5. Object-Oriented Programming, I

Programming and Algorithms II

Degree in Bioinformatics

Fall 2018

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

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:

- Functions
- Classes (see soon)
- Statements, typically initializations. 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

Object-Oriented Programming (OOP)

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

When an object receives a message, <u>that object</u> 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. Attributes

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

Class: template for a set of objects with properties in common

“A human” vs. “John Smith” or “Mary Ryan”

(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 with the same number of parameters)

Notation

Inside the class:

class Point(object):

- We are defining class Point
- All Points are objects = Point is a <u>subclass</u> 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 <u>inherits</u> 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

```
Suppose a class Point that offers attributes .x, .y,
.distance_to_origin(), .angle_from_origin(), plus Point(…) to
create a 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

```
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):
       atan2(self.y,self.x)
```

atan2(y,x) returns atan(y/x) but handles the signs of y and x correctly; think of the pairs (1,1) and (-1,-1); also, it takes care of returning +- pi/2 if x = 0 instead of dividing by 0

@property

Makes a method look like an field

```
@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
```

Class Point: Another Implementation

```
class Point(object):
    def __init__(self,x,y):
        self.mod = sqrt(x*x+y*y)
        self.angle = atan2(y,x)
    def distance_to_origin(self):
        return self.mod
    def angle_to_origin(self):
        return self.angle
    @property
    def x(self):
        return self.mod*cos(self.angle)
    @property
    def y(self):
        return self.mod*sin(self.angle)

(... continues)
```

Class Point: Another Implementation (2)

(… continued)

```
@x.setter
  def x(self,val):
      cury = self.y
      self.mod = sqrt(val*val+cury*cury)
      self.angle = atan2(cury,val)

  @y.setter
  def y(self,val):
      curx = self.x
      self.mod = sqrt(curx*curx+val*val)
      self.angle = atan2(val,curx)
```

This does not work

```
@x.setter
  def x(self,val):
      self.mod = sqrt(val*val+self.y*self.y)
      self.angle = atan2(self.y,val)
```

Function call!
gives wrong result after
mod has been changed

Difference between implementations

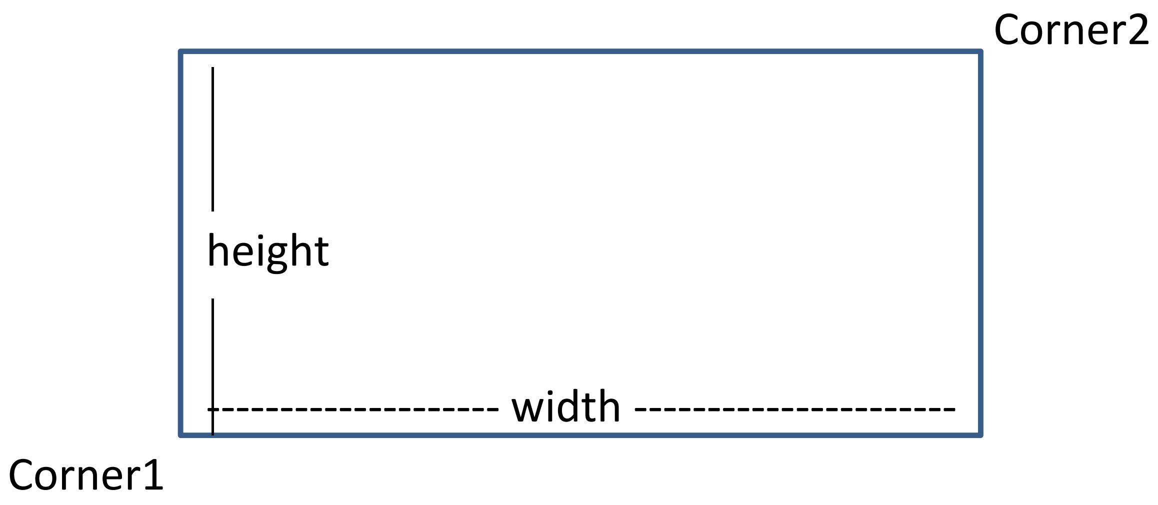
- One keeps information in attributes x, y, and adds two function attributes
- The other keeps information in attributes mod, angle, and offers four function attributes, two of which “look like” variables x, y, the others like true functions
- It is *impossible* to tell, when we are *using* the class, which of the two implementations we are using.
- This lets us change the implementation without changing any class that uses it.
- Recommended practice in most languages: do not access attributes directly, use setter and getter functions:
  - p.set_x(value)
  - p.get_x()
- (in Python we can use @property instead)
- In many programming languages, one can *forbid* accessing some attributes that are considered “private” – to implement stuff, not to be used from outside

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
                       a_point.x <= corner.x + w and
                       corner.y  <= a_point.y and
                       a_point.y <= corner.y + h
      def …
```

