

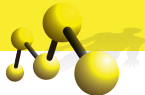
# Defining Your Own Types

Luis Pedro Coelho

Programming for Scientists

September 16, 2012





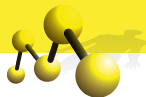
## Built-in Types

- ① lists
- ② dictionaries
- ③ strings
- ④ ...



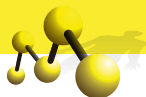
## What's a Type

- ① A domain of values
- ② A set of methods (functions)



## List

- ① Domain: lists
- ② Functions: `L.append(e)`, `L.insert(idx,e)`, ...
- ③ Operators: `L[0]`, `'Rita' in L`

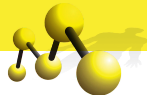


## List

- ① Domain: lists
- ② Functions: `L.append(e)`, `L.insert(idx,e)`, ...
- ③ Operators: `L[0]`, `'Rita' in L`

## Integer

- ① Domain:  $\dots, -2, 1, 0, 1, 2, \dots$
- ② Operators: `A + B`, ...



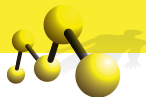
Object-oriented programming languages allow us to define new types.

# Motivating Example



## Simple Population Simulation

- ① We want to simulate a bacterial population.
- ② Our environment is a single float  $e$ .
- ③ Each bacterium has two characteristics: adaptation  $\alpha$  and mutation rate  $\sigma$ .
- ④ The smaller the difference  $|\alpha - e|$ , the better an bacterium is adapted to the world.
- ⑤ When an bacterium reproduces, its offspring has adaptation  $\alpha + \mathcal{N}(0, \sigma)$
- ⑥ At each iteration:
  - ① Bacteria die with a probability given by  $\lambda \exp(-\lambda|\alpha - e|)$
  - ② Bacteria that survive, sometimes reproduce.



## Bacterium Class

We define a bacterium class, with two values:

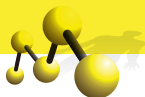
- ① adaptation: its current adaptation value
- ② sigma: its variability parameter

and two methods:

- ① `P_dead(enviro)`: make a stochastic decision on whether the bacterium dies
- ② `reproduce()`: make a new bacterium, derived from current one



# Using our Bacteria



```
population = [Bacterium(random(),random())
               for i in xrange(nr_initia_bacteria)]
for i in xrange(max_iters):
    bi = 0
    while bi < len(population):
        if population[bi].P_dead(enviro) < random():
            del population[bi]
        else:
            bi += 1
    N = len(population)
    for bi in xrange(N):
        if random() < p_reprod:
            population.append(population[bi].reproduce())
    if N >= max_population:
        shuffle(population)
        while len(population) >= max_population:
            population.pop()
```

# Using our Bacteria



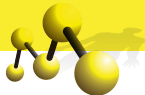
```
...
DeltaAdaptation = [math.abs(envirom-b.adaptation)
                    for b in population]
Sigmas = [b.sigma for b in population]
hist(Sigmas)
```

# Classes As Logical Units



## Class

A class aggregates data and functions that belong together.



## Interface

Functions:

- ➊ Constructor: Takes the initial adaptation value and sigma.
- ➋ `P_dead(enviro)`: Probability of dying in this environment.
- ➌ `reproduce()`: Return a new Bacterium.

Data elements:

- ➊ `adaptation`: Current adaptation.
- ➋ `sigma`: Current sigma.

```

class Bacterium(object):
    """
    Bacterium

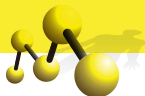
    ...
    """
    def __init__(self, adaptation, sigma):
        self.adaptation = adaptation
        self.sigma = sigma

    def P_dead(self, environ):
        """
        prob = bact.P_dead(environ)

        ...
        """
        return L*math.exp(-abs(self.adaptation-environ)*L)
    def reproduce(self):
        """ ... """
        return Bacterium(self.adaptation +
                           normalvariate(0, self.sigma),
                           self.sigma)
    ...

```

# Calling Methods

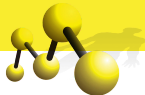


## Defining a method

```
class Bacterium(object):  
    ...  
    def method(self, arg1, arg2):  
        """ ... """  
    ...
```

## Calling a Method

```
anim = Bacterium(random(), random())  
  
anim.method(arg1, arg2)
```



## OOP

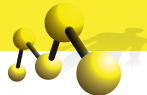
**Aggregation** organise functions & data into classes.

