

PYTHON CLASSES and INHERITANCE

(download slides and .py files from Stellar to follow along!)

6.0001 LECTURE 8

LAST TIME

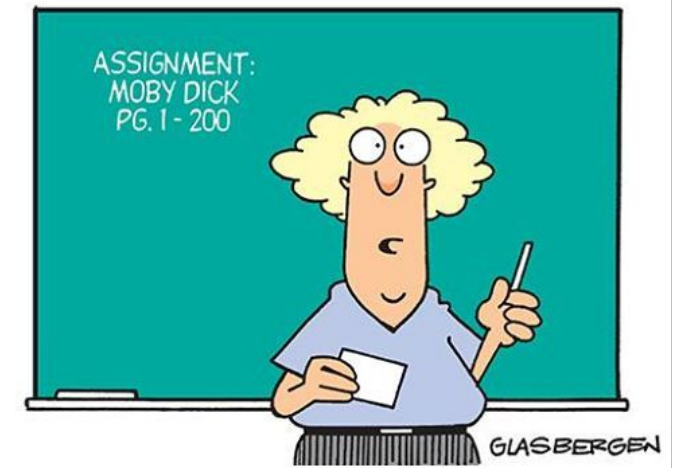
- Abstract data types using classes
- `Coordinate` example
- `Fraction` example

TODAY

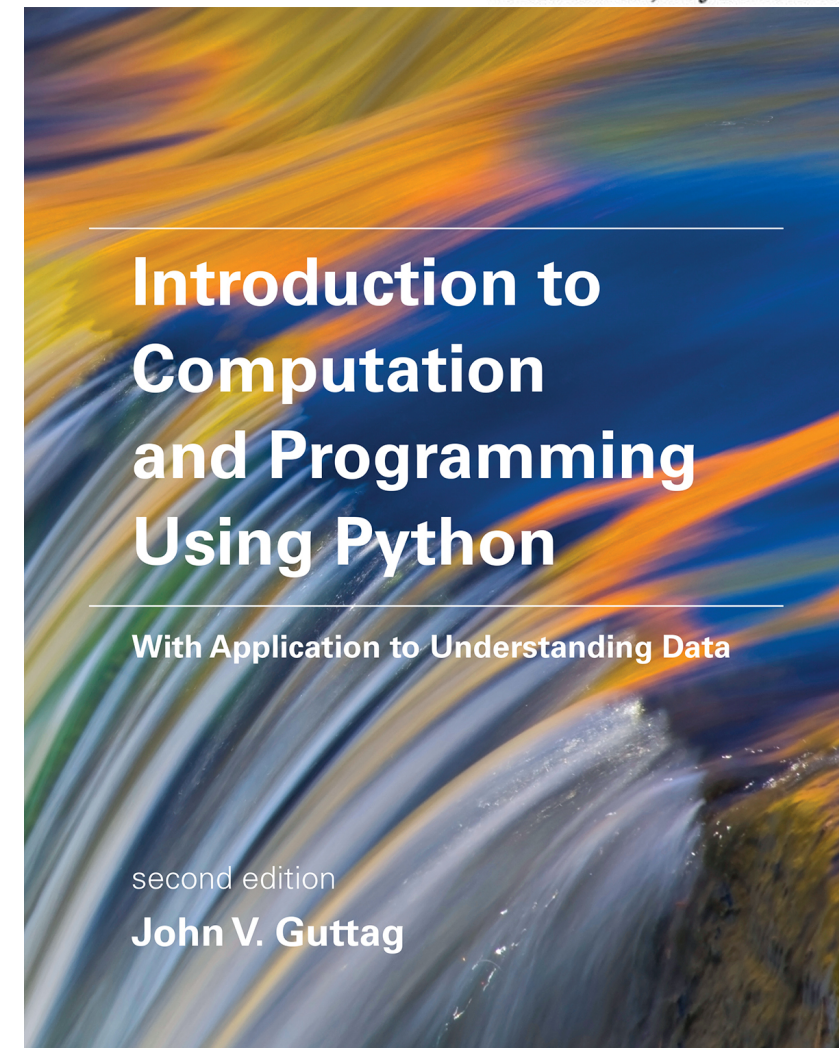
- Review classes
- More details on classes, class variables
- Inheritance and hierarchies of classes

Assigned Reading

- Today
 - 8.2
 - 9.1 – 9.2
- Next lecture
 - 9.3



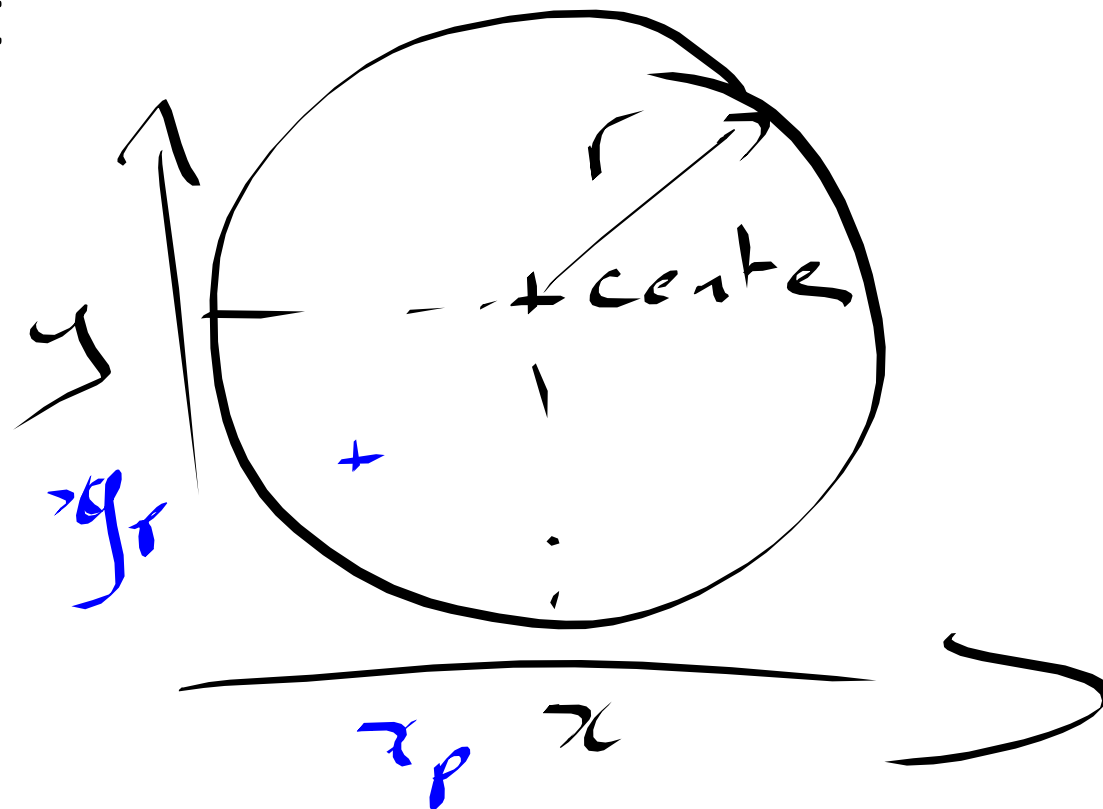
"I don't like to give a lot of homework over the weekend, so just read every other word."



https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

Why objects (or structured types)?

