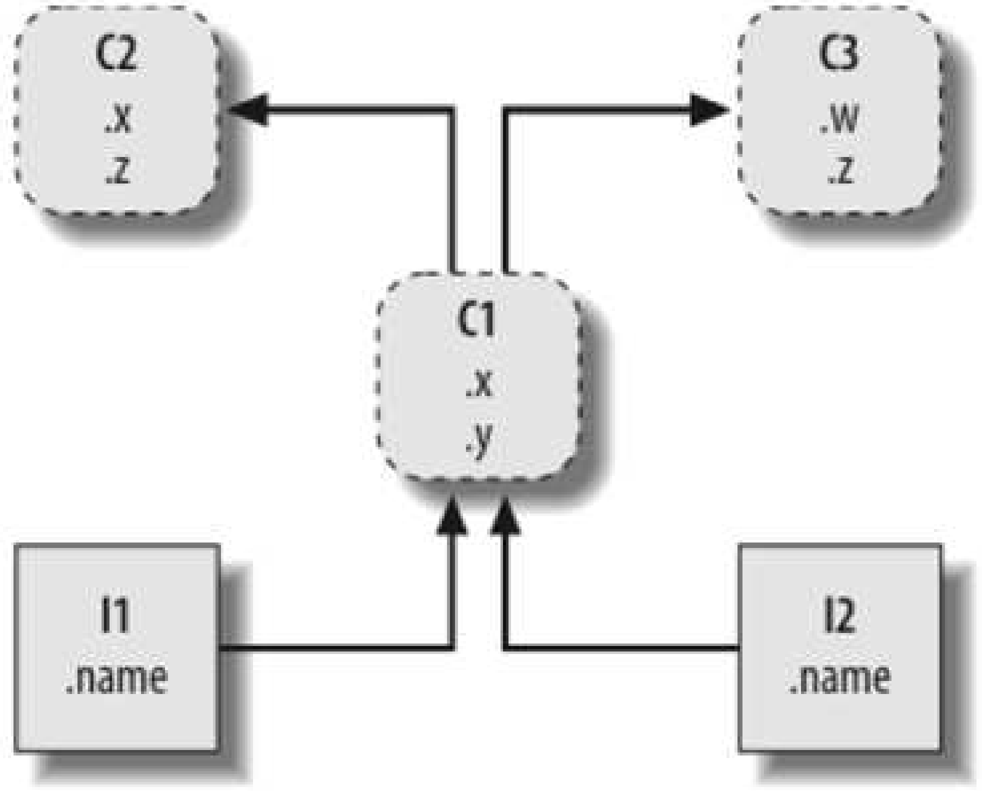
Orientação a Objetos

BSI – UFRPE
Prof. Gustavo Callou
gcallou@gmail.com

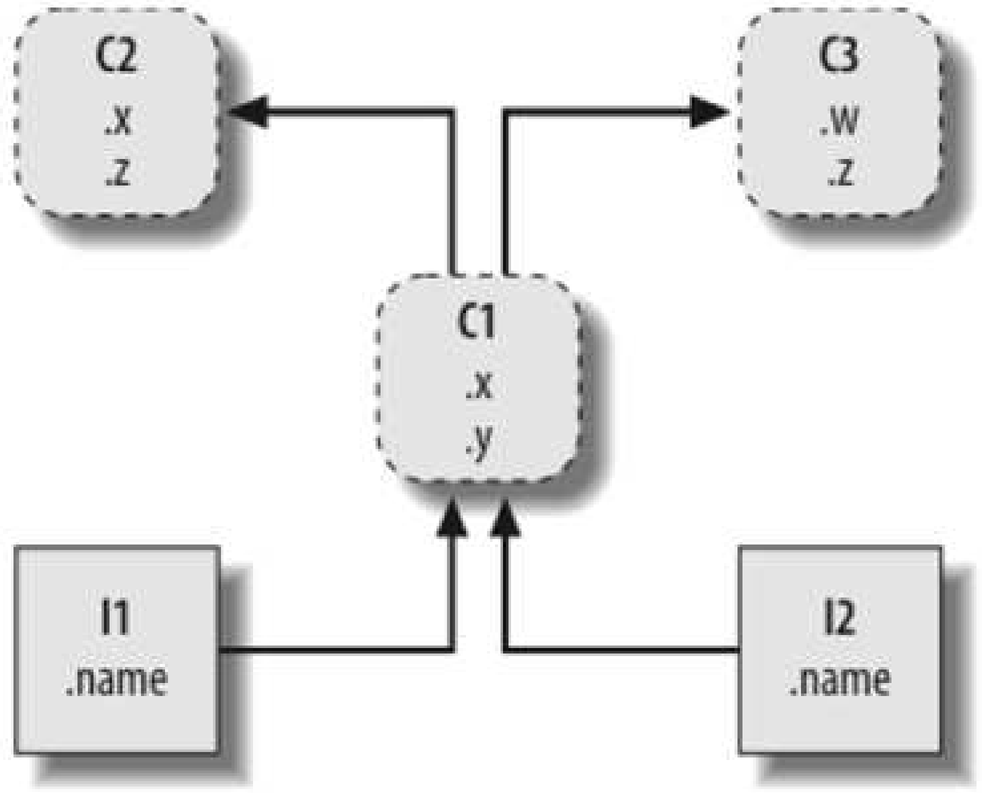
Herança

- Como funciona Herança em Python?
- Como localizar I2.w?



