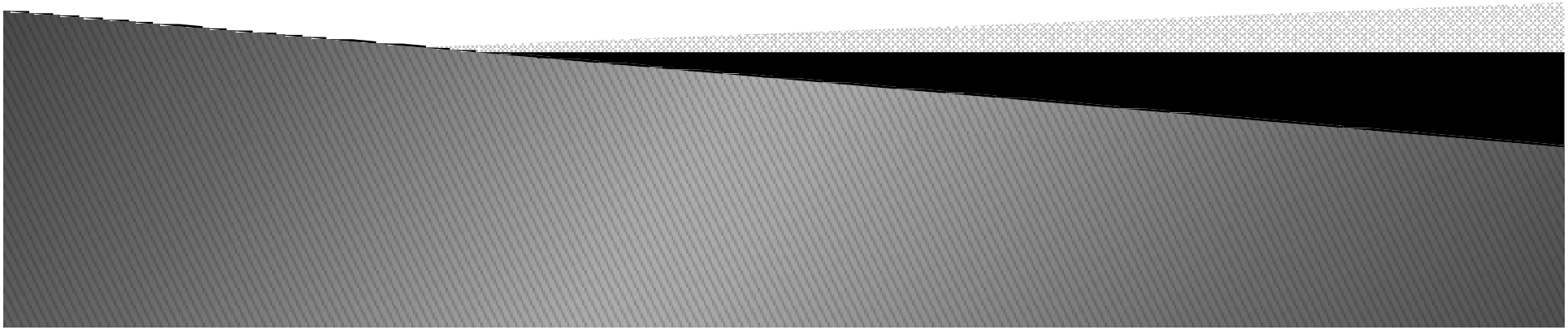
Orientação a Objetos

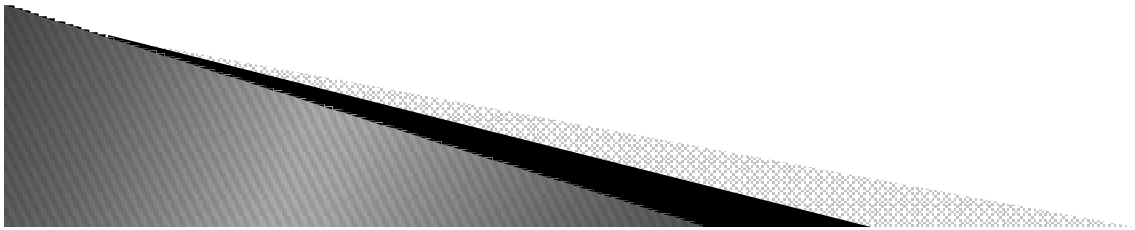
Aula05

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



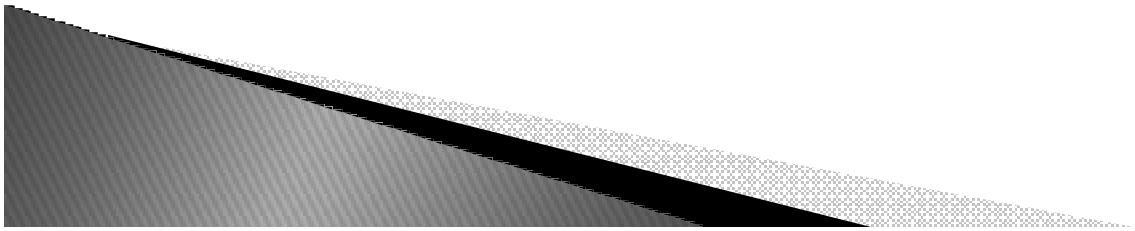
Métodos Estáticos e de Classe

- O que são métodos estáticos?
- E de classe?
- Para que servem?



