PYTHON ACADEMY

O QUE É E COMO UTILIZAR @PROPERTY NO PYTHON

Guia completo de @property: getters/setters Pythônicos, validação de dados, atributos computados, casos reais (temperatura, idade, preço), @property vs atributos públicos.

PYTHONACADEMY.COM.BR

Crie ebooks profissionais incríveis em minutos com IA

Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!

- Capas gerados por IA
- Infográficos feitos por IA
- Edite em Markdown em Tempo Real
- Adicione Banners Promocionais

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS

> ✅ ***Atualizado para Python 3.13*** *(Dezembro 2025) Conteúdo enriquecido com atributos computados, validação e casos práticos.*

Salve salve Pythonista!

@property transforma métodos em atributos, permitindo **validação**, **cálculos** e **encapsulamento** sem mudar a interface pública da classe!

Neste guia, você vai aprender:

- ✅ **Getters/Setters Pythônicos** - Sem `.get_x()` / `.set_x()`
- ✅ **Validação automática** - Dados sempre válidos
- ✅ **Atributos computados** - Cálculos sob demanda
- ✅ **Casos práticos** - Temperatura, idade, preços

O Python é uma linguagem de programação flexível e poderosa que oferece uma série de recursos avançados para os desenvolvedores.

Um desses recursos é o decorador `@property`, que permite transformar métodos de uma classe em propriedades, tornando o código mais elegante e fácil de entender.

Nesse artigo você vai aprender o que é e como utilizar o `@property` no Python.

Sem mais delongas, vamos nessa!

```
<div class="suggestions-header">
    <h2>Leia também</h2>
</div>
<div class="suggestions-body">
    <ul>
        <li>
            <a href="https://pythonacademy.com.br/blog/domine-
decorators-em-python" target="_blank">
                Domine Decoradores em Python
            </a>
        </li>
    </ul>
</div>
```

Ao criar classes em Python, é comum definir métodos que retornam ou modificam valores de atributos.

> *Se ainda não sabe sobre Classes em Python, leia nosso outro artigo sobre Classes no Python*

No entanto, esses métodos muitas vezes têm nomes diferentes dos atributos e são acessados como funções.

Para melhorar a legibilidade do código e fornecer uma sintaxe mais intuitiva, podemos usar o decorador `@property`.

Neste artigo, vamos explorar o que é o `@property` no Python e como utilizá-lo em nossas classes para criar propriedades que podem ser acessadas de uma forma mais intuitiva.

Propriedades em Python

Antes de mergulharmos no `@property`, é importante entender o conceito de propriedades em Python.

Uma propriedade é um atributo de classe que é calculado dinamicamente, em vez de ser armazenado em memória.

Quando um atributo é uma propriedade, ele é acessado e modificado como qualquer outro atributo, mas na verdade, por trás dos panos, um método customizado é chamado.

Ao utilizar propriedades, podemos ter controle sobre o acesso e modificação de atributos de uma classe, permitindo realizar validações, conversões de dados ou cálculos adicionais antes de retornar ou definir o valor.

O decorador `@property`

Em Python, o decorador `@property` é usado para transformar um método em uma propriedade de uma classe.

Ele permite que um método seja acessado como atributo, sem a necessidade de chamá-lo como uma função.

Vamos começar com um exemplo simples para ilustrar como o `@property` funciona.

Suponha que temos a classe `Retangulo` que representa um retângulo e possui os atributos `largura` e `altura`.

Para calcular a área do retângulo, poderíamos ter um método chamado `calcular_area`, como mostrado abaixo:

```
class Retangulo:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def calcular_area(self):
        return self.largura * self.altura
```

Neste caso, para calcular a área do retângulo, precisamos chamar o método `calcular_area()` explicitamente:

```
retangulo = Retangulo(5, 3)
area = retangulo.calcular_area()
print(area)
```

E a saída seria:

```
15
```

Agora, vamos utilizar o `@property` para transformar o método `calcular_area()` em uma propriedade da classe `Retangulo`:

```
class Retangulo:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    @property
    def area(self):
        return self.largura * self.altura
```

Note o uso do decorador `@property` antes do método `area()`. Agora, podemos acessar a área do retângulo como se fosse um atributo:

```
retangulo = Retangulo(5, 3)
print(retangulo.area)  # Saída: 15
```

Agora, a chamada `retangulo.area` retorna o valor da área sem a necessidade de chamarmos explicitamente o método.

