Programação Orientada a Objetos (POO)

- Utiliza o conceito de classes para encapsular dados e funções.

```
class Ponto(object):

    def  _init_(self, x,y):
    self.x=x
    self.y=y

   def distancia(self,outro):
    deltaX=(self.x-outro.x)
    deltaY=(self.y-outro.y)
    return (deltaX**2+deltaY**2)**0.5
```

Programação Orientada a Objetos (POO)

- Objetos são instâncias de uma classe.

  p=Ponto(3,4) ⇒ criou-se o objeto p como uma instância da classe Ponto.
  print(p.x,p.y) ⇒ imprime 3 e 4, valores dos atributos x e y do objeto p.
  q=Ponto(4,8) ⇒ outro objeto da classe Ponto.
  d= p.distancia(q) ⇒ calcula distância entre p e q

Programação Orientada a Objetos (POO)

- Para acessar diretamente um dado de um objeto:

  <objeto>.<atributo> ⇒ p.x ou p.y

- Para acessar um método de um objeto:

  <objeto>.<método> ⇒ p.distancia(q)

Programação Orientada a Objetos (POO)

- Nós já estávamos manipulando objetos em Python:
  - L=[10,20,30] é um objeto que instancia um tipo lista.
  - L.append(40) ocorre onde append() é um método de lista.
  - ‘João’ é uma instância do tipo string.
  - 2.5 é uma instância do tipo float.

Programação Orientada a Objetos (POO)

- Utilizando o paradigma da POO podemos
  - Criar objetos de um novo tipo definindo novas classes.
  - Manipular tais objetos
  - Descartar tais objetos.

Programação Orientada a Objetos (POO)

- Classes podem ser vistas como abstrações para atributos de dados
- As especificações das operações executadas em um classe estabelecem uma interface entre um tipo abstrato de dados e o resto do programa.
- A interface estabelece o comportamento das operações, ou seja, o que elas devem fazer….mas não como elas fazem..
- As classes facilitam a reutilização de código.

Programação Orientada a Objetos (POO)

- Criando uma classe: **class** <nome da classe>**(**<classe pai>**) :**
- **class** Ponto(**object**):
  - Ponto herda todos os atributos da classe **object** definida em Python.
  - Ponto é uma subclasse de **object**
  - **object** é uma superclasse da classe Ponto.

Programação Orientada a Objetos (POO)

- Os atributos de uma classe são representados pelos:
  - dados
  - Métodos

```
def distancia(self,outro):
    deltaX=(self.x-outro.x)
    deltaY=(self.x-outro.y)
    return (deltaX**2+deltaY**2)**0.5
```

Programação Orientada a Objetos (POO)

- Um objeto é instanciado através de uma chamada ao método **__init__**.
- O método **__init__** inicia alguns dados do objeto.

```
def __init__(self, a,b):
    self.a=a
    self.b=b
```

Programação Orientada a Objetos (POO)

```
def __init__(self, a,b):
    self.a=a
    self.b=b
```

- **self** se refere ao próprio objeto instanciado.
- p=Ponto(3,4)
  - Ponto(3,4) chama **_init_**(**self**,3,4)
  - **self** não precisa entrar como argumento em Ponto(3,4).

Programação Orientada a Objetos (POO)

```
def distancia(self,outro):
        deltaX=(self.x-outro.x)
    deltaY=(self.y-outro.y)
    return (deltaX**2+deltaY**2)**0.5
```

- **self** se refere ao próprio objeto instanciado.
- outro é o argumento relativo ao outro objeto instanciado
- d=p.distancia(q)
    - **self** se refere ao objeto daquele método, ou seja, p.

Programação Orientada a Objetos (POO)

- **_init_()** é um método reservado de Python.
- Um método reservado é disponibilizado pela linguagem e tem uma finalidade específica.
- Há outros métodos reservados:

__str__()
__add__() +
__sub__() -
__eq__() ==
__lt__() <

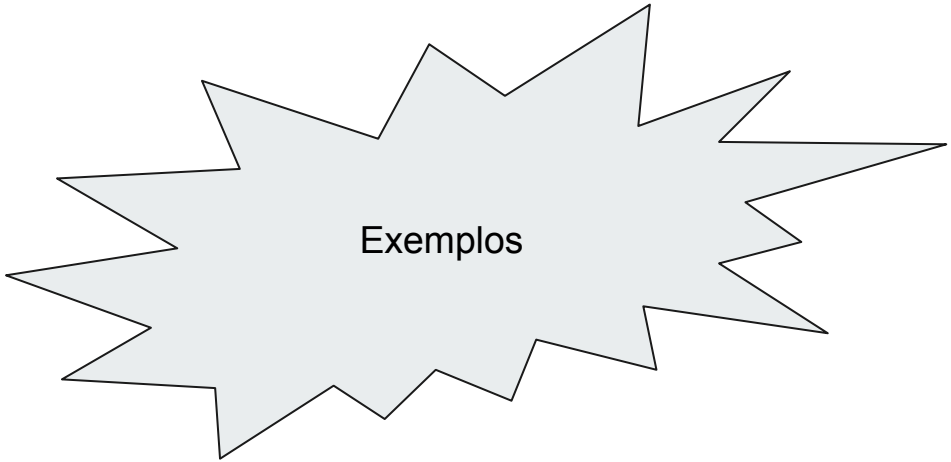
Programação Orientada a Objetos (POO)

```
class Ponto(object):
    def _init_(self, x,y):
            self.x=x
            self.y=y

    def distancia(self,outro):
            deltaX=(self.x-outro.x)
            deltaY=(self.y-outro.y)
            return (deltaX**2+deltaY**2)**0.5
```

```
def _str_(self):
    return '('+str(self.x)+','+str(self.y)+')'
```

```
p=Ponto(3,4)

print(p) ⇒ (3,4)
```

Programação Orientada a Objetos (POO)



Implementar vs utilizar classes

Implementar

- Estabelecer **o que é a classe**.
  - Há atributos de dados que caracterizam a classe.
- Estabelecer **o que a classe faz**.
  - Há métodos que caracterizam as funcionalidades da classe

Utilizar

- Criar instâncias de um objeto.
  - Define-se um novo tipo de objeto a ser manipulado.
- Manipular tais objetos
  - Executar operações com tais objetos.

Implementar vs utilizar classes

- A classe contém a definição de dados e métodos comuns a todas as instâncias da classe.
- O nome da classe define um novo tipo

```
class Ponto(object)
```

- `self é um parâmetro para acessar atributos relativos a uma instância da classe (objeto).`

```
self.x=x
```

