, with $(c_x, c_y) = (x + t\cos(\theta), y + t\sin(\theta))$.

the linear system

Perpendicularity $\perp$ is calculated with a dot product:

$$\begin{array}{rl} & (\alpha - c_x, \beta - c_y) \perp (x - c_x, y - c_y) \\ \Leftrightarrow & (\alpha - c_x)(x - c_x) + (\beta - c_y)(y - c_y) = 0 \\ \Leftrightarrow & (x - c_x)\alpha + (y - c_y)\beta = c_x(x - c_x) + c_y(y - c_y). \end{array}$$

Recall that $2(f_x - c_x)\alpha + 2(f_y - c_y)\beta = f_x^2 + f_y^2 - c_x^2 - c_y^2$.

In matrix form:

$$\left[\begin{array}{cc} 2(f_x - c_x) & 2(f_y - c_y) \\ x - c_x & y - c_y \end{array} \right] \left[\begin{array}{c} \alpha \\ \beta \end{array} \right] = \left[\begin{array}{c} f_x^2 + f_y^2 - c_x^2 - c_y^2 \\ c_x(x - c_x) + c_y(y - c_y) \end{array} \right].$$

With Cramer's rule we find explicit formulas for α and β.

making a parabola callable

```
def __call__(self, t):
    """
    Returns a point on the parabola
    as far from the focus and the point
    obtained by evaluating the directrix.
    """
    line = Line.__call__(self, t)
    (fcx, fcy) = (self.focus.xpt, self.focus.ypt)
    disc = 2*(fcx - line.xpt)*(line.ypt - self.ypt) \
         - 2*(fcy - line.ypt)*(line.xpt - self.xpt)
    rh1 = fcx**2 + fcy**2 - line.xpt**2 - line.ypt**2
```

The discriminant `disc` is the determinant of

$$\begin{bmatrix} 2(f_x - c_x) & 2(f_y - c_y) \\ x - c_x & y - c_y \end{bmatrix}.$$

function continued

```
    if disc + 1.0 != 1.0:
        rh2 = line.xpt*(line.xpt - self.xpt) \
            + line.ypt*(line.ypt - self.ypt)
        sdx = (rh1*(line.ypt - self.ypt) \
            - 2*rh2*(fcy - line.ypt))/float(disc)
        sdy = (2*(fcx - line.xpt)*rh2 \
            - (line.xpt - self.xpt)*rh1)/float(disc)
    else:
        from math import exp, pi
        line = Line.__call__(self, exp(pi))
        disc = 2*(fcx - line.xpt)*(line.ypt - self.ypt) \
             - 2*(fcy - line.ypt)*(line.xpt - self.xpt)
        rh1 = fcx**2 + fcy**2 - line.xpt**2 - line.ypt**2
        rh2 = line.xpt*(line.xpt - self.xpt) \
            + line.ypt*(line.ypt - self.ypt)
        sdx = (rh1*(line.ypt - self.ypt) \
            - 2*rh2*(fcy - line.ypt))/float(disc)
        sdy = (2*(fcx - line.xpt)*rh2 \
            - (line.xpt - self.xpt)*rh1)/float(disc)
    return Point(sdx, sdy)
```

the test function

```
def main():
    """
    Instantiates a parabola and evaluates.
    """
    print('instantiating a parabola ...')
    prb = Parabola(3, 4, -1.23, 10, 0)
    print(prb)
    point = prb(4)
    print('a point on the parabola :', point)

if __name__ == "__main__":
    main()
```

Class Hierarchies

1. Points and Lines
   - points in the plane
   - extending the class `Point`
   - representing lines in the plane
   - visualizing lines
2. Parabolas
   - defining the class Parabola
   - extending the Parabola class for visualization

class definition

```
from Tkinter import Tk, Canvas
from math import pi
from random import randint, uniform
from class_parabola import Parabola
from class_showline import ShowLine

class ShowParabola(Parabola, ShowLine):
    """
    Extends the Parabola class with a draw method.
    """
    def __init__(self, c, d, x, y, a, fx, fy):
        """
        Defines the parabola with directrix the line
        at base point (x, y), angle a, and the focus
        with coordinates (fx, fy).
        Stores the canvas c of dimension d.
        """
        Parabola.__init__(self, x, y, a, fx, fy)
        ShowLine.__init__(self, c, d, x, y, a)
```

drawing the parabola

```
def draw_focus(self):
    """
    Draws the focus on canvas.
    """
    (fxp, fyp) = (self.focus.xpt, self.focus.ypt)
    self.canvas.create_oval(fxp-3, \
        fyp-3, fxp+3, fyp+3, fill='SkyBlue2')

def draw(self):
    """
    Draws the parabola on canvas.
    """
    ShowLine.draw(self)
    self.draw_focus()
    for tpt in xrange(-1000, 1000):
        prb = Parabola.__call__(self, tpt)
        self.canvas.create_oval(prb.xpt-1, \
            prb.ypt-1, prb.xpt+1, prb.ypt+1, \
            fill='SkyBlue2')
```

the main program

```
def main():
    """
    Draws one parabola
    """
    top = Tk()
    dim = 400
    cnv = Canvas(top, width=dim, height=dim)
    cnv.pack()
    (xbs, ybs) = (randint(6, dim-6), randint(6, dim-6))
    agl = uniform(0, 2*pi)
    (fpx, fpy) = (randint(6, dim-6), randint(6, dim-6))
    pbl = ShowParabola(cnv, dim, xbs, ybs, agl, fpx, fpy)
    pbl.draw()
    top.mainloop()
```

Summary + Exercises

By inheritance we defined points, lines, and parabolas.

1. Bind mouse events to the canvas so the user can move the focus of the parabola. The parabola is redrawn automatically after each move of the focus.
2. Make an animation of one directrix sweeping through a canvas with multiple focus points. As the line sweeps, plot the parabolas defined by the fixed foci and the moving directrix.