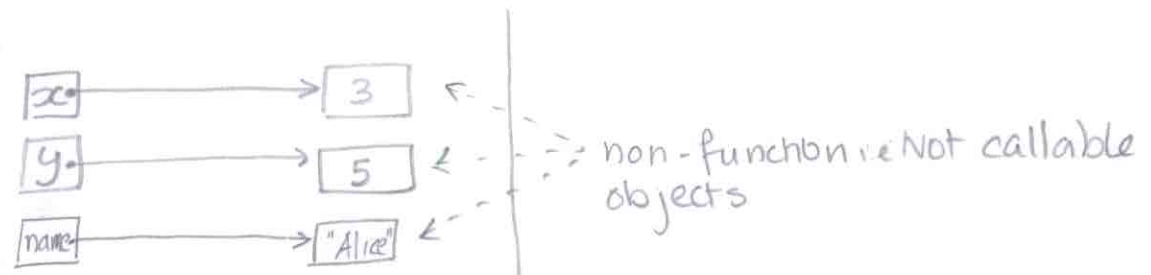


# CLASSES, AND OBJECTS

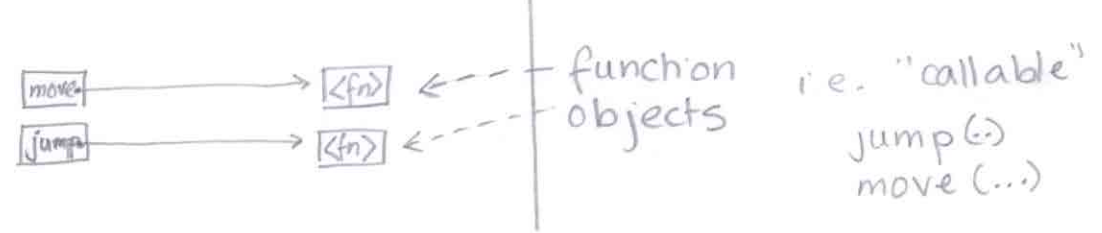
Revelation of the day: OBJECTS are just NAMESPACES!

Think of an object in JAVA. What does it have

① Fields



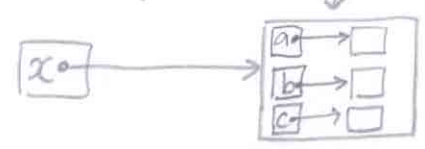
② METHODS



OBJECT: names / attributes / fields / methods → OBJECTS

In Python, a function is also an object so you have a uniform data model. Everything is an object.

So in a namespace: is a namespace!



- ① Everything is an object
- ② Object = Namespace

So, in Python, to access the fields/methods of an object `O` we do `O.x` or `O.m`. (x) (m)

The only difference between ordinary objects (non-function) and functions, is that the latter are callable.

Very similar to ML! Very uniform. Everything is a value can be passed around, returned etc. Only, function values are spl. in that you can do "application" i.e. you can pass params

into them, i.e. "call" them, and you get values output or returned. ok. So, an object is a namespace.

How can we create objects? What techniques do we already know? (To create namespaces)

① Write a function!

- A namespace gets created when the function executes
- The "local" variables of the function are assigned here
- But this namespace "goes away" after function is finished...

- If you "return" a function, then as we saw, the local namespace gets frozen inside the function object like a "closure".

Calls to this function can access / alter namespace but we cannot directly access the namespace.

② Write a new file [a "module" in Python]:

This creates a new namespace with the same name as the file.

```
>>> import pluscount
```

We now have this object "pluscount" called "modules" which are added to the global namespace

(3)

Can access these objects by looking up relevant attributes.

```
>>> pluscount.ctr
```

```
>>> pluscount.next
```

Now notice, that we have packaged inside an object both some data (ctr) as well as some ways to operate on the data.

But this we had before (functions). What's so great about the "object"?

It's that now the data and the operation have been bundled together into an entity, the object, which "knows" how to do things to itself!

In other words we can now view programming quite differently. Rather than data + operations the operations are baked in together with the data.

Now you can write functions like:

```
>>> def doCount(c, n=10):  
    for i in range(n):  
        print c.next()
```

} - We don't know what the data is  
- We don't know how next works!

But we can still write generic routines like this one which will "make the object" count upto "n" times.

```
>>> import minuscount
>>> doCount (minusCount, 20)

0
-1
-2
-3
...
-19
>>> doCount (pluscount)

0
...
9
>>> doCount (minuscount, 5)

-20
-21
-22
-23
-24
```

If the entity, the "object" knows how to react

- to update itself
- to compute

based on external stimuli... message call

Then we have a very powerful new way of structuring programs and computation.