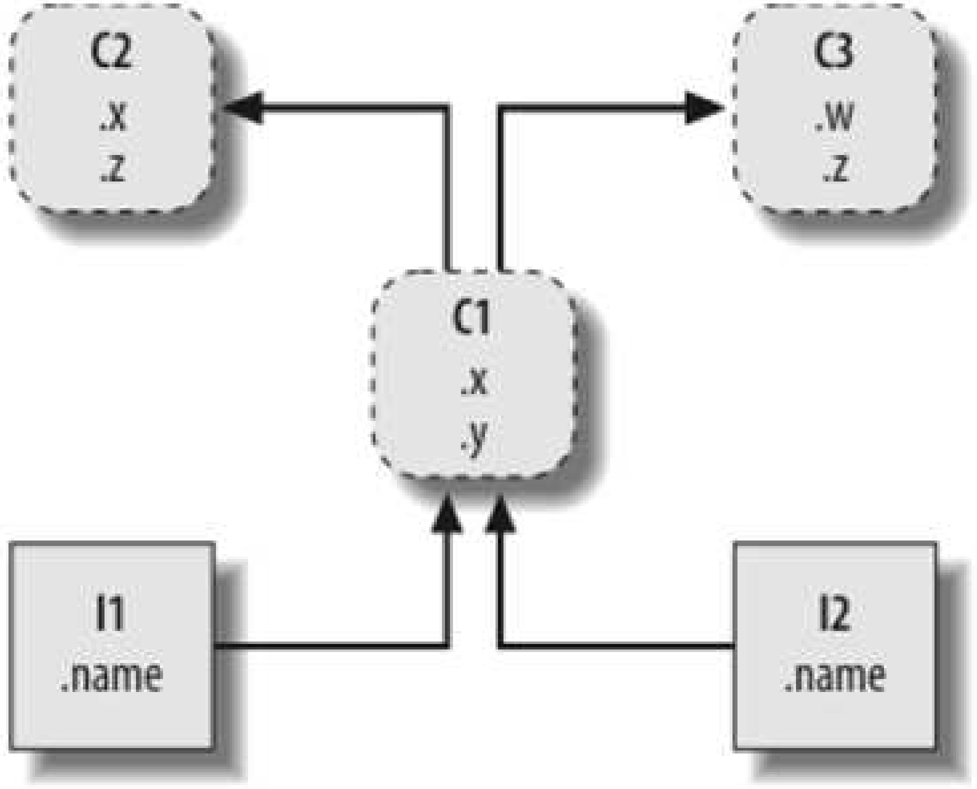
Herança

- Como funciona Herança em Python?
- Como localizar I2.w? I2, C1, C2, C3



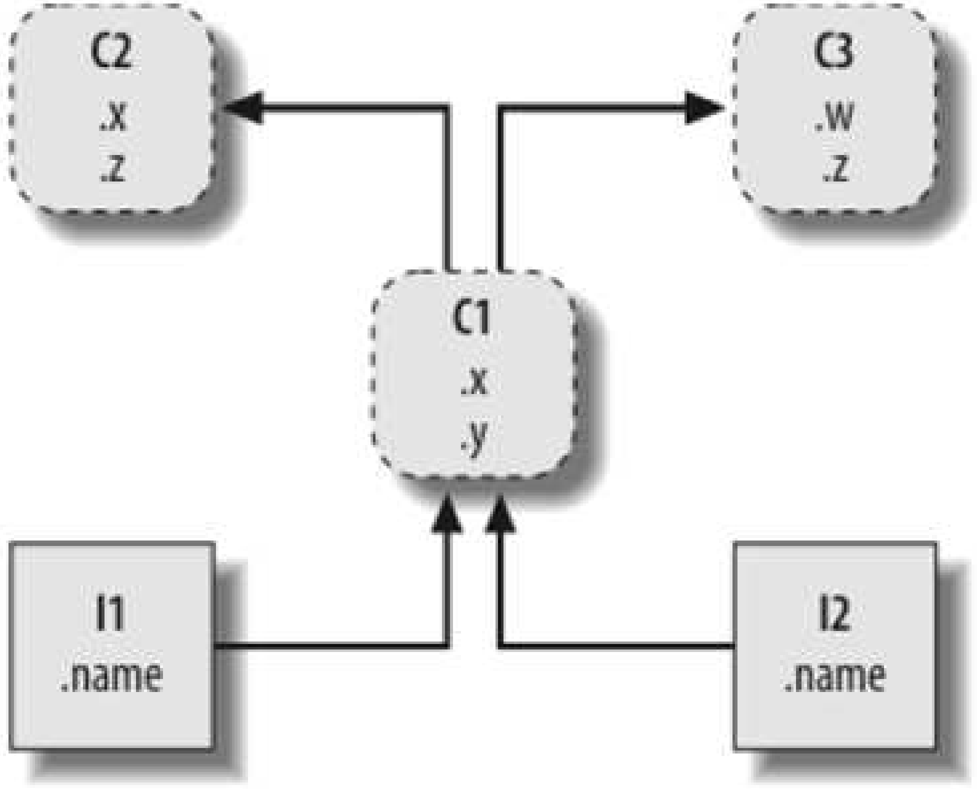
Herança

- I1.x e I2.x?
- I1.y e I2.y?
- I1.z e I2.z?
- I2.name?



