ICC043/IEC582 – Paradigmas de Linguagens de Programação

Paradigma Orientado a Objetos

Prof. Dr. Rafael Giusti
rgiusti@icomp.ufam.edu.br

Baseado em material do Prof. Dr. Marco Cristo (IComp/UFAM)

Programação OO

Paradigma procedural	**Paradigma OO**
Programas são organizados em **procedimentos**, que definem ações	Os programas são organizados em **objetos** que reúnem dados e ações em uma mesma entidade
Cada procedimento tem como finalidade principal realizar alguma ação sobre um conjunto de parâmetros	Cada objeto possui um **estado interno próprio** e um conjunto de ações que podem manipular esse estado
Programas procedurais realizam chamadas de procedimento, manipulando o estado do programa	Os objetos se comunicam entre si trocando **mensagens**, cada qual manipulando seu próprio estado interno

Programação OO

Paradigma funcional	**Paradigma OO**
Programas são declarações e chamadas de funções	Os objetos trocam **mensagens** uns com os oturos
As funções promovem uma separação total dos dados e das ações que devem ser realizada sobre esses dados	Os objetos reúnem em uma mesma entidade dados e as ações que devem ser tomadas sobre esses dados
Não existe estado	Cada objeto possui seu próprio estado interno, que **não deve ser manipulado por outros objetos**

Programação OO

Paradigma lógico	**Paradigma OO**
Não tem funções	As ações são normalmente implementadas como funções
Execução consiste em provar que uma declaração lógica é verdadeira	Execução consiste em troca de mensagens entre os objetos
Declarativo: os programas declaram o quê deve ser provado, não como	Normalmente imperativo: os programas são comandos que dizem quais mensagens devem ser enviadas a quais objetos

Agenda

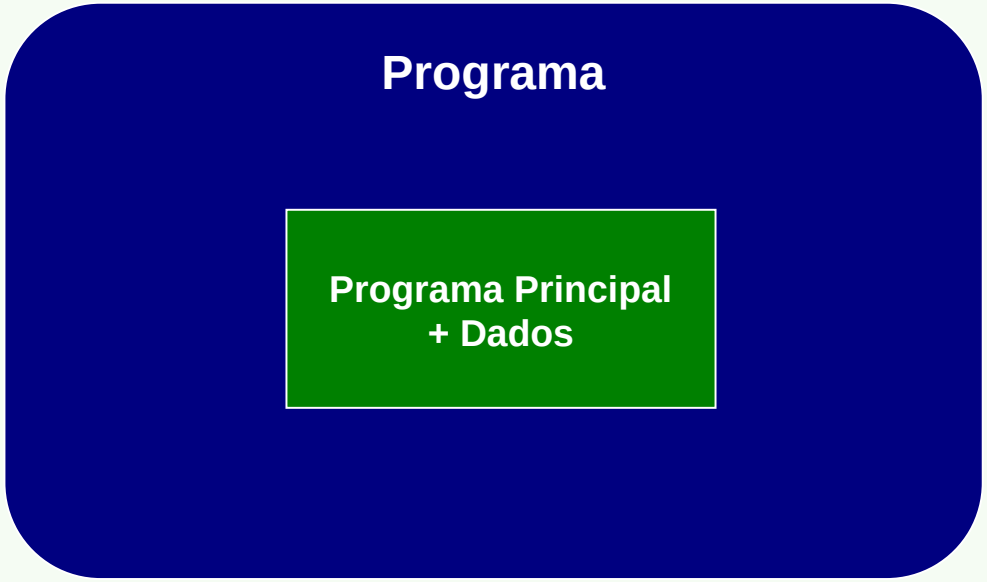
- Prelúdios da orientação a objetos
- Classes e objetos
- Objetos em Python
- Encapsulamento
- Herança
- Polimorfismo de herança

Crise do Software

- Dificuldade em atender demandas
  - Projetos maiores e mais complexos
  - Modificações/melhorias de projetos existentes
- Por quê?
  - Sistemas novos: "reinvenção da roda"
  - Sistemas antigos: mudanças introduzem erros no que já estava funcionando
- Solução: reúso de código

Programação não estruturada

- Dificuldade de reúso devido à programação não estruturada
  - Estilo predominante até a década de 1960
  - Quase um "*go horse*"



Programas não são bem estruturados em funções e módulos

Existe um estado único que todas as diferentes partes do programa manipulam

Uso ostensivo de goto

Programação estruturada

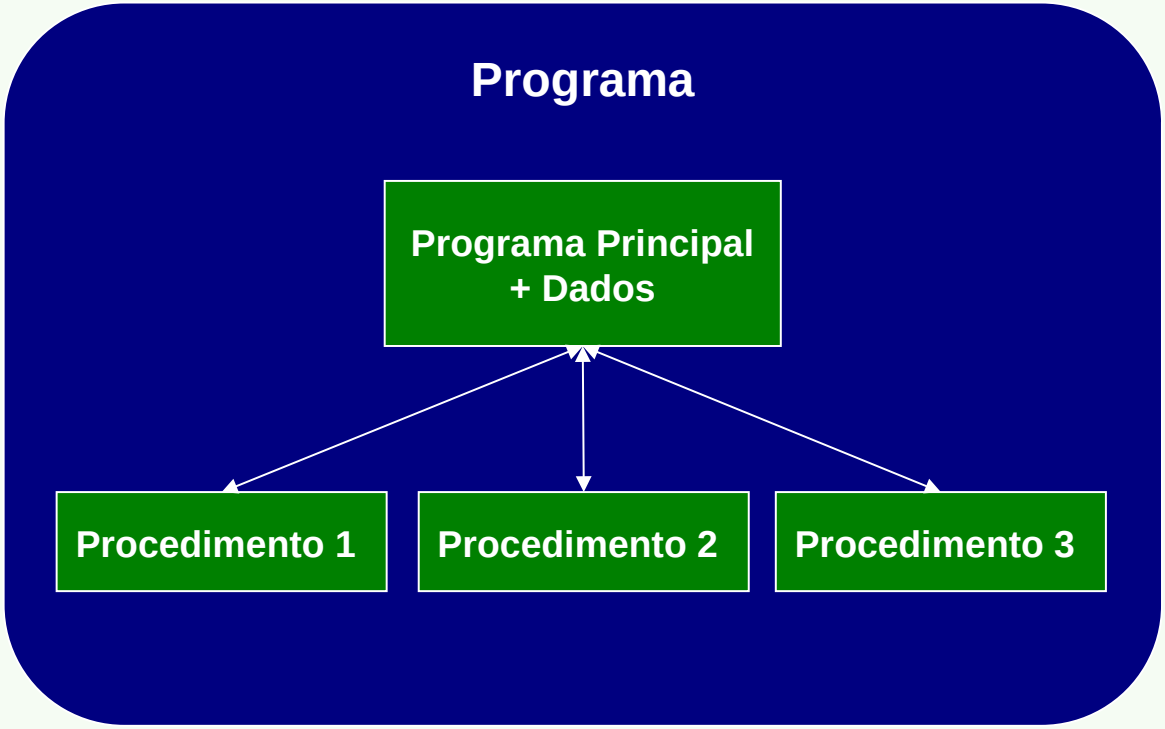
- Avanços da Engenharia de Software na década de 1970
  - Organização do programa em procedimentos parame-trizáveis
    - Estado governado por um programa principal
    - O comportamento dos procedimentos depende, idealmente, apenas dos parâmetros (diminuição dos efeitos colaterais)

Edsger W. Dijkstra

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Programação estruturada

- Programação estruturada
  - Programa principal gerencia o estado e coordena os procedimentos



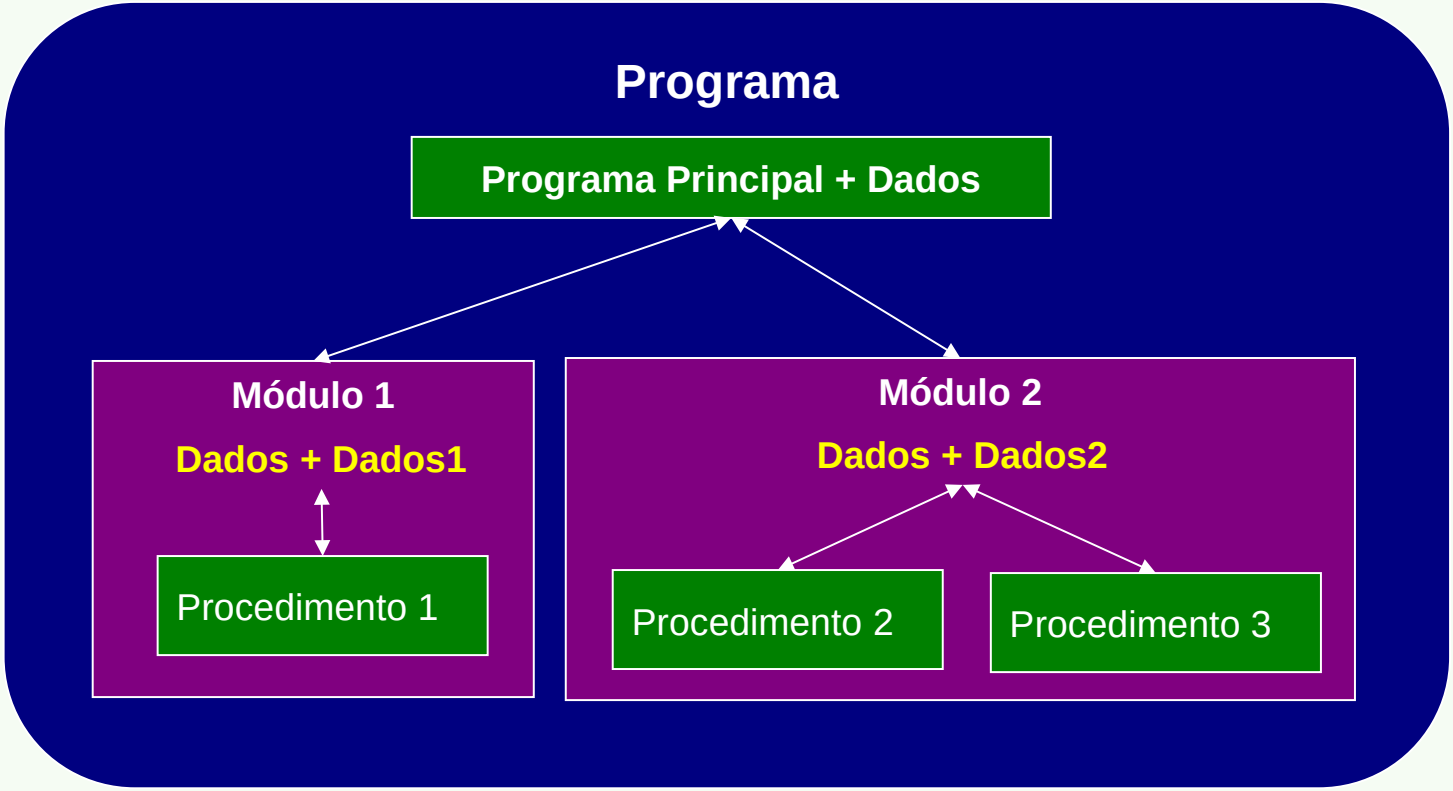
Procedimentos dependem muito mais dos parâmetros do que do estado do programa