- Highly flexible
- Reuse code
- keep operations, data in one place
- create new objects w/ different behavior but all use same code!

Simula - 67 Motivation.

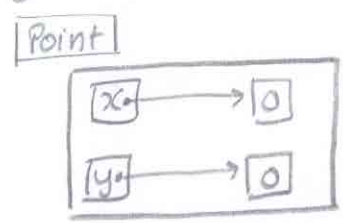
Sounds a bit like Polymorphism it is but now each object could be doing totally different things to implement its "next" method.

All we need on doCount, is that the object be able to receive, i.e. have its namespace, a function called "next".

Ok. So that's one way to create and use "objects" but, to put it mildly, it sucks! Every time we want a new object, we'd have to write a new file!

The standard PL technique to get/build/construct objects is a "class".

```
>>> class Point :  
    x = 0  
    y = 0
```



This creates a NAMESPACE Point, with 2 attributes x, y  
What can we do with this namespace? Well, like any namespace, we can peep into its attributes.

```
>>> Point.x  
0
```

Instantiations : More importantly, we can INSTANTIATE a class namespace, like so:

```
>>> p = Point()  
>>> q = Point()
```

} You can think of instantiation as roughly (very roughly!) "cloning" the namespace Point.

In other words,  $p, q$  are now two independent "copies" of the namespace "Point". So, we can play around with their attributes independently:

```
>> p.x = 7
>> p.x
7
>> q.x
0
>> q.x = 11
>> q.x
11
>> Point.x
0
```

Each of  $p, q, \text{Point}$  corresponds to a different namespace. (Don't forget that  $\text{Point}$  is also a namespace). So, they can be mutated independently.

Notice that by assigning to  $p.x$ , we are MUTATING the namespace i.e. instance  $p$ .

### Class Objects vs Instance Objects

The namespace  $\text{Point}$  is a class object, while  $p, q$  are instances of  $\text{Point}$  called instance objects.

A class object is one that may be instantiated. It is typically defined using the keyword class.

Class NAME :

<binding 1>

<binding 2>

⋮

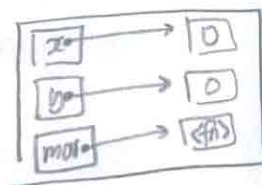
Each binding corresponds to an attribute of the namespace NAME.

So far, all these namespaces are just like records in C/ML. We turn the heat up a little, by adding function definitions (bindings) to the class namespace.

```
>>> class Point:
```

```
    :
    def move(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy
```

This creates an attrib  
move in the namespace



Note that the function "move" takes 3 parameters.

Now when we create some instances:

```
>>> p = Point()
```

```
>>> q = Point
```

We observe that the process of instantiation also creates the attributes in p, q.

```
>>> p.move
```

```
<method object...>
```

This method OBJECT is very critically different from

the FUNCTION OBJECT defined by `def move...`

Intuitively, EVERY function defined for a class must have at least one parameter. This first parameter,

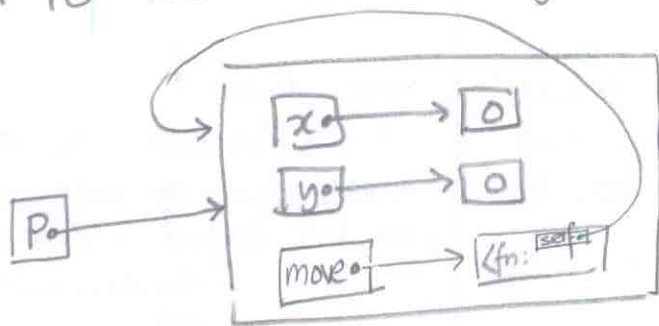
conventionally (but not necessarily) called self

is special. → When the class is instantiated to

get an instance object (eg. p) the first parameters of all the functions defined for the class are bound to the instance created...

(8)

Thus, `p.move` is a "method object" which is essentially, `Point.move` with the first argument (`self`) bound to the instance object `p`.



Thus, if you call `p.move(a, b)` you get the same effect as if you had called `Point.move(p, a, b)`.

So if you do:

```
>>> p.move(10, 100)
```

It executes the function `Point.move`, with

argument `self` bound to `p`

argument `dx` bound to `10`

argument `dy` bound to `100`

This execution assigns to, and thus alters what `self.x`, `self.y` referred to, and thus mutates the object `self` was referring to. As `self` was bound to the object `p`, `p` itself gets mutated.

Thus, by binding the first argument to the instance created, we obtain instances with methods that operate on the particular instance. i.e. each instance object can perform operations on itself!

Thus, during execution,  $A.B$  is evaluated thus:

if  $A$  is an instance object, and  $B$  is a function defined in the class whose instance  $A$  is, then  $A.B$  is the function where the first param is bound to the instance  $A$ .

else  $A.B$  is the attribute  $B$  of the namespace  $A$ .