Implementar vs utilizar classes

- A instância de uma classe é um objeto específico

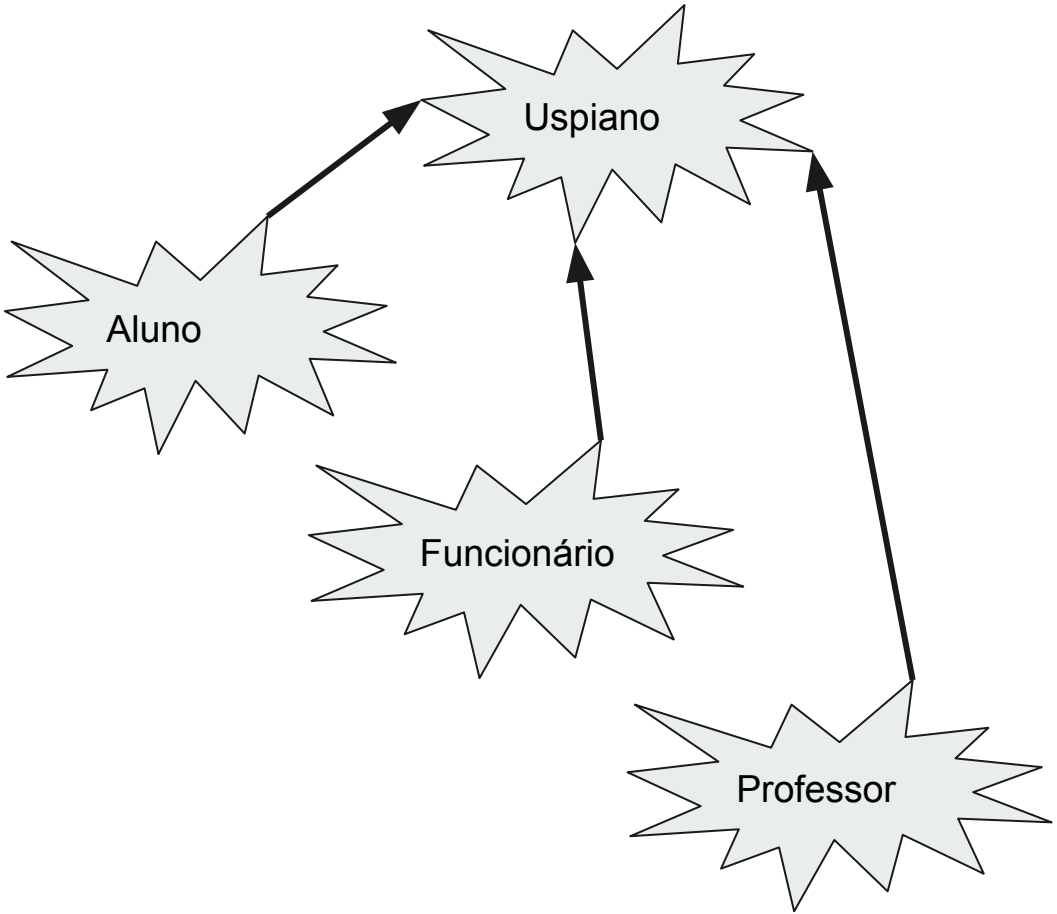
```
p = Ponto(3,4)
```

- Duas instâncias de uma mesma classe podem apresentar valores diferentes

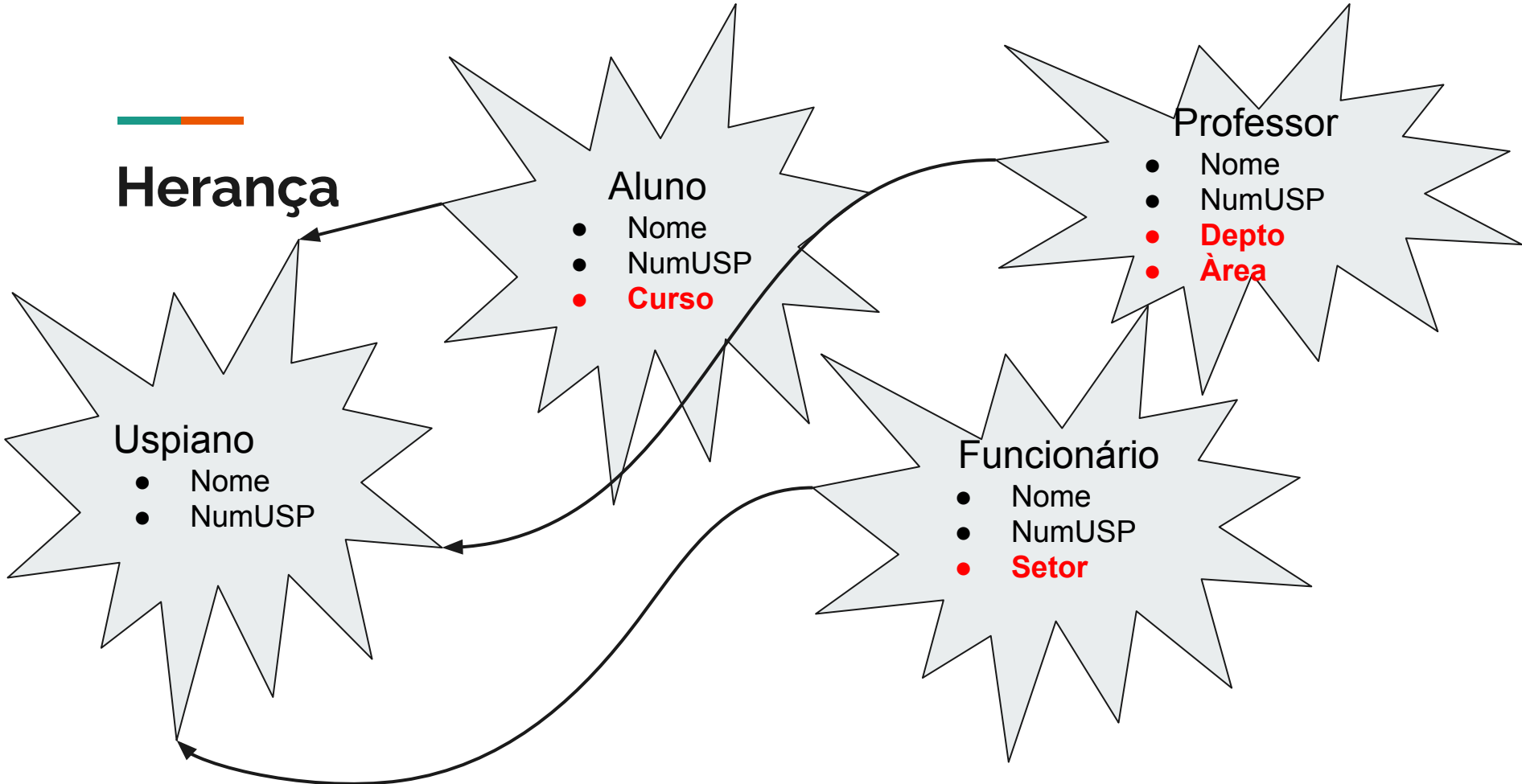
```
p = Ponto(3,4) q = Ponto (2,1)
```

- A estrutura de duas instâncias de uma mesma classe é a mesma.

