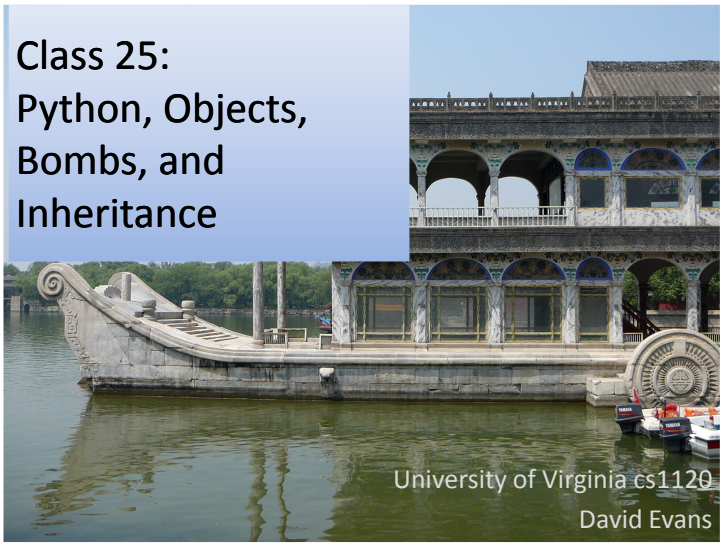
Class 25: Python, Objects, Bombs, and Inheritance



Menu

- PS5: end-auction! running time
- Lists in Python
- Inheritance

end-auction!

```
(define (end-auction!)
  (mmap
   (lambda (item-entry)
    (let ((item-name (list-get-element item-entry (table-field-number items 'item-name))))
     (let ((high-bid (get-highest-bid item-name)))
      (if (null? high-bid)
          (printf "No bids on ~a.~n" (list-get-element item-entry (table-field-number items
'item-name)))
          (printf "Cong
                (list-get
                item-na
                (list-get
           (list-get-eler
   (table-entries item
```

mmap: N applications of mapping procedure
list-get-element running time is in Θ(N)
get-highest-bid running time is in Θ(N)
Overall running time:
N * (Θ(N) + Θ(N)) = Θ(N^2)

WRONG: need to be careful what *N* means!

end-auction!

mmap: *t* applications of mapping procedure
t is the number of entries in the items table
list-get-element running time is in Θ(*N*) => constant time
N is the number of elements in input list:
here, number of fields in items: constant
get-highest-bid running time is in Θ(*N*) => Θ(*b*)
N is number of entries in bids table, *b*

Overall running time:
t * (*O*(1) + Θ(*b*)) = Θ(*tb*) where
t is number of items, *b* is number of bids

Python Lists

Built-in datatypes for both mutable lists [] and immutable tuples ()

```
>>> m = range(1, 1000)
>>> m[0]
1
>>> m[-1]
999
>>> len(m)
999
>>> m[1:]
[2, ..., 999]
```

range(1,1000) ~ (intsto 999)

m[0] ~ (mcar m)

Python lists can access any element in approx. constant time!

len is also constant time

m[1:] ~ (mcdr m) ?

Is m[1:] like mcdr?

```
>>> m1 = m[1:]
>>> m1[0]
2
>>> m[1]
2
>>> m1[0] = 3
>>> m[1]
2
```

m[1:] is a new copy of the list, except for the first element. Uses Θ(*N*) time and space!

Implementing list-map in Python

```
def schemish_list_map(f, p):
    if not p:
        return []
    else:
        return [f(p[0])] + schemish_list_map(f, p[1:])
```

Running time is in $\Theta(N^2)$ where N is number of elements in p.

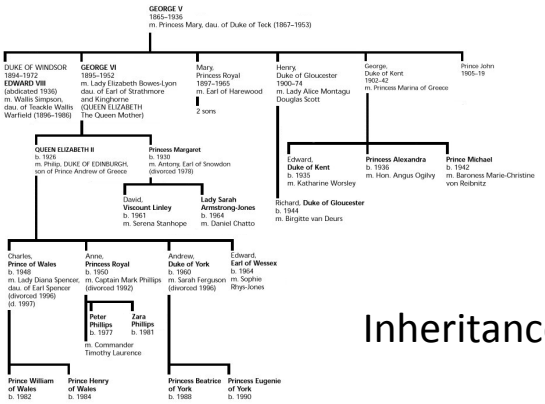
"Literal" translation...not a good way to do this.

Note: there is a built-in **map** in Python.

Pythonic Mapping

```
def mlist_map(f, p):
    for i in range(0, len(p)):
        p[i] = f(p[i])
    return p
```

Unlike the previous one, this mutates p.