Note that  $p.move$  is itself an object which can be passed around (like function objects).

```
>>> f = p.move
```

```
>>> (p.x, p.y)
```

```
(10, 100)
```

```
>>> for i in range(10): f(100, 100)
```

```
>>> (p.x, p.y)
```

```
(1010, 1100)
```

This has the effect of  $p.move(100, 100)$  hence...

By binding the operations / data together, we get the chief benefit of OO - DELAYED BINDING aka VIRTUAL METHODS aka DYNAMIC DISPATCH.

This means we can write generic code like:

```
def spin(c, n=100):
    for i in range(n):
        print c.next()
```

Without at all worrying about what  $c$  and  $next$  are!

We can call `do(...)` with ANY objects which have a method "next" and because of delayed binding, the appropriate operation gets called and changes the object, thus printing out the different sequences.

See the 3 classes `PlusCount`, `MinusCount`, `Count` and associated calls to `spin()` in the python file `lec23.py`.

INITIALIZING OBJECTS (CONSTRUCTORS)

These examples also show another feature common to OO-languages. The initializer which is used to customize the state of the object upon creation. Often we may not want a direct mirror of the class namespace, but instead set up the attributes in a specific way for different instances.

Python lets you do this by defining an `--init--` method for the class.

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
```

Now, to instantiate `Point`, you have to give the parameters (other than the first one, which is bound to the instance created). Python "calls" the `--init--` method right after creating the object

```
>>> p = Point(20, 20)
>>> (p.x, p.y)
(20, 20)
>>> q = Point(100, 200)
>>> (q.x, q.y)
(100, 200)
```

instead of "starting off" at (0,0) we can set them to custom values.

Two other such functions are `--str--`

which is called whenever python sees `str(x)`

i.e. `str(x)` is a pretty way of saying `x.--str--()`

and it is used to get string representations of the object.

```
>>> def Point:
```

```
    def --str--(self):
```

```
        return "x=" + str(self.x) + " y=" + str(self.y)
```

Another standard method is `--repr--` which, like `str` returns a string, but is by convention more precise in that it returns corresponds to how you might construct the object instance:

```
>>> class Point:
```

```
    def --repr--(self):
```

```
        return "Point(' + repr(self.x) + ', ' + repr(self.y) + ')"
```

This method is called if you ask Python in the Read-Eval-Print loop, what the object refers to

```
>>> p = Point(10, 20)
```

```
>>> str(p)
```

```
    x=10 y=20
```

```
>>> p
```

```
Point(10, 20)
```

→ Calls `p.--str--()`

→ Calls `p.--repr--()`

Note that `Point(10, 20)`

would create an object similar in content to `p`.

# INHERITANCE

Now that we've understood the most basic OO-concepts classes and instances let's see some of the nifty things we can do with them: Let's juice up Point so that it has a new method called "draw" which displays it to the screen.

Now, we can write a simple screensaver type thing that takes a list of points and has them fizzing around the screen. (See `screen.py` and `points.py`)

Each "Point" only needs methods move, draw  
and fields x, y, name

```
>>> p1 = Point(...)
>>> p1.move(10,10)
# repr. before, after move

>>> p2 = Point(...)
>>> p2.move(50,30)
# prints out representations
>>> go()
before, after move
```

We have now seen a very simple class: Point.

Each point instance generated by "instantiating" the class by `Point(...)` contains some data and a couple of operations

The data is the location (co-ords  $x, y$ )  
operations are the motion (move/jump) & draw that plots the point on the screen.

We also saw how you can

use any objects that have these properties and plot them.

x ————— x ————— x

Now suppose you're feeling lightheaded or playful, and you think that black is boring.

You would like to have colored points. These have largely the same properties, only you want the ability to alter their color.

So many features or attributes are common - posn, move, draw, jump, but some extra features - color.

x ————— x

INHERITANCE: This brings us to a key idea

in OO programming - Inheritance. We'd like to

- extend the notion of point
  - specialize the notion of point
- } to include color

But without redoing all the code from scratch!