Uso de estruturas de controle (if, while, for etc.) no lugar de goto

Maior possibilidade de reúso

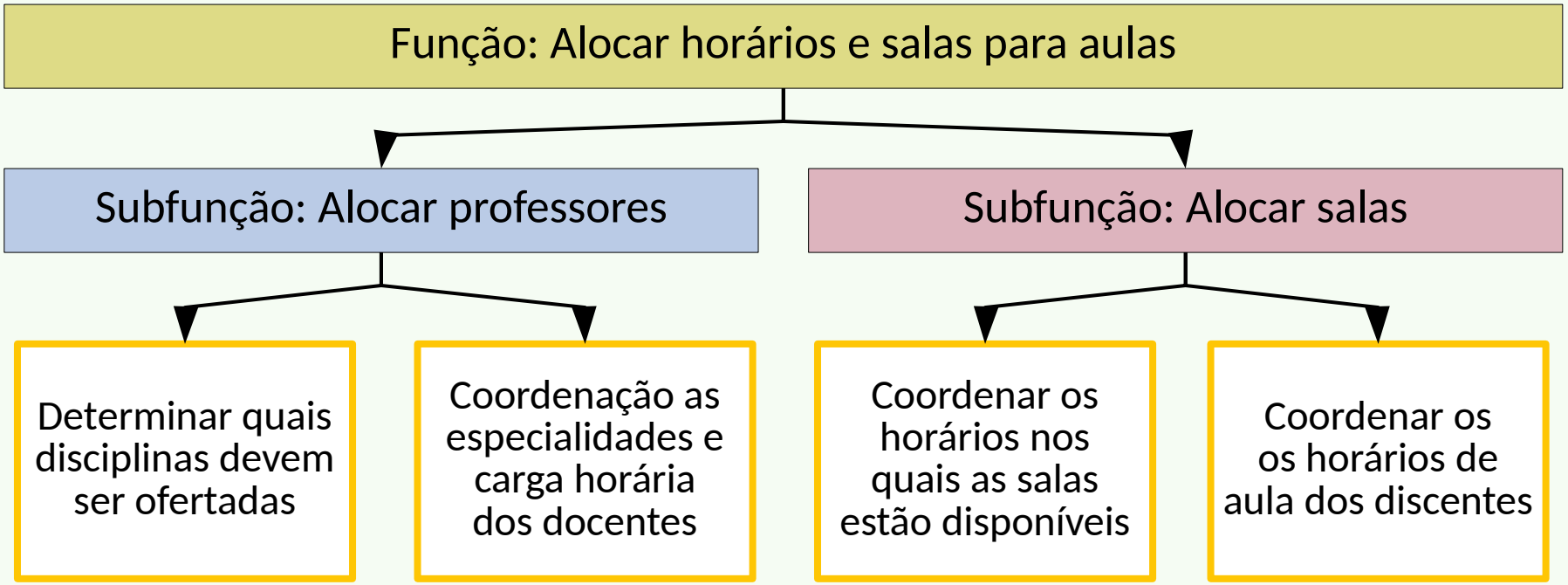
Programação modular

- Organização do programa em módulos
  - Cada módulo executa uma função própria
  - Estilo dominante até a década de 1980



Programação modular

- Modularização por decomposição funcional
  - Módulos determinados pelas funções que devem executar no sistema
    - Quais funcionalidades devem gerar módulos?



Programação modular

- Problemas:
  - Dados e operações não são associados
    - Módulos agrupam operações semelhantes
    - As operações determinam quais dados são necessários
    - Quem detém os dados?
      - Necessidade de compartilhar dados entre vários módulos
      - Dados podem ser manipulados por qualquer operação externa ao módulo

Programação modular

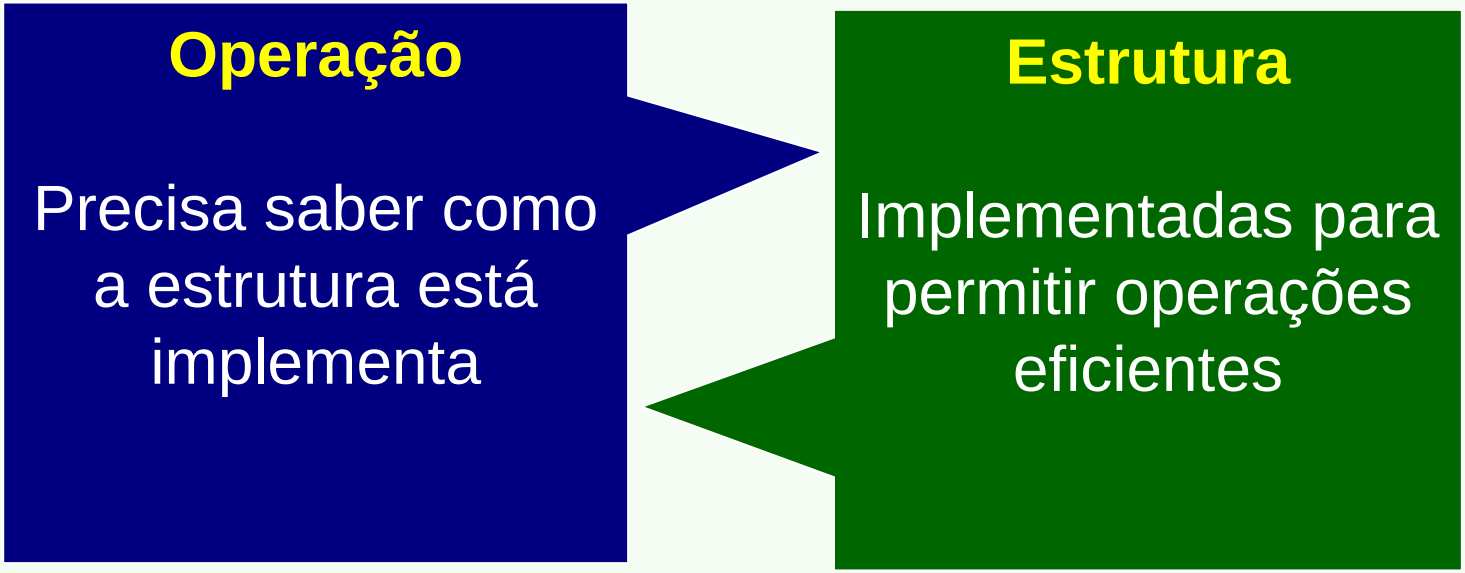
- Problemas:
  - Uso pouco flexível de tipos de dados
    - Dificuldade em permitir que uma estrutura (por exemplo, uma árvore) manipule diferentes tipos de dados (exemplo: *strings* e bytes)
  - Separação entre *representação da estrutura* e *operações de acesso*
    - É difícil definir vários tipos de acesso para uma mesma estrutura

Programação modular

- Problemas:
  - Decomposição funcional gera módulos muito dependentes entre si (acoplamento)
    - Cada módulo desempenha uma função
    - Os módulos se tornam acoplados a muitos outros módulos
    - O reúso de um módulo implica a reutilização de muitos outros módulos
    - A adaptação de um módulo para outro programa pode exigir adaptação de muitos módulos acoplados

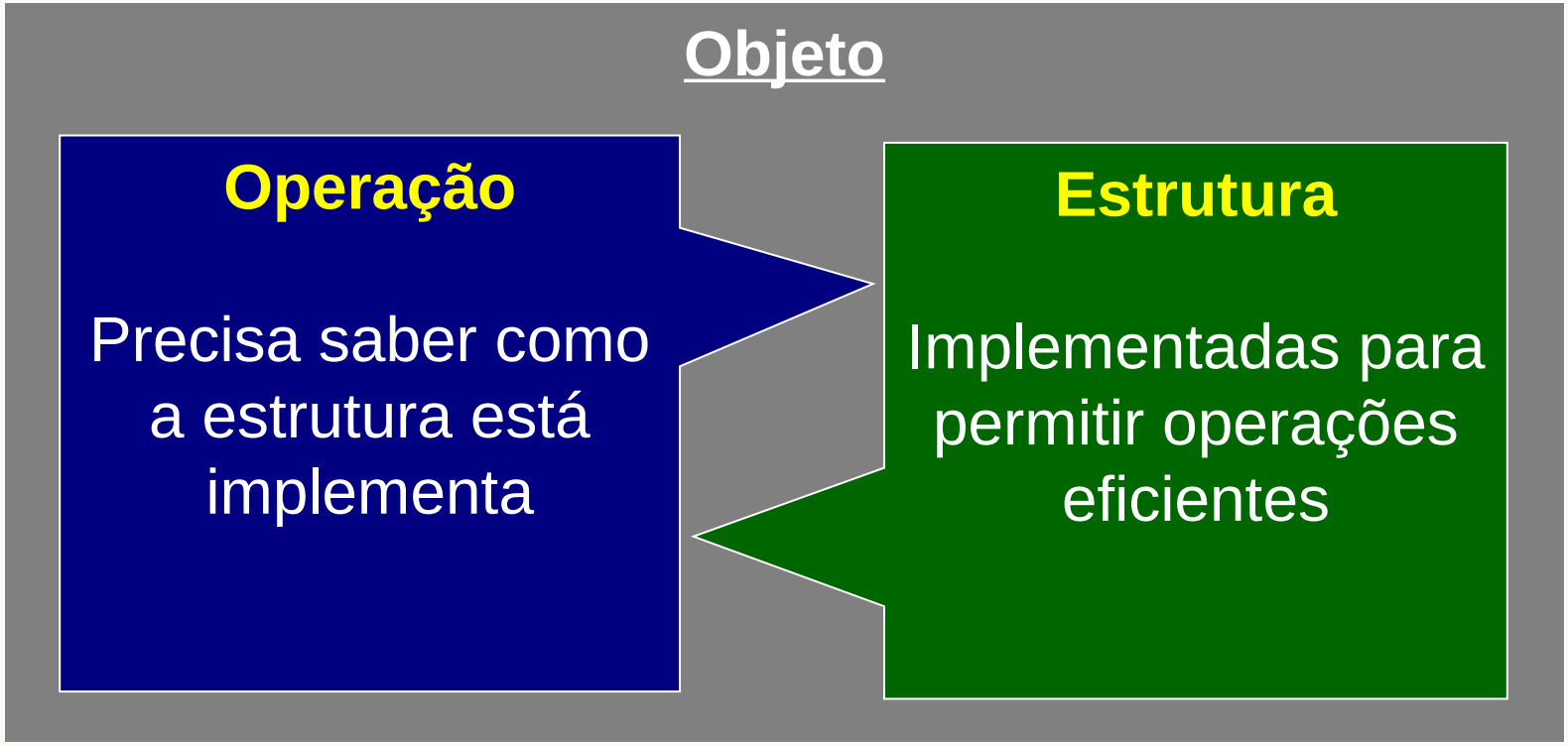
Programação modular

- **Solução:**
  - Considerar a dependência mútua entre dados e operações na modularização!



