6. Object-Oriented Programming, I

Programming and Algorithms II

Degree in Bioinformatics

Fall 2017

Creating complex algorithms vs. creating large programs

Abstraction – Information Hiding

Design so that one can decompose a large problem into many smaller ones

And so that one one is thinking about one subproblem, one can for a while stop thinking about the others

We place the houses on the map. We worry later about how to design each house

When designing this one house, we forget about the others. We asume there are water, gas, electricity connections nearby

Modules

Your program text is split among several files

Someone else may have written modules for you

```
import m
x = m.f1(10)
y = m.f2(30)

myf = m.f1
myg = m.f2
x = myf(10)
y = myg(30)
```

```
from m import f1
from m import f2 as g

x = f1(10)
y = g(30)


from m import *
```

Modules

A module may contain:

- Function definitions
- Statements, typically initializations

In a program execution, the statements are executed only once, at the first import

Modules as abstraction

We can use the module without knowing what is inside the file

But we need to know the specification

= the manual

= all types and operations that we can import,

what parameters they have, what they mean, what do these functions compute; perhaps their costs

We do not need the bodies of the functions to know how to use (if specification is well done…); that's hidden

Abstract Data Types (ADT)

Defined by:

- Set of values of the type
- Operations that can be performed on the type
- Properties of the types

This is the specification of the type – all you need to use it

Examples:

(x+1)-1 = x x*y = y*x (x+y)*z = x*z + y*z

lst.append(x) → lst[len(lst)-1] == x

myset.add(v) → (v is in myset) == True

Advantages of ADT design

Separation of specification and implementation

We do not need to know the implementation to use the type in our programs

Principle (design by contract):

> If a program uses ADT X and works correctly,
> and we replace the implementation of X
> with another (correct) one,
> then the whole program still works correctly

(but it may be more or less efficient)

Implementation of ADTs

Headers of public operations

- Constructor
- Queries
- Modifiers
- Destructors

Private (not to be used from the outside)

- Representation of the type (variables)
- Private (auxiliary) operations

Many languages forbid access to private parts

(keywords "*private*", "*protected*")

In Python, "we are all responsible users": all is visible

ADTs versus OOP

Favorite eternal discussion among programmers

Object-oriented, object-based, prototype-based, class-based languages....

Let me keep it brief

Anyway, people will disagree with me…

ADTs versus OOP

ADT: the programmer defines the type and provides an implementation for it. Then creates variables of the type. The programmer knows (in principle) the type of each object, so the implementation of the operation to use

OOP:

A program is a set of objects that exchange messages telling each other what to do

When an object receives a message, that object decides on the meaning of the message, and what to do about it

Objects with different implementations may understand the same message, but act differently

It may not be possible to determine, from the text of the program, the type of the object that will process a message

Objects and classes

Objects have two kinds of properties:

- methods: functions that "belong" to an object
- attributes or fields: variables that keep state of the object

(in Python, both methods and fields called attributes)

We said "append" is an operation of the list class

But when we write "lst.append(30)", we want to think like

"call the append method that belongs to object lst"

Objects and classes

A set of properties:

- methods: functions that “belong” to an object
- attributes or fields: variables that keep state of the object

(in Python, all called attributes)

Class: template for many objects with properties in common

“A person” vs. “Ricard Gavaldà”

(name,height,weight,age…. walk,eat,talk,sleep…)

An object is then an instance of its class

Using a Class Point

```
p = Point(3.0,10.0)

if p.x > 0 and p.y > 0:
    print(“the point is in first quadrant”)

if p.distance_to_origin() > 5000:
    print(“the point is really far away”)
```

Notation

Outside the class:

p.x object.attribute

(error if p does not have attribute called x)

p.f(…) object.method(parameters)

(error if p does not have attribute called f which is a function)

Notation

Inside the class:

class Point(object):

- We are defining class Point
- All Points are objects = Point is a subclass of object

In the same sense that teachers are persons, or that all cows are both mammals and herbivores. More on this next day

- Instances of Point inherits all properties that objects have, and some more that we are defining for Points now

Notation

Inside the class:

def f(self,otherparameters)

- f can be called as obj.f(…)
- obj is then passed as first parameter (self) to f

self.x = value (inside the code of a method)