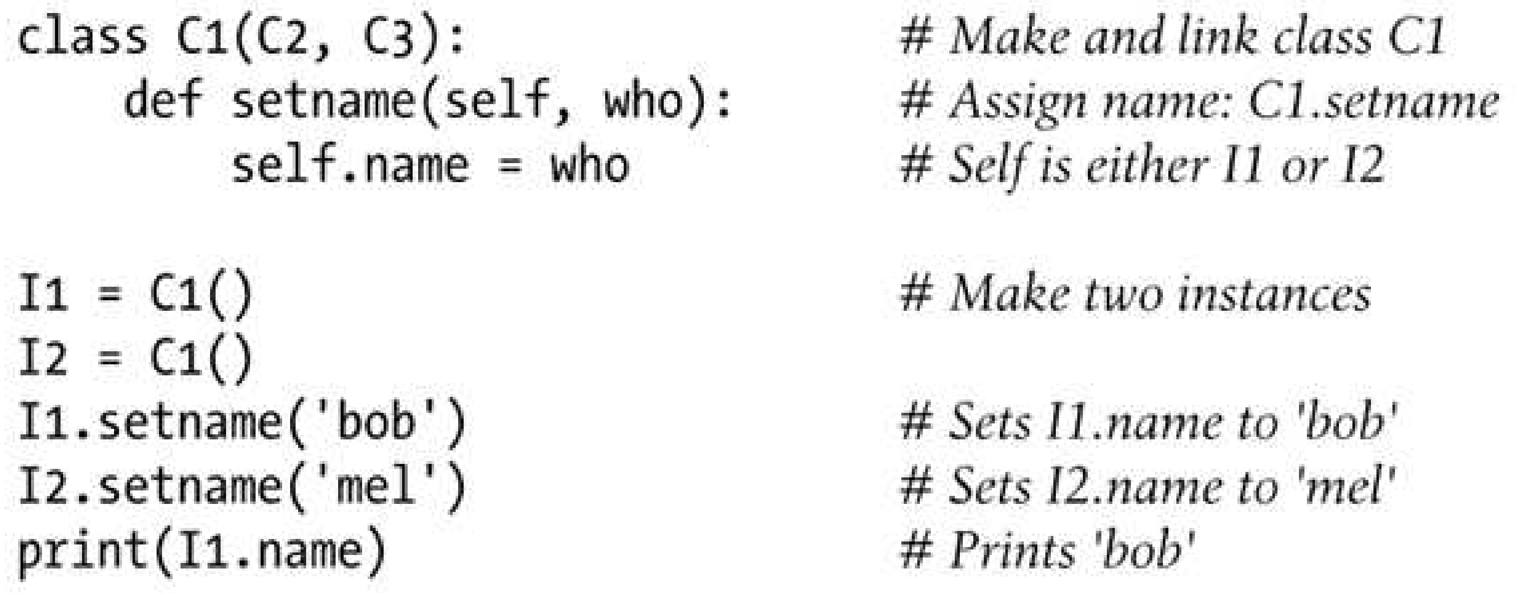
Herança

- I1.x e I2.x? Localizado em C1
- I1.y e I2.y? C1
- I1.z e I2.z? C2
- I2.name? I2



Herança

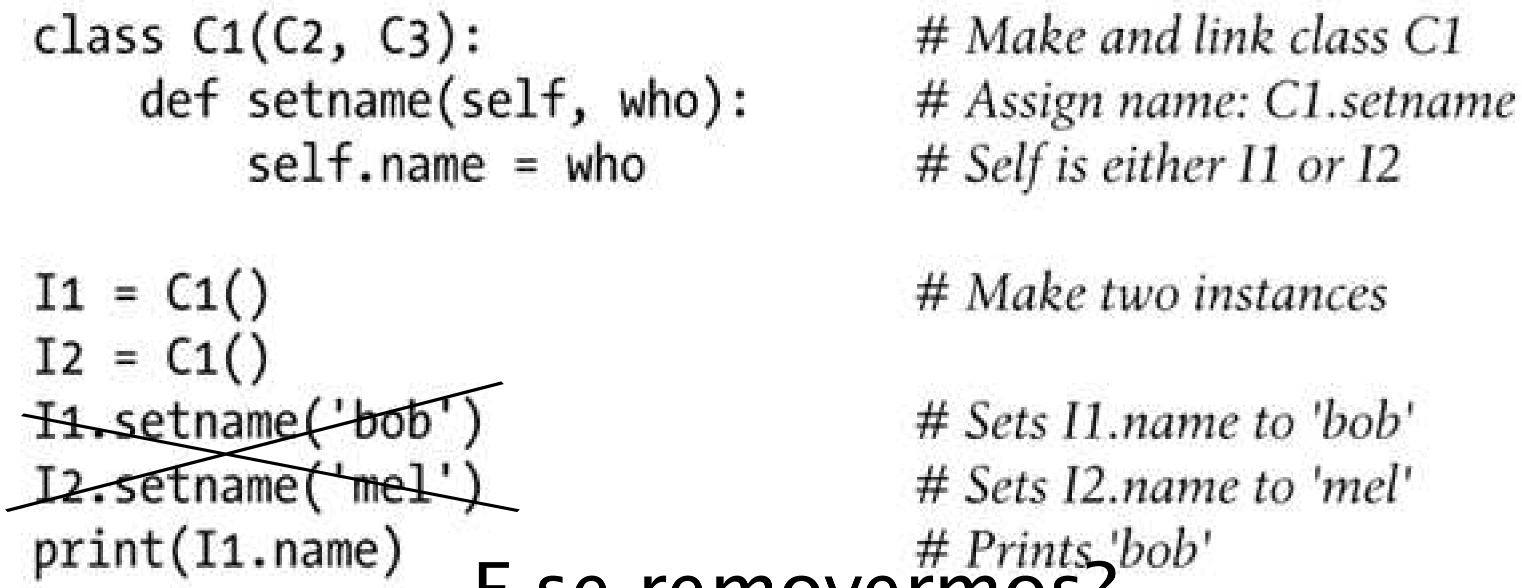
- Código da classe associada a figura dos slides anteriores.



```
class C1(C2, C3):                       # Make and link class C1
    def setname(self, who):             # Assign name: C1.setname
        self.name = who                 # Self is either I1 or I2

I1 = C1()                               # Make two instances
I2 = C1()
I1.setname('bob')                       # Sets I1.name to 'bob'
I2.setname('mel')                       # Sets I2.name to 'mel'
print(I1.name)                          # Prints 'bob'
```

Herança

- Código da classe associada a figura dos slides anteriores.

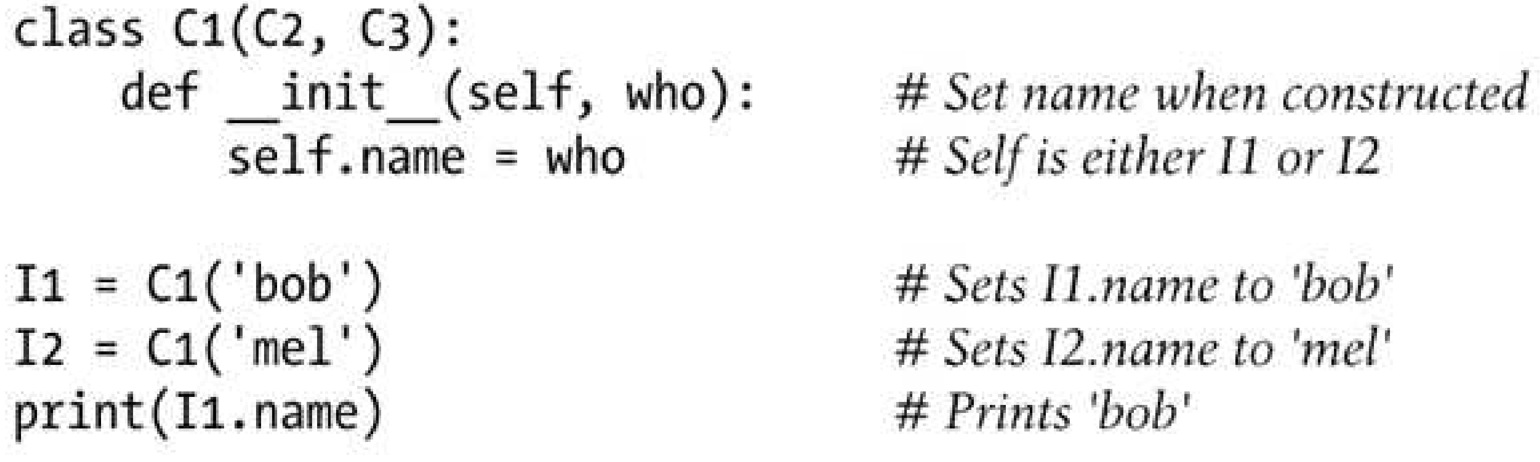


```
class C1(C2, C3):                    # Make and link class C1
    def setname(self, who):          # Assign name: C1.setname
        self.name = who              # Self is either I1 or I2

I1 = C1()                            # Make two instances
I2 = C1()
~~I1.setname('bob')~~                    # Sets I1.name to 'bob'
~~I2.setname('mel')~~                    # Sets I2.name to 'mel'
print(I1.name)                       # Prints 'bob'
```