Programação modular

- **Solução:**
  - Considerar a dependência mútua entre dados e operações na modularização!



O segundo "O" em "POO"

- Em POO, um objeto é
  - Uma estrutura de dados e um conjunto de operações sobre ela
  - Um tipo abstrato de dados
    - Exemplo: em Python, listas são objetos
      - A estrutura de dados é a forma como Python representa os elementos da lista
      - As operações podem ser obter o tamanho da lista, acessar um elemento em particular, concatenar a lista etc.

O primeiro "O" em "POO"

- Questões de projeto
  - Não devemos pensar nas funções dos objetos
  - Pensemos nas características dos objetos e como eles se relacionam

O tabuleiro e as pedras são objetos?

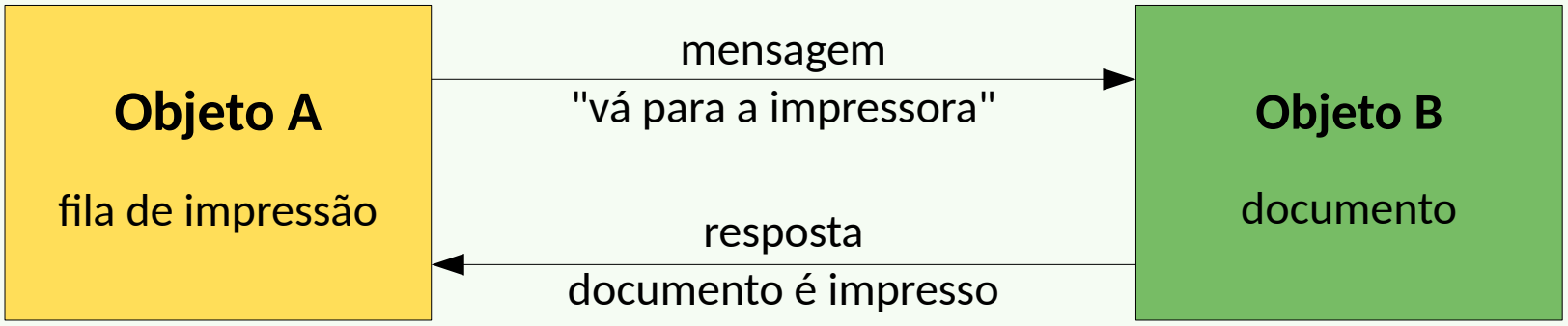
As pedras possuem características em comum?

Como elas se relacionam entre si e com o tabuleiro?

https://www.rei.com/product/871860/outside-inside-magnetic-chess-game

Mensagens

- Os objetos se comunicam trocando mensagens
  - Uma mensagem é uma solicitação para que um objeto realize uma ação
  - A mensagem faz com que um objeto execute um método, provocando uma resposta

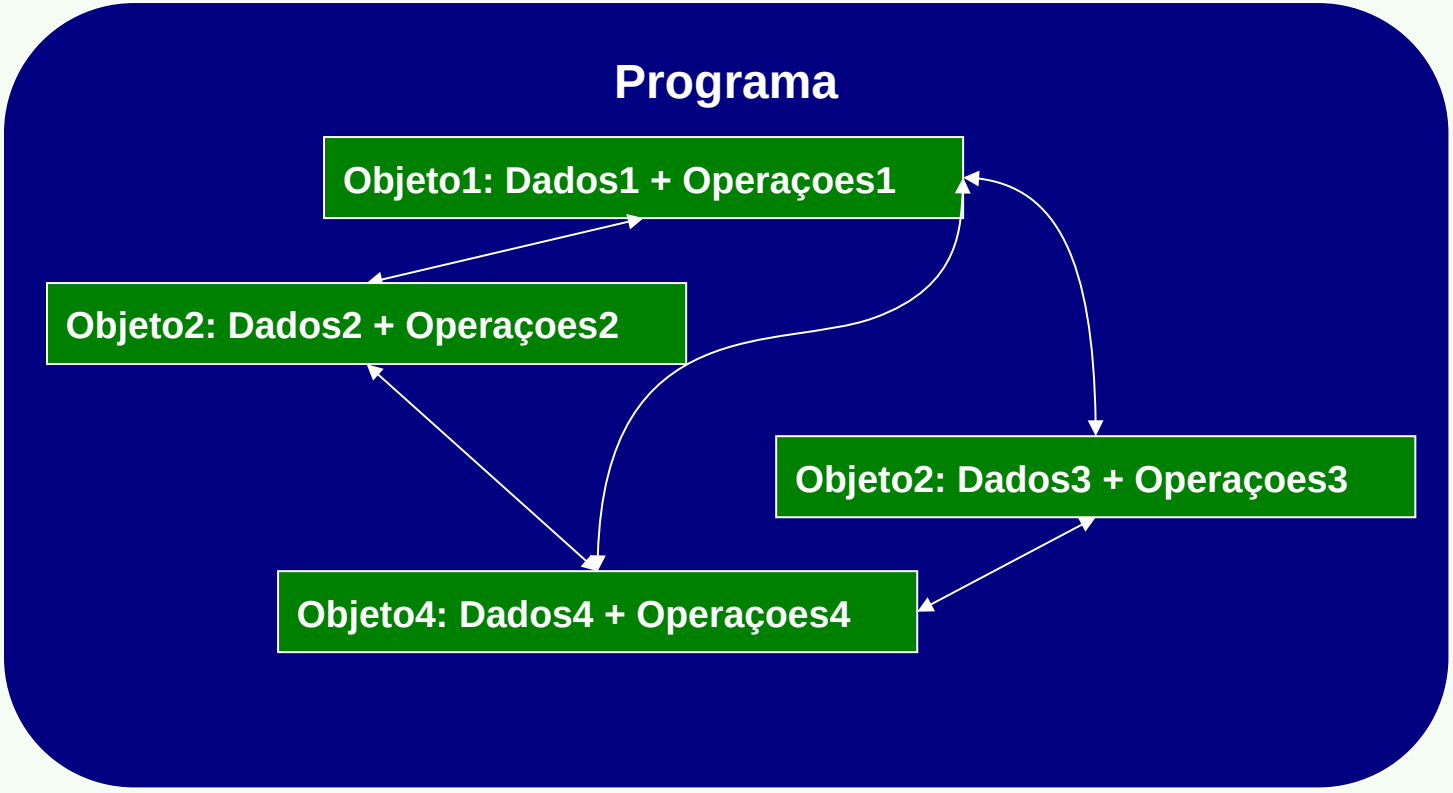


Mensagens

- Pontos importantes
  - A resposta não é necessariamente um valor que o objeto retorna e sim qualquer reação a uma mensagem
  - A resposta pode depender
    - Das variáveis internas ao objeto
    - Dos parâmetros da mensagem
  - O resultado pode ser
    - Um valor
    - Um efeito colateral

Paradigma OO

- Programação orientada a objetos
  - Programas são coleções de objetos que trocam mensagens



Evolução

Época	Dominada por	Características	Linguagens
Até 1955	Sem linguagens de programação	Interpretadores, linguagens de máquina e montagem	Pseudocódigos e Assembly
Até 1955	Programação não estruturada	Programas sem estrutura formal	FORTRAN e COBOL
Até a década de 1970	Programação estruturada	Programas estruturados em procedimentos e funções	C, Pascal, ALGOL
Até a década de 1980	Programação modular	Programas estruturados em módulos (decomposição funcional)	Modula2, ANSI C, Ada, Borland Pascal
A partir da década de 1980 até hoje	Programação orientada a objetos	Sistemas modelados em função das características dos objetos e suas interações	Smalltalk, C++, Java, Python, Swift, Ruby

Agenda

- Prelúdios da orientação a objetos
- Classes e objetos
- Objetos em Python
- Encapsulamento
- Herança
- Polimorfismo de herança

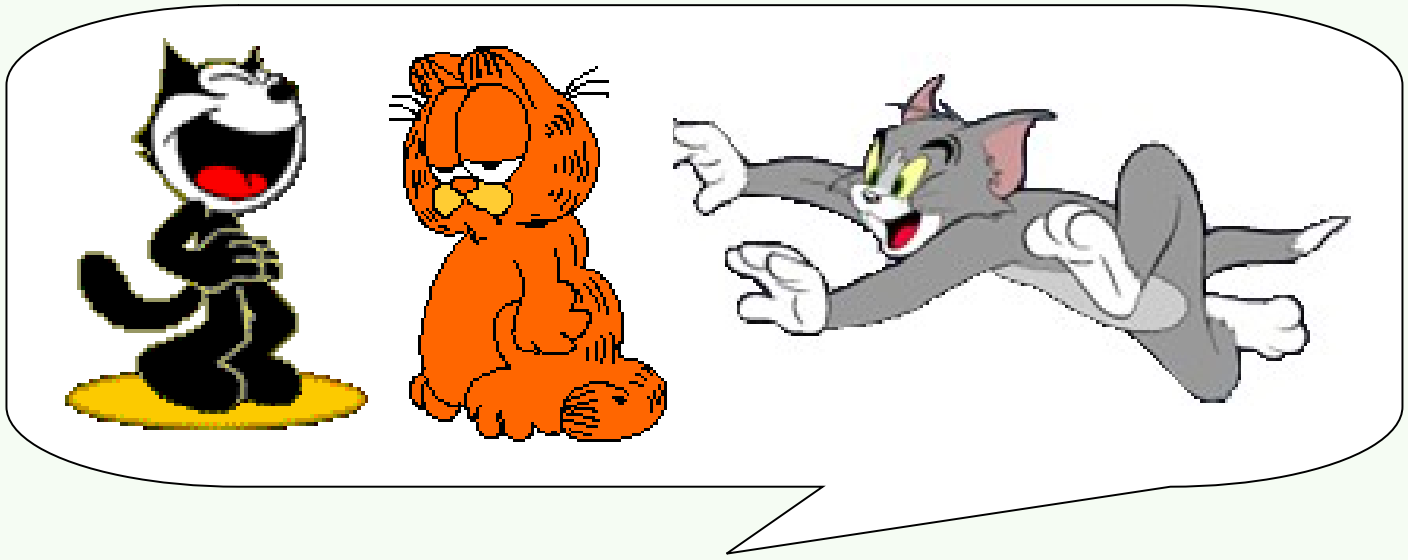
Classe e objeto

- Uma analogia...

O que há em comum entre o Félix, o Garfield e o Tom?

Classe e objeto

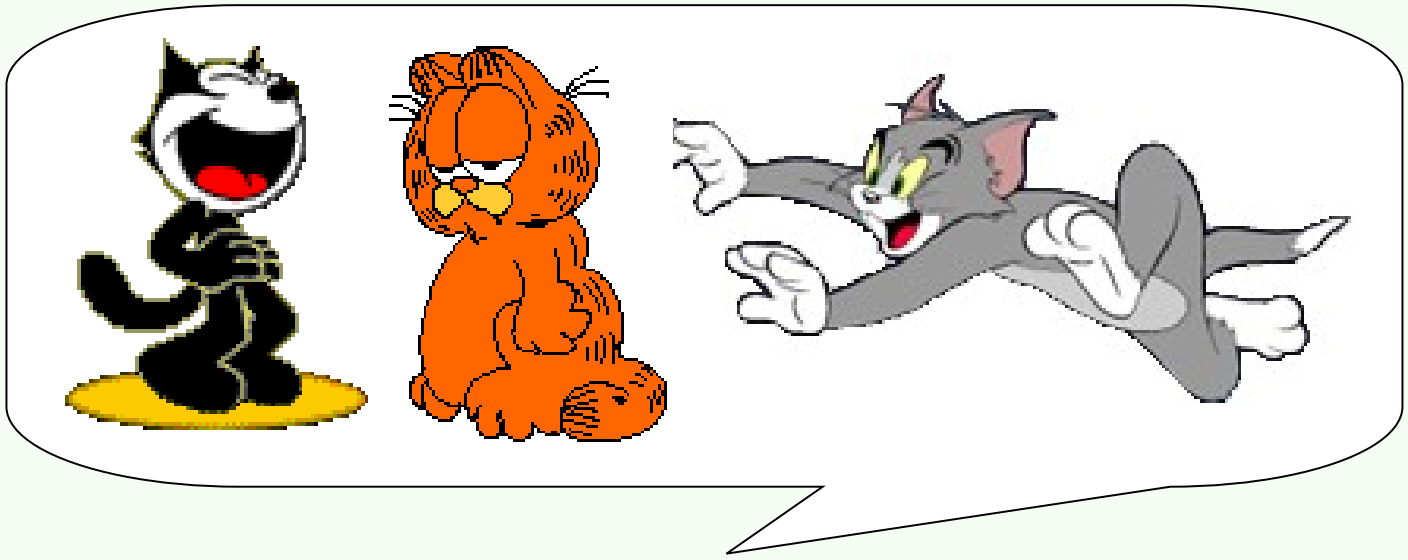
- Uma analogia...



Espécie é um agrupamento de indivíduos que revelam profundas semelhanças entre si, tanto no aspecto estrutural quanto funcional.

Classe e objeto

- Uma analogia...



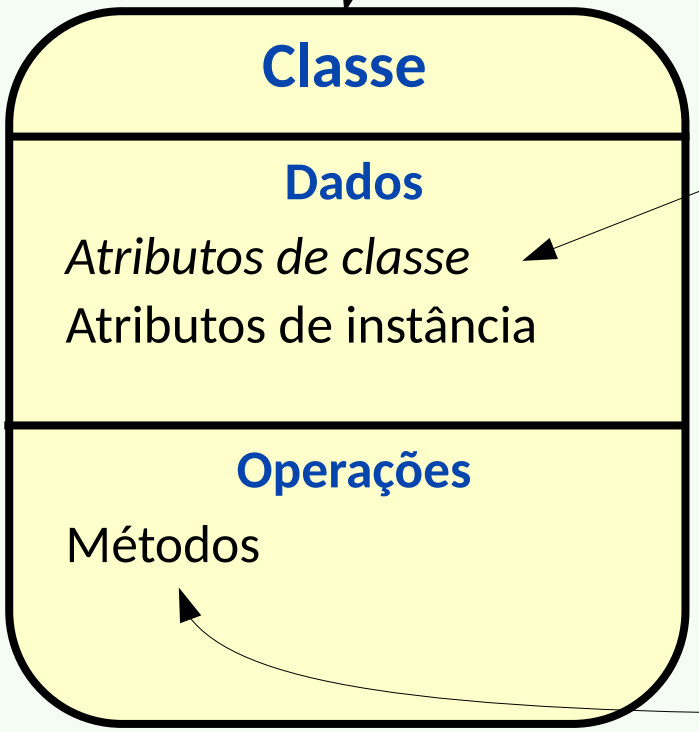
Classe é um agrupamento de objetos que revelam profundas semelhanças entre si, tanto no aspecto estrutural quanto funcional.

Classe e objeto

- A classe é o tipo abstrato de dados
  - Define os dados e as operações
  - É uma espécie de "modelo"
  - Na prática é um um tipo de dados

- O objeto é a realização da classe
  - Também denominado instância
  - Contém os valores concretos dos dados
  - Na prática é uma variável

Classes



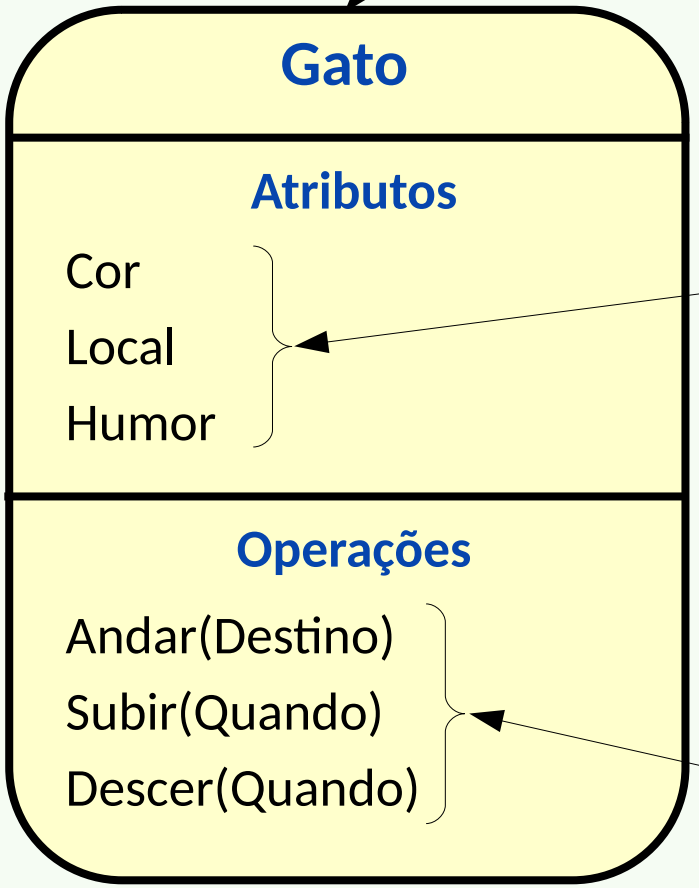
O nome da classe define um tipo abstrato de dados.

Dados (ou atributos) são variáveis que podem pertencer à classe ou às instâncias (objetos).

Coletivamente, referimo-nos aos atributos da classe e das instâncias como os membros da classe.

Métodos são procedimentos que definem as operações assoicadas às classes.

Classes



Gato (espécie) é um tipo de dados que agrupa todos os animais que se parecem e se comportam como gatos.

Os atributos de instância são as características que devem ser especificadas por cada objeto (espécime).

Existe algum atributo para o qual *todos* os gatos possuem o mesmo valor? Isto é, a classe Gato deve ter atributos de classe?

Os métodos definem o comportamento dos gatos. Vamos pensar que gatos podem andar para algum lugar, subir em algum lugar e descer de algum lugar.

Objetos

- Os objetos são as realizações das classes

Garfield: Gato

Atributos

Cor = Laranja

Local = Sala

Humor = Emburrado

Garfield é um gato de pelo laranja, possui 8 anos de idade e é muito emburrado.

Os métodos pertencem à classe, não aos objetos!!

Objetos

Felix: Gato

Atributos

Cor = Preto

Local = Jardim

Humor = Alegre

Garfield: Gato

Atributos

Cor = Laranja

Local = Sala

Humor = Emburrado

Tom: Gato

Atributos

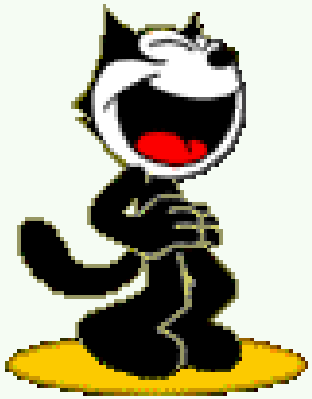
Cor = Cinza

Local = Cozinha

Humor = Arteiro

Estado interno

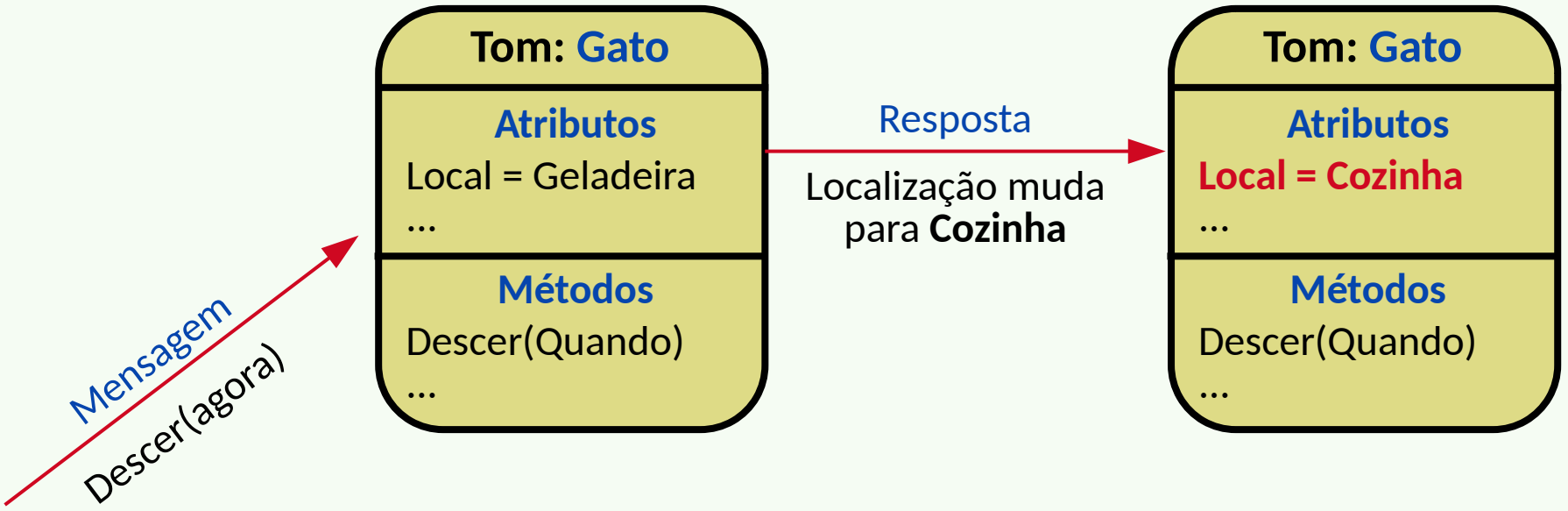
- O conjunto de valores das variáveis de instância é o estado interno de um objeto
  - Quando um objeto é instanciado, seu estado interno deve ser definido
  - O estado interno do objeto pode mudar durante a execução do programa?
    - O Félix *sempre* é alegre?
    - Ele fica sempre no jardim?



