Abstract Data Types (II)
(and Object-Oriented Programming)

Jordi Cortadella and Jordi Petit
Department of Computer Science
Public or private?

• What should be public?
  – Only the methods that need to interact with the external world. Hide as much as possible. Make a method public only if necessary.

• What should be private?
  – All the attributes.
  – The internal methods of the class.

• Can we have public attributes?
  – Theoretically yes (Python and C++ allow it).
  – Recommendation: never define a public attribute.

• Observation: Python does not support public/private attributes and methods. There is no protection to prevents a bad use.

• The naming conventions (underscores) are used to distinguish them.
Class Point: a new implementation

- Let us assume that we need to represent the point with polar coordinates for efficiency reasons (e.g., we need to use them very often).

- We can modify the private section and the implementation of the class without modifying the specification of the public methods.

- The API (a contract between designers and users) should not be modified.

- Do you know what a deprecated method is?
  - Not recommended, possibly superseded by another method
  - Likely to be removed or discontinued in the future
class Point:
    """A class to represent and operate with two-dimensional points""

    # Declaration of attributes (recommended for type checking)
    _radius: float # radius of the polar coordinates
    _angle: float # angle of the polar coordinates

    def __init__(self, x: float = 0, y: float = 0):
        """Constructor with x and y coordinates"
        self._radius = math.sqrt(x*x + y*y)
        self._angle = 0 if x == 0 and y == 0 else math.atan2(y/x)

    def x(self) -> float:
        """Returns the x coordinate"
        return self._radius*math.cos(self._angle)

    def y(self) -> float:
        """Returns the y coordinate"
        return self._radius*math.sin(self._angle)

    def distance(self, p: 'Point' | None) -> float:
        """Returns the distance to point p
        (or the distance to the origin if p is None)"
        dx, dy = self.x(), self.y()
        if p is not None:
            dx -= p.x()
            dy -= p.y()
        return math.sqrt(dx*dx + dy*dy)

Works without any change (but it can be done more efficiently)
Class Point: a new implementation

```python
def angle(self) -> float:
    """Returns the angle of the polar coordinates""
    return self._angle

def __add__(self, p: 'Point') -> 'Point':
    """Returns a new point by adding the coordinates of two points.
    This method is associated to the + operator""
    return Point(self.x() + p.x(), self.y() + p.y())

def __eq__(self, p: 'Point') -> bool:
    """Checks whether two points are equal.
    This method is associated to the == operator""
    return self.x() == p.x() and self.y() == p.y()
```

Discussion:
- How about using _x and _y (or _radius and _angle) as "public" attributes?
- Programs using p._x and p._y would not be valid for the new implementation.
- Programs using p.x() and p.y() would still be valid.

Recommendation:
- All attributes should be private.
A new class: Rectangle

• We will only consider rectilinear rectangles (axis-aligned).

• A rectilinear rectangle can be represented in different ways:

Two points (extremes of diagonal)

One point, width and height
Rectangle: abstract view

Create

Rectangle(11, ur)

Scale

scale(0.5)

Rotate

rotate(false)

Move

(1,8) move(10, 2)

Intersection

Point inside?
from point import Point

class Rectangle:
    """Class to operate with rectilinear rectangles"""
    
def __init__(self, ll: Point, ur: Point):
        """Constructor using the LL and UR corners"""
        ...

def area(self) -> float:
    """Returns the area of the rectangle"""
    ...

def scale(self, s: float) -> None:
    """Scales the rectangle with a factor s > 0"""
    ...

def __mul__(self, r: 'Rectangle') -> 'Rectangle':
    """Returns the intersection with another rectangle"""
    ...
r1 = Rectangle(4,5)  # Creates a rectangle 4x5
r2 = Rectangle(8,4)  # Creates a rectangle 8x4

r1.move(2,3)        # Moves the rectangle
r1.scale(1.2)       # Scales the rectangle
area1 = r1.area()   # Calculates the area

r3 = r1*r2;

if r3.empty(): ...
from point import Point

class Rectangle:
    """Class to operate with rectilinear rectangles"""

