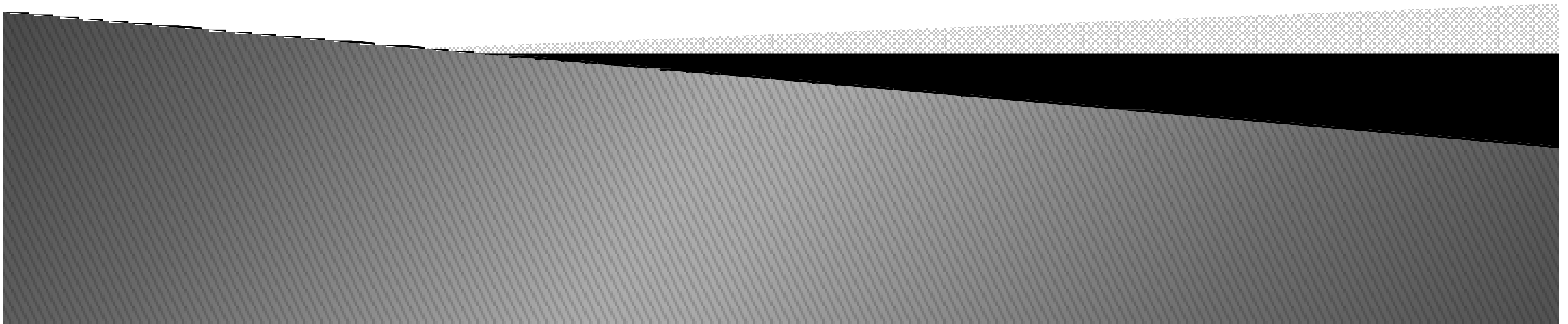
# Orientação a Objetos

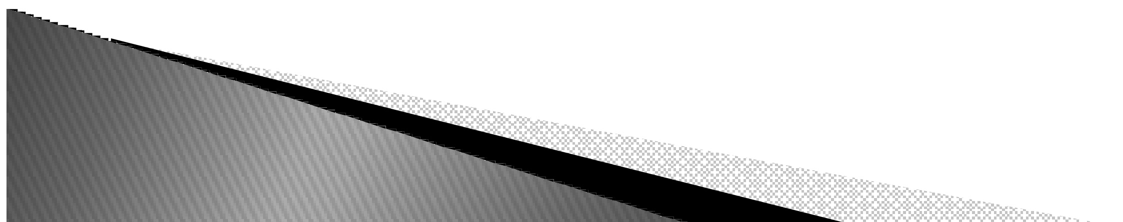
## Aula04

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



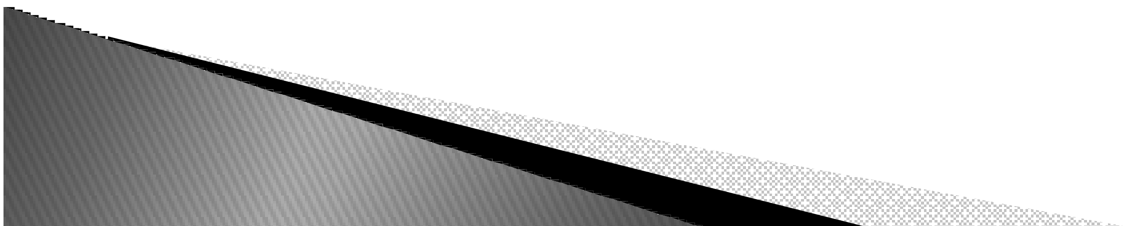
# Criando uma Classe

```
# Add record field initialization

class Person:
    def __init__(self, name, job, pay):        # Constructor takes 3 arguments
        self.name = name                        # Fill out fields when created
        self.job  = job                         # self is the new instance object
        self.pay  = pay
```