Mensagem

- Objetos se comunicam através de mensagens
- Uma mensagem possui duas partes
  - O seletor é o nome da mensagem
  - Os parâmetros são variáveis que podem ser associadas às mensagens
- Quando um objeto recebe uma mensagem, um método é ativado
- O resultado da execução do método é a resposta

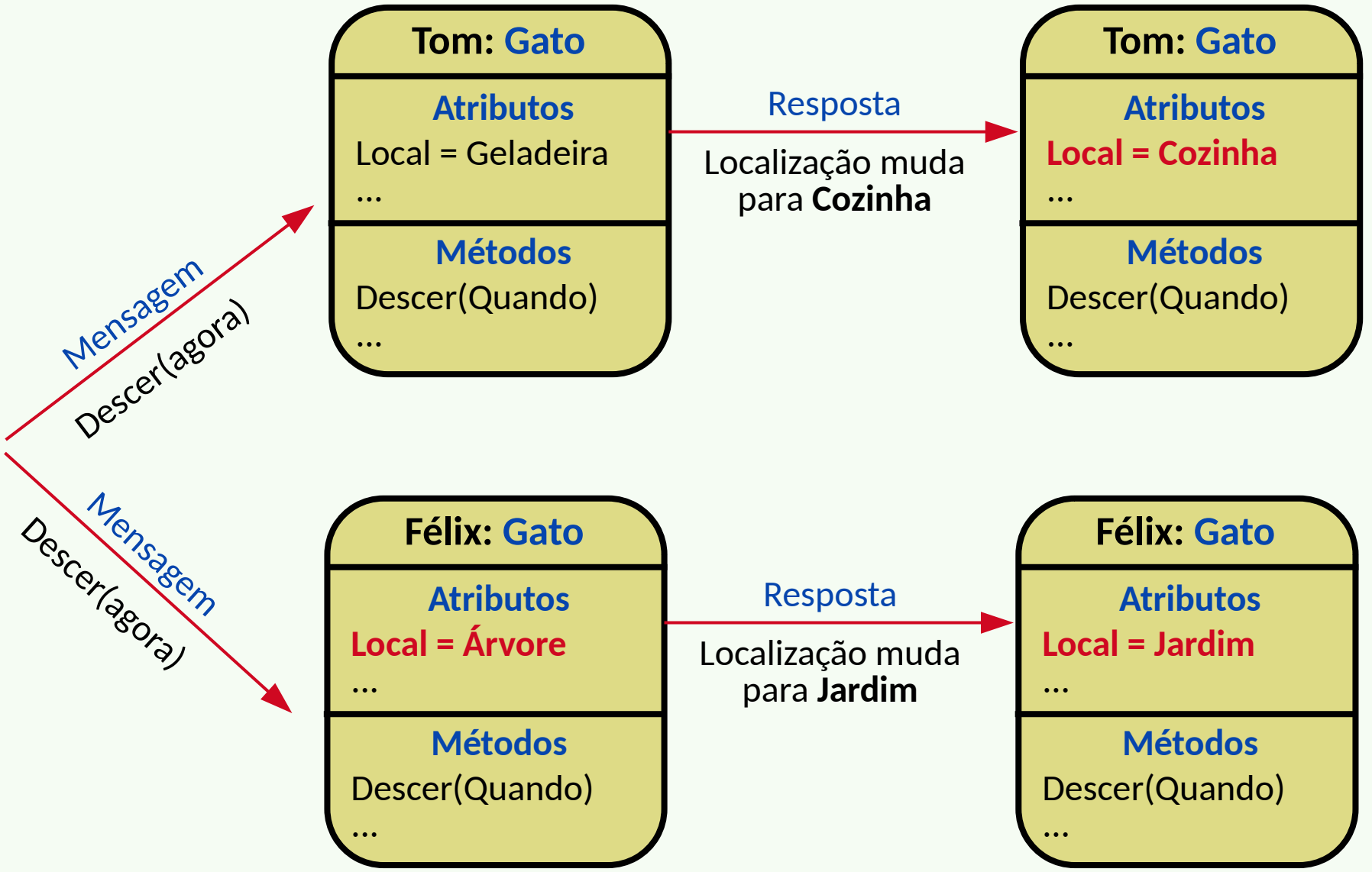
Mensagem



Quando o objeto Tom recebe a **mensagem** **Descer(agora)** o método **Descer(Quando)** é **ativado**. O parâmetro **Quando** está vinculado ao argumento **agora**.

A **resposta** de um método depende **do estado interno do objeto e dos argumentos da mensagem**. Neste exemplo, Tom **responde** imediatamente alterando seu local de cima da geladeira para o chão da cozinha.

Mensagem


Tom: Gato
Atributos
Local = Geladeira
...
Métodos
Descer(Quando)
...
Resposta
Localização muda para Cozinha
Tom: Gato
Atributos
Local = Cozinha
...
Métodos
Descer(Quando)
...
Mensagem
Descer(agora)
Mensagem
Descer(agora)
Félix: Gato
Atributos
Local = Árvore
...
Métodos
Descer(Quando)
...
Resposta
Localização muda para Jardim
Félix: Gato
Atributos
Local = Jardim
...
Métodos
Descer(Quando)
...

Agenda

- Prelúdios da orientação a objetos
- Classes e objetos
- Objetos em Python
- Encapsulamento
- Herança
- Polimorfismo de herança

Objetos em Python

- Tudo em Python são objetos
  - Exemplo: literais reais são objetos da classe `float` que Python instancia automaticamente
    - Expressões são *syntax sugar* para trocas de mensagens entre objetos

```
>>> 3.14 * 2
6.28

>>> 3.14.__mul__(2)
6.28
```

O **objeto** 3.14 recebe a **mensagem** __mul__(2), **ativando** um **método**. A **resposta** do objeto é instanciar um novo objeto da classe float que representa o valor 6,28

Objetos em Python

- Tipos complexos
  - A classe `list` é um tipo abstrato para listas
  - A classe `dict` fornece dicionários (ou vetores de associação) que são estruturas do tipo chave => valor
  - A classe `tuple` é um tipo de lista cujos valores não mudam nunca

Use type(x) para ver o tipo de um objeto

Use dir(x) para ver as variáveis e métodos de um objeto

Objetos em Python

- Tipos atômicos
  - `int`: números inteiros com precisão indeterminada (*big integers*)
  - `float`: números reais
  - `str`: cadeias de caracteres
  - bool: valores *booleanos*

Use type(x) para ver o tipo de um objeto

Use dir(x) para ver as variáveis e métodos de um objeto

Objetos em Python

- Cada objeto possui um identificador
  - Use `id(x)` para descobrir o identificador de uma instância

```
>>> id(42)
9303456

>>> id(40 + 2)
6.28

>>> id(int)
8747648
```

Em Python, até mesmo tipos são objetos!

Objetos em Python

- Cada objeto possui um identificador
  - O operador binário **is** resulta em uma expressão verdadeira se os operandos são o mesmo objeto

```
>>> type([]) is list
True

>>> type(list) is list
True
```

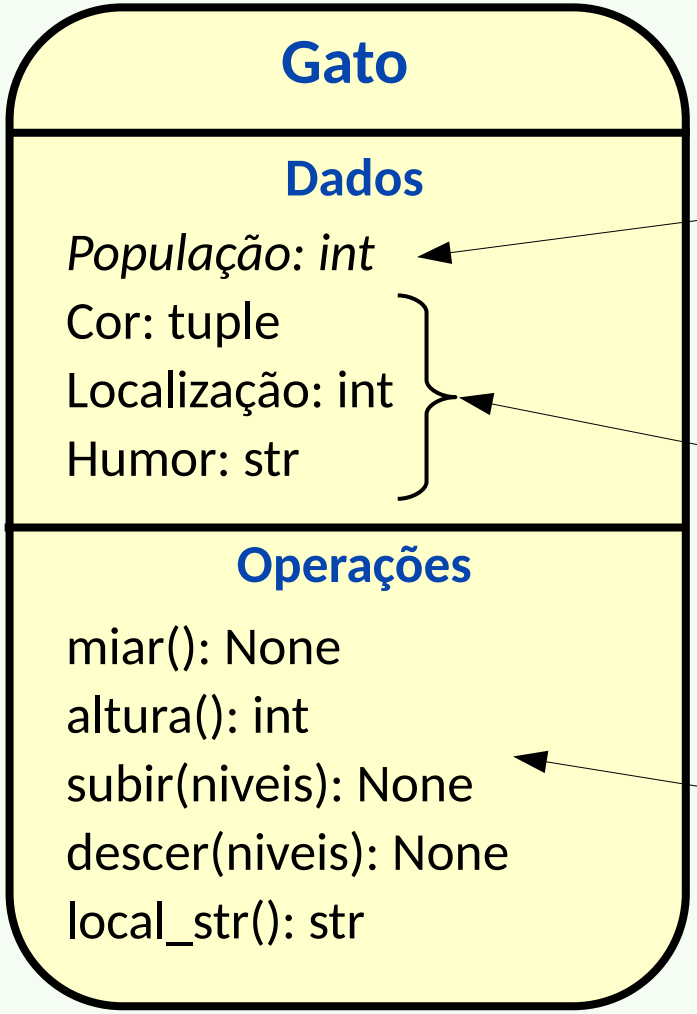
Objetos em Python

- Variáveis são referências
  - O operador de atribuição cria um vínculo entre uma referência e um objeto

```
>>> id(42)
9303456

>>> x = 42
>>> id(x)
9303456

>>> x is 42
True
```

LET'S DO THIS

- Vamos definir uma classe?



Gato.populacao será apenas um exemplo de como usar atributos de classes. Em geral devemos evitar atributos de classes.

Os demais atributos têm os tipos indicados à direita do nome.

Alguns métodos não retornam nada. Nesses casos o tipo de retorno é None. Para os demais, o tipo de retorno é especificado.