- Example: manipulate Geometric circles, point in circle test



is_inside_circle $(x_c, y_c, r_c, x_p, y_p)$

with structured types or classes

circle

~~the~~ center

r

→ fraction is_inside (self, p)

Coordinate

x

y

dirty list based solution

is-inside-circle (L1, L2)

L1 [2] not as readable as

circle-radius

Why objects (or structured types)?

- Example: manipulate Geometric circles

```
class Circle(object):  
    def __init__(self, center, radius):  
        self.center = center  
        self.radius = radius  
    def __str__(self):  
        return "circle: "+str(self.center)+", "+str(self.radius)  
    def is_inside(self, point):  
        return point.distance(self.center)<self.radius
```

- Note that we can use our own classes (Coordinate) in the creation of new classes

Why objects (or structured types)?

- Example: manipulate Geometric circles
 - Without structured type:

Why objects and structured types?

- Example: manipulate Geometric circles
 - Without structured type:
`is_in_circle(x_c, y_c, r, x_p, y_p)`
 - Lots of parameters, hard to read
- We could try to package info into lists, but we would have to remember which index corresponds to coordinates vs. radius

THE POWER OF OBJECT ORIENTED PROGRAMMING

- **Bundle together objects** that share
 - common attributes and
 - procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object versus how to use an object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes (and on top of our own classes)

Another instance of a virtuous cycle – just as defining procedures lets us create new procedures and treat as if built-in, we can create classes and treat as if built in to Python

IMPLEMENTING THE CLASS

USING THE CLASS

- Write code from two different perspectives

Implementing a new object type with a class

- **Define** the class
- Define **data attributes** (WHAT IS the object)
- Define **methods** (HOW TO use the object)

Using the new object type in code

- Create **instances** of the object type
- Do **operations** with them

Class captures common properties and behaviors

Instances have specific values for attributes

CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

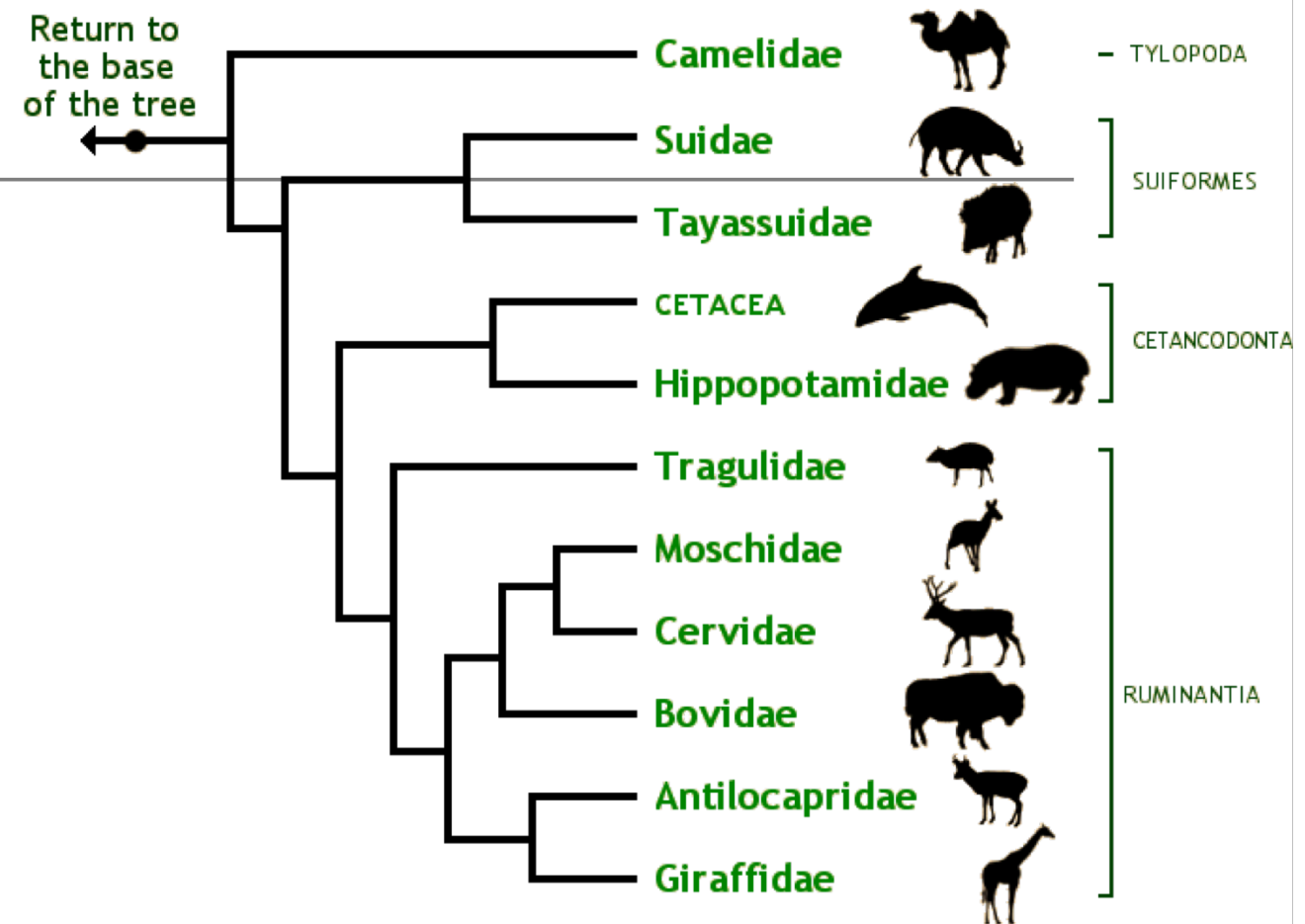
- Class name is the **type**
`class Coordinate(object)`
 - Class is defined generically
 - Use `self` to refer to some instance while defining class
`(self.x - self.y)**2`
 - `self` is a parameter to methods in class definition
 - Class defines data and methods **common across all instances**
- Instance is **one specific object**
`coord = Coordinate(1,2)`
 - Data attribute values vary between instances
`c1 = Coordinate(1,2)`
`c2 = Coordinate(3,4)`
 - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects
 - Instance has the **structure of the class**

Classes & Instances

Classes (~type)
are similar to species

- define a “template”
- can be hierarchically organized

Instances (a.k.a. members)
are similar to individuals



CREATING INSTANCES (Recap)

- Usually when creating an instance of an object type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

Method is another name for a procedural attribute, or a procedure that “belongs” to this class

CREATING INSTANCES (Recap)

- Usually when creating an instance of an object type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

When calling a method of an object, Python always passes the instance as the first argument. By convention, we use `self` as the name of the first argument of methods.