Métodos Estáticos e de Classe

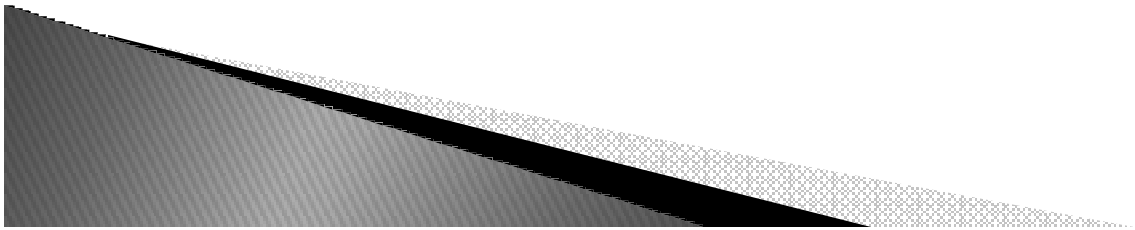
- O que são métodos estáticos?
- E de classe?
- Para que servem?
  - Servem para, por exemplo, contar a quantidade de instâncias criadas de uma classe.



Declarando métodos estáticos

- Vai depender da versão do Python usado 2.6 ou 3.x

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Number of instances created: ", Spam.numInstances)
```



Declarando métodos estáticos

- Esse código não funciona no Python 2.6, veja o erro abaixo.

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Number of instances created: ", Spam.numInstances)
```

```
C:\misc> c:\python26\python
>>> from spam import Spam
>>> a = Spam()                    # Cannot call unbound class methods in 2.6
>>> b = Spam()                    # Methods expect a self object by default
>>> c = Spam()

>>> Spam.printNumInstances()
TypeError: unbound method printNumInstances() must be called with Spam instance
as first argument (got nothing instead)
>>> a.printNumInstances()
TypeError: printNumInstances() takes no arguments (1 given)
```

Declarando métodos estáticos

- Esse código funciona no Python 3.x se for acessado diretamente pela classe e, não funciona pela instância.

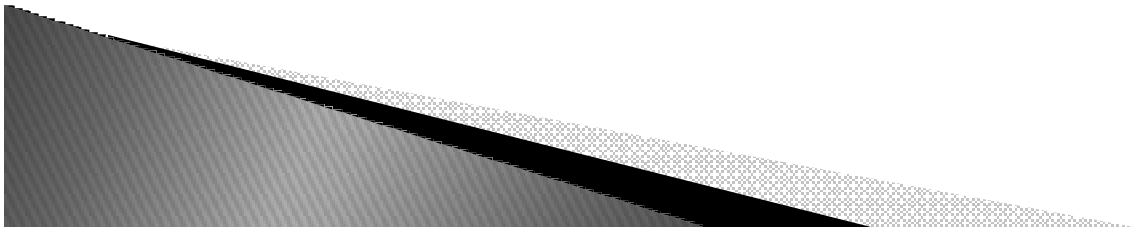
```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Number of instances created: ", Spam.numInstances)
```

```
C:\misc> c:\python30\python
>>> from spam import Spam
>>> a = Spam()                          # Can call functions in class in 3.0
>>> b = Spam()                          # Calls through instances still pass a self
>>> c = Spam()

>>> Spam.printNumInstances()            # Differs in 3.0
Number of instances created:  3
>>> a.printNumInstances()
TypeError: printNumInstances() takes no arguments (1 given)
```

Resumindo

```
Spam.printNumInstances()          # Fails in 2.6, works in 3.0
instance.printNumInstances()      # Fails in both 2.6 and 3.0
```

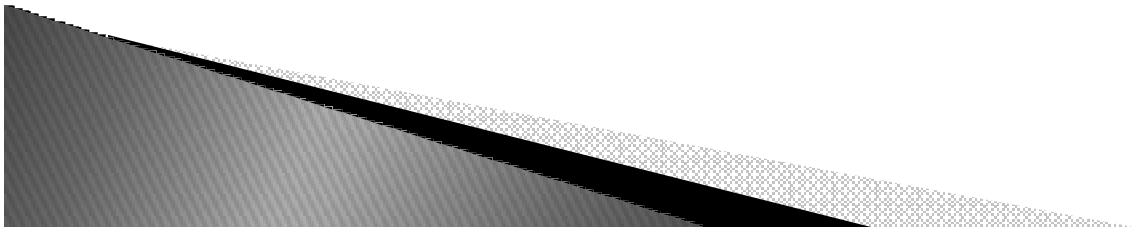
- Como solucionar?
- Como faço rodar no Python 2.6?



1ª Solução

```
def printNumInstances():
    print("Number of instances created: ", Spam.numInstances)

class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
```

```
>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()          # But function may be too far removed
Number of instances created:  3       # And cannot be changed via inheritance
>>> spam.Spam.numInstances
3
```



1ª Solução

```
def printNumInstances():
    print("Number of instances created: ", Spam.numInstances)

class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
```

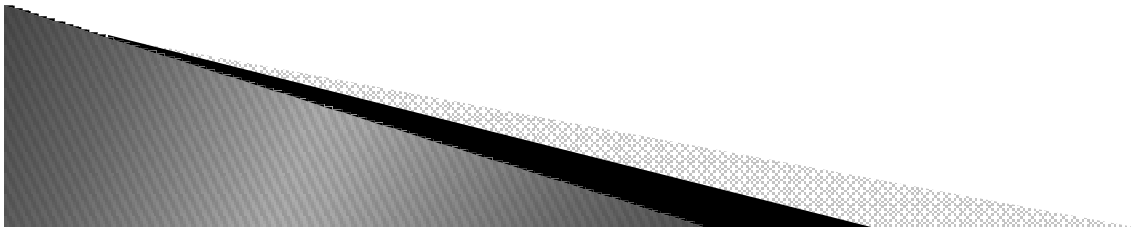
```
>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()          # But function may be too far removed
Number of instances created:  3       # And cannot be changed via inheritance
>>> spam.Spam.numInstances
3
```

Não teremos como sobrescrever o método printNumInstances().

2ª Solução

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances(self):
        print("Number of instances created: ", Spam.numInstances)
```

```
>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created:  3
>>> Spam.printNumInstances(a)
Number of instances created:  3
>>> Spam().printNumInstances()          # But fetching counter changes counter!
Number of instances created:  4
```



2ª Solução

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances(self):
        print("Number of instances created: ", Spam.numInstances)
```

```
>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created:  3
>>> Spam.printNumInstances(a)
Number of instances created:  3
>>> Spam().printNumInstances()          # But fetching counter changes counter!
Number of instances created:  4
```

Somos obrigados a criar uma instância.

3ª Solução

```
class Methods:
    def imeth(self, x):                 # Normal instance method: passed a self
        print(self, x)

    def smeth(x):                       # Static: no instance passed
        print(x)

    def cmeth(cls, x):                  # Class: gets class, not instance
        print(cls, x)

    smeth = staticmethod(smeth)         # Make smeth a static method
    cmeth = classmethod(cmeth)          # Make cmeth a class method
```

```
>>> obj = Methods()                     # Make an instance

>>> obj.imeth(1)                        # Normal method, call through instance
<__main__.Methods object...> 1          # Becomes imeth(obj, 1)

>>> Methods.imeth(obj, 2)               # Normal method, call through class
<__main__.Methods object...> 2          # Instance passed explicitly
```

3ª Solução

```
class Methods:
    def imeth(self, x):                 # Normal instance method: passed a self
        print(self, x)

    def smeth(x):                       # Static: no instance passed
        print(x)

    def cmeth(cls, x):                  # Class: gets class, not instance
        print(cls, x)

    smeth = staticmethod(smeth)         # Make smeth a static method
    cmeth = classmethod(cmeth)          # Make cmeth a class method
```

```
>>> Methods.smeth(3)                   # Static method, call through class
3                                       # No instance passed or expected

>>> obj.smeth(4)                       # Static method, call through instance
4                                       # Instance not passed
```

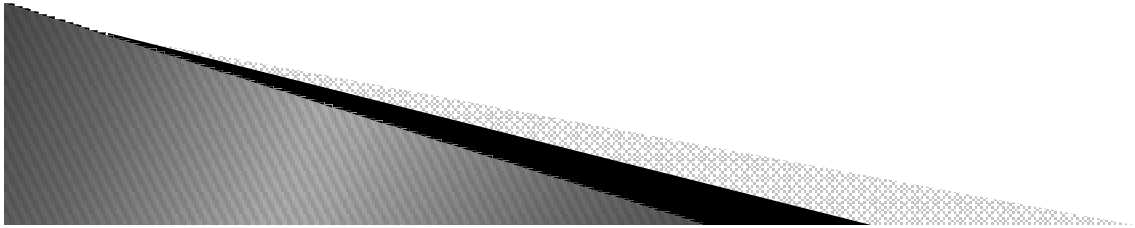
Método Estático

- Usando Método estático para contar o número de instâncias criadas de uma classe.

```
class Spam:
    numInstances = 0                          # Use static method for class data
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print("Number of instances:", Spam.numInstances)
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
```

Como posso saber o número de instâncias?



Método Estático

- Usando Método estático para contar o número de instâncias criadas de uma classe.

```
class Spam:
    numInstances = 0                          # Use static method for class data
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print("Number of instances:", Spam.numInstances)
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
```

Como posso saber o número de instâncias?

```
>>> Spam.printNumInstances()                  # Call as simple function
Number of instances: 3
>>> a.printNumInstances()                     # Instance argument not passed
Number of instances: 3
```

Método Estático

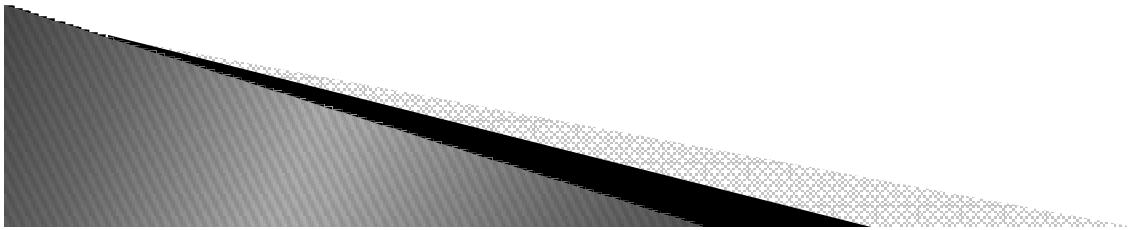
- Acessando o número de instâncias a partir de uma subclasse que redefini o método.

```
class Sub(Spam):
    def printNumInstances():                    # Override a static method
        print("Extra stuff...")                 # But call back to original
        Spam.printNumInstances()
    printNumInstances = staticmethod(printNumInstances)

>>> a = Sub()
>>> b = Sub()
>>> a.printNumInstances()                       # Call from subclass instance
Extra stuff...
Number of instances: 2
>>> Sub.printNumInstances()                     # Call from subclass itself
Extra stuff...
Number of instances: 2
>>> Spam.printNumInstances()
Number of instances: 2
```

Método Estático

- A partir de uma outra subclasse que não redefini o método.

```
>>> class Other(Spam): pass

>>> c = Other()
>>> c.printNumInstances()
Number of instances: 3
```

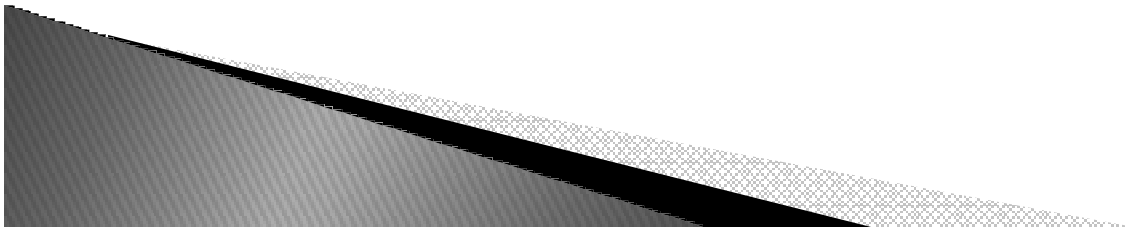


Método de classe

- Contando o número de instância pelo método de classe.

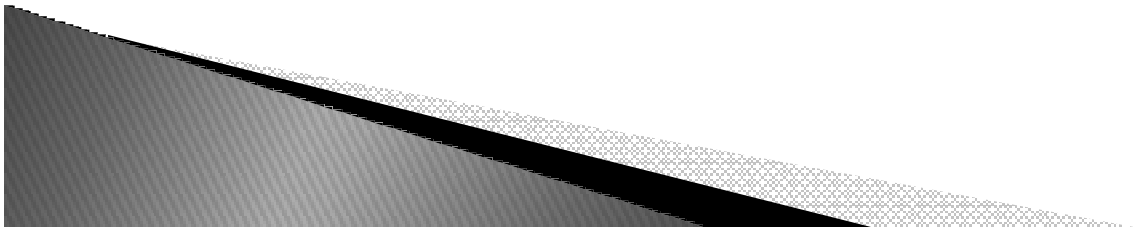
```
class Spam:
    numInstances = 0                          # Use class method instead of static
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances:", cls.numInstances)
    printNumInstances = classmethod(printNumInstances)
```

```
>>> a, b = Spam(), Spam()
>>> a.printNumInstances()                     # Passes class to first argument
Number of instances: 2
>>> Spam.printNumInstances()                  # Also passes class to first argument
Number of instances: 2
```



Diferença

- Métodos Estáticos: util para processar dados referentes a uma classe específica.
- Métodos de Classe:melhor utilizado para processar dados de classes em hierarquia.



Métodos de Classe

```
class Spam:
    numInstances = 0
    def count(cls):                     # Per-class instance counters
        cls.numInstances += 1           # cls is lowest class above instance
    def __init__(self):
        self.count()                    # Passes self.__class__ to count
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0
    def __init__(self):                 # Redefines __init__
        Spam.__init__(self)

class Other(Spam):                      # Inherits __init__
    numInstances = 0
```

```
>>> x = Spam()
>>> y1, y2 = Sub(), Sub()
>>> z1, z2, z3 = Other(), Other(), Other()
>>> x.numInstances, y1.numInstances, z1.numInstances
(1, 2, 3)
>>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```

Decoradores

- São definidos antes dos métodos (def) com o símbolo @função.
- Exemplo:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

    @staticmethod
    def printNumInstances():
        print("Number of instances created: ", Spam.numInstances)

a = Spam()
b = Spam()
c = Spam()
Spam.printNumInstances()        # Calls from both classes and instances work now!
a.printNumInstances()           # Both print "Number of instances created:  3"
```

Decoradores(Exemplo)

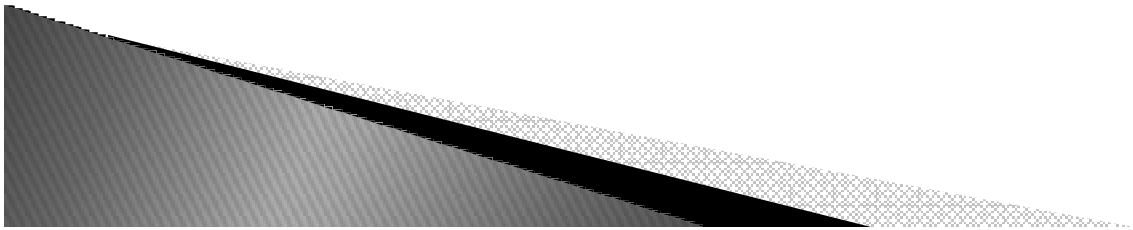
```
class Foo(object):
    def __init__(self,name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self,value):
        if not isinstance(value,str):
            raise TypeError("Must be a string!")
        self.__name = value
    @name.deleter
    def name(self):
        raise TypeError("Can't delete name")

f = Foo("Guido")
n = f.name          # calls f.name() - get function
f.name = "Monty"    # calls setter name(f,"Monty")
f.name = 45         # calls setter name(f,45) -> TypeError
del f.name          # Calls deleter name(f) -> TypeError
```

