

SISB093 - Programação 3

Aula 13

Prof. Thiago Cavalcante

Princípios da POO

- Herança
- Encapsulamento
- Abstração
- Polimorfismo
- Extra: *Modularidade, Composição*

Herança

- Uma classe obtém parte de suas funcionalidades de outra classe
- Quem herda: **classe derivada**, **subclasse** ou **subtipo**
- Quem é herdada: **classe base** ou **superclasse**
- A subclasse **herda**, **deriva** ou **estende** a classe base
- Cria uma relação de **é um(a)** e estabelece uma hierarquia
- Permite a **reutilização ou substituição** de funcionalidades já implementadas e **adição** de novas funcionalidades independentes
- Objetos da subclasse possuem métodos e atributos da superclasse
- Criação de uma classe base genérica e subclasses específicas
- Python dá suporte a herança simples e múltipla
- **!** *Para se aprofundar no assunto: Princípio da substituição de **Liskov** e Princípios **SOLID***

Herança em Python

```
class ClasseBase1:
    # Corpo da classe base 1

class ClasseBase2:
    # Corpo da classe base 2

class Subclasse(ClasseBase1):
    # Corpo da subclasse com herança simples

class SubclasseMultipla(ClasseBase1, ClasseBase2):
    # Corpo da subclasse com herança múltipla
```

Encapsulamento

- O comportamento (métodos) de um objeto é mantido escondido do lado externo do programa **e/ou** objetos mantêm seu estado (propriedades, atributos) privado
- Interfaces **pública** (externa) e **privada** (interna)
- Divisão da classe em três seções: **pública**, **protegida**, **privada**
- Um utilizador da classe não pode alterar ou visualizar o estado de um objeto interagindo **diretamente** com as variáveis de instância, ele deve utilizar métodos expostos na interface (*getters e setters*)

Vantagens do Encapsulamento

- O desenvolvedor tem a liberdade de modificar o código interno da classe sem se preocupar com outros programadores (contanto que a interface pública se mantenha constante e documentada) → **otimização, correção de bugs**
- Adiciona segurança, pois o acesso aos dados é restrito
- Oferece proteção de propriedade intelectual (quando o utilizador da classe não tem acesso ao seu código-fonte)
- Esconde do usuário a complexidade da implementação

Encapsulamento em Python

- Python não oferece suporte completo ao encapsulamento
- Não existem palavras-chave **public**, **protected** e **private**, como em outras linguagens
- Nenhum acesso é realmente restrito na implementação em Python
- O encapsulamento em Python é realizado através de convenções associadas aos usos do caractere underscore (`_`) na nomeação de variáveis e métodos

Abstração

- Esconder (ou abstrair) a complexidade da classe, de forma que o usuário não precise se preocupar com a implementação interna
- Apresentar ao usuário uma interface simples, de alto nível
- Analogia: carro
- Exemplo de código: classe de equação do segundo grau

Polimorfismo

- Chamadas de métodos são determinadas durante a execução do programa, de acordo com o tipo (classe) do objeto → **Busca de métodos dinâmica**
- Permite que objetos de diferentes classes possam ser tratados da mesma forma, se a interface for adequada
- **Duck typing:** *"quando eu vejo um pássaro que caminha como um pato, nada como um pato e grasna como um pato, eu chamo aquele pássaro de pato"*

Polimorfismo: possibilidades

- Um objeto possui várias implementações de um mesmo método, as quais se diferenciam pela quantidade de parâmetros → **Sobrecarga de métodos**
- Um mesmo operador se comporta de forma diferente quando aplicado a diferentes objetos → **Sobrecarga de operadores**
- Um objeto de uma subclasse possui uma implementação diferente de um método da classe base (o nome permanece igual) → **Sobrescrita de métodos** (*override*)
- Vários objetos de diferentes classes possuem um método de mesmo nome (mesma interface)

Polimorfismo em Python

- Python não dá suporte à sobrecarga de métodos com definições múltiplas, mas ela pode ser realizada com a utilização valores padrão, instruções `if/else` ou bibliotecas externas
- Em Python, os operadores e algumas funções integradas estão atrelados a **métodos especiais** (*dunder methods*) que podem ser definidos/sobrescritos para fornecer a funcionalidade desejada

Expressão	Método	Retorno
<code>x + y</code>	<code>__add__(self, y)</code>	objeto
<code>x - y</code>	<code>__sub__(self, y)</code>	objeto
<code>x * y</code>	<code>__mul__(self, y)</code>	objeto
<code>x / y</code>	<code>__truediv__(self, y)</code>	objeto
<code>x // y</code>	<code>__floordiv__(self, y)</code>	objeto
<code>x % y</code>	<code>__mod__(self, y)</code>	objeto
<code>x ** y</code>	<code>__pow__(self, y)</code>	objeto

Expressão	Método	Retorno
<code>x == y</code>	<code>__eq__(self, y)</code>	booleano
<code>x != y</code>	<code>__ne__(self, y)</code>	booleano
<code>x < y</code>	<code>__lt__(self, y)</code>	booleano
<code>x <= y</code>	<code>__le__(self, y)</code>	booleano
<code>x > y</code>	<code>__gt__(self, y)</code>	booleano
<code>x >= y</code>	<code>__ge__(self, y)</code>	booleano
<code>-x</code>	<code>__neg__(self, y)</code>	objeto

Expressão	Método	Retorno
<code>abs(x)</code>	<code>__abs__(self, y)</code>	objeto
<code>float(x)</code>	<code>__float__(self, y)</code>	float
<code>int(x)</code>	<code>__int__(self, y)</code>	inteiro
<code>str(x)</code> ou <code>print(x)</code>	<code>__repr__(self, y)</code>	string
<code>x = NomeClasse()</code>	<code>__init__(self, y)</code>	objeto

Modularidade

- Princípio de organização em que diferentes componentes de um programa são divididos em unidades funcionais separadas (módulos)
- Analogia: casa
- Em Python, os módulos são coleções de funções e classes intimamente relacionadas, os quais são definidas, geralmente, em um único arquivo de código (ex.: `math`)
- A utilização da modularidade facilita a correção de problemas no código (rastreamento)
- Permite também a reusabilidade do código em diferentes contextos onde o módulo se encaixa

Composição

- Forma de combinar objetos e/ou classes para a criação de tipos de dados mais complexos
- Um método de uma determinada classe chama métodos de um objeto de uma outra classe, integrando sua funcionalidade sem a utilização de herança
- Cria uma relação de **tem um(a)**, sem a definição de uma hierarquia
- Exemplo: empregados e endereços





Exercícios

1. Implemente uma superclasse `Pessoa`. Faça duas outras classes, `Estudante` e `Professor`, que herdam da superclasse `Pessoa`. Uma pessoa possui um nome e uma data de nascimento. Um estudante possui um curso e o professor possui um salário. Escreva as declarações de classe, os construtores e o método `__repr__` para todas as classes.

2. Criar uma classe para números racionais (frações) com as operações:

- Criar um número racional
- Acessar os valores de numerador e denominador individualmente
- Determinar se o número é negativo ou zero
- Fazer operações matemáticas em dois números racionais (soma, subtração, multiplicação, divisão e exponenciação)
- Comparar dois números racionais
- Criar uma representação em string de um número racional

Referências

- GOODRICH, M. T.; TAMASSIA, R.; GOLDWASSER, M. H. **Data Structures and Algorithms in Python** ()
- GIRIDHAR, C. **Learning Python Design Patterns** (/)
- HORSTMANN, C.; NECAISE, R. **Python for Everyone** ()
- **DAN BADER: The Meaning of Underscores in Python**
- **EDUCATIVE.IO: What is Object Oriented Programming?**
- **EDUCATIVE.IO: How to Use OOP in Python**
- **PYTHON DOCS: Classes > Private Variables**
- **PYTHON DOCS: Data Model > Special Method Names**
- **REAL PYTHON: Inheritance and Composition: A Python OOP Guide**
- **REAL PYTHON: Operator and Function Overloading in Custom Python Classes**