CREATING INSTANCES (Recap)

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

When calling a method of an object, Python always passes the instance as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The “.” operator accesses an attribute of an object, so `__init__` defines two attributes for new object: `x` and `y`.

CREATING INSTANCES (Recap)

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

```
c = Coordinate(3, 4)  
origin = Coordinate(0, 0)  
print(c.x, origin.x)
```

The expression
`classname(values...)`
creates a new object of type
`classname` and then calls its
`__init__` method with the new
object and `values...` as the
arguments. When the method is
finished executing, Python returns
the initialized object as the value.

Note that don't provide
argument for `self`, Python does
this automatically

Getters and setters

- For good encapsulation and abstraction, it is often advised to not presume how internal data is encoded
 - You should not use directly `coord.x = 1.0` from outside the class
 - Instead, implement getters and setters
 - `Coordinate.set_x`
 - `Coordinate.get_x`

~~directly
Coordinate
p.x = 1~~ `p(0,0)` bad style

Getters and setters

- For good encapsulation and abstraction, it is often advised to not presume how internal data is encoded
 - You should not use directly `coord.x = 1.0` from outside the class
 - Instead, implement getters and setters
 - `Coordinate.set_x`
 - `Coordinate.get_x`

~~directly
Coordinate
p.x = 1~~ `p(0,0)` bad style

Motivation for getters and setters (aka accessors)

- Information hiding, abstraction
- Adding validation
- Support undo, history tracking
- Debugging insertion point
- Separate control of get vs. set
- Later in this lecture, with inheritance:
change semantics for child class
- For fans of controversy, there are counter arguments

PYTHON NOT GREAT AT INFORMATION HIDING



- Allows you to **access data** from outside class definition in an instance

```
print(p.x)
```

- Allows you to **write to data** from outside class definition to an instance

```
p.x = 2
```

- Allows you to **create data attributes** for an instance from outside class definition

```
a.new_field = "whatever"
```

- It's usually considered **NOT GOOD STYLE** to do any of these!

Example of encapsulation and data hiding

- Credit card class that stores operation history

Default argument

- For any function, including class function attribute

```
def f(x, y=1):  
    return x+y  
print(f(1, 4))  
print(f(2))
```

- When argument is not provided, use default
- Can only be done at end of argument list
- Kind of similar to default arguments of range(::)

Default argument examples

- `def f(x, y, debug_mode_on=False)`
- `def f(x, y, z=0.0)`
- `Class Person(object):`
 `def __init__(first_name, last_name, title="")`
- `def sqrt(x, precision=1e-6)`
- `Def save_jpeg(image, quality=80)`

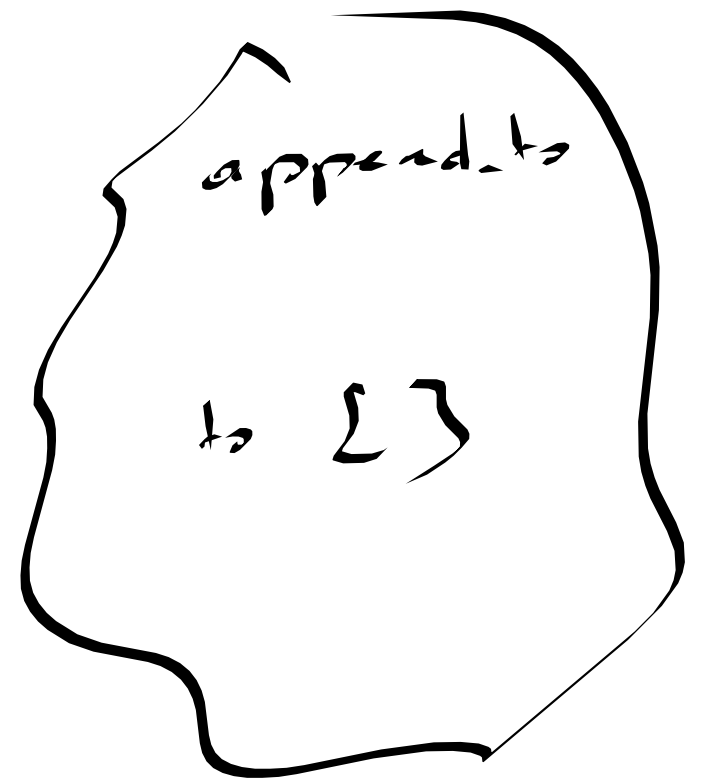
Careful

- Careful with mutable default arguments (lists, dictionaries)
 - DON'T MUTATE THEM
- They get created only once

```
def append_to(element, to=[]):  
    to.append(element)  
    return to
```

```
my_list = append_to(12)  
print(my_list)
```

```
my_other_list = append_to(42)  
print(my_other_list)
```



better solution:
default of None
+ test inside function

CREDIT CARD example

- Operations
 - spend(amount)
 - pay_bill(amount)
 - monthly_update: apply interest rate
- data:
 - balance, rate
- Initialize with:
 - yearly rate in %, set balance to zero

CREDIT CARD Class

```
class CreditCard(object):
    def __init__(self, yearly_rate_in_percent):
        self.balance = 0
        self.yearly_rate_in_percent = yearly_rate_in_percent

    def get_balance(self):
        return self.balance

    def spend(self, amount):
        self.balance += amount

    def pay_bill(self, amount):
        self.balance -= amount

    def monthly_update(self):
        interests = self.get_balance()*self.yearly_rate_in_percent/12.0/100.0
        self.balance += interests

    def __str__(self):
        return "amount due: "+str(self.get_balance())
```

*#alternative that stores the monthly multiplier
follows the same specifications
example of abstraction where internal representation
can be changed without affecting external behavior*

```
class CreditCard(object):
    def __init__(self, yearly_rate_in_percent):
        self.balance = 0
        self.monthly_interests_fraction = yearly_rate_in_percent/12.0/100.0

    def get_balance(self):
        return self.balance

    def spend(self, amount):
        self.balance += amount

    def pay_bill(self, amount):
        self.balance -= amount

    def monthly_update(self):
        interests = self.get_balance()*self.monthly_interests_fraction
        self.balance += interests

    def __str__(self):
        return "amount due: "+str(self.get_balance())
```

*#alternative that stores a list of operations
follows the same specifications*

```
class CreditCard(object):  
    def __init__(self, yearly_rate_in_percent):  
        self.operations = []  
        self.monthly_interests_fraction = yearly_rate_in_percent/12.0/100.0  
  
    def get_balance(self):  
        return sum(self.operations) + compute interest  
  
    def spend(self, amount):  
        self.operations.append(amount)  
  
    def pay_bill(self, amount):  
        self.operations.append(-amount)  
  
    def monthly_update(self):  
        interests = self.get_balance()*self.monthly_interests_fraction  
        self.operations.append(interests)  
  
    def __str__(self):  
        return "amount due: "+str(self.get_balance())
```


5 Minute Break

Debugging your pset

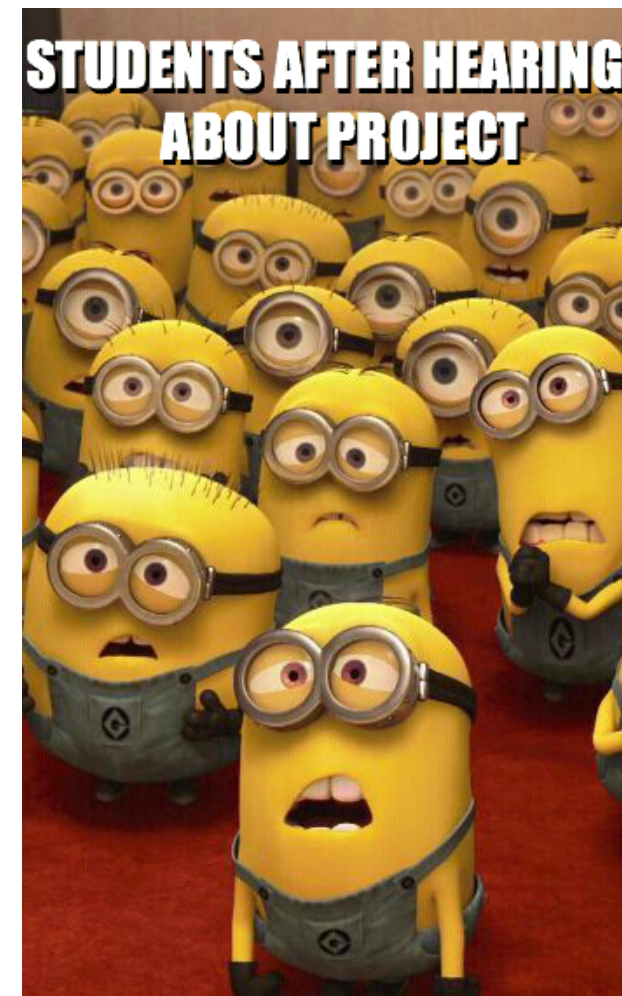


MY CODE DOESN'T WORK

LETS CHANGE NOTHING AND RUN IT AGAIN



STUDENTS AFTER HEARING ABOUT PROJECT



INHERITANCE



- Idea: “extend” a class with new/different capability
 - Reuse code from parent class in child class
 - Create sets of classes with similar capabilities
- Motivating examples
 - Checking account class
gets extended into saving account class
 - Credit card card extends to reward card
 - Drawable class
gets extended into various shapes (point, line, circle)
 - Clickable User Interface element
parent of buttons, windows, text field
 - Output stream gets extended into file, print

REWARDS CARD

- Credit card
- Reward card
 - same as credit card
 - but also accumulates miles each time we spend money
 - add more behavior/information (miles)
 - augments the behavior of the credit card “spend” operator

HOW TO INHERIT

- Define parent class

```
class CreditCard(object):
```

child :

```
class RewardCard(CreditCard):
```

HOW TO INHERIT

- Define parent class

```
class CreditCard(object):
```

Define child class:

```
class RewardCard(CreditCard):
```

Automatically inherits all data and function attributes of parent

Can be extended with

- new data attributes

- new function attributes

- Modification of existing behavior
(function overloading)

CONSTRUCTOR

- We still need to initialize balance (or operations) and monthly_interests_fraction (or yearly)
- We could copy-paste the code from CreditCard
 - but it would be dirty and make the code harder to maintain
- Solution: call the parent function in the child function
 - using syntax:
ParentClass.function(self, ...)

*Rev. def __init__
CreditCard.__init__(self)*

CONSTRUCTOR

```
class RewardCard(CreditCard):  
    def __init__(self, yearly_rate_in_percent, miles_per_dollar):  
        CreditCard.__init__(self, yearly_rate_in_percent) → call  
        self.miles_per_dollar = miles_per_dollar  
        self.miles = 0
```

new data attributes

parent constructor

- Note that here we use self in the call
 - yes, Python can be confusing
- Notice the dot notation after CreditCard?
 - Yes, classes are objects too!

NEW DATA ATTRIBUTE

■ Parent class

```
class CreditCard(object):  
    def __init__(self, yearly_rate_in_percent):  
        self.balance = 0  
        self.yearly_rate_in_percent = yearly_rate_in_percent
```

■ Child class:

```
class RewardCard(CreditCard):  
    def __init__(self, yearly_rate_in_percent, miles_per_dollar):  
        CreditCard.__init__(self, yearly_rate_in_percent)  
        self.miles_per_dollar = miles_per_dollar  
        self.miles = 0
```

RewardCard has all attributes of credit card
(balance, yearly_rate_in_percent)

It has two extra one: miles, miles_per_dollar

NEW BEHAVIOR

```
def get_miles(self):  
    return self.miles
```

MODIFIED BEHAVIOR

- Change spend to increment miles
- In this case, call parent version of spend to preserve parent behavior
- Augment with extra behavior

```
def spend(self, amount):  
    CreditCard.spend(self, amount)  
    self.miles += amount * self.miles_per_dollar
```

call parent function

SUBSTITUTION PRINCIPLE

```
def spend(self, amount):  
    CreditCard.spend(self, amount)  
    self.miles += amount * self.miles_per_dollar
```

- When modifying a method, make sure that code that works correctly with the parent method still works with the modified child

How does it work?

- When using dot notation (data, function)
 - E.g. `card.spend()`
- Python tries to find it at level of child class
- If it can't, it looks for it in parent class
 - Potentially recursively if the parent has a parent
- Note: if you want to augment the old behavior, you have to call it explicitly (here `CreditCard.spend()`)

```
def spend(self, amount):  
    CreditCard.spend(self, amount)  
    self.miles += amount * self.miles_per_dollar
```

Good abstraction

```
class RewardCard(CreditCard):
    def __init__(self, yearly_rate_in_percent, miles_per_dollar):
        CreditCard.__init__(self, yearly_rate_in_percent)
        self.miles_per_dollar = miles_per_dollar
        self.miles = 0
    def spend(self, amount):
        CreditCard.spend(self, amount)
        self.miles += amount * self.miles_per_dollar
    def get_miles(self):
        return self.miles
    def __str__(self):
        return CreditCard.__str__(self)+"; reward "+str(self.get_miles())+" miles"
```

Our RewardClass still works if we change the implementation of CreditCard

- balance vs. operations
- yearly vs. monthly rate

Questions?

POLYMORPHISM

- Functions like `spend` are called polymorphic
- They can function on different datatypes
- Inheritance and function overloading is one mechanism for polymorphism
- Other examples:
 - Most arithmetic operators
 - `len()`
 - `min()`

 - `Draw()`, `area()`
 - `Mouse_up()`
 - `Update_state()`

HIERARCHIES

Animal



Cat

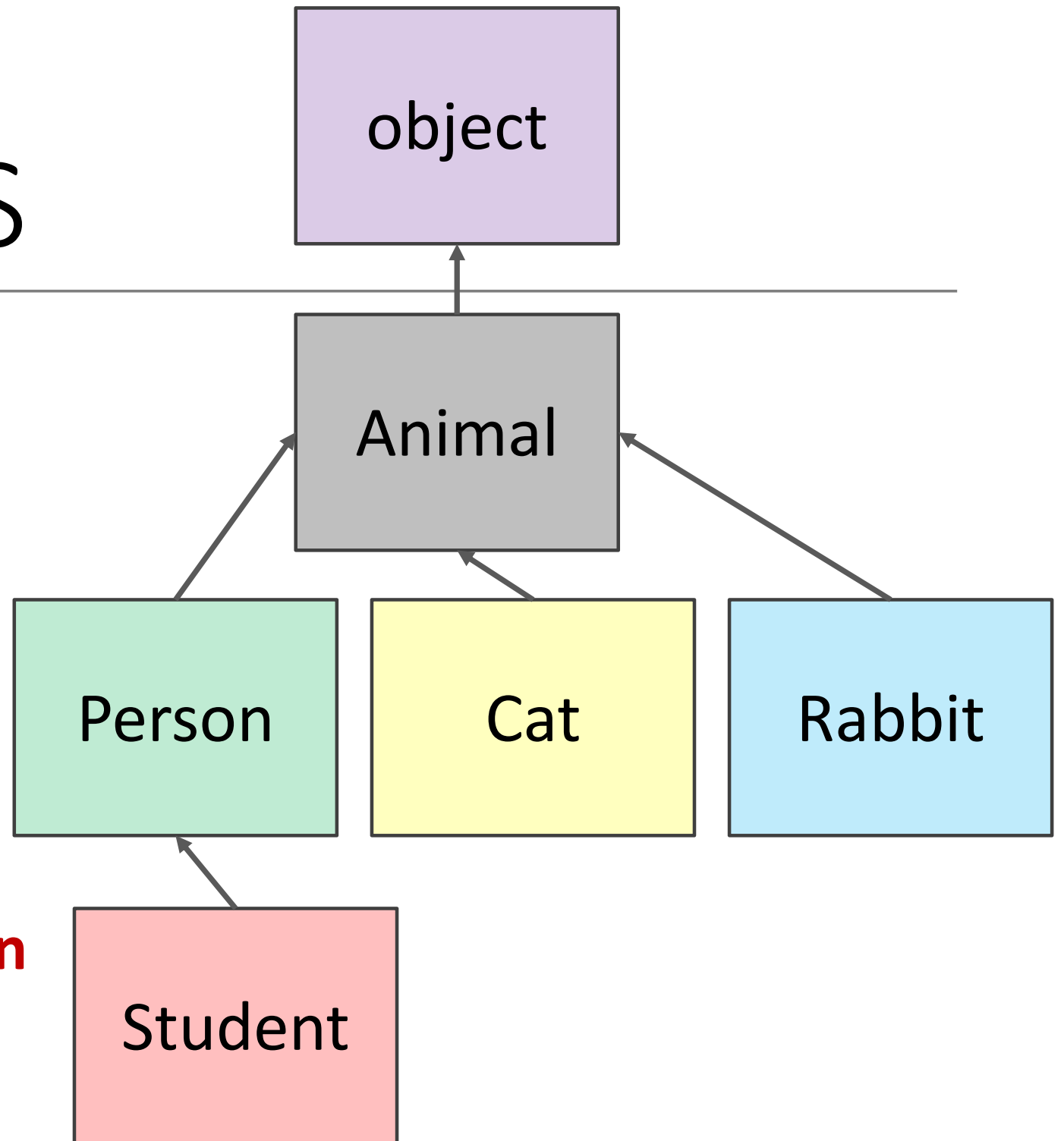


Rabbit



HIERARCHIES

- **Parent class**
(superclass)
- **Child class**
(subclass)
 - **Inherits** all data and behaviors of parent class
 - **Add** more **information**
 - **Add** more **behaviors**
 - **Override** behavior



Class variables

- So far we have used instance variables
 - Each object instance has a different value for the data attribute
- New:
class variables are shared for all members of a class
 - But not across parents and children
 - Declared inside class definition but not in `__init__`

Example of class variables

```
class Animal(object):  
    #class variables will be shared by all instances  
    species = ""  
    counter = 0  
    #class variable will be shared by all instances  
    def __init__(self, name, age):  
        self.age = age  
        self.name = name  
        #increment counter to count all animals  
        (Animal.counter += 1)  
    def __str__(self):  
        return self.species+" "+str(self.name)+":"+str(self.age)
```

```
class Rabbit(Animal):  
    species = "Rabbit"  
    #insert here whatever it is rabbits do
```

```
class Fox(Animal):  
    species = "Fox"  
    #insert here whatever it is foxes do
```

INSTANCE VARIABLES

vs

CLASS VARIABLES

- we have seen **instance variables** so far in code
- specific to an instance
- created for **each instance**, belongs to an instance
- used the generic variable name `self` within the class definition