Alterando atributos de classe

```
>>> class X:
...     a = 1          # Class attribute
...
>>> I = X()
>>> I.a               # Inherited by instance
1
>>> X.a
1

>>> X.a = 2           # May change more than X
>>> I.a               # I changes too
2
>>> J = X()           # J inherits from X's runtime values
>>> J.a               # (but assigning to J.a changes a in J, not X or I)
2
```

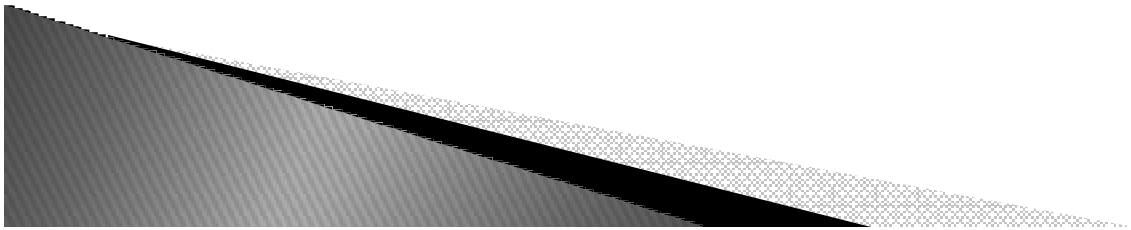
Checagem de tipo da classe

```
class A(object): pass
class B(A): pass
class C(object): pass

a = A()             # Instance of 'A'
b = B()             # Instance of 'B'
c = C()             # Instance of 'C'

type(a)             # Returns the class object A
isinstance(a,A)     # Returns True
isinstance(b,A)     # Returns True, B derives from A
isinstance(b,C)     # Returns False, C not derived from A
```



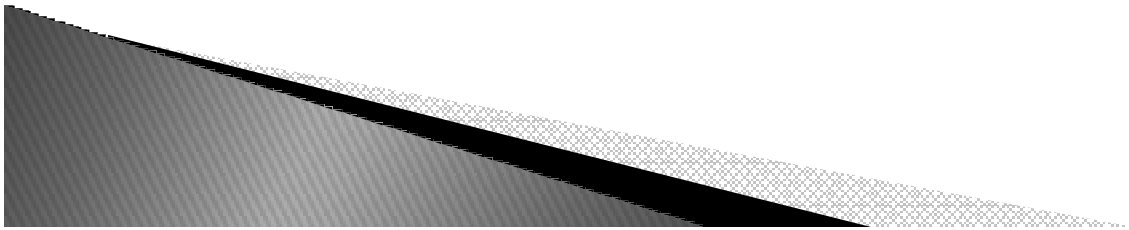
Classe Abstrata

- O que é?
- Para que serve?



Classe Abstrata

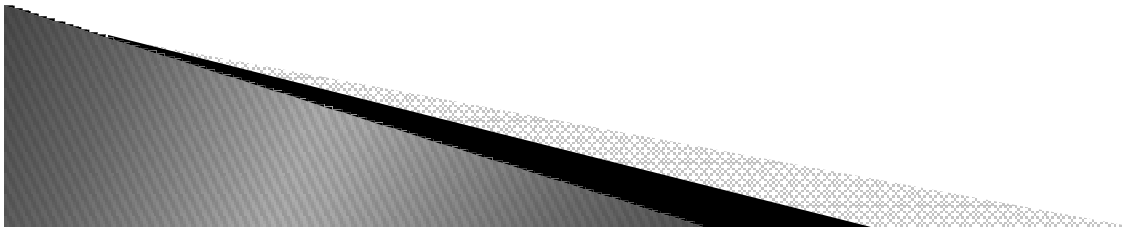
- O que é?
- Para que serve?
- Quando desejamos especificar quais métodos uma classe vai ter sem termos de implementar os métodos, declaramos esses métodos como abstratos e conseqüentemente a classe também.