- we create an attribute named x for the object self, if it didn't exist yet
- we change its value if it already existed
- it keeps existing even after the method finishes

Notation

Inside the class:

Special method

```
__init__(self, parameters)
```

Is called “the constructor”. Called whenever a new class instance is created

```
p = Point(30.0,40.0)
```

will call __init__(self,30.0,40.0)

Using a Class Point

```
p = Point(3.0,10.0)

If p.x > 0 and p.y > 0:
    print(“point is in first quadrant”)

if p.distance_to_origin() > 5000:
    print(“point is really far away”)
```

Class Point: Implementation

```
from math import *
class Point(object):

      def __init__(self,x,y):
            self.x = x
            self.y = y

      def distance_to_origin(self):
            return sqrt(self.x*self.x+self.y*self.y)

      def angle_to_origin(self):
         if x != 0: return atan(self.y/self.x)
         else: return pi/2 * sign(y)    # sign in {1,-1,0}
```

(for brevity, I will ignore the case x==0 in further slides)

Class Point: Another Implementation

```
class Point(object):

    def __init__(self,x,y):
        self.mod = sqrt(x*x+y*y)
        self.angle = atan(y/x)

    def distance_to_origin(self):
        return self.mod

    def angle_to_origin(self):
        return self.angle

    def x(self):
        return self.mod*cos(self.angle)

    def y(self):
        return self.mod*sin(self.angle)
```

Using a Class Point

```
p = Point(3.0,10.0)

if p.x() > 0 and p.y() > 0:
    print(“point is in first quadrant”)

if p.distance_to_origin() > 5000:
    print(“point is really far away”)
```

Recommended practice in most languages: do not access attributes directly, use setter and getter functions:

p.set_x(value)

p.get_x()

more representation Independence

(Python offers another way: @property)

@property

Makes a function look like an attribute

```
@property
def my_attribute(self):
    return …

@my_attribute.setter
def my_attribute(self,val):
    change real attributes so that my_attribute becomes val
```

now we can do:

```
x = obj.my_attribute
obj.my_attribute = value
```

@property

Makes a function look like an attribute

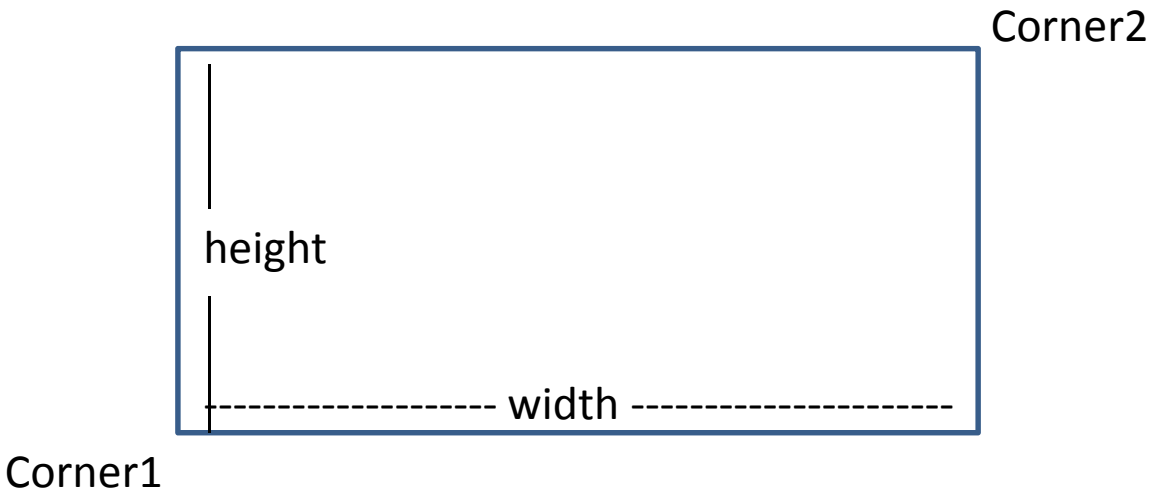
```
@property
def mod(self):
    return sqrt(self.x*self*x+self.y*self.y)

@property
def rho(self):
    return atan(y/x)
```

Impossible to tell from the outside whether we are representing the point as (x,y) or as (mod,rho)