LET'S DO THIS

- Uma classe pode ser declarada e definida com a palavra reservada `class`
  - Métodos são declarados como funções no escopo da classe
  - Todo método precisa ter, pelo menos, o parâmetro `self`
    - Lembre-se que todo método é ativado quando algum objeto recebe uma mensagem
    - O `self` referencia o objeto que recebeu a mensagem

LET'S DO THIS

```
>>> class Gato:
...     def miar(self):
...         print("Miau!")
...

>>> mingau = Gato()

>>> type(mingau)
__main__.Gato

>>> mingau.miar()
Miau
```

Gato é uma classe que tem apenas um método e nenhum atributo (por enquanto).

O comando Gato() **instancia** um objeto da classe, isto é, cria uma nova instância da classe

Para enviar uma **mensagem** a um objeto, usa-se a notação **objeto.seletor(argumentos)**

LET'S DO THIS

- Variáveis de classes são definidas dentro do escopo da classe, fora de funções

```
>>> class Gato:
...     populacao = 0
...
...     def miar(self):
...         print("Miau!")
...

>>> Gato.populacao
0
```

O atributo de classe é vinculado à classe imediatamente e pode ser acessado com a notação **Classe.atributo**

LET'S DO THIS

- O método `__init__` é o construtor da classe
  - O objetivo do construtor da é definir o estado interno inicial das instâncias

```
>>> class Gato:
...     populacao = 0
...
...     def __init__(self, cor, humor):
...         self.humor = humor
...         self.cor = cor
...         self.localizacao = 0
...         Gato.popuplacao += 1
```

LET'S DO THIS

- Para a altura, vamos simplificar e estabelecer que o gato pode estar
  - No chão (altura = 0)
  - Em cima da mesa
  - Sobre a pia
  - Em cima da geladeira (altura = 3)
- O gato só pode subir e descer desses lugares

```
>>> class Gato:
...     # atributos de classe e outros metodos
...
...     def altura(self):
...         return self.localizacao
...
...     def subir(self, niveis):
...         if self.localizacao + niveis > 3:
...             print("Nao e' possivel subir acima "
...                 "da geladeira!")
...         else:
...             localizacao += niveis
...
...     def descer(self, niveis):
...         if self.localizacao - niveis < 0:
...             print("Nao e' possivel descer "
...                 "abaixo do chao!")
...         else:
...             localizacao -= niveis
```

LET'S DO THIS

- O código completo, incluindo *docstrings*, pode ser encontrado no ColabWeb
  - As *docstrings* podem ser utilizadas para consultar a documentação da classe e dos métodos
    - Abra o ipython3 no mesmo diretório em que se encontra o arquivo `gato.py`
    - Execute, no ipython3
      - `% run gato.py`
      - `help(Gato)`

Agenda

- Prelúdios da orientação a objetos
- Classes e objetos
- Objetos em Python
- Encapsulamento
- Herança
- Polimorfismo de herança

Controle de acesso

- Nossa classe Gatos possui alguns problemas
  - O que acontece se mudarmos forçadamente a localização do gato?

```
>>> mingau = Gato(cores["laranja"], "sereno")
>>> mingau.subir(2)
>>> mingau.local_str()
'pia'
>>> mingau.subir(2)
Nao e' possivel subir acima da geladeira!
>>> mingau.localizacao = 4
>>> mingau.local_str()
IndexError: list index out of range
```

Controle de acesso

- O estado interno da classe está exposto
  - Qualquer classe do sistema consegue manipular os dados da nossa classe
    - Os atributos deveriam ser privados!
      - Apenas objetos de uma determinada classe deveriam ter acesso a eles
      - Isso impede que um objeto mude o estado interno de outros objetos
    - As classes podem fornecer métodos de acesso para os atributos

Controle de acesso

- Em linguagens como Java, cada atributo ou método pode ter um nível de acesso
  - Público: qualquer classe do sistema pode acessar dados públicos de um objeto
  - Privado: apenas objetos da própria classe podem acessar os dados privados
  - Protegido: utilizado com herança
- Os modificadores de acesso podem ser utilizados tanto para atributos quanto métodos

Controle de acesso

```
class Pilha {
   private Object[] elem;
   private int tamanho;

   public Pilha(int capacidade) {
      elem = new Object[capacidade];
      tamanho = 0;
   }

   public void push(int valor) {
      elem[tamanho++] = valor;
   }

   public Object pop() {
      return elem[--tamanho];
   }
}
```

Controle de acesso

- O "padrão" em Java é privado
  - Princípio do menor privilégio
  - Tudo o que não deve necessariamente ser público, deve ser privado
  - Declarar o vetor `elem` privado permite à classe coordenar o acesso aos elementos através dos métodos `push()` e `pop()`
    - Impede que elementos sejam inseridos ou removidos da pilha em ordem incorreta

Controle de acesso

```
class Pilha {
public:
  void push(int valor) {
    dados.push_back(valor);
  }

  int pop() {
    int topo = peek();
    dados.pop_back();
    return topo;
  }

  int peek() {
    return dados.back();
  }

private:
  std::vector<int> dados;
};
```

Controle de acesso em Python

- Python não possui modificadores de acesso
  - Não existe mecanismo da linguagem para forçar um atributo a ser privado
  - Python segue uma filosofia inversa à de linguagens como Java e C++
    - Tudo deve ser público, por padrão
    - A menos que haja um bom motivo para esconder alguma coisa

Dados "privados" em Python

- Python segue um "código de honra"
  - Dados que não devem ser modificados por classes externas podem ser prefixado com uma única barra baixa

```
>>> class Gato:
...     _populacao = 0
...
...     def __init__(self, cor, humor):
...         self.humor = humor
...         self.cor = cor
...         self._localizacao = 0
...         Gato._populacao += 1
```

Dados "privados" em Python

- Python segue um "código de honra"
  - Entretanto, isso é apenas uma convenção
  - Cabe ao programador observar que aquele dado é privado e não deve ser manipulado

```
>>> garfield = Gato(laranja, "emburrado")

>>> garfield._localizacao = 10
```

Dados "privados" em Python

- Mas podemos "forçar" a consciência do usuário
  - Se um atributo for *realmente* importante para a classe, podemos utilizar *name mangling*
    - O verbo "*mangle*" pode ser traduzido como "desfigurar"
    - O *name mangling* altera o nome de um atributo ou método de modo que ele não pode ser diretamente acessado
  - Basta adicionar duas barras baixas no começo de um nome e ele será "desfigurado"

```
>>> class Gato:
...     __populacao = 0
...
...     def __init__(self, cor, humor):
...         self.humor = humor
...         self._cor = cor
...         self.__localizacao = 0
...         Gato.__populacao += 1
...
...     def get_cor(self):
...         return _cor
...
```

```
>>> tom = Gato(cinza, "bobo")
>>> tom.humor
bobo
>>> tom._cor
(200, 200, 200)
>>> tom.__localizacao
AttributeError: 'Gato' object has no attribute
'__localizacao'
```

Name mangling

- *Name mangling* faz parte do código de honra
  - Quando usamos *name mangling*, o nome de um atributo ou método é alterado
    - Ele recebe o prefixo `_Classe`, sendo que "`Classe`" é o nome da classe
    - No exemplo anterior, `__localizacao` virou `_Gato__localizacao`
  - O usuário está "fortemente encorajado" a *ignorar* esse nome
    - Mas é possível acessar o nome alterado

```
>>> class Gato:
...     __populacao = 0
...
...     def __init__(self, cor, humor):
...         self.humor = humor
...         self._cor = cor
...         self.__localizacao = 0
...         Gato.__populacao += 1
...
...     def get_cor(self):
...         return _cor
...
```

```
>>> tom.__localizacao
AttributeError: 'Gato' object has no attribute
'__localizacao'

>>> tom._Gato__localizacao
0

>>> tom._Gato__localizacao = 10
```

Métodos de acesso

- E os métodos `get` e `set`?
  - Em Java e C++, *getters* e *setters* são podem ser usados para controlar e coordenar acesso
    - Atributos somente-leitura não têm *setter*
    - Atributos que exigem verificação ou acesso especial podem ter lógica embutida no *getter*
    - Atributos que possuem valores específicos podem ter lógica embutida no *setter*
    - Também usamos esses métodos quando a implementação pode mudar

```
class Discente:
    def __init__(self, nome, matricula):
        """Instancia um novo aluno com nome e matricula"""
        self.__nome = nome
        self.__nome_efetivo = nome
        self.__matricula = matricula

    def get_nome(self):
        """Retorna o nome de um aluno. Este nome pode
        mudar durante o curso."""
        return self.__nome_efetivo

    def set_nome(self, nome):
        """Muda o nome de um aluno."""
        self.__nome_efetivo = nome

    def get_nome_original(self):
        """Obtem o nome de matricula do aluno. Este
        nome nao pode mudar"""
        return self.__nome

    def get_matricula(self):
        """Retorna a matricula do aluno"""
        return self.__matricula
```

```
class Discente:
    def __init__(self, nome, matricula):
        """Instancia um novo aluno com nome e matricula"""
        self.__nome = nome
        self.__nome_e
        self.__matri

    def get_nome(sel
        """Retorna o
        mudar durante
        return self._

    def set_nome(sel
        """Muda o nor
        self.__nome_efetivo = nome

    def get_nome_original(self):
        """Obtem o nome de matricula do aluno. Este
        nome nao pode mudar"""
        return self.__nome

    def get_matricula(self):
        """Retorna a matricula do aluno"""
        return self.__matricula
```

Em Python, uma *string* com três aspas é simplesmente uma *string* que pode ocupar várias linhas.

Mas quando utilizada no começo de uma classe ou função, ela se torna um *docstring*. O *docstring* é a documentação *externa* do código e serve para usuários saberem o que a função faz, para que servem os parâmetros e o que a função retorna.

Docstrings também podem ser utilizados para documentar classes.

```
class Discente:
    def __init__(self, nome, matricula):
        """Instancia um novo aluno com nome e matricula"""
        self.__nome = nome
        self.__nome_efetivo = nome
        self.__matricula = matricula

    def get_nome(self):
        """Retorna o nome de um aluno.
        mudar durante o curso."""
        return self.__nome_efetivo

    def set_nome(self, nome):
        """Muda o nome de um aluno."""
        self.__nome_efetivo = nome

    def get_nome_original(self):
        """Obtem o nome de matricula do aluno. Este
        nome nao pode mudar"""
        return self.__nome

    def get_matricula(self):
        """Retorna a matricula do aluno"""
        return self.__matricula
```

Seguimos a filosifia "não Pythônica" de deixar todos os atributos privados por padrão. Definimos funções de acesso para os que podem ser lidos e/ou alterados.

```
class Discente:
    def __init__(self, nome, matricula):
        """Instancia um novo aluno com nome e matricula"""
        self.__nome = nome
        self.__nome_efetivo = nome
        self.__matricula = matricula

    def
        nome nao pode mudar"""
        return self.__nome

    def get_matricula(self):
        """Retorna a matricula do aluno"""
        return self.__matricula
```

Como fazer isso da maneira "Pythônica"?