Classe Abstrata

- Exemplo:

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Foo:                          # In Python 3, you use the syntax
    __metaclass__ = ABCMeta         # class Foo(metaclass=ABCMeta)
    @abstractmethod
    def spam(self,a,b):
        pass
    @abstractproperty
    def name(self):
        pass
```

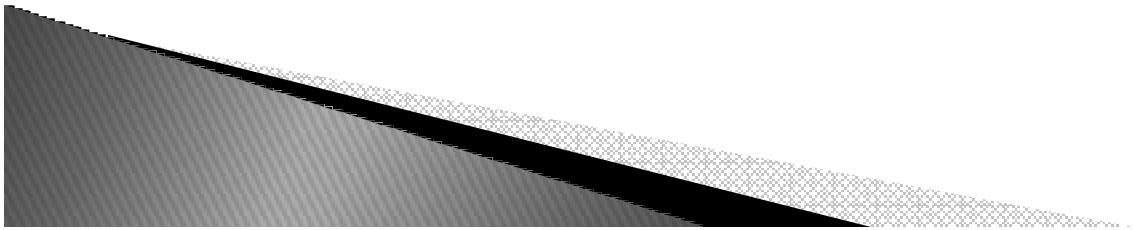


Classe Abstrata

- Característica:
  - Não pode ser instanciada.

```
>>> f = Foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract methods spam
>>>
```

  - As subclasse que vai ter de implementar todos os métodos definidos como abstratos da classe abstrata.



Exercício

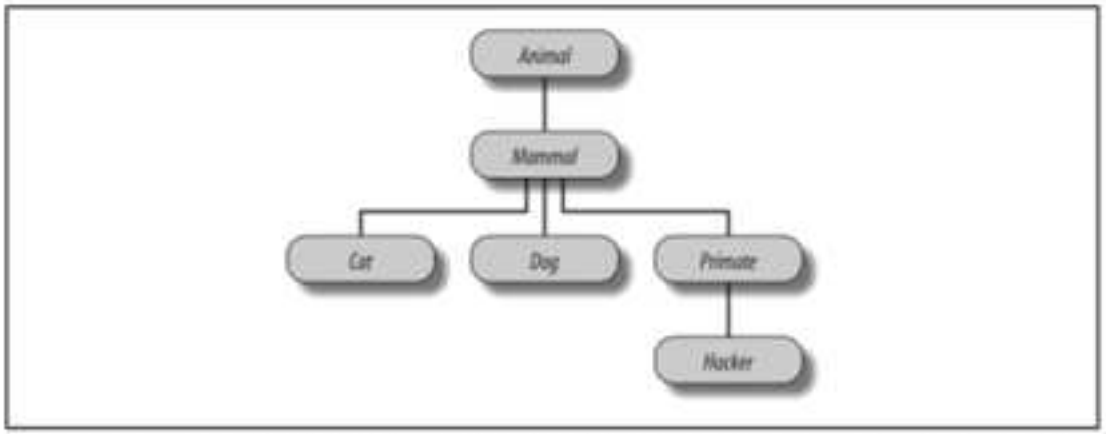


Figure 31-1. A zoo hierarchy composed of classes linked into a tree to be searched by attribute inheritance. Animal has a common "reply" method, but each class may have its own custom "speak" method called by "reply".

3. *Zoo animal hierarchy*. Consider the class tree shown in Figure 31-1.

   Code a set of six `class` statements to model this taxonomy with Python inheritance. Then, add a `speak` method to each of your classes that prints a unique message, and a `reply` method in your top-level `Animal` superclass that simply calls `self.speak` to invoke the category-specific message printer in a subclass below (this will kick off an independent inheritance search from `self`). Finally, remove the `speak` method from your `Hacker` class so that it picks up the default above it. When you're finished, your classes should work this way:

```
% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply()                    # Animal.reply; calls Cat.speak
meow
>>> data = Hacker()                 # Animal.reply; calls Primate.speak
>>> data.reply()
Hello world!
```