Inheritance

There are many kinds of Dogs…

```
class Dog:
    def __init__(self, n):
        self.name = n
    def bark(self):
        print "wuff wuff wuff wuff"
```

```
class TalkingDog (Dog):
    def speak(self, stuff):
        print stuff
```

Subclasses

ClassDefinition ::= **class** *SubClassName* **(** *SuperClassName* **) :**
FunctionDefinitions

```
class TalkingDog (Dog):
    def speak(self, stuff):
        print stuff
```

TalkingDog is a **subclass** of Dog.
Dog is the **superclass** of TalkingDog.

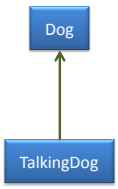
Every Dog has its Day

```
class Dog:
    def __init__(self, n):
        self.name = n
    def bark(self):
        print "wuff wuff wuff wuff"
```

```
class TalkingDog (Dog):
    def speak(self, stuff):
        print stuff
```

```
>>> bo = Dog('Bo')
>>> scooby = TalkingDog('Scooby Doo')
>>> scooby.speak('Ta-da!')
Ta-da!
>>> bo.speak('Ta-da!')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    bo.speak('Ta-da!')
AttributeError: Dog instance has no attribute 'speak'
>>> scooby.bark()
wuff wuff wuff wuff
```

Speaking about Inheritance



Inheritance is using the definition of one class to define another class.

TalkingDog ***inherits*** from Dog.

TalkingDog is a ***subclass*** of Dog.

The ***superclass*** of TalkingDog is Dog.

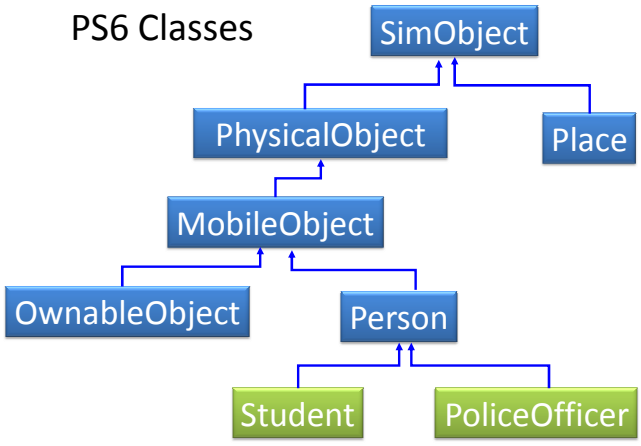
These all mean the same thing.

PS6

Make an adventure game programming with objects

Many objects in our game have similar properties and behaviors, so we use inheritance.

PS6 Classes

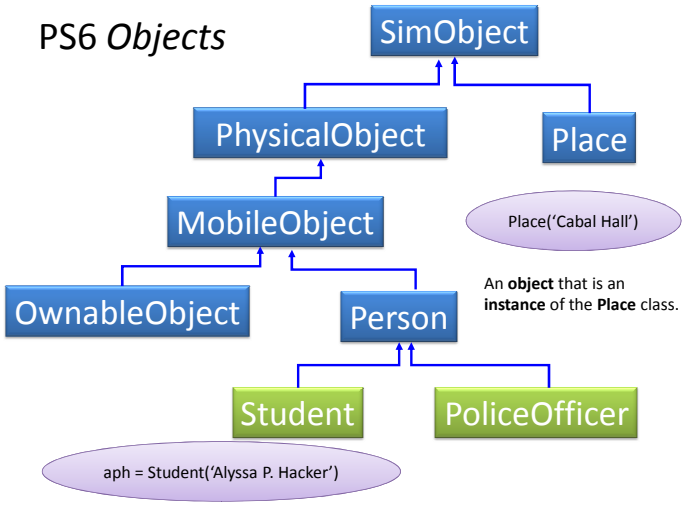



```
class SimObject:
    def __init__(self, name):
        self.name = name
    def note(self, msg):
        print "%s: %s" % (self, msg)
```

```
class PhysicalObject (SimObject):
    def __init__(self, name):
        SimObject.__init__(self, name)
        self.location = None
    def install(self, loc):
        self.note ("Installing at " + str(loc))
        self.location = loc
        loc.add_thing(self)
```

```
class MobileObject (PhysicalObject):
    def change_location(self, loc):
        self.location.remove_thing(self)
        loc.add_thing(self)
        self.location = loc
```

PS6 *Objects*



Object-Oriented Summary

- An **object packages state and procedures.**
- A **class** provides procedures for making and manipulating a type of object.
- The procedures for manipulating objects are called **methods.** We invoke a method on an object.
- **Inheritance allows one class to refine and reuse the behavior of another. This is a good thing.**
- Friday: Excursion on Exponential Growth
  - **Please ready Tyson essay before Friday!**