# Criando uma Classe

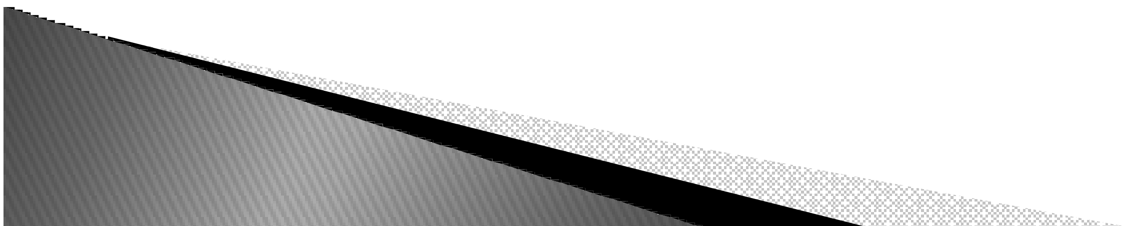
```
# Add defaults for constructor arguments

class Person:
    def __init__(self, name, job=None, pay=0):          # Normal function args
        self.name = name
        self.job  = job
        self.pay  = pay
```



# Criando uma Classe

```
# Add incremental self-test code

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

bob = Person('Bob Smith')                              # Test the class
sue = Person('Sue Jones', job='dev', pay=100000)       # Runs __init__ automatically
print(bob.name, bob.pay)                               # Fetch attached attributes
print(sue.name, sue.pay)                               # sue's and bob's attrs differ
```

Quais as saídas?



# Criando uma Classe

```
# Add incremental self-test code

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

bob = Person('Bob Smith')                          # Test the class
sue = Person('Sue Jones', job='dev', pay=100000)   # Runs __init__ automatically
print(bob.name, bob.pay)                           # Fetch attached attributes
print(sue.name, sue.pay)                           # sue's and bob's attrs differ
```

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000
```

# Criando uma Classe

```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                          # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

Quais as Saídas?

# Criando uma Classe

```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                    # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000
```

Saídas

# Criando uma Classe

```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                          # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

Qual a finalidade?

# Criando uma Classe

```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                    # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

```
>>> import person
>>>
```

# Criando uma Classe

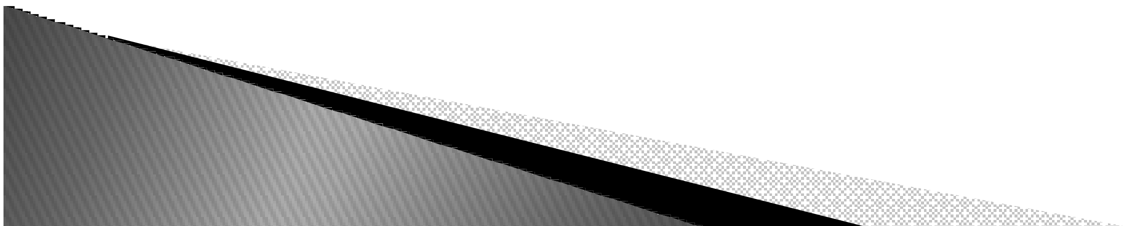
```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                          # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

```
>>> import person
>>>
```

Ao incerir *if __name__ ==‘__main__’:*
Quando a classe é importada, os códigos dentro desse bloco não são executados.

# Recordando

```
>>> name = 'Bob Smith'          # Simple string, outside class
>>> name.split()                # Extract last name
['Bob', 'Smith']
>>> name.split()[-1]            # Or [1], if always just two parts
'Smith'
```



# Criando uma Classe

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])          # Extract object's last name
    sue.pay *= 1.10                      # Give this object a raise
    print(sue.pay)
```

Quais as saídas?

# Criando uma Classe

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])          # Extract object's last name
    sue.pay *= 1.10                      # Give this object a raise
    print(sue.pay)
```

Quais as saídas?

```
Bob Smith 0
Sue Jones 100000
Smith
110000.0
```

# Criando uma Classe

```
# Add methods to encapsulate operations for maintainability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                              # Behavior methods
        return self.name.split()[-1]                 # self is implied subject
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))     # Must change here only

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())            # Use the new methods
    sue.giveRaise(.10)                               # instead of hardcoding
    print(sue.pay)
```

Saídas?

# Criando uma Classe

```
# Add methods to encapsulate operations for maintainability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                              # Behavior methods
        return self.name.split()[-1]                 # self is implied subject
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))     # Must change here only

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue.pay)
```

Saídas?

```
Bob Smith 0
Sue Jones 100000
Smith Jones
110000
```

# Criando uma Classe

```
# Add __str__ overload method for printing objects

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):                                          # Added method
        return '[Person: %s, %s]' % (self.name, self.pay)       # String to print

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

Saídas?

# Criando uma Classe

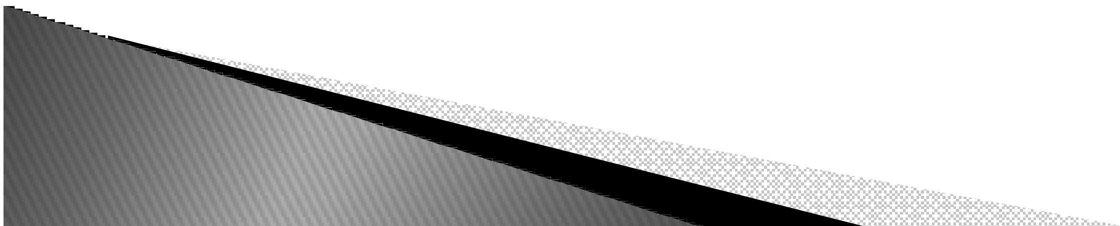
```
# Add __str__ overload method for printing objects

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):                                          # Added method
        return '[Person: %s, %s]' % (self.name, self.pay)       # String to print

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

Saídas?

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
```

# Herança

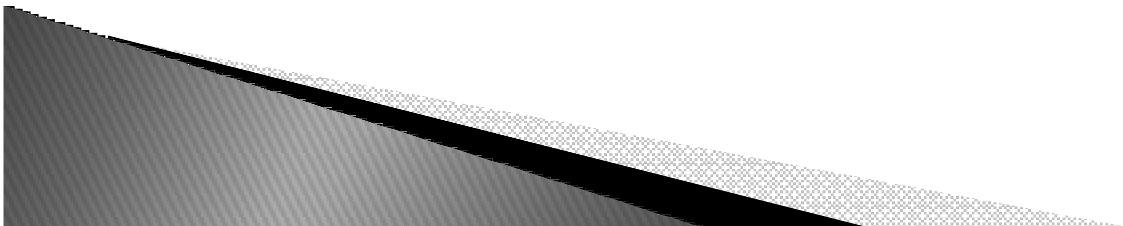
- Vamos criar uma classe Manager que vai sobrescrever o método *giveRaise*.



# Herança

- Vamos criar uma classe Manager que vai sobrescrever o método *giveRaise*.

```
class Manager(Person):                              # Inherit Person attrs
    def giveRaise(self, percent, bonus=.10):        # Redefine to customize
```

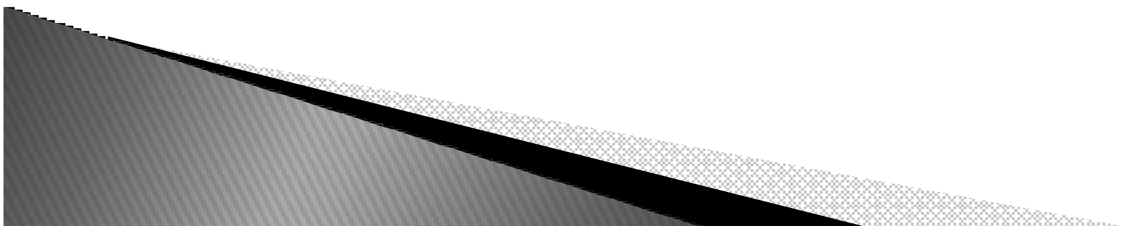


# Herança

- Vamos criar uma classe Manager que vai sobrescrever o método *giveRaise*.

```
class Manager(Person):                               # Inherit Person attrs
    def giveRaise(self, percent, bonus=.10):         # Redefine to customize
```

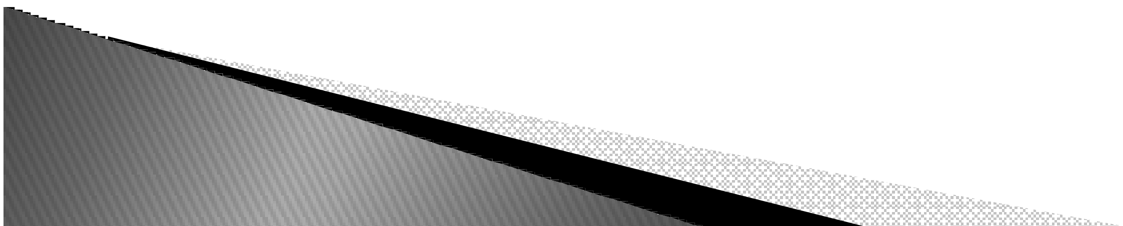
- Tenho duas formas de implementar esse método. Uma “boa”e uma “ruim”.



# Herança

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))    # Bad: cut-and-paste
```

Essa implementação é ruim ou boa?

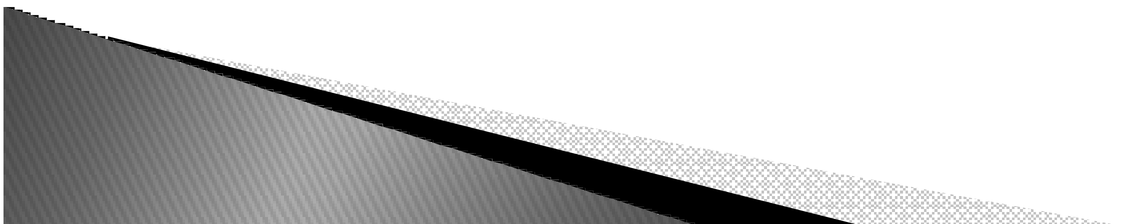


# Herança

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))    # Bad: cut-and-paste
```

Essa implementação é ruim ou boa?
Ruim.
Se alterarmos a forma de dar aumento na classe Person, teremos de fazer retrabalho aqui também.

Qual a solução?



# Herança

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)          # Good: augment original
```

Normalmente fazemos uso de:

```
instance.method(args...)
```

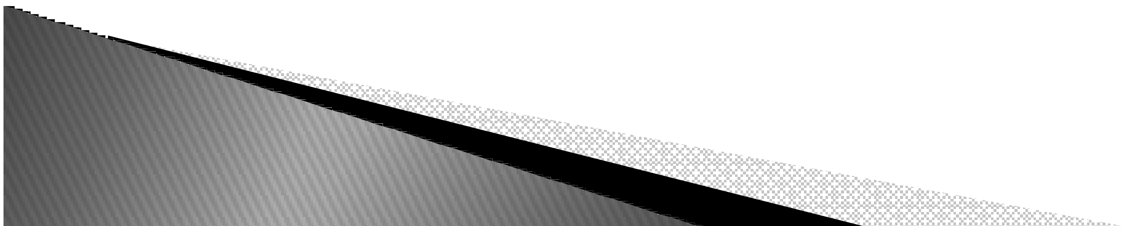
O Python, internamente, vai fazer isso:

```
class.method(instance, args...)
```

# Herança

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)          # Good: augment original
```

Nessa implementação, fazemos uso do próprio método para dar aumento de *Person.*



# Analisando o exemplo

```
# Add customization of one behavior in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):          # Redefine at this level
        Person.giveRaise(self, percent + bonus)       # Call Person's version

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)          # Make a Manager: __init__
    tom.giveRaise(.10)                                # Runs custom version
    print(tom.lastName())                             # Runs inherited method
    print(tom)                                        # Runs inherited __str__
```

Saídas?

# Analisando o exemplo

```
# Add customization of one behavior in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):          # Redefine at this level
        Person.giveRaise(self, percent + bonus)       # Call Person's version

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)          # Make a Manager: __init__
    tom.giveRaise(.10)                                # Runs custom version
    print(tom.lastName())                             # Runs inherited method
    print(tom)                                        # Runs inherited __str__
```

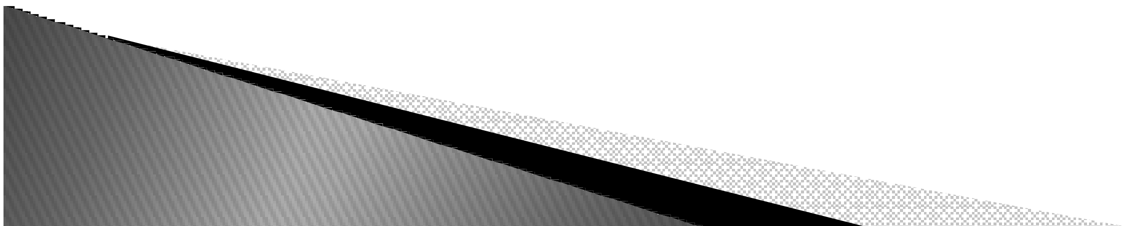
Saídas?

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

# Polimorfismo em Ação

```
if __name__ == '__main__':
    ...
    print('--All three--')
    for object in (bob, sue, tom):
        object.giveRaise(.10)
        print(object)
```

Saídas?



# Polimorfismo em Ação

```
if __name__ == '__main__':
    ...
    print('--All three--')
    for object in (bob, sue, tom):
        object.giveRaise(.10)
        print(object)
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
--All three--
[Person: Bob Smith, 0]
[Person: Sue Jones, 121000]
[Person: Tom Jones, 72000]
```

# Herança

- Com Herança podemos realizar um implementação específica, ao redefinir e acrescentar métodos.

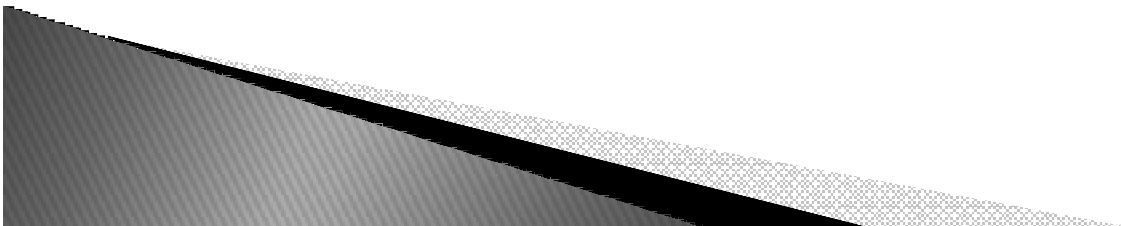
```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __str__(self): ...

class Manager(Person):                          # Inherit
    def giveRaise(self, ...): ...               # Customize
    def someThingElse(self, ...): ...           # Extend

tom = Manager()
tom.lastName()                                  # Inherited verbatim
tom.giveRaise()                                 # Customized version
tom.someThingElse()                             # Extension here
print(tom)                                      # Inherited overload method
```

# Redefinindo um construtor

```
# Add customization of constructor in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay):                   # Redefine constructor
        Person.__init__(self, name, 'mgr', pay)      # Run original with 'mgr'
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)                 # Job name not needed:
    tom.giveRaise(.10)                                # Implied/set by class
    print(tom.lastName())
    print(tom)
```

# Redefinindo um construtor

```
class A():
    def __init__(self,value):
        self.prog1=value
        print (self.prog1)

class B(A):
    def __init__(self):
        #A.__init__(self,"teste")
        super(B, self).__init__("teste1")
        self.prog2="teste2"
        print(self.prog1)
        print(self.prog2)

a1=A("testeInicial")
a2=B()
```



# Combinando Classes

```
# Aggregate embedded objects into a composite

...
bob = Person(...)
sue = Person(...)
tom = Manager(...)

class Department:
    def __init__(self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
            print(person)

development = Department(bob, sue)          # Embed objects in a composite
development.addMember(tom)
development.giveRaises(.10)                 # Runs embedded objects' giveRaise
development.showAll()                       # Runs embedded objects' __str__s
```