```
>>> class ColoredPoint(Point):
```

```
    def setColor(self, color=(0,0,0))
        self.color = color
```

```
    def draw(self, screen, pos):
        ... # use the color to draw.
```

```
    def __str__(self):
        return Point.__str__(self) + "color:" + str(self.color)
```

This new class inherits from Point. We say ColoredPoint is a subclass of the superclass Point for ColPoint extends Point all the names defined in the namespace of Point

are automatically included for ColoredPoint.

So, an instance of colored point gets from Point the methods:

`--init--`, `move`, `jump`

In addition, it extends the parent class by adding the new methods setColor that creates the additional field color.

What about `draw` and `--str--`? The super (parent) class had these fields too, but by giving new definitions ~~here~~ in the subclass, we get to OVERRIDE the previous definitions.

Once again, we see at work the essence of OO-late-binding / Dynamic dispatch.

If you say `x.draw(..)`, which method gets called? Depends on the object `x` is at run-time...

if it is a `ColoredPoint`, then `ColoredPoint.draw` is called, if it is a vanilla point, `Point.draw` gets called.

But again, be very careful. Lets see what happens if

```
>>> p3 = ColoredPoint(...)
>>> p3.setColor((0,255,0))
>>> p3.jump(20,20)
p3 : ... # prints out color !
      jumps to ...
p3 : ... # prints out color !
```

What happened? Lets look at the code for move in the class `Point`.

```
def jump(self, dx, dy)
    print str(self)
    self.x = self.x + dx
    self.y = self.y + dy
    print "moves to ", str(self)
```

← These two call `self...str..()`

Due to dynamic dispatch, the code defined in the super class, ends up calling the overriding `-- str --` defined in sub/ext class ColoredPoint

In this slick way, overriding / dynamic dispatch let you extend the functionality of older objects without even knowing what future extensions may be.

Here, the same code for "move" works perfectly, indep of how you may extend with color, sparkles, shapes, because dd. ensures the appropriate `--repr--` gets called!

But actually, there is a subtle problem with what I just showed you, illustrating some problems with inheritance. Clues?

```
>>> p5 = ColoredPoint()  
>>> p5.move(10,10)  
ERROR!
```

} What happened?  
Well, we didn't set the color field! The intended `--init--` doesn't know about color so until `setColor` called — there is no color defined!

Be careful — many subtle errors like this due to overriding / dispatch.

Fix: add a default color field to the class...

so when `move` calls the overriding `"--repr--"` an error gets thrown!

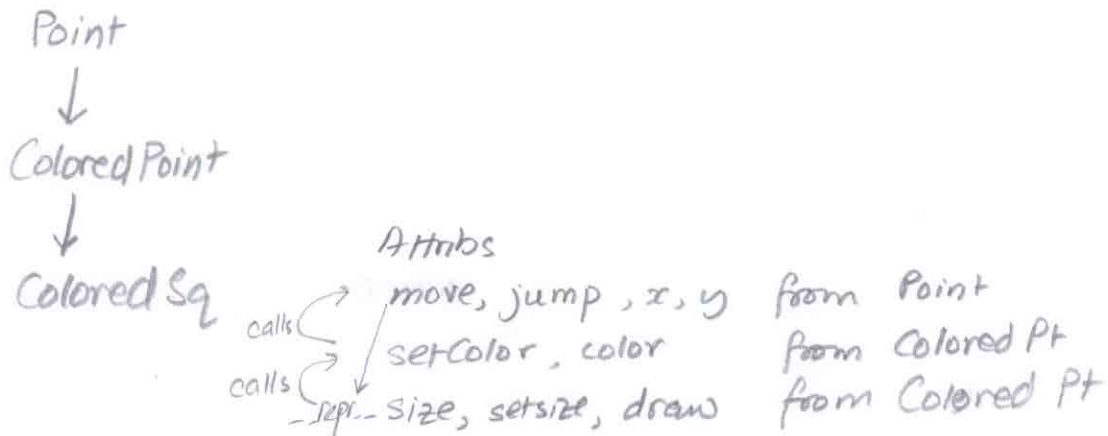
Now we can continue to use these new points

```
>>> p5 = setColor((0,255))  
>>> p6 = ColoredPoint(...)  
>>> p6.setColor((0,0,255))  
>>> points = points + [p5, p6]  
>>> do(points)
```

Now, nothing prevents us from further extending colored point to get, say - colored squares

```
>>> class ColoredSquare (ColoredPoint):
    size = 10
    def setSize (self, n):           # new method
        self.size = n
    def draw (self, screen, pos):    # overrides.
        ...
    def __str__ (...):
        return ColoredPoint.__str__(self) + ...
```

So we've built a hierarchy of classes:



Thus gradually extending the functionality

```
>>> p7 = ColoredSq(...)
>>> p7.move(50, 20)
>>> p7.setColor(20, 90, 90)
>>> p7.setSize(30, 30)
>>> points.append(p7)
>>> 50 (points)
```

Say you'd like text instead of shapes:

⑦