Class Rectangle: implementation

We can define positional parameters and 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
```

Example

With the above definition, we can call

```
r = Rectangle(Point(1,2),p2=Point(3,4))
```

or

```
r = Rectangle(Point(1,2),width=2,height=2)
```

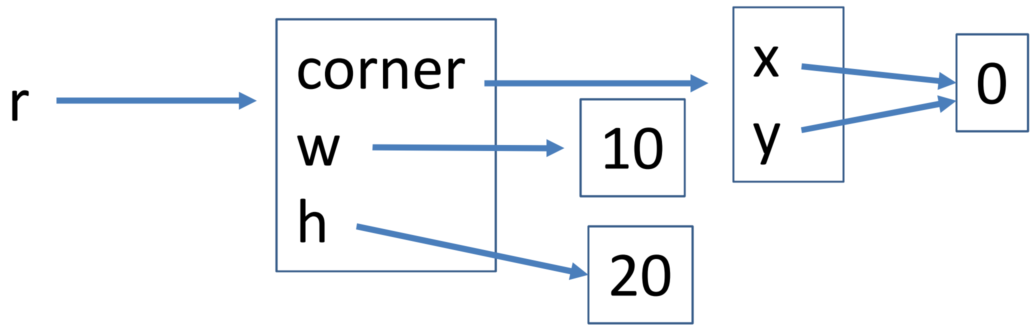
or

```
r = Rectangle(Point(1,2),height=2,width=2)
```

```
(we can also call
r = Rectangle(Point(1,2), p2=Point(3,4), width=10, height=22)
but our implementation will, in this case, ignore width and height)
```

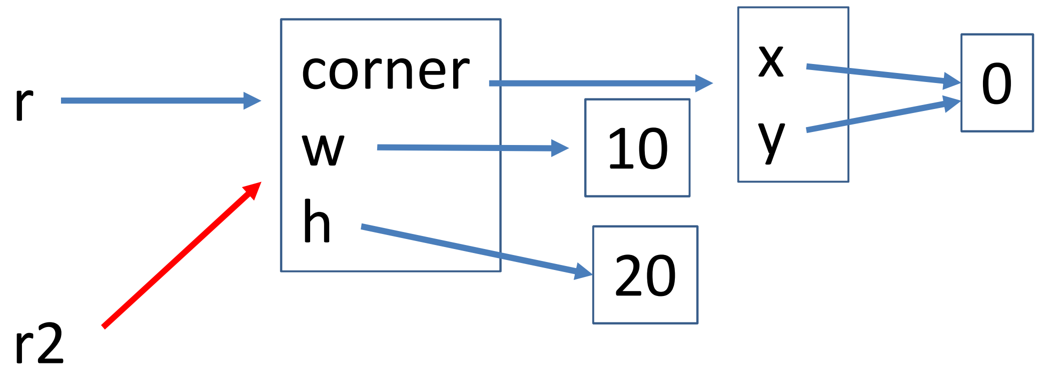
Objects and copying

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



Objects and copying

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

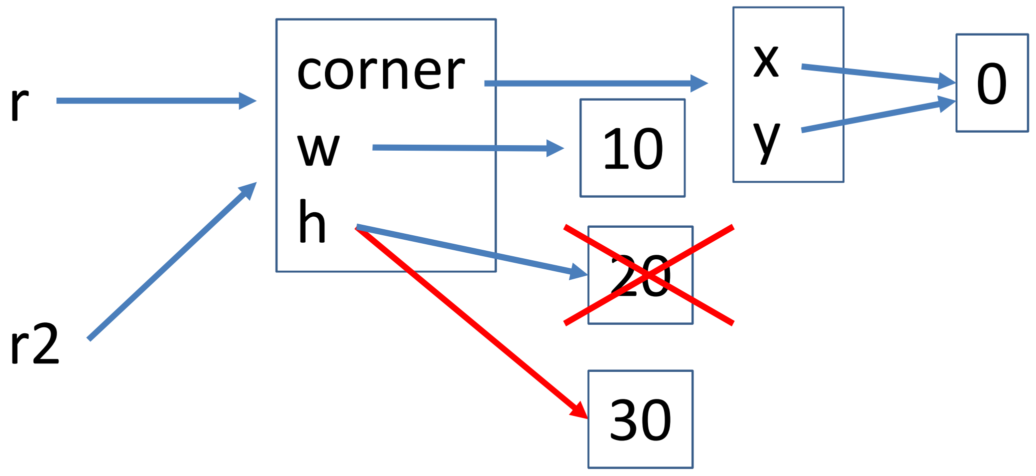


```
=
assign to r2 a
reference
to the object
to which
r refers
```

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`

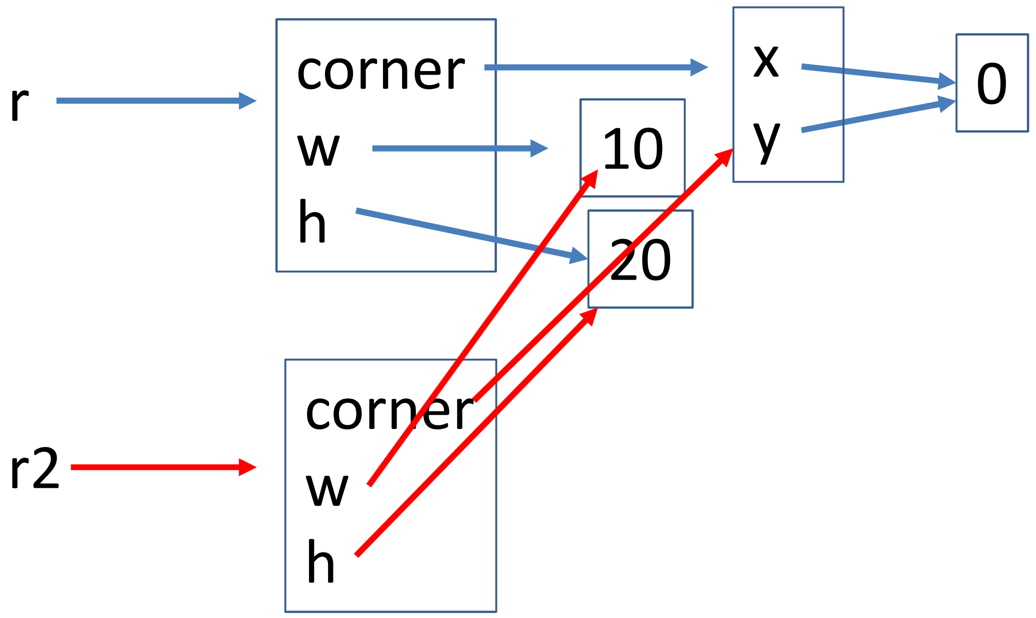
`to the object to which`

`r refers`

`aliasing` `r, r2`

(Shallow) copy

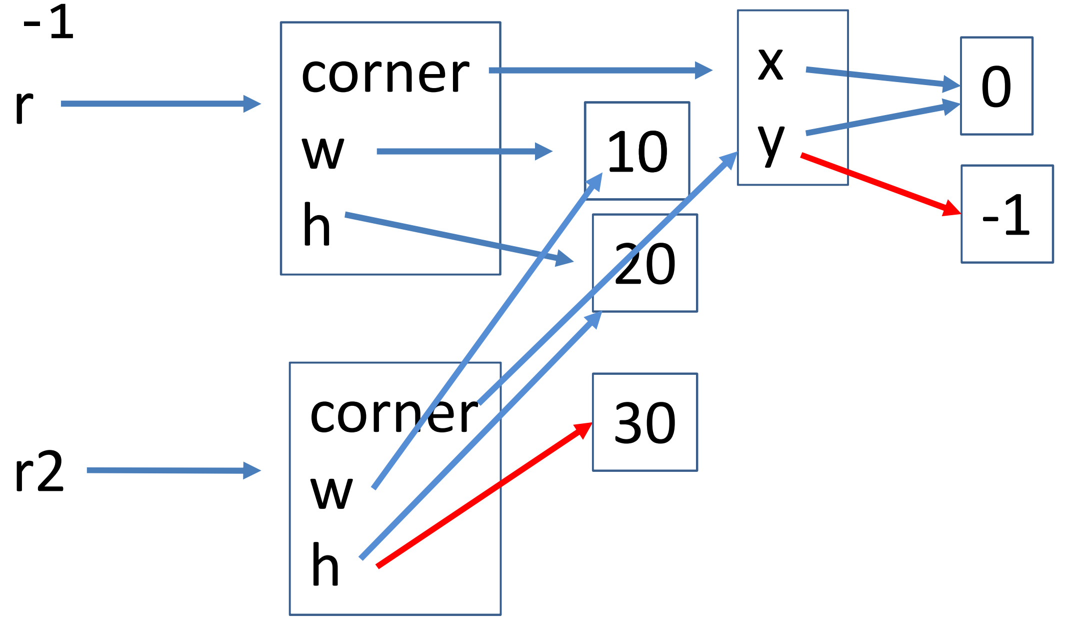
```
import copy
r = Rectangle(Point(0,0),width=10,height=20)
r2 = copy.copy(r)
```



```
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:

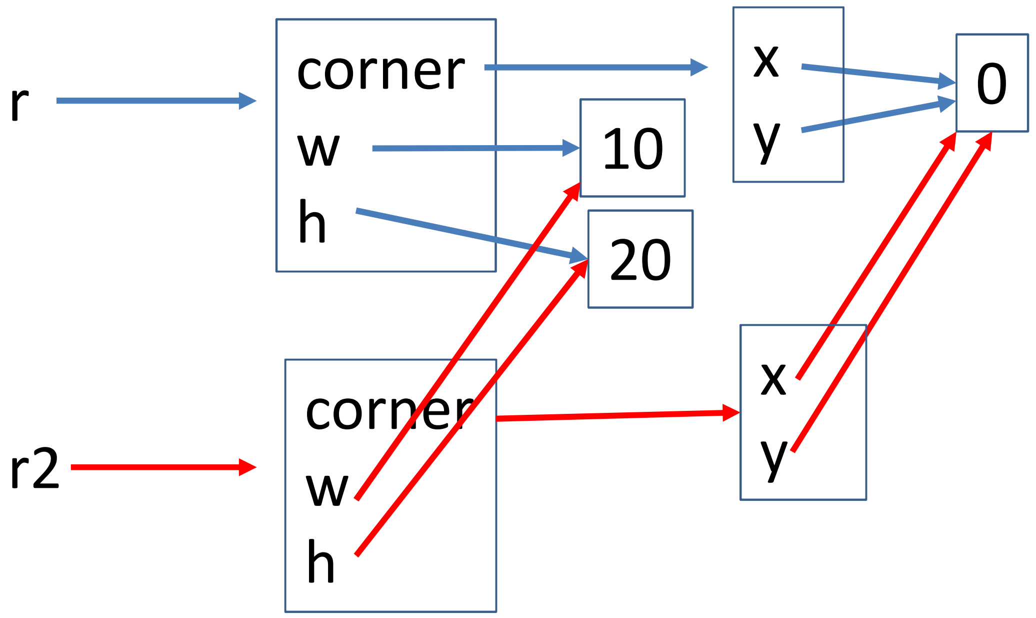
1. create a new object,

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:

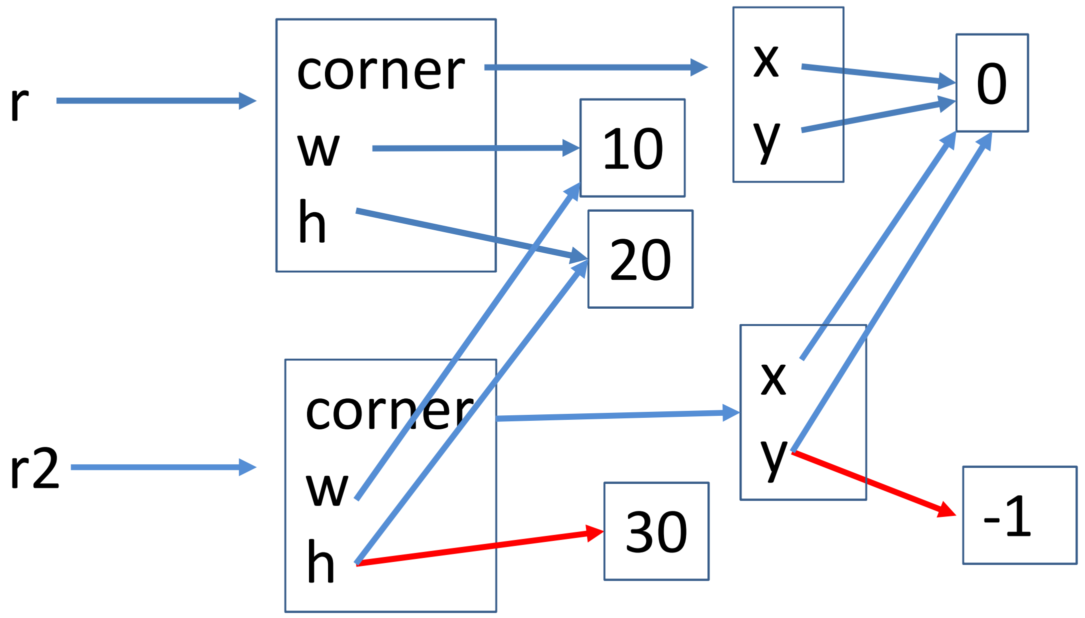
1. create a new object,
2. assign to destination a reference to new object
3. copy fields from source to it,

recurse!

but stop at unmutable values

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(object):
       count = 0
       def __init__(self):
             count = 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(object):
   …
   def f(value):
      print(value)
```

can be called as

```
>>> Myclass.f(10)
10
```

Summary

- Abstraction / information hiding
- Modules
- 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