Métodos de acesso em Python

- Use métodos de acesso só quando necessário
  - Python possui propriedades
    - Em linguagens orientadas a objetos, uma propriedade é um atributo vinculado a métodos da classe
    - Tentativas de ler ou atribuir um valor a uma propriedade são substituídas pela chamada do método
    - É possível criar métodos para ler, escrever e remover um atributo

Propriedades em Python

- Um método pode ser definido como uma propriedade através de um decorador

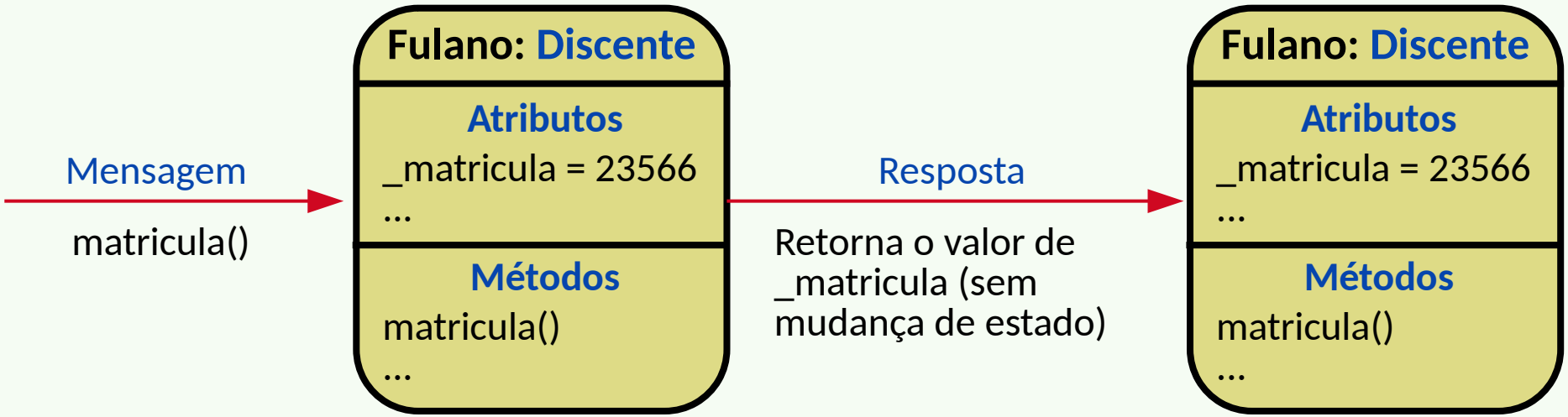
Define um método para ler a propriedade matricula

```
class Discente:
    def __init__(self, nome, matricula):
        """Instancia um novo aluno com
        nome e matricula"""
        self._nome = nome
        self.nome_efetivo = nome
        self._matricula = matricula

    @property
    def matricula(self):
        """Retorna o numero de matricula
        do discente. A matricula nao pode
        ser alterada"""
        return self._matricula
```

Propriedades em Python

- Quando tentamos acessar uma propriedade como se fosse um atributo, uma mensagem para o método de leitura é passada

```
>>> fulano.matricula
```



Propriedades em Python

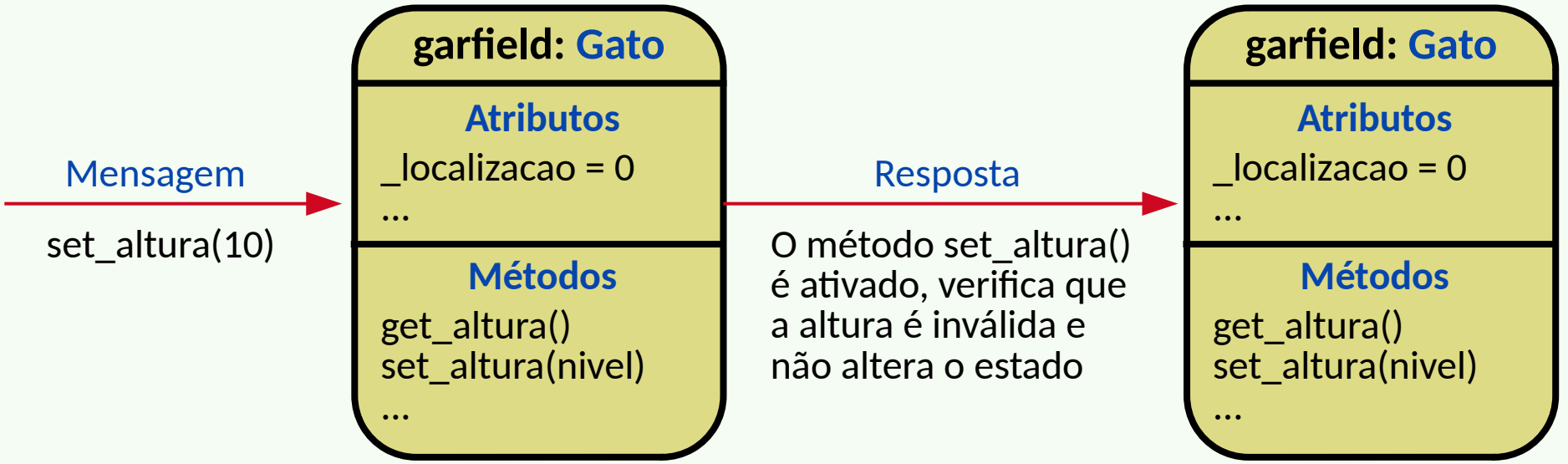
- O decorador `@nome.setter` pode ser usado para definir um método de escrita da propriedade

```
class Gato:
    # outros métodos

    @property
    def altura(self):
        """Retorna onde o gato esta"""
        return self._localizacao

    @localizacao.setter
    def altura(self, nivel):
        """Muda a altura do gato"""
        if 0 <= nivel <= 3:
            self._localizacao = altura
```

Propriedades em Python

- Quando tentamos mudar uma propriedade como se fosse um atributo, uma mensagem para o método de escrita é passada

```
>>> garfield.localizacao = 10
```

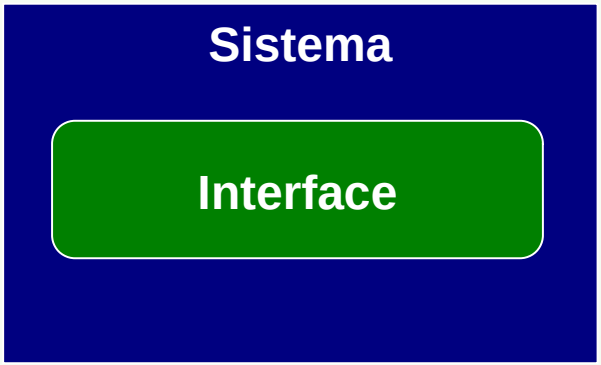


Encapsulamento em Python

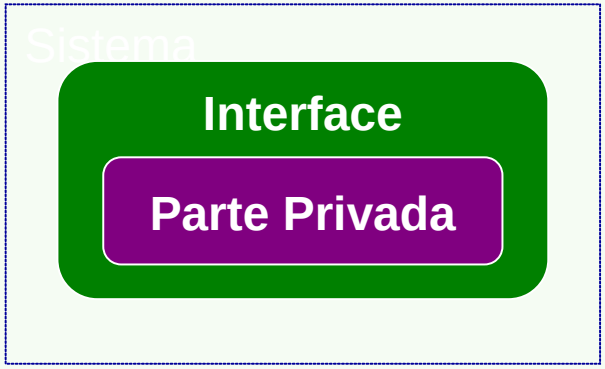
- Resumo
  - O encapsulamento deve ser respeitado pelo usuário
  - É possível desencorajar o acesso a um nome pela convenção _nome
  - É possível esconder um nome com *name mangling* (__nome)
  - Mas não há como *impedir* o acesso a um nome
  - Use atributos quando necessário para restringir a escrita ou impor uma lógica no acesso aos valores do atributo

Encapsulamento

- Agrupa dados e métodos
- Oculta detalhes de informação do resto do programa
- Apenas a interface permanece visível



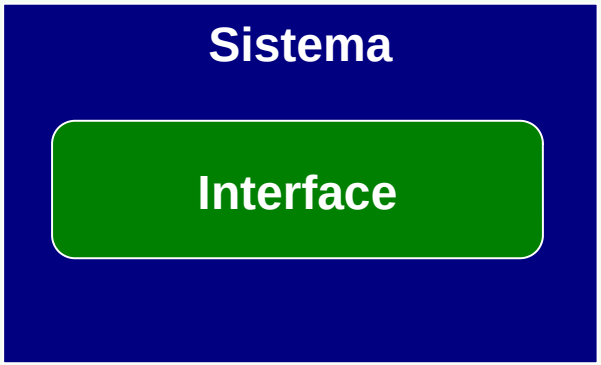
Visão Usuário



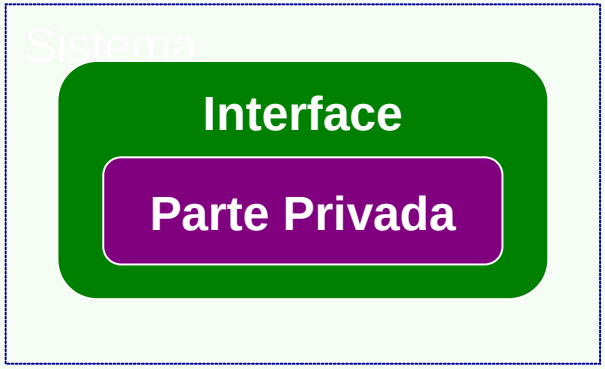
Visão Implementador

Encapsulamento

- Consequências:
  - Ocultação de informação
  - Decisões de projeto localizadas
  - Objetos independentes de aplicação