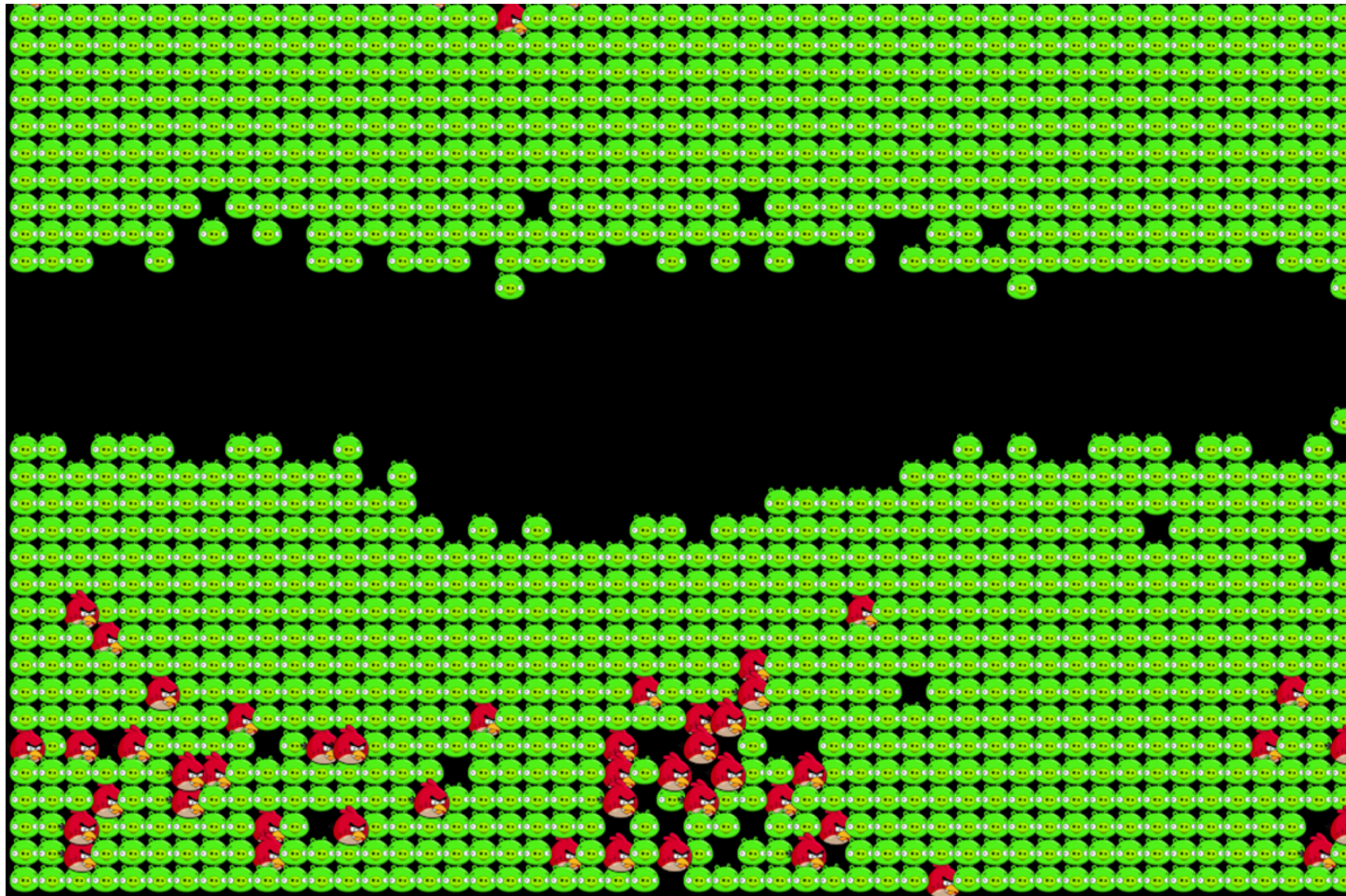
```
self.variable_name
```

- introduce **class variables** that belong to the class
- defined inside class but outside any class methods, outside `__init__`
- **shared** among all objects/instances of that class

THE POWER OF OBJECT ORIENTED PROGRAMMING

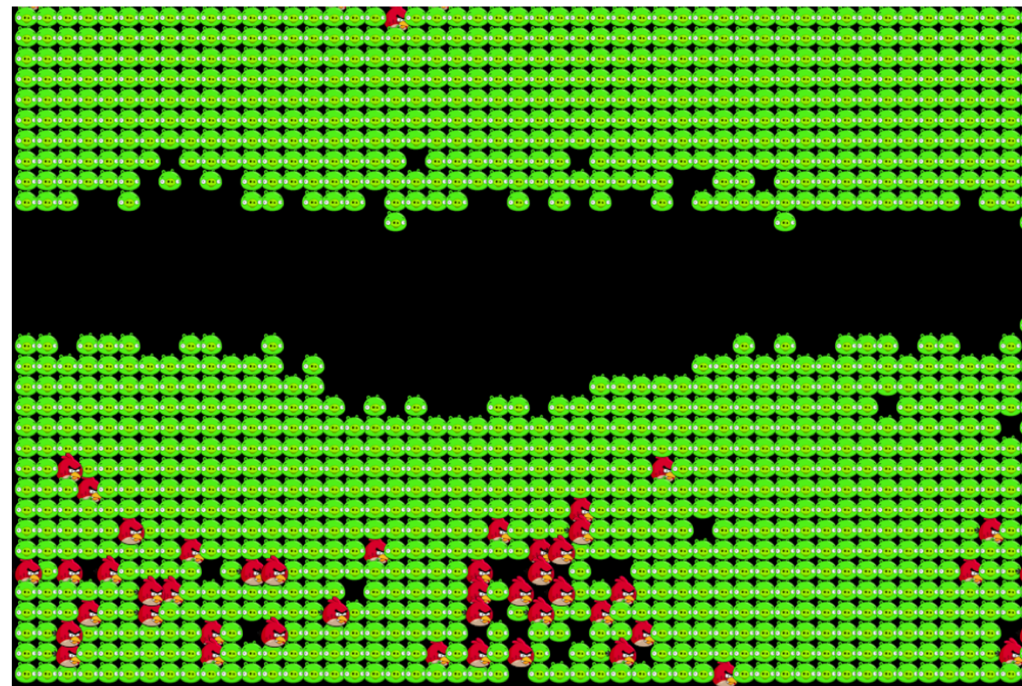
- **Bundle together objects** that share
 - common attributes and
 - procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object versus how to use an object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes

Predator-Prey Simulation



PREDATOR/PREY SIMULATION

- Animals move on discrete grids
- Reproduce at given rates by parthenogenesis
- Predators need to eat preys to survive



Predator-Prey OOP

■ Animals

- **Behavior:** move, reproduce, eat
- Maintain **information:** location, time without eating
- Many instances of the same class
- Hierarchy:
Animal
 - Prey
 - Predator
- share properties/behavior, differ in others