E se removermos?
Tem impacto?

Construtor

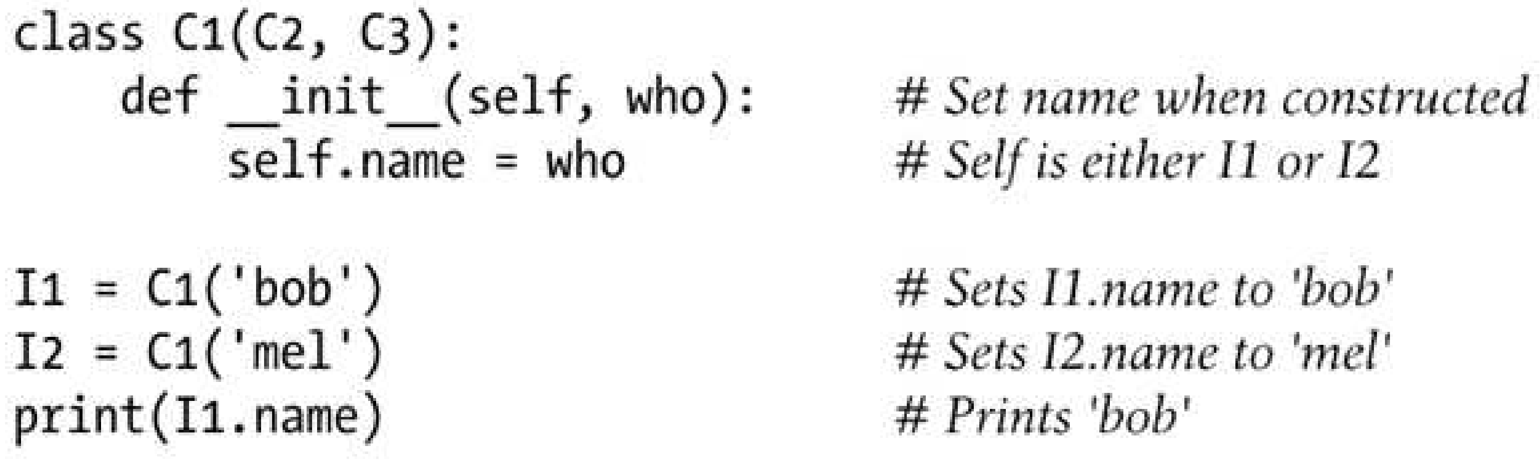
- Porque precisamos de um construtor?
- Qual a finalidade do construtor?



```
class C1(C2, C3):
    def __init__(self, who):        # Set name when constructed
        self.name = who             # Self is either I1 or I2

I1 = C1('bob')                      # Sets I1.name to 'bob'
I2 = C1('mel')                      # Sets I2.name to 'mel'
print(I1.name)                      # Prints 'bob'
```

Construtor

- O construtor vai garantir que, nesse caso, o atributo *name* seja inicializado.

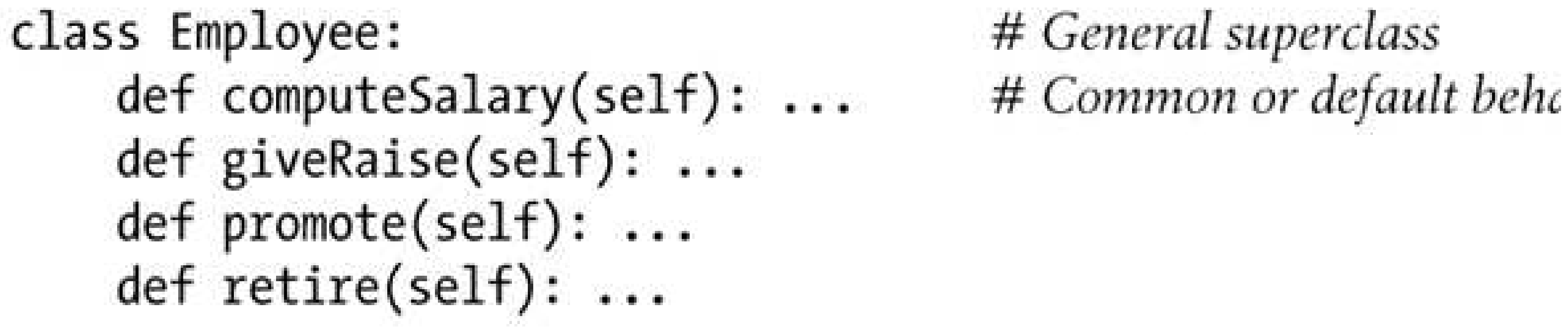


```
class C1(C2, C3):
    def __init__(self, who):          # Set name when constructed
        self.name = who               # Self is either I1 or I2

I1 = C1('bob')                        # Sets I1.name to 'bob'
I2 = C1('mel')                        # Sets I2.name to 'mel'
print(I1.name)                        # Prints 'bob'
```