```
class ColoredText(ColoredPoint):
```

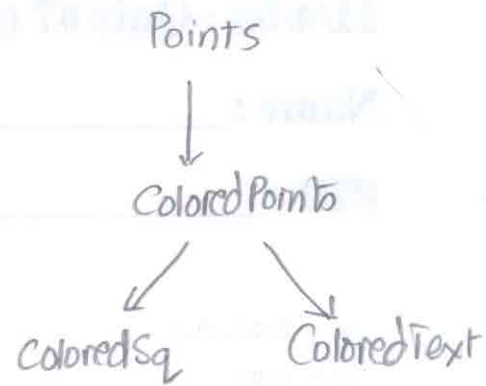
```
    def __init__(... text):
```

```
        ColoredPoint.__init__(...)
        self.setColor(color)
        self.setText(text)
```

```
    def draw():
```

```
    def __str__():
```

```
    def setText(self, text):
```



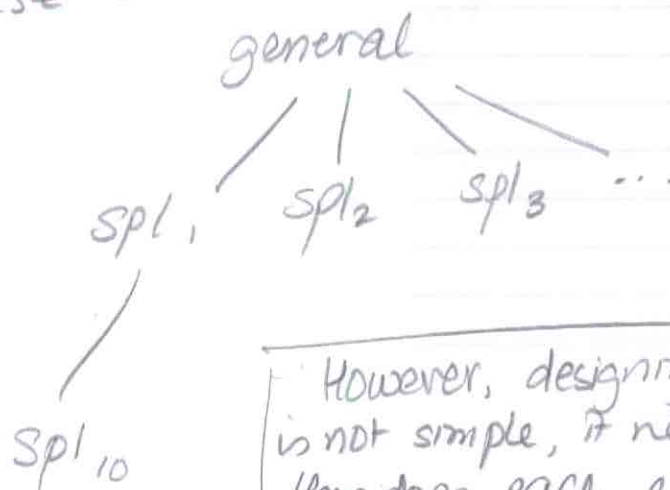
Can say:

```
>>> p8 = ColoredText(...)
```

```
>>> points.append(p8)
```

```
>>> do(points)
```

Moral, if you find yourself reimplementing the same code (eg move, jump) again again in a bunch of different places, you should think about gathering all that is common — making a superclass, and extending it for each spl case:



However, designing a proper hierarchy is not simple, it needs thought. How does each class extend the parent? Only shares code or also properties? More on this later...

MULTIPLE INHERITANCE : of course, we can take inheritance a step further. Suppose that you want to extend the Screensaver doodad, so that you have boxes with text in them.

Now, this is an object which shares or combines the elements of 2 of our classes - colored Box and Text Point

- we'd like to be able to change the dimensions of the box, the color of the box
- we'd like to be able to change the text ...

Thus, to avoid rewriting a bunch of code, we can do something like :