LOOK AT CODE

```
class Animal:
    baseProbaToReproduce=0.0  ← class variable
    def __init__(self, x, y, world):
        self.x=x
        self.y=y
        self.world=world
        world.addAnimal(self)
        self.dead=False
    def simulate(self): pass;
    def reproduce(self, x, y): pass
    def die(self):
        self.world.setCell(self.x, self.y, None)
        self.dead=True
        self.world.listOfAnimals.remove(self)
    def move(self, x, y):
        self.world.setCell(self.x, self.y, None)
        self.x=x
        self.y=y
        self.world.setCell(x, y, self)
    def moveAndReproduceProba(self, x, y):
        oldX, oldY=self.x, self.y
        self.move(x,y)
        LFriends=self.world.neighborsOfType(self.x, self.y, type(self))
        if rnd.random()<self.baseProbaToReproduce*len(LFriends):
            offspring=self.reproduce(oldX, oldY)
```

```
class Prey(Animal):
    baseProbaToReproduce=2.0/8.0
    def reproduce(self, x, y):
        offspring=Prey(x, y, self.world)
    def simulate(self):
        if self.dead: return
        LFree=self.world.freeNeighbors(self.x, self.y)
        LFriends=self.world.neighborsOfType(self.x, self.y, Prey)
        if len(LFree)>0:
            i=rnd.randint(len(LFree))
            self.moveAndReproduceProba(LFree[i][0], LFree[i][1])
```

```

class Predator(Animal):
    baseProbaToReproduce=0.1/8.0
    baseProbaToCatchPrey=0.3
    maxTimeWithoutEating=5
    def __init__(self, x, y, world):
        Animal.__init__(self, x, y, world)
        self.timeWithoutEating=0
    def display(self):
        displayUtilities2.displayBird(self.x*scale, self.y*scale)
    def reproduce(self, x, y):
        offspring=Predator(x, y, self.world)
    def simulate(self):
        if self.dead: print('predator dead:'), self.dead
        if self.dead: return
        self.timeWithoutEating=self.timeWithoutEating+1
        L=self.world.listOfNeighbors(self.x, self.y)
        LFree=self.world.freeNeighbors(self.x, self.y)
        LFriends=self.world.neighborsOfType(self.x, self.y, Predator)
        LPrey=self.world.neighborsOfType(self.x, self.y, Prey)
        if rnd.random()< self.baseProbaToCatchPrey*len(LPrey): #eat a prey
            i=rnd.randint(len(LPrey))
            prey=self.world.cell(LPrey[i][0], LPrey[i][1])
            prey.die()
            self.moveAndReproduceProba(LPrey[i][0], LPrey[i][1])
            self.timeWithoutEating=0
            return
        if self.timeWithoutEating>self.maxTimeWithoutEating: #starvation
            self.die()
            return
        if len(LFree)>0:
            i=rnd.randint(len(LFree))
            self.moveAndReproduceProba(LFree[i][0], LFree[i][1])

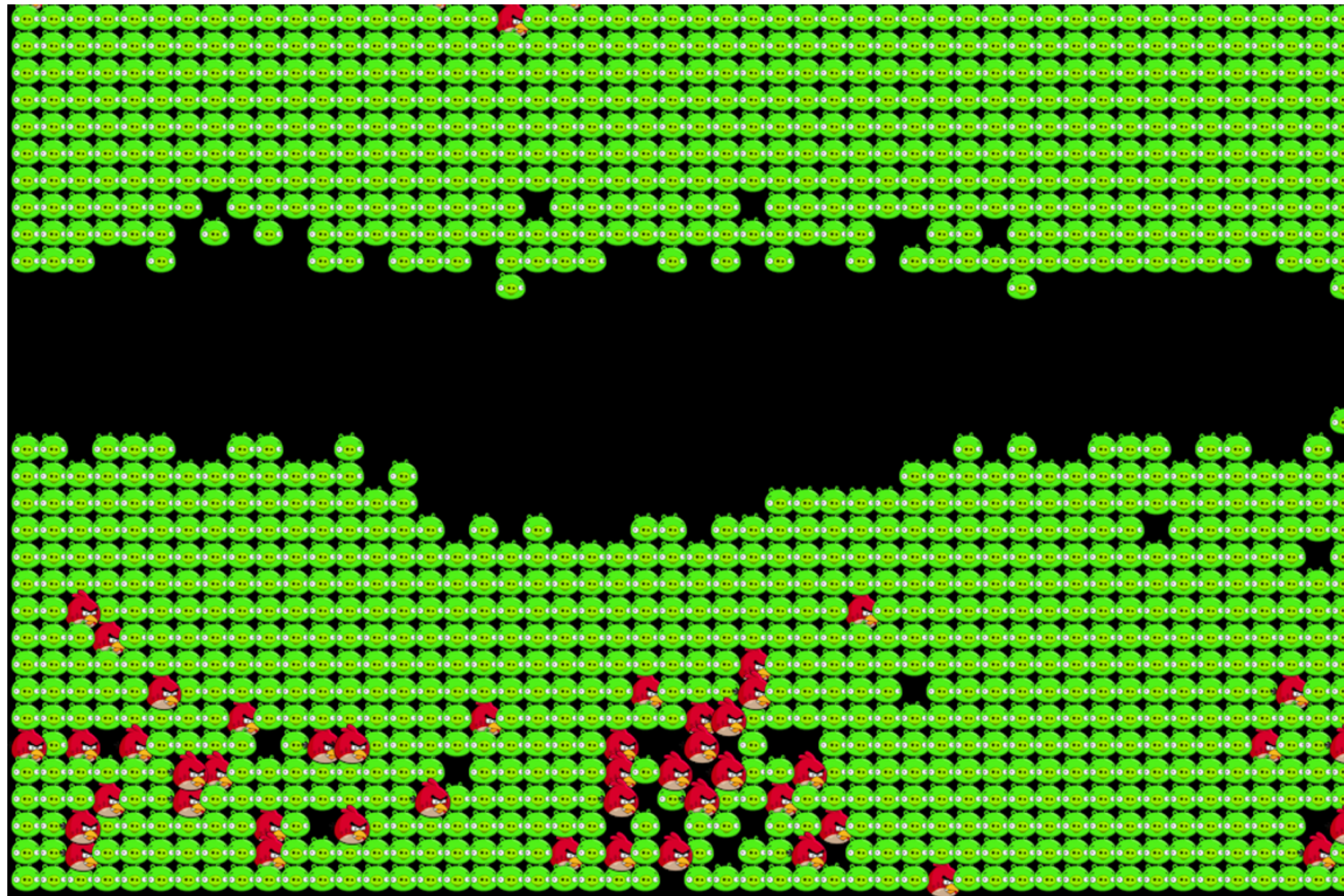
```



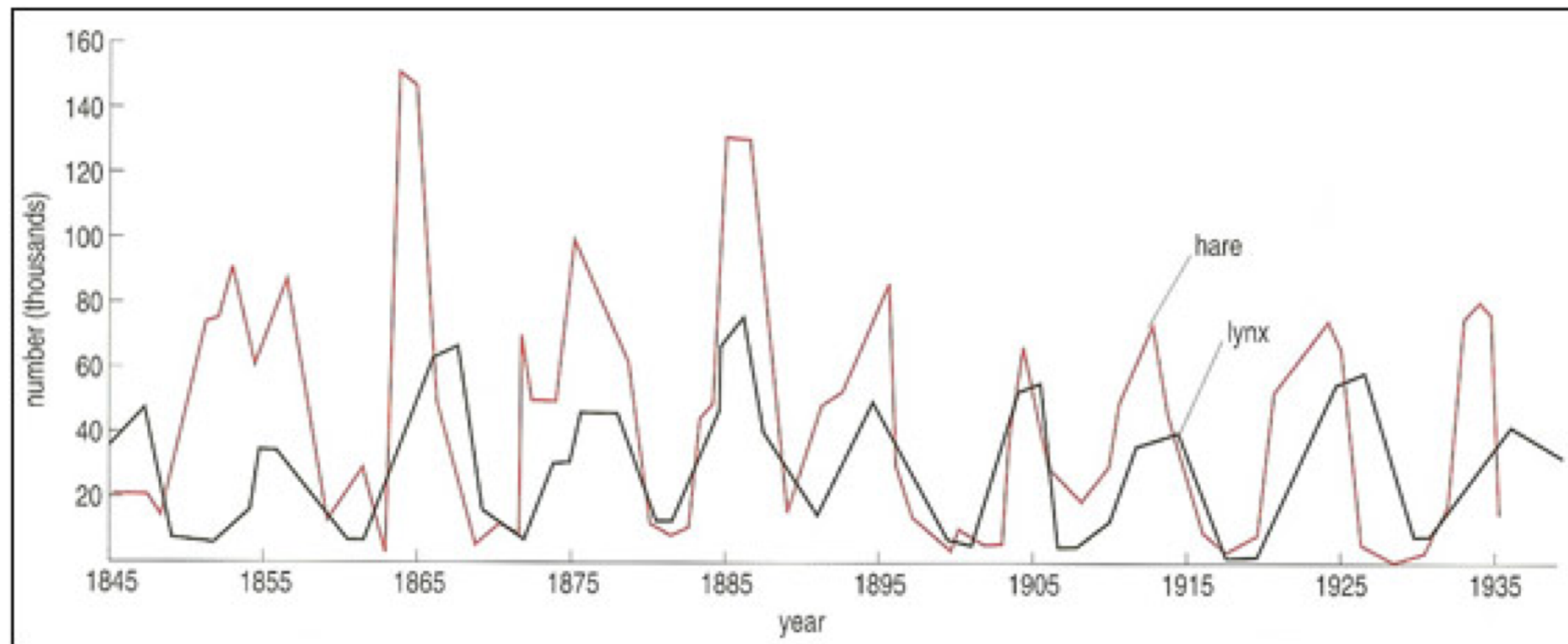
```
class World:
    def __init__(self, width, height):
        self.width=width
        self.height=height
        self.worldMap=[]
        for y in xrange(height):
            row=[]
            for x in xrange(width):
                row.append(None)
            self.worldMap.append(row)
        self.listOfAnimals=[]
```

....

SIMULATE



Lynx vs. hare



Differential equation: Lotka-Volterra

<http://labspace.open.ac.uk/mod/oucontent/view.php?id=474950&printable=1>

<http://mathnathan.com/2010/12/predator-prey-model/>

Other applications

Gang territories

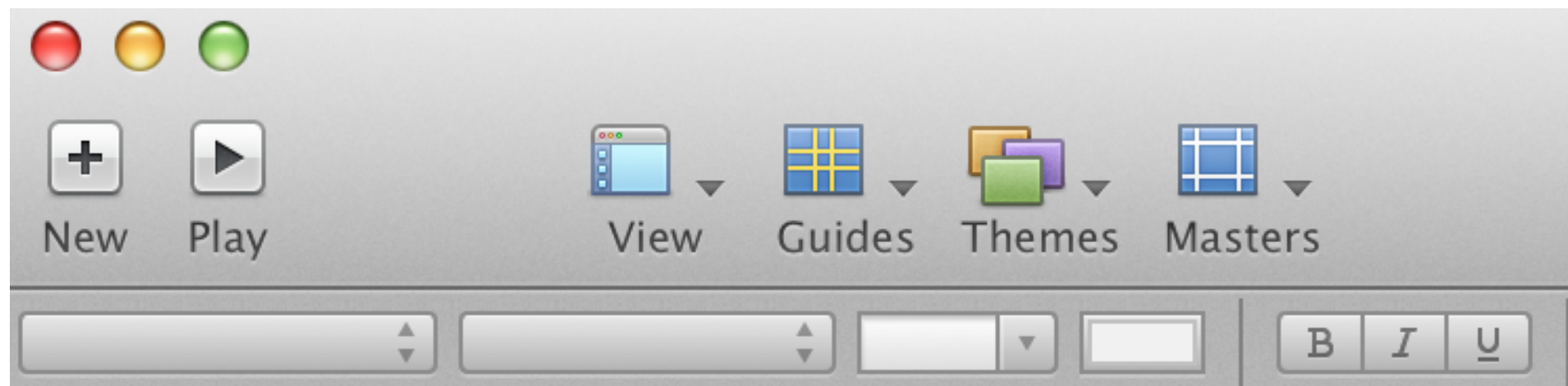
- <http://www.scientificamerican.com/podcast/episode.cfm?id=predator-prey-equations-govern-gang-12-07-02>

Stock market and bubbles

- <http://www.catataxis.com/index.php/the-cultural-theory-of-risk/>
- http://en.wikipedia.org/wiki/Competitive_Lotka%E2%80%93Volterra_equations

Object Oriented Programming (OOP) biggest successes

- Graphical User Interface
 - Many types of buttons/window/sliders
 - state: on/off, active or not, pushed or not
 - some behavior is similar
 - some differs



- Agent-based simulation, games

OBJECT ORIENTED PROGRAMMING LIMITATIONS

- Inheritance can create rigidity and complexity
- Changes to parent classes can still break a lot of things
- The world or your problem cannot always be decomposed into a hierarchy
- <http://wiki.c2.com/?ArgumentsAgainstOop>
- <https://github.com/raganwald-deprecated/homoiconic/blob/master/2010/12/oop.md>
- <https://medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53>
- <https://cacm.acm.org/magazines/2010/9/98017-objects-never-well-hardly-ever/fulltext>

IN THE END

- Structured types are good
- Object are useful
- Grouping things is useful. OOP is one way of grouping
- Polymorphism is great (but may not need inheritance)
- Inheritance is sometimes useful, sometimes dangerous, but it's also used a lot (e.g. for GUIs)