Importância de OO

```
class Employee:                          # General superclass
    def computeSalary(self): ...         # Common or default beh
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

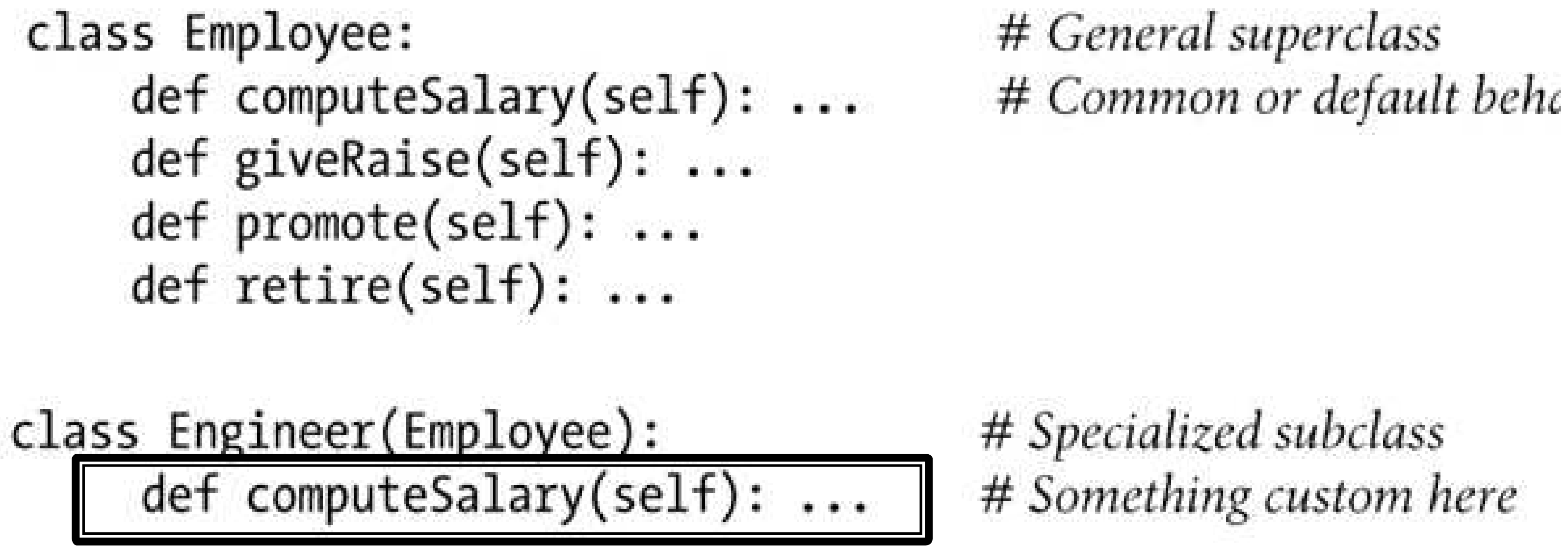
Importância de OO



```
class Employee:                          # General superclass
    def computeSalary(self): ...         # Common or default beha
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

Reuso de Código,
mas como???

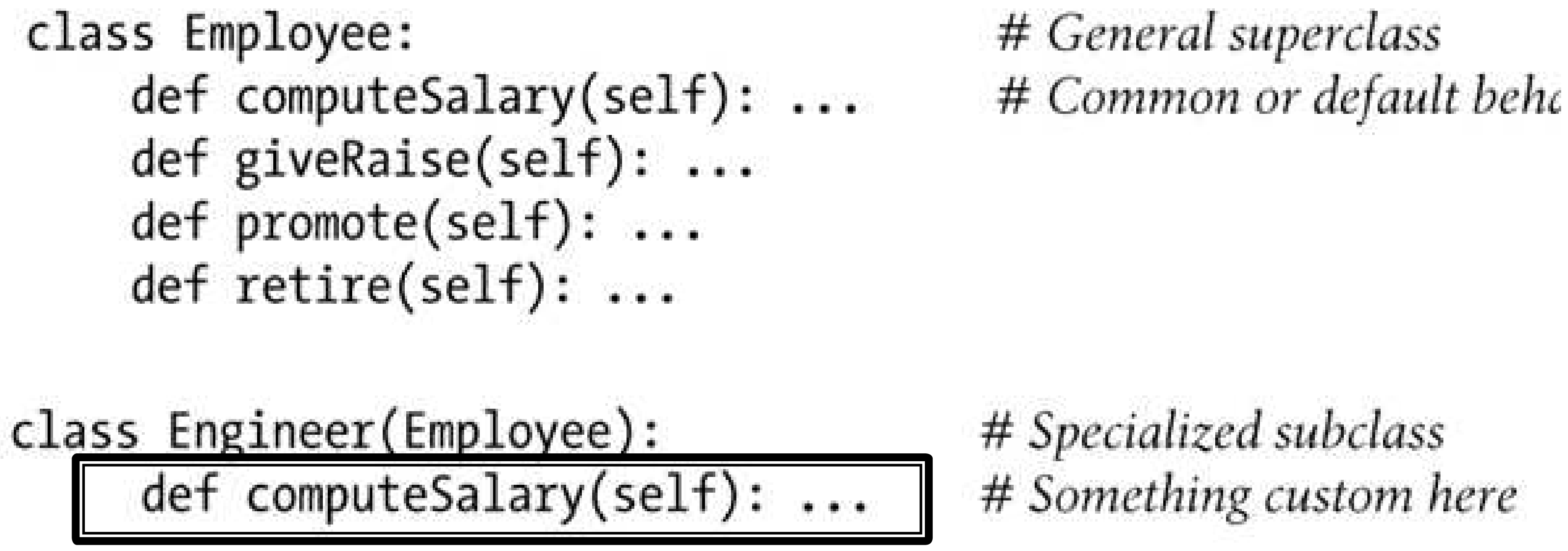
Importância de OO



```
class Employee:                          # General superclass
    def computeSalary(self): ...         # Common or default beh
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):                # Specialized subclass
    def computeSalary(self): ...         # Something custom here
```

Exemplo de?