```
>>> class TextBox(coloredBox, TextPoint):  
    def draw(self, screen, pos):  
        ColoredBox.draw(self, screen, pos)  
        TextPoint.draw(self, screen, pos)  
    def __str__(self): ...
```

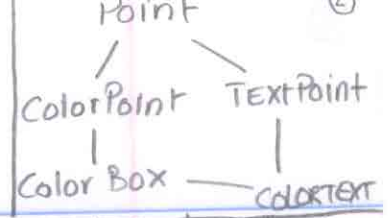
That's it ! The line : `class TextBox(coloredBox, TextPoint):` means that `TextBox` inherits from BOTH `coloredBox`, `TextPoint`.

So it gets :  

|                             |                   |     |                 |                  |
|-----------------------------|-------------------|-----|-----------------|------------------|
| setSize,<br>width<br>height | } from coloredBox | and | setText<br>text | } from TextPoint |
|-----------------------------|-------------------|-----|-----------------|------------------|

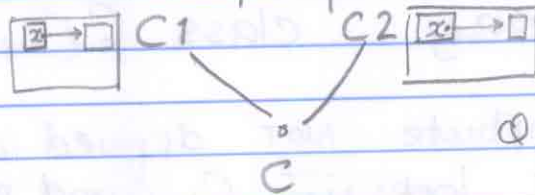
And it defines its own `draw` and `--str--` methods by combining those of the superclasses.

Notice that now the class hierarchy is not a tree but a DAG.



This seems like a great way to extend and compose code, but there's always a catch.

Problem 1: What if the same attribute (field, method... NAME) is defined in multiple parents (superclasses)?

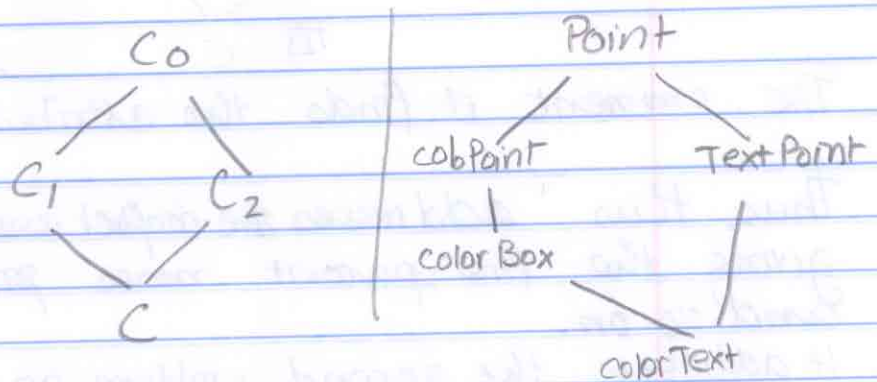


Q: What is  $C.x$ ?

In this case, how does one resolve the field in the subclass's namespace?

Of course, if the subclass has an overriding definition, then we are fine, but what if we don't have one?

Problem 2: The "diamond" Problem. It is often the case, as in our example, that several parents have a common root.



In this case, an attribute  $x$  may be inherited along several paths to  $Co$  — does that mean we have several "copies" of  $x$ ?

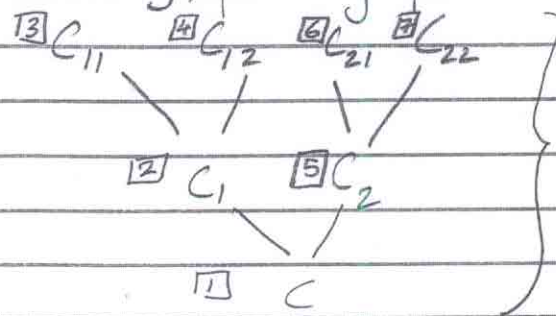
In our example  $x, y, --init--$  are all inherited from  $Point$ . Are there multiple  $x, y$ ?

Because of complications like this JAVA did away with multiple inheritance. In Python, there is a way to "solve" the problem, but the jury is out on how good the solution is:

When you say `class C(C1, C2, ...)`:

For any attribute NOT defined in (the namespace of) C python first looks up C<sub>1</sub> (and recursively, the parents of C<sub>1</sub>) if not found it then looks up C<sub>2</sub> (and recursively, the parents of C<sub>2</sub>) and so on.

In other words, Python does a DEPTH-FIRST-SEARCH in the class hierarchy, picking parents in the ORDER defined



Python searches the namespaces in the order:

C, C<sub>1</sub>, C<sub>11</sub>, C<sub>12</sub>, C<sub>2</sub>, C<sub>21</sub>, C<sub>22</sub>

The moment it finds the attribute, it stops.

Thus, this addresses the conflict issue of Problem 1 by giving the first parent more priority over the second and so on.

It addresses the second problem as there is only ONE copy.

However, this situation is not great as now one parent has more authority, greater custody if you will.

It may be that for certain methods we want one parent while for other methods we want the other parent...

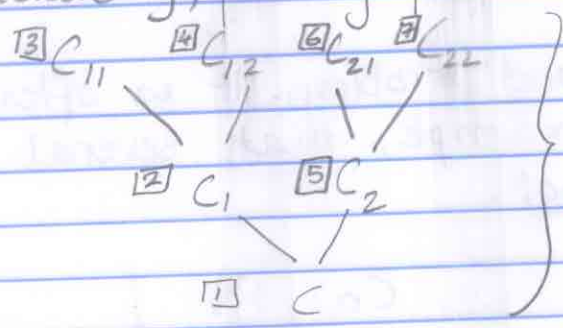
③

Because of complications like this, JAVA did away with multiple inheritance. In Python, there is a way to "solve" the problem, but the jury is out on how good the solution is:

When you say `class C(C1, C2, ...)`:

For any attribute NOT defined in (the namespace of) C, python first looks up C<sub>1</sub> (and recursively, the parents of C<sub>1</sub>, if not found it then looks up C<sub>2</sub> (and recursively, the parents of C<sub>2</sub> and so on.

In other words, Python does a DEPTH-FIRST-SEARCH in the class hierarchy, picking parents in the ORDER defined.



Python searches the namespaces in the order:  
C, C<sub>1</sub>, C<sub>11</sub>, C<sub>12</sub>, C<sub>2</sub>, C<sub>21</sub>, C<sub>22</sub>

The moment it finds the attribute, it stops.

Thus, this addresses the conflict issue of Problem 1 by giving the first parent more priority over the second and so on.

It addresses the second problem as there is only ONE copy.

However, this situation is not great as now one parent has more authority, greater custody if you will.

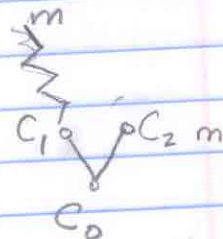
It may be that for certain methods we want one parent while for other methods we want the other parent...

(4)

Also, some may prefer BFS instead of DFS.

If  $C_2$  is the second parent and it has defined  $m$ , does it make sense to instead

use  $m$  defined in a remote ancestor of the first class  $C_1$ ?