Herança



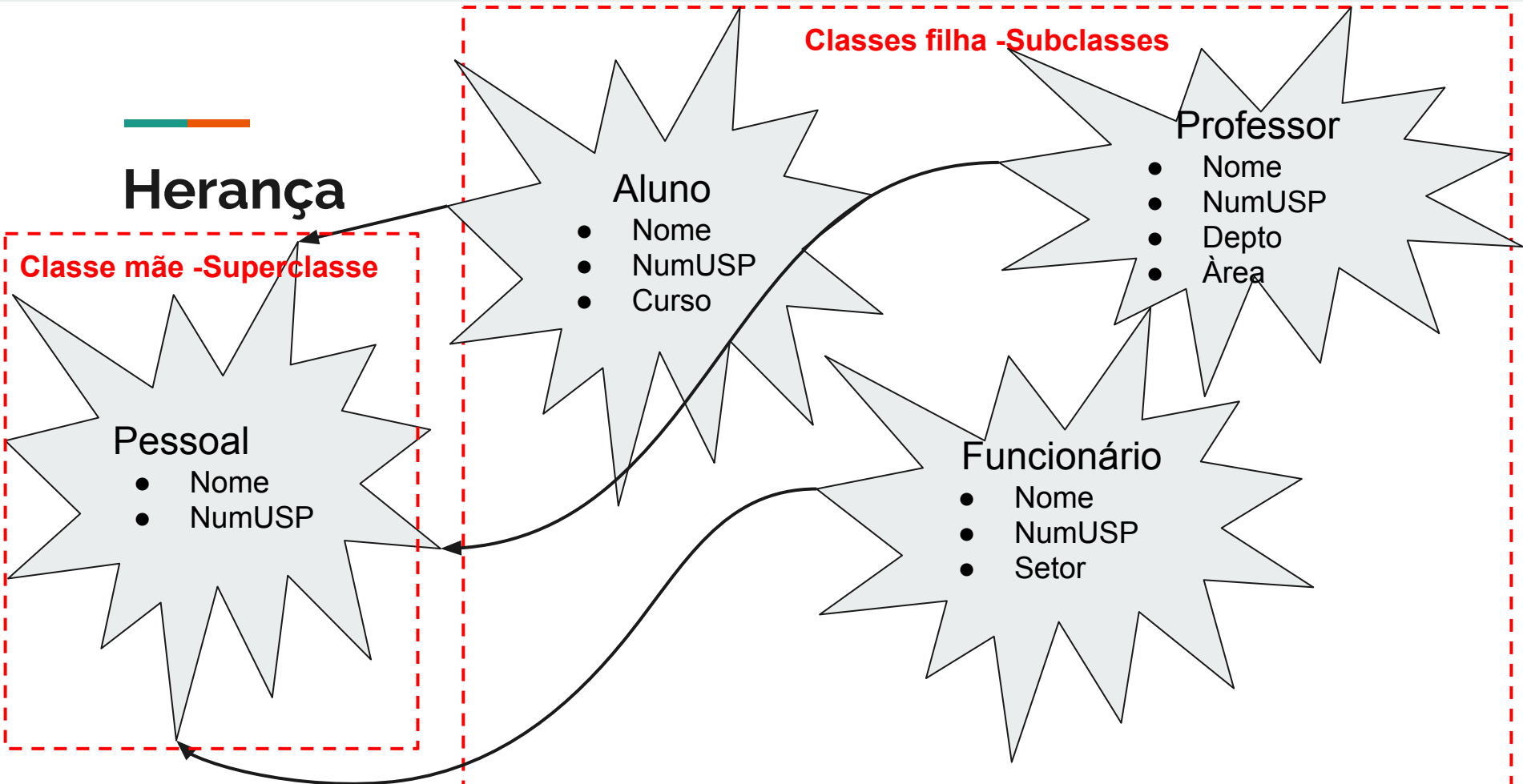
Herança


Aluno
Nome
NumUSP
Curso
Professor
Nome
NumUSP
Depto
Àrea
Uspiano
Nome
NumUSP
Funcionário
Nome
NumUSP
Setor

Herança

- Trata-se de uma forma bastante conveniente de relacionar grupos de abstrações.
- Torna-se possível estabelecer uma hierarquia onde classes filha herdam atributos das classes mãe.