> 💡 *Estou construindo o **Ebookr.ai**, uma plataforma onde você cria ebooks profissionais com IA sobre qualquer assunto — do zero ao PDF pronto, com capas e infográficos gerados automaticamente. Dá uma olhada!*

Getters e Setters

Em muitos casos, queremos não apenas obter o valor calculado de uma propriedade, mas também definir seu valor.

Para isso, utilizamos os métodos `getter` e `setter`.

O método `getter` é responsável por retornar o valor da propriedade quando ela é acessada.

Utilizamos o decorador `@property` para definir o `getter`, como já vimos anteriormente.

O método `setter`, por sua vez, é usado para definir o valor da propriedade quando ela é modificada.

Para definir o `setter`, utilizamos o mesmo nome da propriedade seguido pelo decorador `@nomedapropriedade.setter`.

Vamos expandir nosso exemplo anterior para incluir um `setter` para modificar a largura do retângulo:

```python
class Retangulo:
    def __init__(self, largura, altura):
        self._largura = largura
        self.altura = altura

    @property
    def largura(self):
        return self._largura

    @largura.setter
    def largura(self, nova_largura):
        if nova_largura > 0:
            self._largura = nova_largura
        else:
            raise ValueError("A largura deve ser maior que 0.")

    @property
    def area(self):
        return self.largura * self.altura
```

Neste exemplo, criamos um método `largura` para funcionar como o `getter` da propriedade `largura` e um método `largura.setter` para funcionar como o `setter`.

O método `setter` verifica se o novo valor da largura é maior que zero e, em caso positivo, atualiza o atributo `_largura`.

Caso contrário, lança uma exceção `ValueError` informando que a largura deve ser maior que zero.

Agora, podemos usar a propriedade `largura` para obter e modificar o valor da largura do retângulo:

```
retangulo = Retangulo(5, 3)
print(retangulo.largura)  # Saída: 5

retangulo.largura = 7
print(retangulo.largura)  # Saída: 7

retangulo.largura = -1  # Lança uma exceção ValueError
```

Podemos ver que, ao acessar a propriedade `largura`, o método `largura()` é chamado e retorna o valor atual da largura.

Da mesma forma, ao atribuir um novo valor à propriedade `largura`, o método `largura.setter` é executado e valida o novo valor antes de atualizar `_largura`.

Deletar propriedades

Além de obter e definir o valor de uma propriedade, também podemos excluí-la utilizando o método `deleter`.

Para definir o `deleter` de uma propriedade, utilizamos o decorador `@nomedapropriedade.deleter`.

Vamos adicionar um método `deleter` à classe `Retangulo` para excluir a propriedade `largura`:

```python
class Retangulo:
    def __init__(self, largura, altura):
        self._largura = largura
        self.altura = altura

    @property
    def largura(self):
        return self._largura

    @largura.setter
    def largura(self, nova_largura):
        if nova_largura > 0:
            self._largura = nova_largura
        else:
            raise ValueError("A largura deve ser maior que 0.")

    @largura.deleter
    def largura(self):
        del self._largura

    @property
    def area(self):
        return self.largura * self.altura
```

Agora podemos excluir a propriedade `largura` utilizando o comando `del`:

```python
retangulo = Retangulo(5, 3)
print(retangulo.largura)  # Saída: 5

del retangulo.largura

print(retangulo.largura)
```

Ao tentar acessar a propriedade na linha `print(retangulo.largura)` o seguinte erro será lançado:

```
AttributeError: 'Retangulo' object has no attribute '_largura'
```

Casos de uso do `@property`

O decorador `@property` é muito útil em situações em que precisamos controlar o acesso aos atributos de uma classe.