**Encapsulation** hide information inside methods.

**Polymorphism** re-use code for multiple types.

**Inheritance** re-use code from one class to build another.

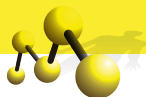
# Simulation of Changing Bacteria



Why should only adaptation change? Why not  $\sigma$  too?



# Evolving Bacterium



```
class EvolveSigmaBacterium(object):
    """ ... """
    def __init__(self, adapt, sigma, sigmafact):
        self.adaptation = adapt
        self.sigma = sigma
        self.sigmafact = sigmafact

    def P_dead(self, environ):
        """ ... """
        return L*math.exp(-
            math.abs(self.adaptation-environ)*L)

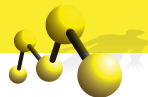
    def reproduce(self):
        """ ... """
        return EvolveBacterium(
            self.adaptation + normalvariate(0, self.sigma),
            self.sigma + normalvariate(0, self.sigma*self.sigmafact),
            self.sigmafact)
```

```

population = [EvolveSigmaBacterium(random(),random(),0.5)
               for i in xrange(nr_initial_bacteria)]
for i in xrange(max_iters):
    bi = 0
    while bi < len(population):
        if population[bi].P_dead(enviro) < random():
            del population[bi]
        else:
            bi += 1
    N = len(population)
    for bi in xrange(N):
        if random() < p_reprod:
            population.append(population[bi].reproduce())
    if N >= max_population:
        shuffle(population)
        while len(population) >= max_population:
            population.pop()

```

# Mixing populations



We can have a mixed population of  $\sigma$ -fixed and  $\sigma$ -changing bacteria!

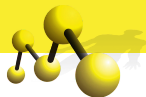
```

population = [EvolveSigmaBacterium(random(),random(),0.5)
               for i in xrange(nr_inital_bacteria//2)] + \
               [Bacterium(random(),random())
               for i in xrange(nr_inital_bacteria//2)]

for i in xrange(max_iters):
    bi = 0
    while bi < len(population):
        if population[bi].P_dead(envirom) < random():
            del population[bi]
        else:
            bi += 1
    N = len(population)
    for bi in xrange(N):
        if random() < p_reprod:
            population.append(population[bi].reproduce())
    if N >= max_population:
        shuffle(population)
        while len(population) >= max_population:
            population.pop()

```

# Polymorphism

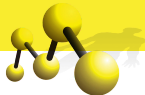


## Type Polymorphism

Code is **polymorphic** if it can use different types without change

# Duck Typing





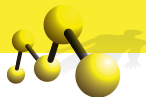
## OOP

**Aggregation** organise functions & data into classes.

**Encapsulation** hide information inside methods.

**Polymorphism** re-use code for multiple types.

**Inheritance** re-use code from one class to build another.



## Typical examples

- Actors in a simulation.
- File-like objects.
- Widgets.
- ...





The code for `EvolveSigmaBacterium` is very similar to the code for `Bacterium`.

```
class EvolveSigmaBacterium(Bacterium):
```

```
    '''
```

A type of Bacterium, where  $\sigma$  (which controls the rate of adaptative mutation) is itself subject to mutation (subject to  $\sigma * \sigma_{\text{fact}}$ ).

Methods

```
-----
```

```
    * Constructor:
```

```
    * P_dead(enviro): inherited from Bacterium
```

```
    * reproduce():
```

```
    '''
```

```
def __init__(self, adaptation, sigma, sigmafact):
```

```
    Bacterium.__init__(self, adaptation, sigma)
```

```
    self.sigmafact = sigmafact
```

```
def reproduce(self):
```

```
    ''' ... '''
```

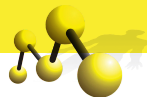
```
    return EvolveSigmaBacterium(
```

```
        self.adaptation + normalvariate(0, self.sigma),
```

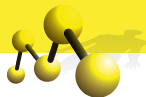
```
        self.sigma + normalvariate(0, self.sigma * self.sigmafact),
```

```
        self.sigmafact)
```

# Lyskov Substitution Principle

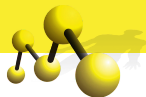


If D inherits from C, then  
you should be able to use D anywhere you previously used C.



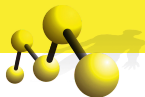
If D inherits from C, then  
D should behave-like C.

# New-Style vs. Old-Style Classes



```
class Bacterium(object):  
    ...
```

Are we inheriting from **object**?



## OOP

**Aggregation** organise functions & data into classes.

**Encapsulation** hide information inside methods.

**Polymorphism** re-use code for multiple types.

**Inheritance** re-use code from one class to build another.