Class Rectangle

A rectangle can be given (and represented) by:

- Bottom left corner + width + height
- Two opposite corners



Class Rectangle

Constructors:

- create a point from two opposite corners
- create a point from bottom left corner, width, height

Attributes or @properties:

- corner (bottom left corner)
- corner2 (upper right corner)
- w (width)
- h (height)

True methods:

- contains(a_point)
- is_empty()
- overlaps(another_rectangle)
- intersect(another_rectangle)
- translate_by(x,y)

Class Rectangle

```
r = Rectangle(Point(10,10),Point(30,40))

if r.w > r.h
    print(“this rectangle is wider than taller”)


print("this triangle sits at (“,
          r.corner.x, ",", r.corner.y,”)”)
```

Class Rectangle: An implementation

```
class Rectangle(object):
      def __init__(self, p1,p2):
          self.corner = p1
          self.w = p2.x - p1.x
          self.h = p2.y - p1.y
      def contains(a_point):
          return corner.x <= a_point.x and
                 corner.y  <= a_point.y and
                 a_point.x <= corner.x + w and
                 a_point.y <= corner.y + h
      def …
```

Class Rectangle: An implementation

We may also want to create a point given corner, width, and height

```
r1 = Rectangle(Point(1,2),Point(3,4))
```

or

```
r2 = Rectangle(Point(1,2),2,2)
```

```
(this exactly won't work, but something similar will…)
```

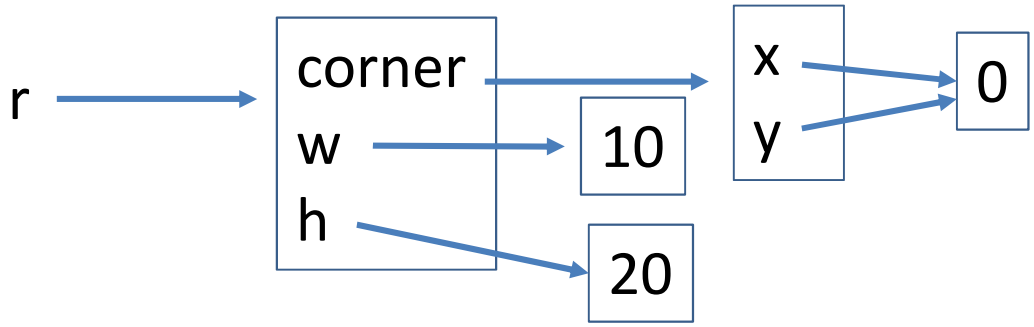
Class Rectangle: implementation

Creator that can be called both ways, using keyword parameters:

```
class Rectangle(object):
      def __init__(self,p1,p2=None,width=0,height=0):
            """gets a point plus, to choose,
               either another point p2 OR width
               and height parameters"""
            self.corner = p1
            if p2 is None:
                self.w = width
                self.h = height
            else:
                self.w = p2.x - p1.x
                self.h = p2.y - p1.y
           # we could check for too few
           # or too many arguments
```

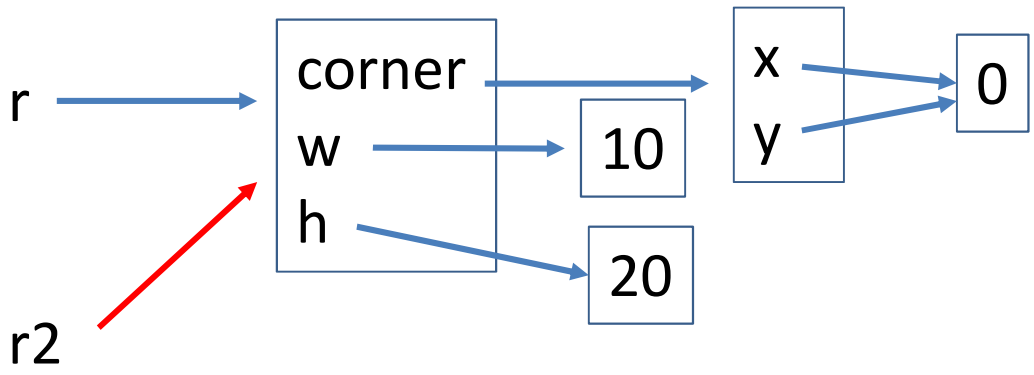
Objects and copying

```
r = Rectangle(Point(0,0),width=10,height=20)
```