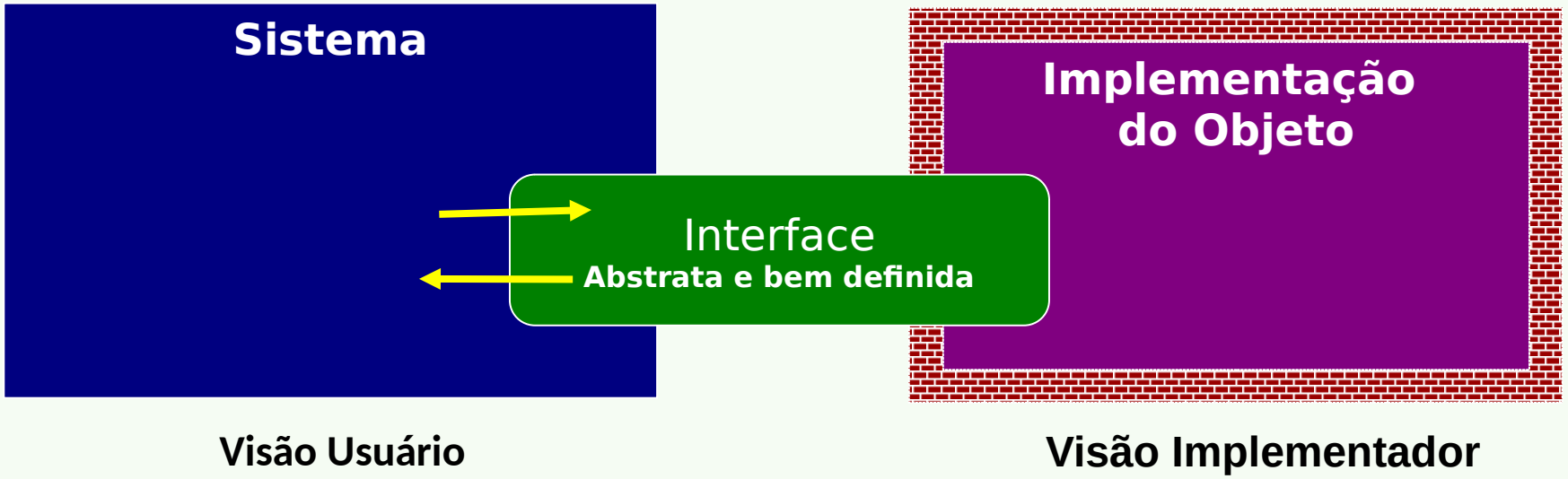
Visão Usuário



Visão Implementador

Encapsulamento

- Consequências:
  - Apenas o objeto pode modificar seu próprio estado interno
  - A resposta de uma mensagem é definida apelas pelo estado interno e pelos parâmetros



Agenda

- Prelúdios da orientação a objetos
- Classes e objetos
- Objetos em Python
- Encapsulamento
- Herança
- Polimorfismo de herança

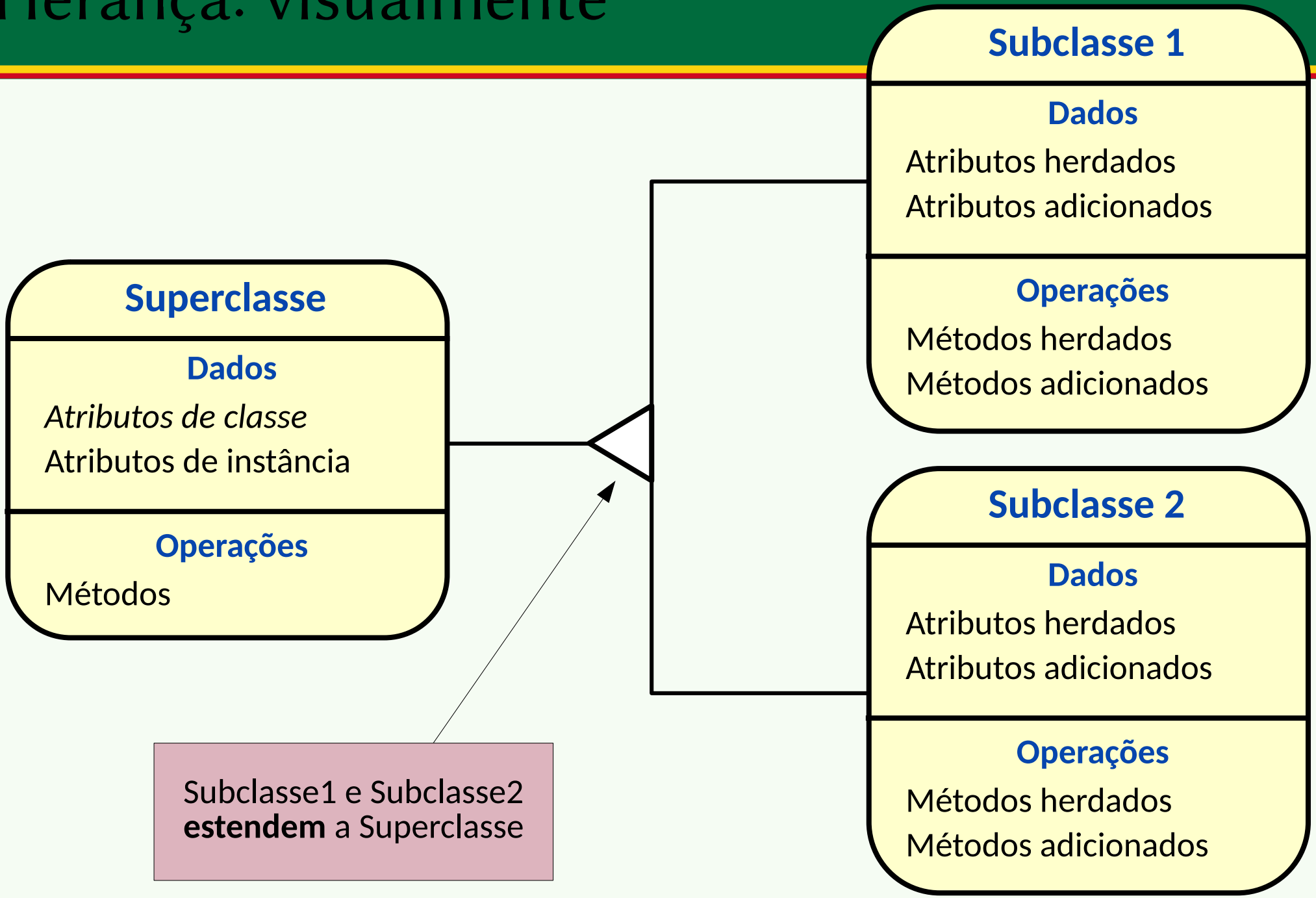
Reúso

- Um dos principais objetivos da Programação Orientada a Objetos é o reúso de código
- Mas o que fazemos quando uma classe não fornece todos os recursos desejados?
  - Podemos modificar a classe
    - Mas e como fica o encapsulamento?
    - E se introduzirmos erros nessa classe?
    - E se precisarmos alterar a interface da classe? Como ficam os outros objetos que fazem uso dela?

Herança

- Podemos reutilizar objetos através de herança
  - Na herança, uma nova classe estende as funcionalidades de uma classe já existente
    - A nova classe é filha ou subclasse
    - A classe já existente é base ou superclasse
    - A subclasse herda todos os atributos e métodos da superclasse
    - A subclasse **pode ser** a superclasse, mas especializada

Herança: visualmente



Herança: em Python

```
class Gato:
    def __init__(self):
        self.nome = 'Gato Sem Nome'

    def nomear(self, nome):
        self.nome = nome

    def miar(self):
        print("miau...")

class GatoGuerreiro(Gato):
    pass
```

```
>>> mingau = Gato()
>>> mingau is Gato
True
>>> gattacus = GatoGuerreiro()
>>> gattacus is Gato
False
>>> issubclass(GatoGuerreiro, Gato)
True
```

Herança: métodos e propriedade

- A subclasse herda todos os métodos e propriedades da superclasse
  - `GatoGuerreiro` possui um método `miar()`
  - `GatoGuerreiro` possui os atributos definidos em `Gato.__init__()`
- A subclasse também pode adicionar métodos e propriedades
  - Adicione o método `Gatoguerreiro.urrar()`
    - Existe `Gato.urrar()`?

Herança: sobrescrita

- A subclasse pode sobrescrever elementos presentes na superclasse
  - Por que os gatos guerreiros têm um miado tão fraquinho?
    - Sobrescreva o método `miar()`
  - O que acontece se dermos um construtor para a subclasse?
    - A subclasse pode acessar elementos da superclasse usando `super().`*nome*

Exemplo de herança

- Uma classe Lista, com métodos para
  - Inserção no início e no fim
  - Inserção no meio
  - Acesso aleatório
- Estenda
  - Uma classe Pilha, que só permite inserção, acesso e remoção em uma ponta
  - Uma classe Fila, que só permite inserção em uma ponta e inspeção e remoção na outra

Agenda

- Prelúdios da orientação a objetos
- Classes e objetos
- Objetos em Python
- Encapsulamento
- Herança
- Polimorfismo de herança

Linguagens sem *duck typing*

- Python possui *duck typing*
  - ***Duck typing* não deve ser confundido com tipagem dinâmica**
  - Tipagem dinâmica significa que o vínculo de uma variável com o tipo pode mudar em tempo de execução
  - *Duck typing* significa que uma operação é permitida entre duas variáveis se elas suportam essa operação, independentemente dos tipos envolvidos

Linguagens sem *duck typing*

- Em linguagens sem *duck typing*, herança é frequentemente usada para estabelecer uma relação do tipo "é um"
  - Objetos da superclasse **são objeto da superclasse**
  - Um objeto da subclasse pode ser utilizado em um contexto no qual se espera um objeto da superclasse
    - Princípio de substituição de Leskov

Linguagens com *duck typing*

- Em Python, o polimorfismo de herança não é tão frequentemente utilizado
  - Qualquer objeto que tem o comportamento esperado do parâmetro pode ser usado como argumento

```
def soma_elementos(colecao):
  soma = 0
  for item in colecao:
    soma += item
  return soma
```

Qualquer coleção cujos valores são iteráveis pode ser passada para esta funçao

Exemplos: dicionários, intervalos, conjuntos, listas, geradores etc.

Polimorfismo

- Polimorfismo significa que um objeto pode ter múltiplas formas
- No polimorfismo de herança, objetos da superclasse podem assumir as formas dos objetos das subclasses
- Polimorfismo exige vínculo dinâmico entre mensagens e métodos
  - O método não pode ser associado ao tipo da variável
  - Deve ser associado ao objeto

Vínculo estático

- Em C++, métodos são vinculados estaticamente
- Isso significa que, se um objeto recebe uma mensagem, quem vai determinar o método ativado será o tipo da variável
  - Pode ser ruim para o polimorfismo de herança
  - Exemplo no ColabWeb:
    - Como fazer o professor apresentar-se como professor através de polimorfismo de herança?

Vínculo dinâmico

- Para forçar o vínculo dinâmico, temos que declarar um método como virtual
  - Em C++, isso é chamado despacho dinâmico
    - O despacho da mensagem é feito dinamicamente, de acordo com o objeto
  - O vínculo dinâmico ou despacho dinâmico é menos eficiente do que o vínculo estático
    - Todo objeto que possui um método virtual precisa de uma tabela
    - Em tempo de execução os endereços dos métodos são consultados na tabela

Vínculo dinâmico

- Lembrando: em Python e Java, o vínculo entre o método e a mensagem é sempre dinâmico
  - Em Java, o despacho é sempre feito através de uma tabela que relaciona as mensagens aos métodos dos objetos
  - Em Python, o mesmo ocorre, mas devido à existência do *duck typing*
    - O polimorfismo de herança funciona assim como o polimorfismo entre objetos de classes quaisquer