    # Private attributes to represent a rectangle
    _ll: Point  # Lower-left corner of the rectangle
    _w: float   # width of the rectangle (>= 0)
    _h: float   # height of the rectangle (>= 0)

    def __init__(self, ll: Point, ur: Point):
        """""Constructor using the LL and UR corners"""
        assert ll.x() <= ur.x() and ll.y() <= ur.y()
        self._ll = Point(ll.x(), ll.y())
        self._w = ur.x() - ll.x()
        self._h = ur.y() - ll.y()

Note: Python does not support function overloading. Classes can only have one constructor.
class Rectangle:

    def width(self) -> float:
        """Returns the width of the rectangle"""
        return self._w

    def height(self) -> float:
        """Returns the height of the rectangle"""
        return self._h

    def ll(self) -> Point:
        """Returns the LL corner of the rectangle"""
        return Point(self._ll.x(), self._ll.y())

    def ur(self) -> Point:
        """Returns the UR corner of the rectangle"""
        return self._ll + Point(self.width(), self.height())

Discussion: what if ll() would return self._ll (instead of a copy)?
class Rectangle:

    def area(self) -> float:
        """Returns the area of the rectangle""
        return self.width()*self.height()

    def scale(self, s: float) -> None:
        """Scales the rectangle by s, keeping the LL corner fixed""
        self._w *= s
        self._h *= s

    def empty(self) -> bool:
        """Checks whether the rectangle is empty""
        return self.area() <= 0  # Gives some tolerance
class Rectangle:
    :
    
def __mul__(self, r: 'Rectangle') -> 'Rectangle':
        """Returns the intersection of two rectangles"""

        # Calculate the ll coordinates
        r1ll, r2ll = self.ll(), r.ll()
        ll_x = max(r1ll.x(), r2ll().x())
        ll_y = max(r1ll.y(), r2ll().y())

        # Calculate the ur coordinates
        r1ur, r2ur = self.ur(), r.ur()
        ur_x = min(r1ur.x(), r2ur().x())
        ur_y = min(r1ur.y(), r2ur().y())

        # Check if no intersection: return empty rectangle
        if ur_x <= ll_x or ur_y <= ll_y:
            return Rectangle(Point(0, 0), Point(0, 0))

        return Rectangle(Point(ll_x, ll_y), Point(ur_x, ur_y))
Let us work with rectangles

```python
r1 = Rectangle(Point(2,3), Point(6,8))
area1 = r1.area()  # area1 = 20

r2 = Rectangle(Point(3,5), Point(5, 9))

# Check whether the point (4,7) is inside the intersection of r1 and r2.
bool in = (r1*r2).isPointInside(Point(4,7))

r2.rotate(false)  // r2 is rotated counterclockwise
r2 *= r1;         // Intersection with r1
```

Exercise: draw a picture of R1 and R2 after the execution of the previous code.
A Python session with rational numbers

```python
>>> from rational import Rational  # from file rational.py
>>> a = Rational(4, -6)  # construct with num and den
>>> print(a)
-2/3
>>> b = Rational(4)  # integer value
>>> print(b)
4
>>> print((a+b).num(), (a+b).den())
10 3
>>> c = Rational()  # c = 0
>>> if a < c:
...    print(a, "is negative")
...
-2/3 is negative
>>> print(a*b)  # uses the __str__ method (see later)
-8/3
>>> a/b  # uses the __repr__ method (see later)
Rational(-1/6)
```
class Rational:
    """Class to represent rational numbers"""

    # Private attributes:
    # Invariant: _den > 0 and gcd(_num, _den) = 1
    _num: int  # numerator
    _den: int  # denominator (invariant: _den > 0)

    def __init__(self, num: int = 0, den: int = 1):
        assert isinstance(num, int)
        assert isinstance(den, int)
        assert den != 0

        self._num, self._den = num, den
        self._simplify()
class Rational:
    :
    def num(self) -> int:
        """Returns the numerator""
        return self._num

    def den(self) -> int:
        """Returns the denominator""
        return self._den

    def _simplify(self) -> None:  # Private method
        """Simplifies the representation""
        if self._den < 0:
            self._num *= -1
            self._den *= -1
        d = math.gcd(abs(self._num), self._den)
        self._num //= d
        self._den //= d
class Rational:

    :

def __str__(self) -> str:
    """Returns a user-friendly string with information about the value of the object. It is invoked by str(x) or print(x).""
    if self._den == 1:
        return str(self._num)
    return str(self._num) + "/" + str(self._den)

def __repr__(self) -> str:
    """Returns a string with information about the representation of the class. It is invoked by repr(x) or simply 'x'.""
    return "Rational(" + str(self) + ")"
class Rational:

    :

    def __neg__(self) -> 'Rational':
        """Returns -self.""
        return Rational(-self._num, self._den)

    def __add__(self, rhs: 'Rational') -> 'Rational':
        """Returns self + rhs.""
        num = self._num*rhs._den + self._den*rhs._num
        den = self._den*rhs._den
        return Rational(num, den)

    # Similarly for __sub__, __mul__, __truediv__
class Rational:
    : 

def __eq__(self, rhs: 'Rational') -> bool:
    """Checks whether self == rhs.""
    return self._num == rhs._num and self._den == rhs._den

def __ne__(self, rhs: 'Rational') -> bool:
    """Checks whether self != rhs.""
    return not self == rhs

def __lt__(self, rhs: 'Rational') -> bool:
    """Checks whether self < rhs.""
    return self._num*rhs._den < self._den*rhs._num

def __le__(self, rhs: 'Rational') -> bool:
    """Checks whether self <= rhs.""
    return not rhs < self

# Similarly for __gt__ and __ge__
Python documentation: docstrings

```python
>>> from rational import Rational
>>> help(Rational.__add__)
Help on function __add__ in module rational:

__add__(self, rhs)
    Returns self + rhs.

>>> help(Rational)
class Rational(builtins.object)
    Rational(num=0, den=1)

    Class to represent rational numbers.

    The class includes the basic arithmetic and relational operators.

    Methods defined here:

    __add__(self, rhs)
        Returns self + rhs.

    __eq__(self, rhs)
        Checks whether self == rhs.
```

© Dept. CS, UPC
Python documentation: *docstrings*

- The first line after a module, class or function can be used to insert a string that documents the component.

- Triple quotes ("""""""") are very convenient to insert multi-line strings.

- The *docstrings* are stored in a special attribute of the component named `__doc__`.

- Different ways of print the *docstrings* associated to a component:
  - `print(Rational.num.__doc__)`
  - `help(Rational.num)`
Designing a module: example

# geometry.py

"""geometry.py
Provides two classes for representing Polygons and Circles."""

# author: Euclid of Alexandria

from math import pi, sin, cos

class Polygon:
    """Represents polygons and provides methods to calculate area, intersection, convex hull, etc."""

    def __init__(self, list_vertices: list[Point]):
        """Creates a polygon from a list of vertices."""
        ...

class Circle:
    ...

Documentation of the module

Documentation of the class

Documentation of the method
Using a module: example

```
import geometry
p = geometry.Poligon(...)
c = geometry.Circle(...)  
```
Imports the module. Now all classes can be used with the prefix of the module.

```
import geometry as geo
p = geo.Poligon(...)  
c = geo.Circle(...)  
```
Imports and renames the module.

```
from geometry import *
p = Poligon(...)  
c = Circle(...)  
```
Imports all classes in the module. No need to add the prefix of the module.

```
from geometry import Poligon as plg, Circle as cir
p = plg(...)  
c = cir(...)  
```
Imports and renames the classes in the module.
Conclusions

• Finding the appropriate hierarchy is a fundamental step towards the design of a complex system.

• User-friendly documentation is indispensable.
Implement the following methods for the class Rectangle:

```python
def rotate(self, clockwise: bool) -> None:
    """Rotate the rectangle 90 degrees clockwise or counterclockwise, depending on the value of the parameter. The rotation should be done around the lower-left corner of the rectangle""

def flip(self, horizontally: bool) -> None:
    """Flip horizontally (around the left edge) or vertically (around the bottom edge), depending on the value of the parameter.""

def isPointInside(self, p: Point) -> bool:
    """Check whether point p is inside the rectangle""
```
Re-implement the class Rectangle using an internal representation with two Points:

- Lower-Left (LL)
- Upper-Right (UR)