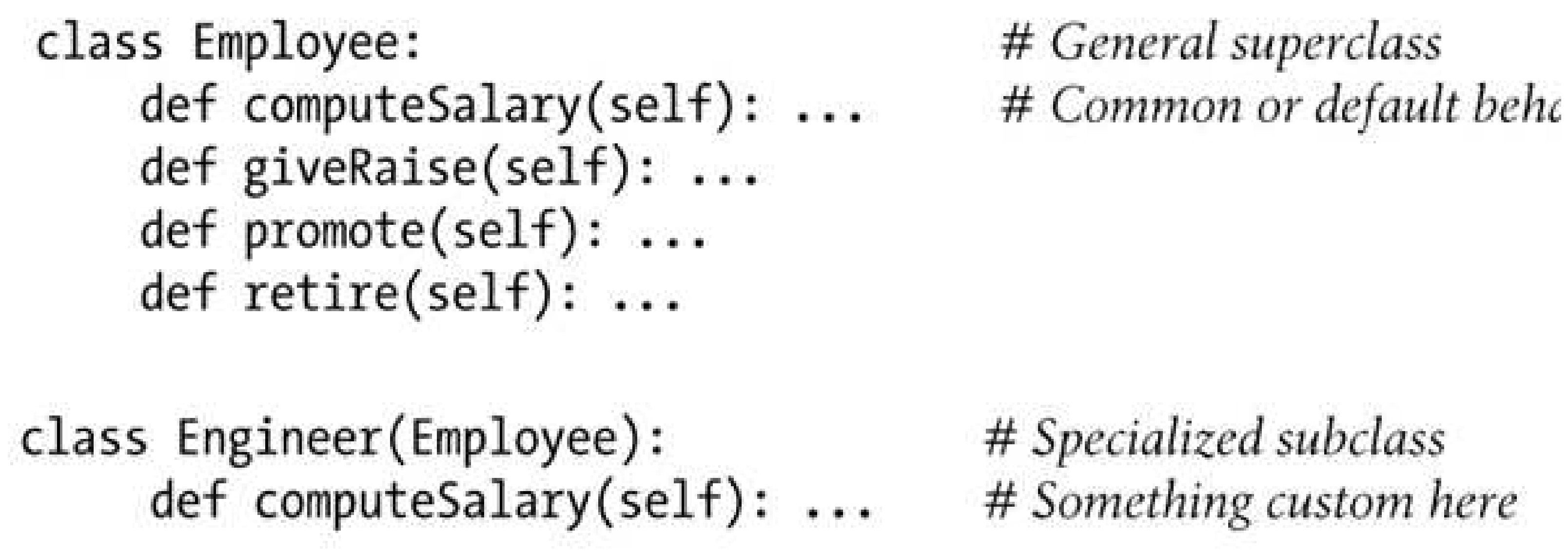
Importância de OO



```
class Employee:                          # General superclass
    def computeSalary(self): ...         # Common or default beh
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):                # Specialized subclass
    def computeSalary(self): ...         # Something custom here
```

Sobrecarga

Importância de OO



```
class Employee:                              # General superclass
    def computeSalary(self): ...             # Common or default beh
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):                    # Specialized subclass
    def computeSalary(self): ...             # Something custom here
```

Como instanciar os objetos
dessas classes?

Importância de OO

```
class Employee:                          # General superclass
    def computeSalary(self): ...         # Common or default beh
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):                # Specialized subclass
    def computeSalary(self): ...         # Something custom here

bob = Employee()                         # Default behavior
mel = Engineer()                         # Custom salary calculator
```

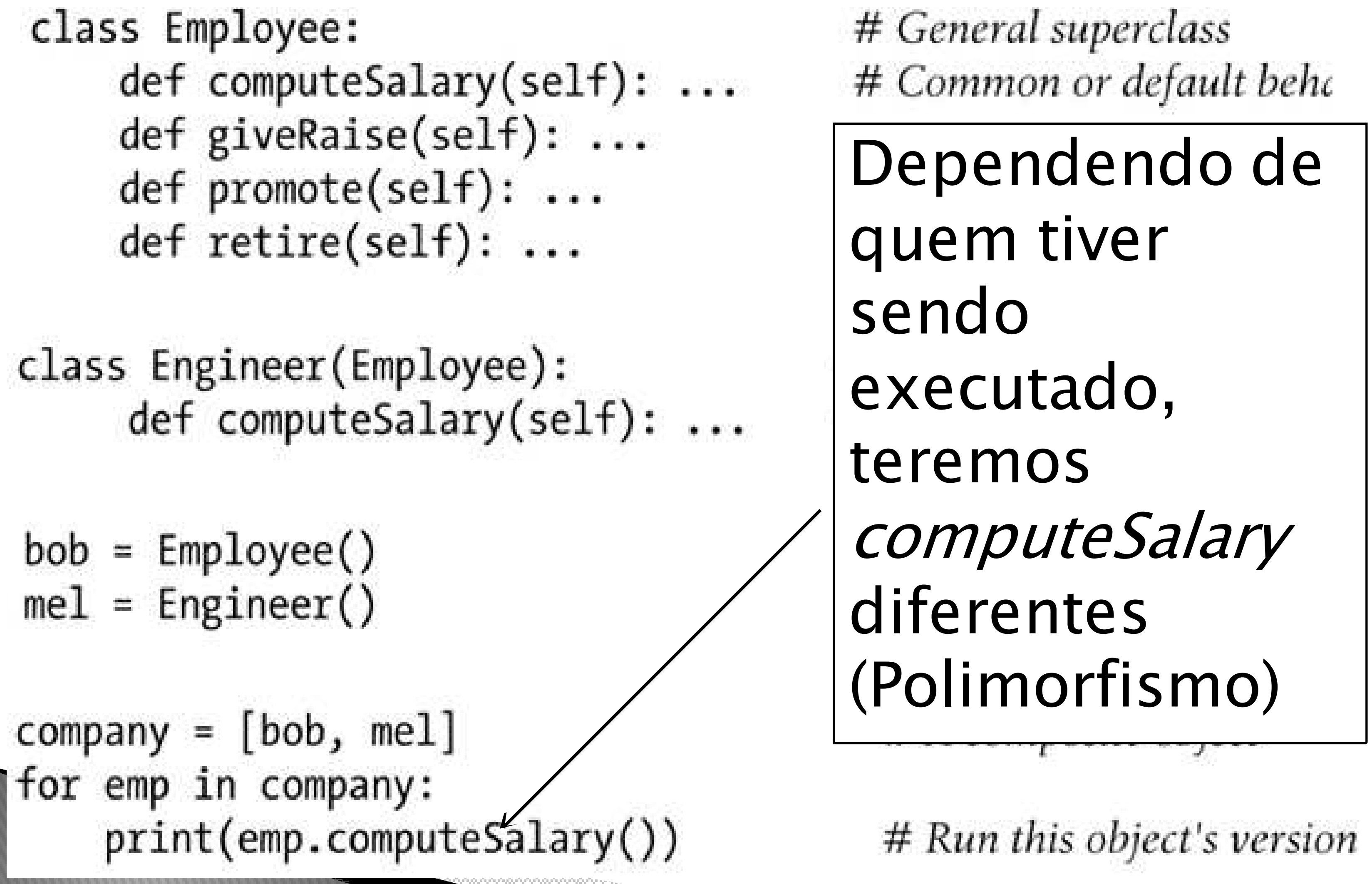
Importância de OO

```
class Employee:                              # General superclass
    def computeSalary(self): ...             # Common or default beh
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):                    # Specialized subclass
    def computeSalary(self): ...             # Something custom here

bob = Employee()                             # Default behavior
mel = Engineer()                             # Custom salary calculator
```

Importância de OO

```
class Employee:                             # General superclass
    def computeSalary(self): ...            # Common or default beh
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):                   # Specialized subclass
    def computeSalary(self): ...            # Something custom here

bob = Employee()                            # Default behavior
mel = Engineer()                            # Custom salary calculator

company = [bob, mel]                        # A composite object
for emp in company:
    print(emp.computeSalary())              # Run this object's version
```

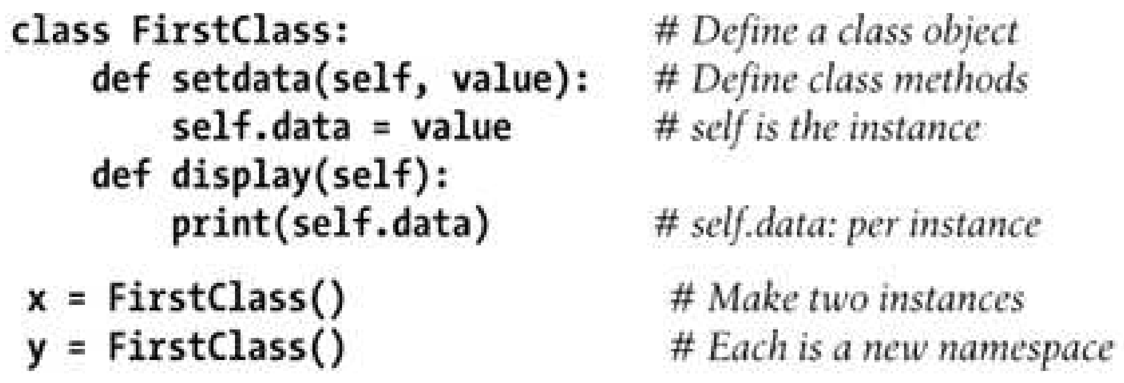
Importância de OO



```
class Employee:                          # General superclass
    def computeSalary(self): ...         # Common or default beha
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):
    def computeSalary(self): ...

bob = Employee()
mel = Engineer()

company = [bob, mel]
for emp in company:
    print(emp.computeSalary())           # Run this object's version
```

Dependendo de quem tiver sendo executado, teremos *computeSalary* diferentes (Polimorfismo)

Exemplo Simples



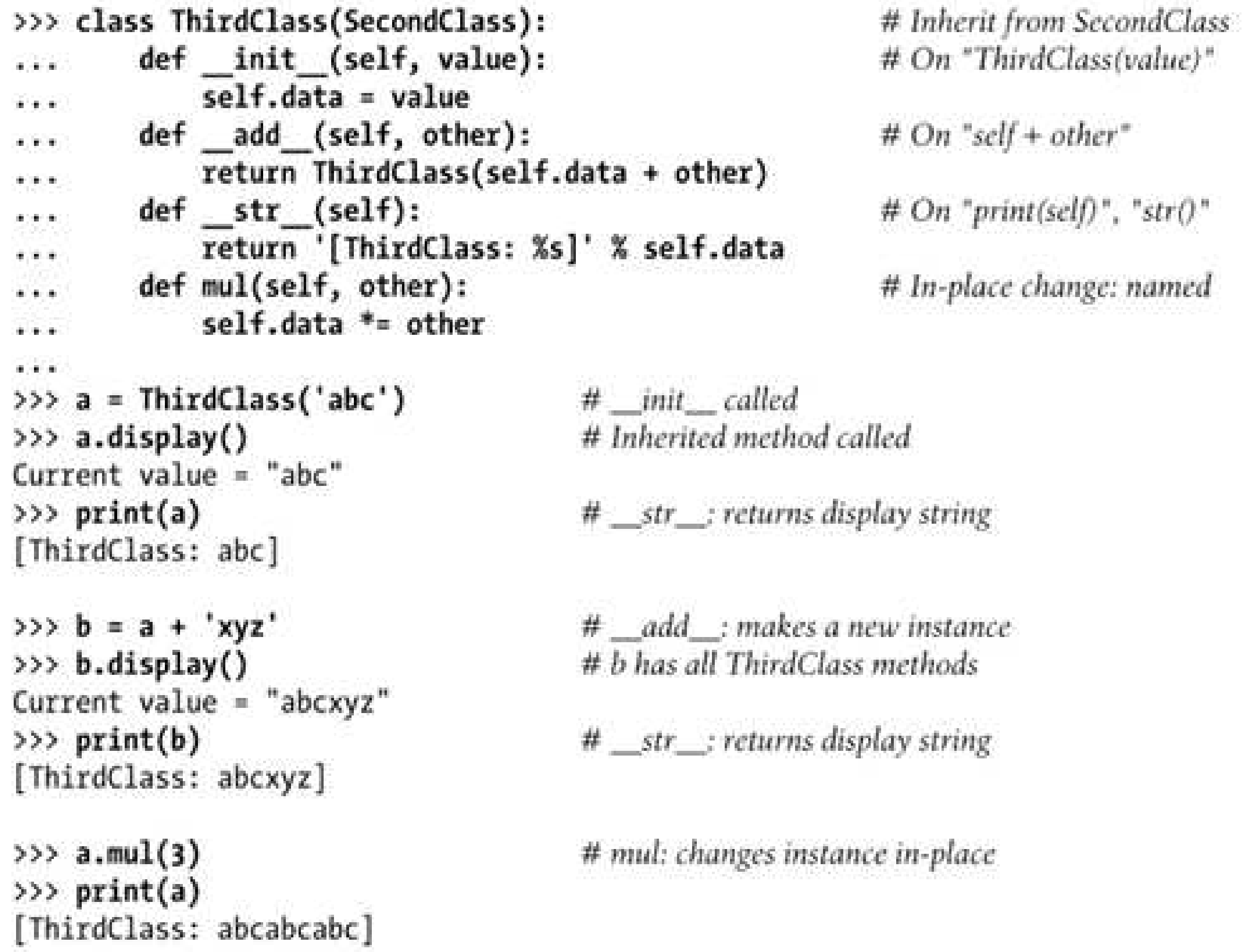
```
class FirstClass:                      # Define a class object
    def setdata(self, value):          # Define class methods
        self.data = value              # self is the instance
    def display(self):
        print(self.data)               # self.data: per instance

 x = FirstClass()                      # Make two instances
 y = FirstClass()                      # Each is a new namespace
```

X “is a” FirstClass
Y “is a” FirstClass

Exemplo Simples

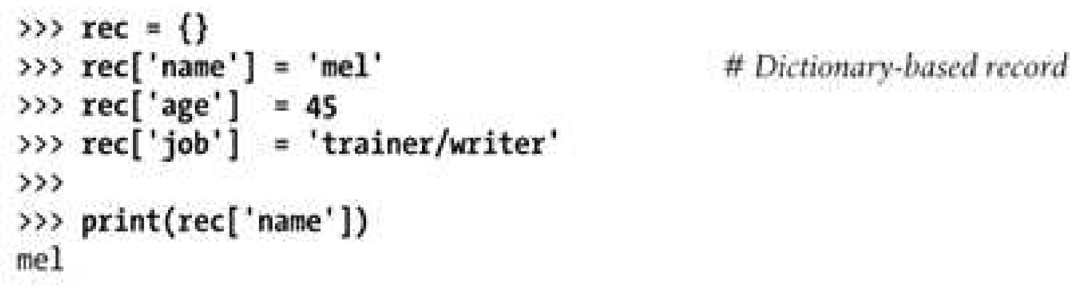
```
class FirstClass:                   # Define a class object
    def setdata(self, value):       # Define class methods
        self.data = value           # self is the instance
    def display(self):
        print(self.data)            # self.data: per instance

x = FirstClass()                    # Make two instances
y = FirstClass()                    # Each is a new namespace

>>> x.setdata("King Arthur")        # Call methods: self is x
>>> y.setdata(3.14159)              # Runs: FirstClass.setdata(y, 3.14159)

>>> x.data = "New value"
>>> x.display()
New value

>>> class SecondClass(FirstClass):                      # Inherits setdata
...     def display(self):                              # Changes display
...         print('Current value = "%s"' % self.data)
...

>>> z = SecondClass()
>>> z.setdata(42)                   # Finds setdata in FirstClass
>>> z.display()                     # Finds overridden method in SecondClass
Current value = "42"
>>> x.display()                     # x is still a FirstClass instance (old message)
New value
```

Sobrecarga de operadores

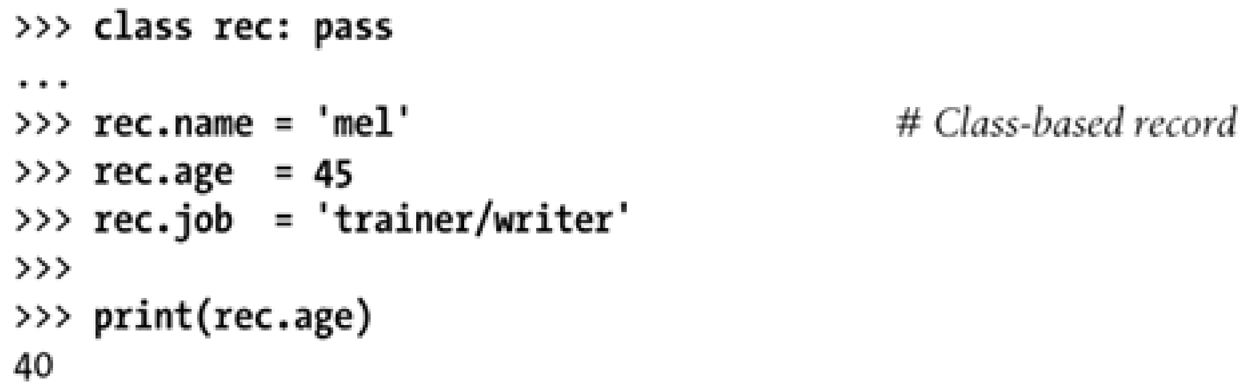


```
>>> class ThirdClass(SecondClass):                         # Inherit from SecondClass
...     def __init__(self, value):                         # On "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):                          # On "self + other"
...         return ThirdClass(self.data + other)
...     def __str__(self):                                 # On "print(self)", "str()"
...         return '[ThirdClass: %s]' % self.data
...     def mul(self, other):                              # In-place change: named
...         self.data *= other
...
>>> a = ThirdClass('abc')            # __init__ called
>>> a.display()                      # Inherited method called
Current value = "abc"
>>> print(a)                         # __str__: returns display string
[ThirdClass: abc]

>>> b = a + 'xyz'                    # __add__: makes a new instance
>>> b.display()                      # b has all ThirdClass methods
Current value = "abcxyz"
>>> print(b)                         # __str__: returns display string
[ThirdClass: abcxyz]

>>> a.mul(3)                         # mul: changes instance in-place
>>> print(a)
[ThirdClass: abcabcabc]
```

Classes x Dicionários



```
>>> rec = {}
>>> rec['name'] = 'mel'                         # Dictionary-based record
>>> rec['age']  = 45
>>> rec['job']  = 'trainer/writer'
>>>
>>> print(rec['name'])
mel
```



```
>>> class rec: pass
...
>>> rec.name = 'mel'                            # Class-based record
>>> rec.age  = 45
>>> rec.job  = 'trainer/writer'
>>>
>>> print(rec.age)
40
```

Melhorando a Classe

```
>>> class rec: pass
...
>>> pers1 = rec()                                   # Instance-based records
>>> pers1.name = 'mel'
>>> pers1.job  = 'trainer'
>>> pers1.age   = 40
>>>
>>> pers2 = rec()
>>> pers2.name = 'vls'
>>> pers2.job  = 'developer'
>>>
>>> pers1.name, pers2.name
('mel', 'vls')
```

Melhorando a Classe 2

```
>>> class Person:
...     def __init__(self, name, job):      # Class = Data + Logic
...         self.name = name
...         self.job  = job
...     def info(self):
...         return (self.name, self.job)
...
>>> rec1 = Person('mel', 'trainer')
>>> rec2 = Person('vls', 'developer')
>>>
>>> rec1.job, rec2.info()
('trainer', ('vls', 'developer'))
```