Objects and copying

```
r = Rectangle(Point(0,0),width=10,height=20)
r2 = r
```



=

assign to r2 a reference

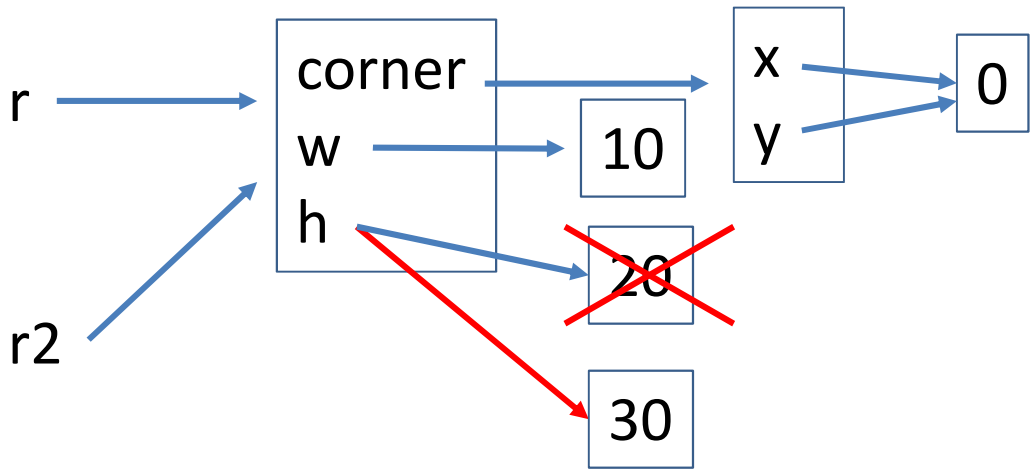
to the object to which

r refers

aliasing r, r2

Objects and copying

```
r = Rectangle(Point(0,0),width=10,height=20)
r2 = r
r2.h = 30
print(r.h)  # r, not r2!
>>> 30
```



=

assign to r2 a reference

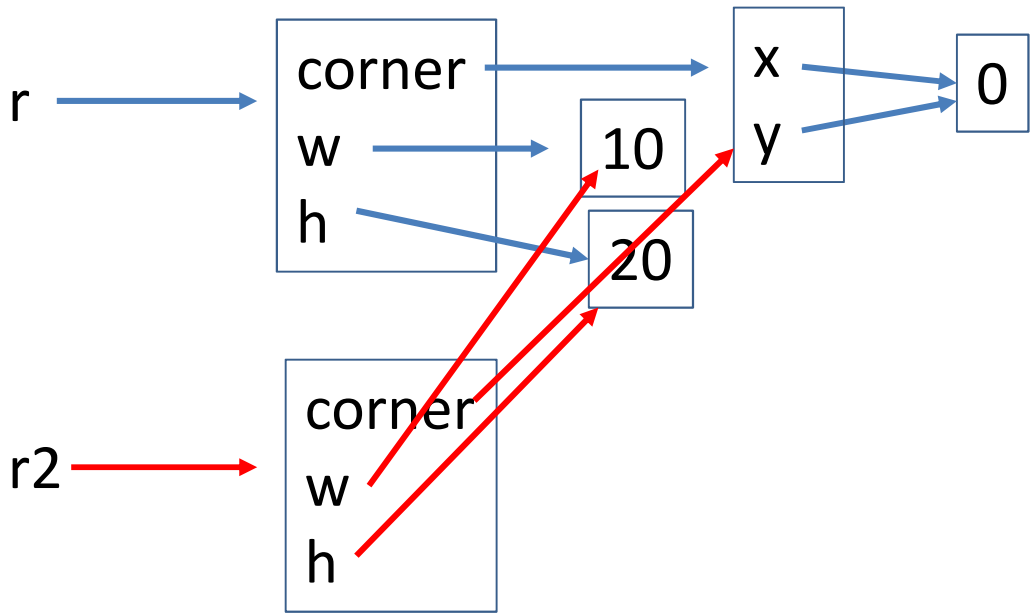
to the object to which

r refers

aliasing r, r2

(Shallow) copy

```
import copy
r = Rectangle(Point(0,0),width=10,height=20)
r2 = copy.copy(r)
```