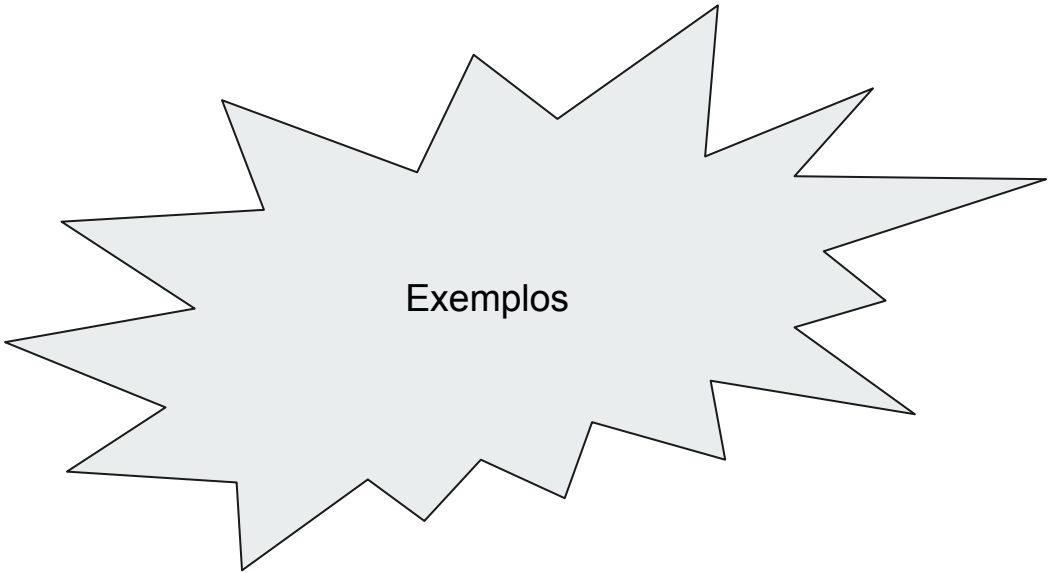
Herança


Classes filha -Subclasses
Classe mãe -Superclasse
Aluno
Nome
NumUSP
Curso
Professor
Nome
NumUSP
Depto
Àrea
Pessoal
Nome
NumUSP
Funcionário
Nome
NumUSP
Setor

Herança

- Subclasse herda os atributos da superclasse: dados e métodos.
- Na subclasse é possível:
  - Adicionar novos dados e métodos
  - Sobrescrever métodos da superclasse.

Herança



Atributos Privados

- Linguagens orientadas a objetos (Java, C++, etc.) controlam ou restrigem o acesso a atributos das classes.
- Nesse caso, os atributos da classe passam a ser classificados como público (*public*), privado (*private*) ou protegido (*protected*).
- Não há mecanismos em Python que impeçam de fato o acesso a qualquer variável ou método de uma classe.
- Todos os atributos de uma classe em Python são públicos por definição.
- **Atributos públicos** (*public*) podem ser acessados fora da classe.

Atributos Privados

- Python permite simular acessos do tipo protegido ou privado através da adição de prefixos aos nomes de variáveis e métodos de uma classe.
- Atributos protegidos (protected) de uma classe podem ser acessados dentro da própria classe e pelas classes filhas.
- Atributos protegidos não podem ser acessados a partir de outros ambientes.
- Utiliza-se 1 underscore: _<nomeAtributo>

Atributos Privados

- Atributos privados têm seu acesso efetivamente impedido a partir de outros ambientes.
- Tentativas de acesso externo resultam em erro.
- Utiliza-se dois underscore: __<nomeAtributo>

Classes Abstratas

- As classes abstratas possuem um ou mais métodos do tipo abstrato.
- Um método abstrato é declarado, mas não é implementado.

```
@abstractmethod
def precisa_implementar(self):
    pass
```

- O decorador `@abstractmethod` torna o método abstrato.
- Python por default não fornece classes abstratas.
- Deve-se importar a Abstract Base Classes (ABCs)

```
from abc import ABC
```

Classes Abstratas