

Aula 13

Atributos Públicos e Privados

Repare nos códigos abaixo:

Java

```
class Pessoa {
    ... private String nome;
    ... private int idade;
    ... private String cpf;

    ... public Pessoa() {}
    ... public Pessoa(String nome, int idade, String cpf) {
    ...     this.nome = nome;
    ...     this.idade = idade;
    ...     this.cpf = cpf;
    ... }

    ... public String getNome() { return this.nome; }
    ... public void setNome(String p_nome) {
    ...     if (p_nome.length() > 0) {
    ...         this.nome = p_nome;
    ...     }
    ... }

    ... public int getIdade() { return this.idade; }
    ... public void setIdade(int p_idade) {
    ...     if (p_idade > 0) {
    ...         this.idade = p_idade;
    ...     }
    ... }

    ... public String getCpf() { return this.cpf; }
    ... public void setCpf(String p_cpf) {
    ...     if (p_cpf.length() == 11) {
    ...         this.cpf = p_cpf;
    ...     }
    ... }
}

public class Main {
    ... public static void main(String[] args) {
    ...     Pessoa alguem;

    ...     // definindo os valores dos atributos do objeto
    ...     alguem.setNome("João");
    ...     alguem.setIdade(20);
    ...     alguem.setCpf("12345678901");

    ...     // imprimindo os valores dos atributos do objeto
    ...     System.out.println("Nome: " + alguem.getNome());
    ...     System.out.println("Idade: " + alguem.getIdade());
    ...     System.out.println("CPF: " + alguem.getCpf());

    ...     System.out.println("Nome: " + alguem.nome);
    ... }
}
```

C++

```
class Pessoa {
private:
    ... string nome;
    ... int idade;
    ... string cpf;

public:
    ... Pessoa() {}
    ... Pessoa(string nome, int idade, string cpf) {
    ...     this->nome = nome;
    ...     this->idade = idade;
    ...     this->cpf = cpf;
    ... }

    ... string getNome() const { return this->nome; }
    ... void setNome(string p_nome) {
    ...     if (p_nome.length() > 0) {
    ...         this->nome = p_nome;
    ...     }
    ... }

    ... int getIdade() const { return this->idade; }
    ... void setIdade(int p_idade) {
    ...     if (p_idade > 0) {
    ...         this->idade = p_idade;
    ...     }
    ... }

    ... string getCpf() const { return this->cpf; }
    ... void setCpf(string p_cpf) {
    ...     if (p_cpf.length() == 11) {
    ...         this->cpf = p_cpf;
    ...     }
    ... }
};

int main() {
    ... Pessoa alguem;

    ... // definindo os valores dos atributos do objeto
    ... alguem.setNome("Joao");
    ... alguem.setIdade(20);
    ... alguem.setCpf("12345678901");

    ... // imprimindo os valores dos atributos do objeto
    ... cout << "Nome: " << alguem.getNome() << endl;
    ... cout << "Idade: " << alguem.getIdade() << endl;
    ... cout << "CPF: " << alguem.getCpf() << endl;

    ... cout << "Nome: " << alguem.nome << endl;

    ... return 0;
}
```

Reparou em algo em comum nos códigos acima? Tanto o código em Java quanto em C++ possuem termos para se referir aos atributos como privados e públicos.

Veja agora como fica o mesmo código em Python:

```
class Pessoa:
    def __init__(self) -> None:
        self.nome = ''
        self.idade = 0
        self.cpf = ''

    def get_nome(self):
        return self.nome

    def set_nome(self, nome):
        if len(nome) > 0:
            self.nome = nome

    def get_idade(self):
        return self.idade

    def set_idade(self, idade):
        if idade > 0:
            self.idade = idade

    def get_cpf(self):
        return self.cpf

    def set_cpf(self, cpf):
        if len(cpf) == 11:
            self.cpf = cpf

if __name__ == '__main__':
    alguem = Pessoa()

    # definindo os valores dos atributos do objeto
    alguem.set_nome('João')
    alguem.set_idade(20)
    alguem.set_cpf('12345678901')

    # imprimindo os valores dos atributos do objeto
    print(f'Nome: {alguem.get_nome()}')
    print(f'Idade: {alguem.get_idade()}')
    print(f'CPF: {alguem.get_cpf()}')
```

Na programação orientada a objetos, os termos "público" e "privado" são usados para controlar o acesso aos membros (atributos e métodos) de uma classe. Esses termos ajudam a garantir a encapsulação e a modularidade do código, permitindo que os objetos interajam uns com os outros de maneira controlada.

Vamos entender como funcionam os termos públicos e privados em detalhes:

- **Membros Públicos:**
 - Atributos e métodos marcados como públicos são acessíveis por qualquer parte do programa que tenha uma referência a um objeto dessa classe;
 - Os atributos públicos podem ser lidos e modificados diretamente de fora da classe;
 - Os métodos públicos definem a interface pública da classe, permitindo que outros objetos interajam com o objeto através desses métodos;
 - Geralmente, métodos públicos são usados para realizar operações e fornecer informações úteis sobre o objeto;
- **Membros Privados:**
 - Atributos e métodos marcados como privados são acessíveis somente dentro da própria classe onde são declarados;
 - Os atributos privados não podem ser acessados diretamente de fora da classe. Para acessar ou modificar esses atributos, são utilizados métodos públicos da classe, conhecidos como "métodos de acesso" (getters e setters);
 - Os métodos privados são utilizados para auxiliar nas operações internas da classe e não são diretamente acessíveis fora dela;
 - O encapsulamento de atributos e métodos privados fornece segurança e evita que outros objetos manipulem diretamente os dados internos da classe;

A importância de distinguir entre membros públicos e privados é manter um controle sobre o acesso e a manipulação dos dados dentro de uma classe. A ideia central é esconder a implementação interna dos objetos, permitindo que apenas as interfaces públicas sejam usadas externamente. Isso promove a modularidade do código, pois outras partes do programa não precisam conhecer os detalhes internos de uma classe para interagir com ela.

A convenção geralmente adotada em muitas linguagens de programação é usar um prefixo ou um sublinhado para indicar que um membro é privado. Por exemplo, um atributo privado pode ser declarado como `"_atributo"` ou um método privado como `"_metodoPrivado"`.

Em Java e C++, a diferença entre atributos públicos e privados é mais rigorosamente definida em comparação com Python. Vamos explorar as diferenças em cada linguagem:

- Java:
 - Atributos públicos em Java são declarados com o modificador `"public"` e podem ser acessados diretamente por qualquer objeto que possua uma referência a essa classe;
 - Atributos privados em Java são declarados com o modificador `"private"` e só podem ser acessados internamente pela própria classe;
 - Para acessar ou modificar atributos privados em Java, geralmente são usados métodos públicos chamados "métodos de acesso" (getters e setters) para garantir o encapsulamento. Esses métodos fornecem um controle sobre como os atributos privados são acessados e modificados externamente;
- C++:
 - Atributos públicos em C++ são declarados com a palavra-chave `"public"` e podem ser acessados por qualquer objeto que tenha acesso a um objeto da classe;
 - Atributos privados em C++ são declarados com a palavra-chave `"private"` e só podem ser acessados internamente pela própria classe;
 - Assim como em Java, em C++, é comum usar métodos públicos para acessar e modificar atributos privados, seguindo o princípio do encapsulamento;
- Python:
 - Em Python, a convenção é usar um sublinhado único (por exemplo, `"_atributo"`) para indicar que um atributo é privado, mas isso é apenas uma convenção, e o acesso aos atributos é determinado pela filosofia do "consentimento adulto";
 - Atributos públicos e privados em Python são acessíveis diretamente por qualquer objeto que tenha uma referência à classe, não há uma restrição rigorosa de acesso;
 - Embora seja comum seguir a convenção de acesso a atributos privados por meio de métodos "getters" e "setters" em Python, não há uma obrigatoriedade estrita de implementá-los. Os atributos podem ser acessados e modificados diretamente em Python, mesmo os que são considerados privados;

@property

Em Python, o decorador `@property` é uma ferramenta poderosa para definir comportamentos de atributos públicos e privados. Ele permite que você defina métodos especiais chamados de "métodos de acesso" para manipular o acesso e a modificação dos atributos da classe. Vamos explorar como usar o `@property` para criar atributos públicos e privados:

Atributos Públicos:

- Declare um atributo normalmente na classe;
- Use o decorador `@property` acima de um método com o mesmo nome do atributo;
- Dentro do método, defina a lógica para retornar o valor do atributo;

Exemplo:

```

class Pessoa:
    ... def __init__(self):
    ...     self._nome = ''

    ... @property
    ... def nome(self):
    ...     return self._nome

    ... @nome.setter
    ... def nome(self, valor):
    ...     if len(valor) > 0:
    ...         self._nome = valor

objeto = Pessoa()
objeto.nome = ''
print(objeto.nome)

objeto.nome = 'João'
print(objeto.nome)

print(objeto._nome)

```

Nesse exemplo, o atributo "_nome" é considerado público, mas o acesso a ele é feito através do método "nome". O decorador @property permite que o método "nome" seja acessado como um atributo, sem a necessidade de chamá-lo como um método. Caso o nome do atributo seja igual ao nome do método, irá gerar uma recursão infinita.

Atributos Privados:

- Declare um atributo com um nome diferente, geralmente com um sublinhado no início, para indicar que é privado (convenção);
- Use o decorador @property acima de um método com um nome público que representa o atributo;
- Dentro do método, defina a lógica para retornar o valor do atributo;

Exemplo:

```

class Pessoa:
    ... def __init__(self):
    ...     self._nome_privado = ''

    ... @property
    ... def nome(self):
    ...     return self._nome_privado

    ... @nome.setter
    ... def nome(self, valor):
    ...     if len(valor) > 0:
    ...         self._nome_privado = valor

objeto = Pessoa()
objeto.nome = ''
print(objeto.nome)

objeto.nome = 'João'
print(objeto.nome)

print(objeto._nome_privado)

```

Nesse exemplo, o atributo "_nome_privado" é considerado privado, mas o acesso a ele é feito através do método "nome". O decorador @property permite que o método "nome" seja acessado como um atributo público.

Com o uso do decorador @property, você pode controlar o acesso e a modificação de atributos em Python, definindo lógicas personalizadas em métodos especiais. Isso permite que você mantenha a flexibilidade dos atributos enquanto mantém a capacidade de adicionar lógica extra quando necessário, como validações ou cálculos computacionais.

Embora os exemplos apresentados mostrem que ambos os atributos podem ser acessados de forma pública, a utilização de atributos privados ainda possui vantagens importantes na programação orientada a objetos. Aqui estão algumas razões para usar atributos privados:

1. Encapsulamento e controle de acesso: A utilização de atributos privados permite controlar o acesso aos dados internos de uma classe. Ao fornecer acesso apenas por meio de métodos públicos (como os getters e setters), você pode garantir que a manipulação dos atributos seja realizada de acordo com regras específicas de validação, segurança ou consistência. Isso ajuda a evitar que partes externas do programa manipulem os dados de forma incorreta ou insegura;
2. Manutenção e evolução: A utilização de atributos privados permite que você tenha maior flexibilidade para modificar a implementação interna de uma classe sem afetar diretamente as partes externas que a utilizam. Se um atributo privado precisar ser alterado ou removido, você pode fazer isso sem afetar o código cliente, desde que mantenha a interface pública consistente. Isso facilita a manutenção, a evolução e o refatoramento do código;
3. Abstração e ocultação de detalhes: A utilização de atributos privados ajuda a promover uma maior abstração e ocultação dos detalhes internos de uma classe. Ao esconder a implementação dos atributos privados, você pode fornecer uma interface pública mais simples e focada nos comportamentos e funcionalidades essenciais da classe. Isso permite que outras partes do programa usem a classe de forma mais intuitiva e reduz a dependência em relação aos detalhes internos;
4. Segurança e integridade dos dados: A utilização de atributos privados permite que você defina lógicas de validação e restrição de acesso aos dados da classe. Isso ajuda a garantir a integridade dos dados e a prevenir que valores inválidos ou inconsistentes sejam atribuídos aos atributos. Além disso, você pode adicionar mecanismos adicionais de segurança, como controle de permissões, criptografia ou auditoria, ao acessar ou modificar os atributos privados;

Embora seja possível criar atributos públicos em Python, a utilização de atributos privados proporciona maior controle, segurança, modularidade e encapsulamento ao projeto. A prática recomendada é utilizar atributos privados sempre que você precisar controlar o acesso, adicionar lógica extra ou garantir a integridade dos dados internos de uma classe.

Atributos Privados e Herança

Em Python, os atributos privados em uma classe não são herdados pelas subclasses. No entanto, é importante entender como os atributos privados podem afetar a herança. Vou explicar duas situações que envolvem atributos privados e herança em Python:

1. Atributos privados não são acessíveis diretamente pelas subclasses: quando uma classe define um atributo privado, por exemplo, _atributo_privado, esse atributo não é acessível diretamente pelas subclasses. As subclasses não podem acessar ou modificar esse atributo diretamente, pois ele é tratado como privado. No entanto, as subclasses podem acessar e modificar o atributo privado por meio dos métodos públicos da classe pai, se esses métodos estiverem disponíveis.
2. Atributos privados com o mesmo nome em classes pai e subclasses: se uma classe pai e uma subclasse têm atributos privados com o mesmo nome, os atributos são considerados como entidades distintas em cada classe. Ou seja, não há sobrescrita de atributos privados em herança. A subclasse pode ter seu próprio atributo privado com o mesmo nome, mas eles não compartilham nenhum valor ou estado.

Vamos ver um exemplo para ilustrar essas situações:

```
class Pai:
    def __init__(self):
        self._atributo_privado = 10

    def get_atributo_privado(self):
        return self._atributo_privado

class Filho(Pai):
    def __init__(self):
        super().__init__()
        self._atributo_privado = 20

pai = Pai()
filho = Filho()

print(pai.get_atributo_privado()) # Saída: 10
print(filho.get_atributo_privado()) # Saída: 20
```

Neste exemplo, a classe Pai define um atributo privado `_atributo_privado` e um método público `get_atributo_privado()` para acessá-lo. A classe Filho herda da classe Pai e também define seu próprio atributo privado `_atributo_privado`. Quando chamamos o método `get_atributo_privado()` em cada instância, obtemos os valores correspondentes aos atributos privados da respectiva classe. Se tirarmos a chamada da função `super()` e tentarmos chamar o método `get_atributo_privado()`, vai gerar um erro:

```
class Pai:
    def __init__(self):
        self._atributo_privado = 10

    def get_atributo_privado(self):
        return self._atributo_privado

class Filho(Pai):
    def __init__(self):
        pass

pai = Pai()
filho = Filho()

print(pai.get_atributo_privado()) # Saída: 10
print(filho.get_atributo_privado()) # AttributeError
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
10
Traceback (most recent call last):
  File "C:\Users\gutoh\OneDrive\0.Python\main.py", line 21, in <module>
    print(filho.get_atributo_privado()) # Saída: 20
    ~~~~~^
  File "C:\Users\gutoh\OneDrive\0.Python\main.py", line 9, in get_atributo_privado
    return self._atributo_privado
    ~~~~~^
AttributeError: 'Filho' object has no attribute '_atributo_privado'. Did you mean:
'get_atributo_privado'?
PS C:\Users\gutoh\OneDrive\0.Python>
```

Exercícios para Praticar

1. Crie uma classe chamada Pessoa com um atributo privado `_nome`. Implemente métodos para definir e obter o nome da pessoa.
2. Crie uma classe ContaBancaria com atributos privados `_saldo` e `_limite`. Implemente métodos para depositar, sacar e obter o saldo.
3. Crie uma classe Círculo com atributo privado `_raio`. Implemente métodos para calcular a área e o perímetro do círculo.
4. Crie uma classe Retângulo com atributos privados `_comprimento` e `_largura`. Implemente métodos para calcular a área e o perímetro do retângulo.
5. Crie uma classe Veículo com atributo privado `_velocidade`. Implemente métodos para acelerar, frear e obter a velocidade atual do veículo.
6. Crie uma classe Funcionário com atributos privados `_salário` e `_horas_trabalhadas`. Implemente métodos para calcular o salário mensal com base no salário por hora e nas horas trabalhadas.
7. Crie uma classe Animal com atributo privado `_nome` e um método para emitir um som característico do animal.
8. Crie uma classe Triângulo com atributos privados `_lado1`, `_lado2` e `_lado3`. Implemente um método para verificar se os lados formam um triângulo válido.
9. Crie uma classe Estudante com atributos privados `_nome` e `_matrícula`. Implemente um método para exibir as informações do estudante.
10. Crie uma classe Produto com atributos privados `_nome` e `_preço`. Implemente métodos para alterar o preço em uma porcentagem e obter o preço atualizado.
11. Crie uma classe Biblioteca com atributo privado `_livros`. Implemente métodos para adicionar e remover livros da biblioteca.
12. Crie uma classe Quadrado com atributo privado `_lado`. Implemente métodos para calcular a área e o perímetro do quadrado.
13. Crie uma classe Carro com atributo privado `_marca` e um método para exibir a marca do carro.
14. Crie uma classe Cachorro com atributo privado `_idade` e um método para converter a idade do cachorro em idade humana.
15. Crie uma classe Data com atributos privados `_dia`, `_mês` e `_ano`. Implemente métodos para verificar se uma data é válida e exibir a data no formato dd/mm/aaaa.
16. Crie uma classe Telefone com atributo privado `_numero`. Implemente métodos para verificar se um número de telefone é válido e exibir o número formatado.
17. Crie uma classe Lâmpada com atributo privado `_ligada`. Implemente métodos para ligar, desligar e verificar o estado da lâmpada.
18. Crie uma classe ContaCorrente com atributos privados `_saldo` e `_cheque_especial`. Implemente métodos para realizar saques e depósitos, levando em consideração o cheque especial.
19. Crie uma classe Funcionário com atributos privados `_nome` e `_salário`. Implemente métodos para calcular o salário anual com base no salário mensal.
20. Crie uma classe Casa com atributo privado `_endereço` e um método para exibir o endereço da casa.
21. Crie uma classe Animal com um atributo privado `_nome`. Crie uma subclasse chamada Cachorro que defina um método para exibir o nome do cachorro.
22. Crie uma classe FiguraGeometrica com um atributo privado `_cor`. Crie uma subclasse chamada Quadrado que defina um método para exibir a cor do quadrado.
23. Crie uma classe Pessoa com um atributo privado `_nome`. Crie uma subclasse chamada Estudante que defina um método para exibir o nome e a matrícula do estudante.
24. Crie uma classe Veiculo com um atributo privado `_velocidade`. Crie uma subclasse chamada Carro que defina um método para acelerar o carro em uma determinada velocidade.
25. Crie uma classe Conta com um atributo privado `_saldo`. Crie uma subclasse chamada ContaCorrente que defina métodos para depositar, sacar e verificar o saldo da conta corrente.
26. Crie uma classe Forma com um atributo privado `_cor`. Crie uma subclasse chamada Circulo que defina um método para exibir a cor do círculo.
27. Crie uma classe Funcionario com um atributo privado `_salario`. Crie uma subclasse chamada Gerente que defina um método para aumentar o salário do gerente.
28. Crie uma classe Animal com um atributo privado `_idade`. Crie uma subclasse chamada Gato que defina um método para exibir a idade do gato.

29. Crie uma classe ContaBancaria com um atributo privado `_titular`. Crie uma subclasse chamada ContaPoupanca que defina um método para exibir o nome do titular da conta poupança.
30. Crie uma classe Pessoa com um atributo privado `_idade`. Crie uma subclasse chamada Estudante que defina um método para verificar se o estudante é maior de idade.
31. Crie uma classe Animal com um atributo privado `_som`. Crie uma subclasse chamada Cachorro que defina um método para emitir o som de latido do cachorro.
32. Crie uma classe Produto com um atributo privado `_preco`. Crie uma subclasse chamada ProdutoImportado que defina um método para adicionar uma taxa de importação ao preço do produto.
33. Crie uma classe Veiculo com um atributo privado `_combustivel`. Crie uma subclasse chamada CarroEletrico que defina um método para verificar o nível de carga da bateria do carro elétrico.
34. Crie uma classe Forma com um atributo privado `_area`. Crie uma subclasse chamada Triangulo que defina um método para calcular a área do triângulo.
35. Crie uma classe Pessoa com um atributo privado `_cpf`. Crie uma subclasse chamada Cliente que defina um método para exibir o CPF do cliente.
36. Crie uma classe Conta com um atributo privado `_numero`. Crie uma subclasse chamada ContaSalario que defina um método para exibir o número da conta salário.
37. Crie uma classe Animal com um atributo privado `_especie`. Crie uma subclasse chamada Cachorro que defina um método para exibir a espécie do cachorro.
38. Crie uma classe Produto com um atributo privado `_descricao`. Crie uma subclasse chamada ProdutoPerecivel que defina um método para verificar se o produto perecível está vencido.
39. Crie uma classe Forma com um atributo privado `_perimetro`. Crie uma subclasse chamada Quadrado que defina um método para calcular o perímetro do quadrado.
40. Crie uma classe Pessoa com um atributo privado `_email`. Crie uma subclasse chamada Cliente que defina um método para exibir o e-mail do cliente.