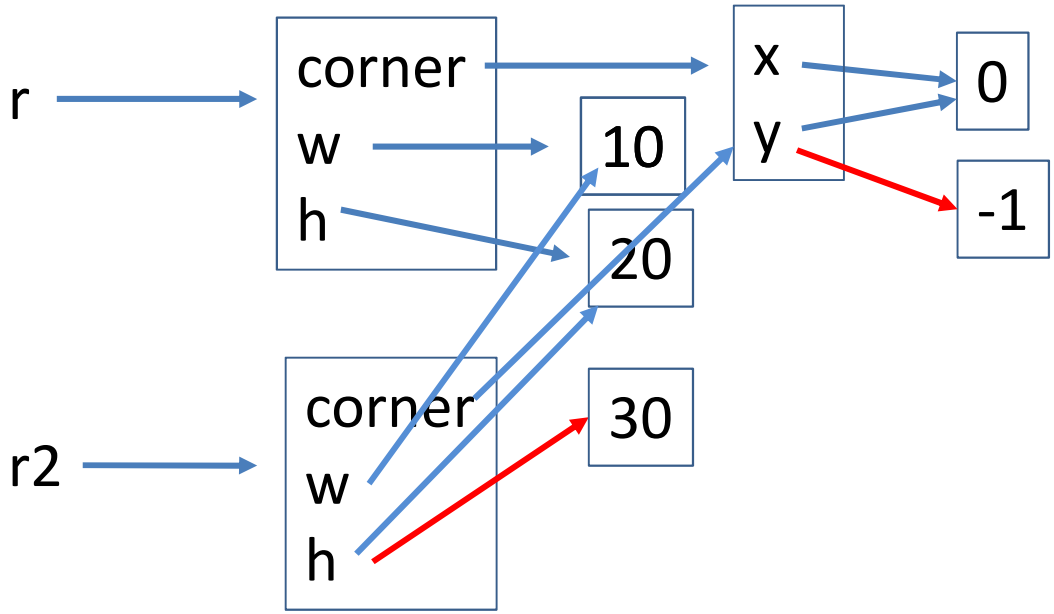


shallow copy:

1. create a new object,
2. assign to destination a reference to new object
3. copy fields from source to it

(Shallow) copy

```
import copy
r = Rectangle(Point(0,0),width=10,height=20)
r2 = copy.copy(r)
r2.h = 30
print(r.h)  # r, not r2!
>>> 20
r2.corner.y = -1
print(r.corner.y)  # r, not r2!
>>> -1
```

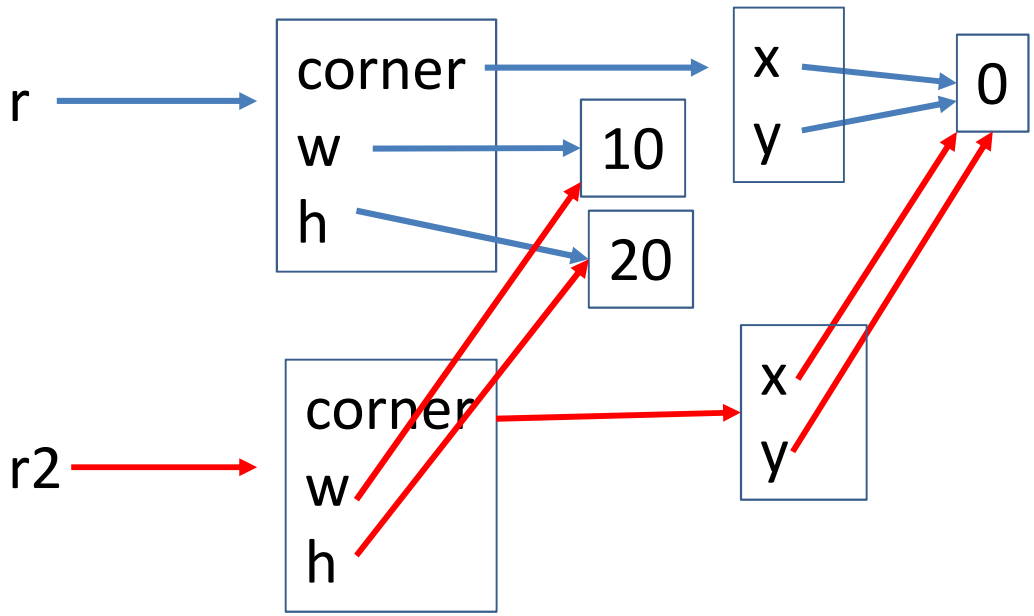


shallow copy:

1. create a new object,
2. assign to destination a reference to new object
3. copy fields from source to it

Deep copy

```
import copy
r = Rectangle(Point(0,0),width=10,height=20)
r2 = copy.deepcopy(r)
```



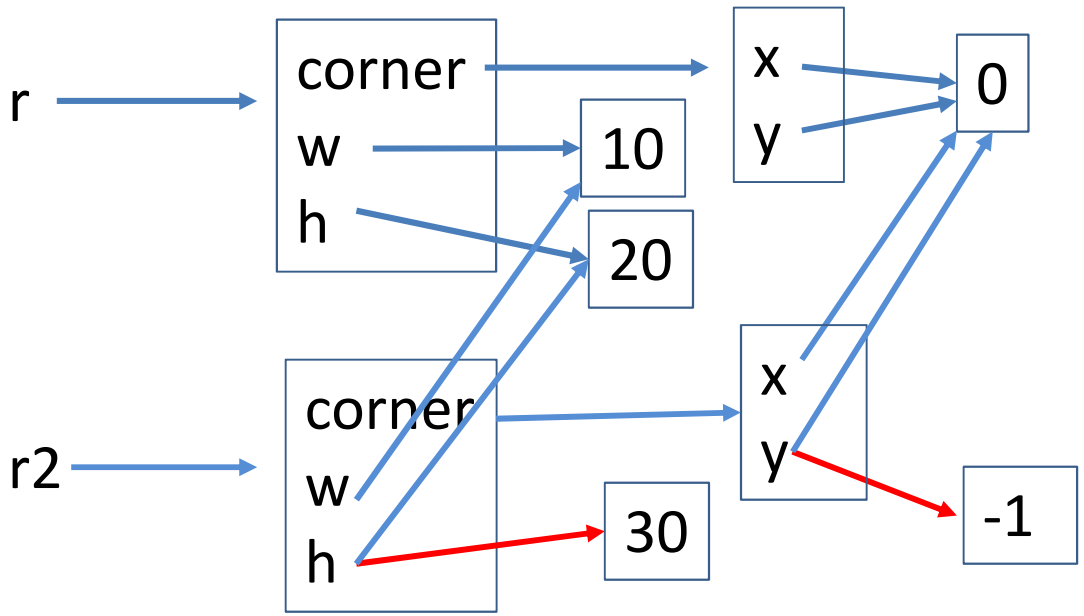
deep copy:

1. create a new object,
2. assign to destination a reference to new object
3. copy fields from source to it,

recurse!

Deep copy

```
import copy
r = Rectangle(Point(0,0),width=10,height=20)
r2 = copy.deepcopy(r)
r2.h = 30
r2.corner.y = -1
print(r.h,r.corner.y)  # r, not r2!
>>> 20 0
```



deep copy:

1. create a new object,
2. assign to destination a reference to new object
3. copy fields from source to it,

recurse!

Class properties

```
class sheep:
        count = 0
        def __init__(self):
              count += 1


...
print("you have created", sheep.count,
                              "instances of sheep")
```

Note: count is created outside any method

So: it is a property of the class, not of an instance of the class

It is common to all instances

Class properties

Similarly, a method without “self” parameter is a class method.

```
class myclass:
    …
    def f(value):
        print(value)
```

can be called as

```
>>> myclass.f(10)
10
```

Summary

- Abstraction / information hiding
- Modules
- Abstract Data Types; representation independence
- Objects, methods, attributes, classes, instances
- Think: each object carries its own attributes and methods
- = and copy.copy may create aliases
  - to fully avoid aliasing, use copy.deepcopy
- aliasing may be useful for efficiency
  - *but it makes code confusing and error-prone. Use at your own risk*
- class properties may be occasionally useful
  - but often indicate bad design