Aqui estão alguns exemplos de casos de uso comuns:

- Conversão de tipos: podemos usar `@property` para converter automaticamente tipos de dados. Por exemplo, podemos ter um atributo `data` que é armazenado como uma string e uma propriedade `data` que devolve o valor convertido em um objeto `datetime`.
- Verificação de validade: podemos adicionar validações em um `setter` para garantir que os atributos estão dentro dos limites aceitáveis. Por exemplo, podemos ter um atributo `idade` que precisa ser um número positivo e, caso contrário, levanta um erro.
- Uso de cache: podemos utilizar uma propriedade para fazer cache de um valor calculado, evitando recalcular a cada vez que a propriedade é acessada.
- Acesso a dados externos: podemos usar propriedades para acessar e atualizar dados em bancos de dados externos ou sistemas remotos. Dessa forma, podemos manter a interface do objeto consistente, independentemente de onde os dados são armazenados.

Casos Práticos Adicionais

1. Conversor de Temperatura

```python
class Temperatura:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, valor):
        if valor < -273.15:
            raise ValueError("Temperatura abaixo do zero absoluto!")
        self._celsius = valor

    @property
    def fahrenheit(self):  # Somente leitura
        return self._celsius * 9/5 + 32

    @property
    def kelvin(self):  # Somente leitura
        return self._celsius + 273.15

temp = Temperatura(25)
print(f"{temp.celsius}°C = {temp.fahrenheit}°F = {temp.kelvin}K")
# 25°C = 77.0°F = 298.15K
```

2. Produto com Preço Calculado

```python
class Produto:
    def __init__(self, nome, preco_base, imposto=0.1):
        self.nome = nome
        self._preco_base = preco_base
        self.imposto = imposto

    @property
    def preco_base(self):
        return self._preco_base

    @preco_base.setter
    def preco_base(self, valor):
        if valor <= 0:
            raise ValueError("Preço deve ser positivo")
        self._preco_base = valor

    @property
    def preco_final(self):  # Calculado automaticamente
        return self._preco_base * (1 + self.imposto)

produto = Produto("Mouse", 100)
print(produto.preco_final)  # 110.0
produto.preco_base = 150
print(produto.preco_final)  # 165.0
```

3. Pessoa com Idade Calculada

```python
from datetime import datetime, date

class Pessoa:
    def __init__(self, nome, data_nascimento):
        self.nome = nome
        self._data_nascimento = data_nascimento

    @property
    def data_nascimento(self):
        return self._data_nascimento

    @data_nascimento.setter
    def data_nascimento(self, valor):
        if valor > date.today():
            raise ValueError("Data de nascimento no futuro!")
        self._data_nascimento = valor

    @property
    def idade(self):  # Sempre atualizado!
        hoje = date.today()
        return hoje.year - self._data_nascimento.year - (
            (hoje.month, hoje.day) < (self._data_nascimento.month,
        self._data_nascimento.day)
        )

pessoa = Pessoa("Alice", date(1990, 5, 15))
print(pessoa.idade)  # Calcula idade atual automaticamente
```

@property vs Atributos Públicos

Quando Usar @property?

✅ **Use @property quando:**

- Precisa **validar** valores
- Atributo é **calculado** (não armazenado)
- Precisa **controlar** acesso (read-only)
- Pode precisar **mudar lógica** no futuro
- **Lazy loading** (calcular só quando necessário)

Quando NÃO usar @property?

❌ **Use atributo simples quando:**

- Atributo é apenas **armazenamento** (sem lógica)
- **Performance crítica** (property tem overhead mínimo)
- Não precisa validação
- Classe é simples (dataclass)

```python
# ❌ Property desnecessária
class Usuario:
    def __init__(self, nome):
        self._nome = nome

    @property
    def nome(self):
        return self._nome  # Só retorna, sem lógica

# ✅ Atributo público direto
class Usuario:
    def __init__(self, nome):
        self.nome = nome  # Mais simples!
```

Conclusão

Neste guia de **@property**, você aprendeu:

✅ **Getters/Setters** - Interface Pythônica sem `.get_x()` ✅ **Validação** - Dados sempre válidos automaticamente ✅ **Atributos computados** - Cálculos sob demanda (idade, temperatura) ✅ **Read-only** - Propriedades somente leitura ✅ **@property vs públicos** - Quando usar cada um

Principais lições:

- **@property** torna métodos acessíveis como atributos
- Use para **validação** e **cálculos**
- **Setter** controla modificações
- **Deleter** permite `del objeto.atributo`
- Property tem overhead **mínimo** (99% dos casos, use!)

