Creating complex algorithms vs. creating large programs
Design so that one can decompose a large problem into many smaller ones.

And so that 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 assume there are water, gas, electricity connections nearby.
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 *
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:
\[(x+1)-1 = x \quad x*y = y*x \quad (x+y)*z = x*z + y*z\]
\[\text{lst.append(x) } \rightarrow \text{ lst[len(lst)-1] == x}\]
\[\text{myset.add(v) } \rightarrow \text{ (v is in myset) == True}\]
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

```python
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:

\[ p.x \quad \text{object.attribute} \]
  (error if \( p \) does not have attribute called \( x \))

\[ p.f(...) \quad \text{object.method(parameters)} \]
  (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
Inside the class:

def f(self, other_parameters)
    • 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
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”)
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):
        return atan(self.y/self.x)
class Point(object):

    def __init__(self, x, y):
        self.mod = sqrt(x*x+y*y)
        if x != 0.0:
            self.angle = atan(y/x)
        else:
            self.angle = 0.0

    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

```python
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)
A rectangle can be given (and represented) by:

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

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

corner (bottom left corner)
w (width)
h (height)

contains(a_point)
is_empty()
overlaps(another_rectangle)
intersect(another_rectangle)
translate_by(x,y)
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(object):
    def __init__(self, p1, p2):
        self.corner = p1
        self.w = p2.x() - p1.x()
        self.h = p2.y() - p1.y()
    def contains(self, a_point):
        return self.corner.x <= a_point.x and
              self.corner.y <= a_point.y and
              a_point.x <= self.corner.x + self.w and
              a_point.y <= self.corner.y + self.h
    def ...
Class Rectangle: An implementation

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

\[
r_1 = \text{Rectangle}(\text{Point}(1,2), \text{Point}(3,4))
\]

or

\[
r_2 = \text{Rectangle}(\text{Point}(1,2), 2, 2)
\]
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

```python
r = Rectangle(Point(0,0), width=10, height=20)
r2 = r
r2.w = 30
print(r.w)
```
import copy
r = Rectangle(Point(0,0), width=10, height=20)
r2 = copy.copy(r)
r2.w = 30
r2.corner.x = -1
print(r.w, r.corner.x)

```python
>>> 10, -1
```

**Shallow copy:**
- Create a new object,
- Assign to destination a reference to new object
- Copy fields from source to it
import copy
r = Rectangle(Point(0,0), width=10, height=20)
r2 = copy.deepcopy(r)
r2.w = 30
r2.corner.x = -1
print(r.w, r.corner.x)
>>> 10, 0

deep copy:
create a new object,
assign to destination a reference to new object
copy fields from source to it,
recurse!
Class properties

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

...

print("you have created", myclass.count, "instances of c")

c is created outside any method
it is a property of the class, not of instances of the class
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
• Objects, methods, attributes, classes, instances
• Think: each object carries its own attributes and methods
• Representation Independence
• =, copy, deepcopy
• class properties occasionally useful