Thus, the trouble with Multiple Inheritance is that it's hard to tell exactly which piece of code gets to run and so the jury is still out on whether or not it's more trouble than it is worth.

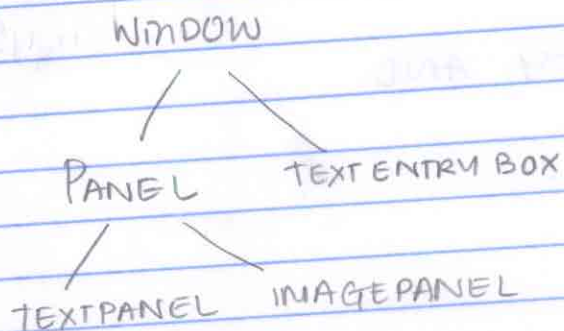
x ————— x ————— x ————— x

So, if you find yourself rewriting the same code in a bunch of places then you should stop hacking and figure out how you can reorganize the code so that the common parts can be reused via inheritance.

This process is called "refactoring" pulling out the common parts from a large piece of code where it's scattered inside.

In fact, well before you even get to the keyboard you have to — you MUST whip out a pencil and paper and figure out how things are organised and "compartmentalized" into modules / classes / structures — each with a well defined interface. How to do this is still a bit of an art — it requires great care.

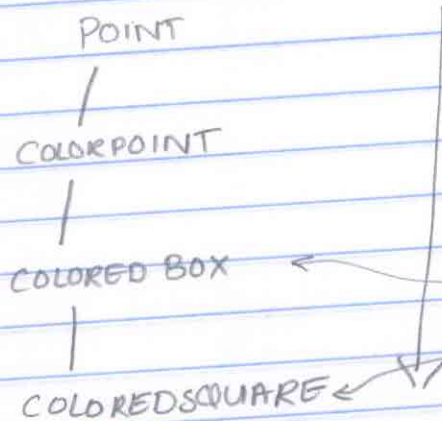
Often, your application will dictate what the organization will be. For example, if you are working on a GUI of some kind:



In each case, notice that the subclass is a "special case" of the "superclass" but also extends the functionality of the super class. That is, it supports all those operations and then adds some.

When designing a hierarchy, never forget this "extends" requirement. Ask yourself "how does each subclass extend the parent"?

For our example, suppose I wanted now a colored square



SPECIALIZATION.

But there is a problem!

has methods to change dim indep inherited, but it doesn't make sense for a square!

## MODULARITY AND ENCAPSULATION / HIDING.

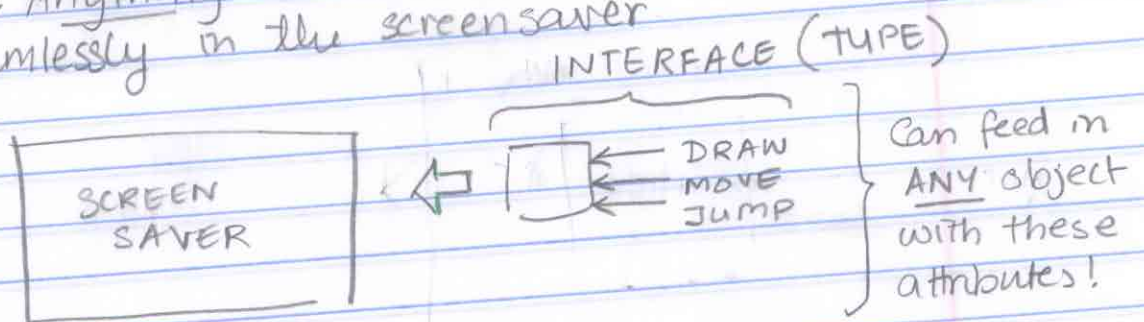
As we've seen with our example, a primary benefit of classes is to enable MODULAR code, where data and operations that are closely related can be bundled together.

This is similar to the structures we saw in SML.

We can now have several instances of Point, of several different kinds (point, box, text...), inheritance allows us to reuse common code.

Separately, we wrote a little screensaver that USES

the points, boxes etc. The only thing the screensaver requires is that each point have move, jump, draw methods. Anything with these attributes works seamlessly in the screensaver

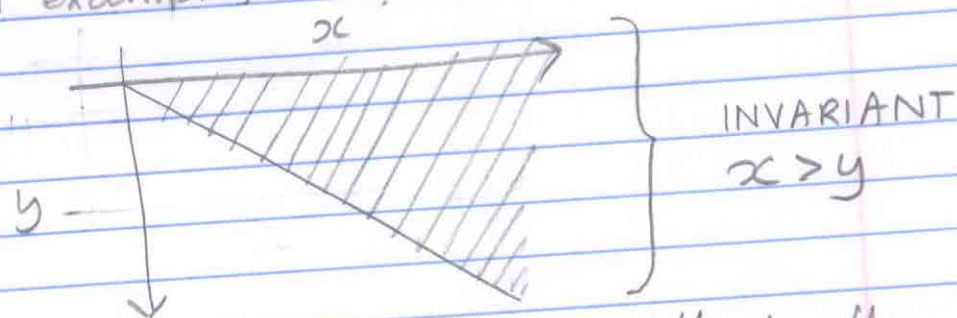


The Points need not even know the screensaver exists!  
The screensaver needs to know nothing else about points.

① Fewer interface req  $\Rightarrow$  More stuff can be plugged in

② If you don't know what's inside, you CANT BREAK what's inside!

For example, we may have a class of points constrained to be within a certain ZONE, i.e. stay within some boundaries, for example,



This gives an invariant for the object that the  $x > y$

Such a class can adapt its move method to prohibit or restrict moves that take object outside the zone, thus prohibiting the point from leaving the zone.

But then, to ensure the property (INVARIANT) holds, we must ensure that the ONLY way to mutate the object is by calling jump/move. In particular the attributes  $x, y$  must not be visible or at any rate WRITABLE outside the objects own methods!

Several other examples of such invariants:

|              |   |                          |
|--------------|---|--------------------------|
| BANK ACCOUNT | : | BALANCE must be $\geq 0$ |
| FILE HANDLER | : | REFERS TO EXISTING FILE  |

- In JAVA, you can make some fields PRIVATE, only visible from inside object. Same for methods, only callable from inside. Thus certain attributes are invisible from outside and compiler prohibits you from calling / reading / writing the attributes.
- To control how the fields are read / updated a good practice is to have "getter / setter" methods to read from, write to fields (instead of making them invisible).
- In ML we saw SIGNATURES which give a way to hide functions, data and even how types are implemented to get "abstract datatypes". Java has a similar mechanism called interfaces that we will see later.

PYTHON: Alas has no mechanism for encapsulation / information hiding. Everything is a namespace, so you can see what names are defined in the namespace, what they refer to. You can add names (and hence fields, methods to obj) you can overwrite fields. Its mayhem.

Can smash `__x`, `__y` or even the `move` / `jump` / `draw` methods. Convention is names starting w/ `__` are considered "hidden" but no compiler support. I expect this may change over time...

Because you can read the namespaces:

```
>>> p.__dict__
```

it makes it trivial to do things like reflection.  
If you don't know what that is: good. keep it that way. Its evil.

x ————— x ————— x ————— x

CLASSES == NAMESPACES == OBJECTS

However, there are some pleasing consequences of the fact that in Python classes are also just namespaces. This means that unlike in Java, classes are themselves OBJECTS! And hence, can be passed around, assigned to variables, sent and returned as parameters and be defined locally inside other namespaces. Just like ANY OBJECT.

Thus, we say that classes are first-class citizens in Python. They are "equal" to all other objects (Just like "functions" were first-class citizens in ML, "equal" i.e. can be passed around etc. like any value).

So why is this cool? For several reasons.

### (i) Classes as "FACTORIES":

Suppose you are writing a big GUI app and you want it to be platform independent so it should work on Linux / WINDOWS / MAC.

So you have 3 classes:

LinuxFrame  
WindowsFrame  
MacFrame

} One for each platform.

Now of course you will be creating frames all over the place. But which one?

Maybe you have a variable `os` and whenever you want a new frame you say

```
if (os == LINUX) :
    LinuxFrame()
elif (os == WINDOWS) :
    WINDOWSFRAME...
elif ...
```

→ Tedious. Could put into a method / global function, and call the function every time.

→ But theres still this ugly `os` global variable...  
→ or make `os` local and pass it around.... tedious

Instead we can say at the start of the program

```
if (os == LINUX) : Frame = LinuxFrame
elif (os == WINDOWS) : Frame = WINDOWSFRAME
elif (os == MAC) : Frame = MacFRAME
```

That's it! We can now just use `Frame` everywhere w/o worrying about platforms! So `Frame` acts as a factory generating the right kind of frames. Define factory once using platform and keeps rest of code clean!

- ② Can do interesting stuff with classes, such as "wrap" stuff around them to track extra info -  
E.g. suppose you want to count how many instances of a class have been made:

```
>>> def countedClass(C):  
    class CC(C):  
        __inst = 0  
        def __init__(self, *args, **dargs):  
            try:  
                C.__init__(self, *args, **dargs)  
            except: pass  
            CC.__inst = CC.__inst + 1  
    return CC
```