Próximos passos:

- Pratique Classes e Objetos
- Explore Dataclasses (reduz boilerplate)
- Aprenda `__getattr__` e `__setattr__`
- Estude descriptors (property avançado)

O decorador `@property` é uma ferramenta poderosa que nos permite transformar métodos em propriedades de uma classe, oferecendo um acesso mais intuitivo e controlado a essas propriedades.

Além de fornecer uma sintaxe mais elegante, as propriedades também nos permitem adicionar validações, conversões de dados e realizar cálculos adicionais antes de retornar ou definir o valor de um atributo.

Com esse conhecimento em mãos, você está pronto para utilizar o `@property` em suas classes Python e tornar seus códigos mais legíveis e eficientes.

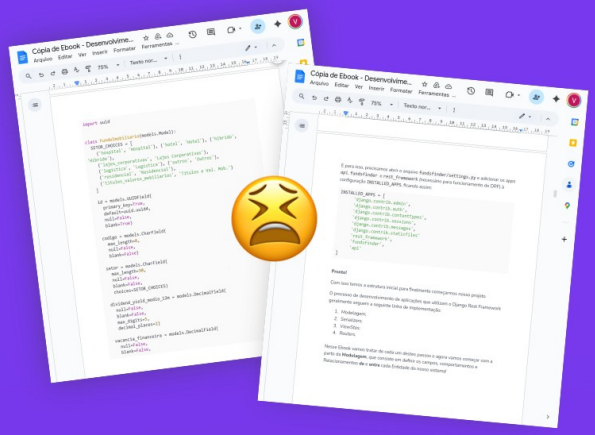
Experimente utilizar propriedades em seus projetos e descubra como elas podem simplificar o acesso e a manipulação de atributos em suas classes.

É isso por hoje! Nos vemos no próximo artigo 😉

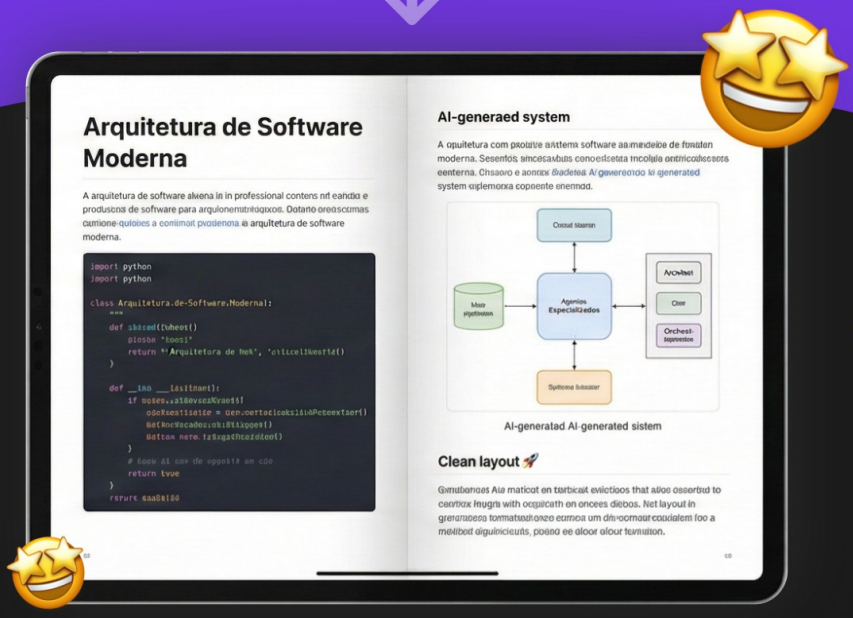
Não se esqueça de conferir!

Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS