5

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

5

Criação de classes em Python

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo vamos ver como implementar classes em Python e aplicar os conceitos de abstração e encapsulamento. Vamos aprender a convenção de nomenclatura de classes adotada pela comunidade Python, revisitar os diagramas de classe da UML, e criar a nossa primeira classe em Python. Vamos estudar o conceito de self, o parâmetro que permite aos objetos terem consciência sobre si mesmos, aprender como inicializar um objeto no momento da sua criação e veremos duas formas de aplicar o encapsulamento para ocultar informações do objeto.

5.1. Introdução

Python é uma linguagem multiparadigma, que permite a programação simultânea em diferentes paradigmas de programação, em especial procedural, funcional e orientado a objetos. Este é um dos motivos que contribuiu para a popularidade atual do Python nos mais diversos contextos.

O Python trata todas as entidades da linguagem como objetos, o que faz dele uma linguagem naturalmente apta ao paradigma de POO. No entanto, devido às características da linguagem, a aplicação de alguns conceitos difere do que vemos em linguagens exclusivamente orientadas a objetos, como Java, C# e PHP, por exemplo.

5.2. PEP-8 aplicada às classes

Antes de ver propriamente como implementar as classes em Python, vamos ver as regras para nomeá-las, pois isso irá nos ajudar a identificar e diferenciar classes de instâncias ao lermos um código que segue as recomendações da PEP8.

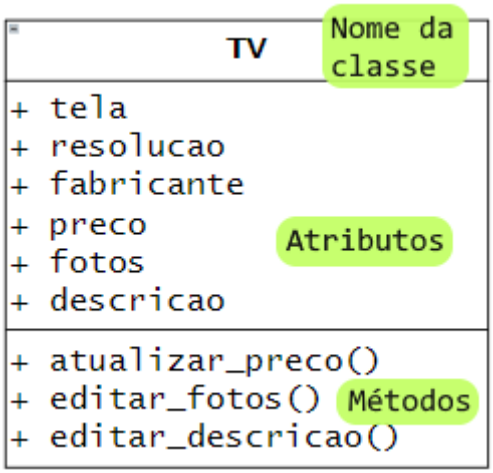
A convenção para nomes de classes é `MinhaClasse`, na qual todas as palavras possuem a primeira letra maiúscula e são unidas diretamente, sem espaço ou sublinhado. Para os atributos e métodos das classes, seguimos a mesma convenção das variáveis e funções, todas as letras minúsculas, unidas por sublinhado (PSF, 2021a).

Uma exceção a essa regra são as classes integradas do Python, como `int`, `float`, etc. que possuem nomes curtos e todos minúsculos. O raciocínio por trás disso é que tais classes são usadas pelo programador principalmente como funções para converter dados entre tipos diferentes, e seguem a convenção para nomear funções.

5.3. POO em Python

Vamos começar relembrando a definição que vimos para a classe TV (Figura 5.1), incluindo alguns atributos novos e também alguns métodos. Lembre-se que estamos modelando uma TV que irá representar um produto a ser vendido, então precisamos pensar nos métodos e atributos que sejam relevantes ao contexto. Por exemplo, um método `aumentar_volume` faz sentido na programação do sistema operacional da TV, mas não é necessário aqui. Isso faz parte do processo de abstração do objeto real para sua representação no código.

Figura 5.1: Diagrama de classes UML da classe TV.



Fonte: do autor, 2021.

A Figura 5.1 mostra o diagrama de classe UML[1] para a classe TV. É comum que este diagrama indique os tipos de cada atributo, os parâmetros de cada método e seus tipos e muitas vezes o tipo do retorno de cada método. Por hora vamos usar esta versão simplificada para focar na tradução desse diagrama para o Python.

5.3.1. Implementando classes em Python

Em Python, a criação de uma classe é feita com o uso da palavra chave `class`, seguida do nome da classe. Todas as classes em Python herdam de uma classe especial chamada *object*, que garante que novas classes terão os métodos comumente esperados de uma classe em Python. Isso facilita muito a programação e a partir da versão 3 do Python, foi introduzida uma nova sintaxe para a criação de classes, na qual não é preciso mais indicar explicitamente tal herança.

```
class NomeDaClasse:
    <bloco de código da classe>
```

Vamos fazer alguns exemplos na *Shell* do Python para entender o funcionamento deste comando. Abra a IDLE ou digite o comando do Python referente ao seu sistema operacional (`python`, `py` ou `python3`) no terminal do VSCode para abrir uma *Shell*.

```
>>> class TV:
...     pass
```

O comando acima cria uma classe cujo nome é TV, mas por hora não definimos nenhum atributo ou método ainda. Podemos verificar isso acessando o nome da classe.

```
>>> TV
<class '__main__.TV'>
```

5.3.2. Instanciando objetos a partir de uma classe em Python

Já para criar um objeto a partir desta classe devemos chamá-la, de maneira análoga a forma como chamamos funções, com parênteses.

```
>>> TV()
<__main__.TV object at 0x7f2c6f106310>
```

Ao chamarmos a classe, ela nos retorna um objeto de TV, dizemos que esse objeto é do tipo TV, pois uma classe define um novo tipo de dado. Em Python, podemos adicionar atributos diretamente ao objeto, então se quisermos criar uma nova TV podemos fazer:

[1] UML é a Linguagem Unificada de Modelagem, da sigla em inglês *Unified Modelling Language*.

```
>>> tv_1 = TV()
>>> tv_1.tela = 43
>>> tv_1.resolucao = 'FullHD'
>>> tv_1.fabricante = 'Samsung'
>>> tv_1.preco = 2400.0
>>> tv_1.fotos = []
>>> tv_1.descricao = 'TV FullHD 43" - Samsung'
```

Podemos agora usar a função integrada `vars`, que devolve um dicionário com os atributos de um objeto em Python, para conferir que nosso objeto de fato possui os atributos que criamos nele.

```
>>> vars(tv_1)
{'tela': 43, 'resolucao': 'FullHD', 'fabricante': 'Samsung', 'preco':
2400.0, 'fotos': [], 'descricao': 'TV LED FullHD de 43" - Samsung'}
```

Agora se quisermos criar um segundo objeto, podemos repetir o processo. Vamos criar outro objeto e verificar os atributos que ele possui inicialmente.

```
>>> tv_2 = TV()
>>> vars(tv_2)
{}
```

Como não colocamos nada em nossa classe, o novo objeto começa vazio, então criar novos objetos dessa forma não é algo prático na maioria das situações.

5.3.3. Como inicializar um objeto em Python

Para resolver esse problema, o Python possui um método especial que podemos definir para que o objeto seja inicializado. Esse método deve obrigatoriamente ser chamado `__init__` e será executado uma vez no momento da criação de cada objeto. Os métodos especiais em Python são métodos que começam e terminam com dois sublinhados, também chamados de *dunder*[2] *methods*. A lista completa desses métodos pode ser vista na documentação (PSF, 2021b).

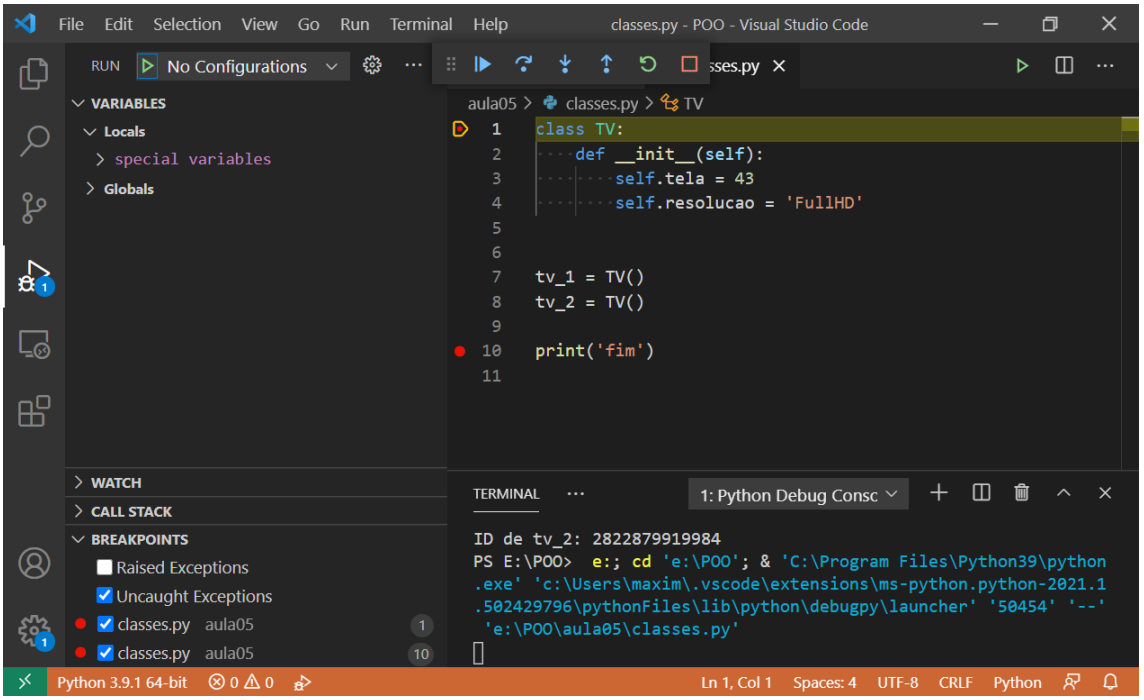
[2] Do inglês *double underscore*.

Vamos agora pro editor do VSCode. Crie um arquivo chamado “classes.py” na pasta “aula05” e digite o seguinte código nele:

```
class TV:
    def __init__(self):
        self.tela = 43
        self.resolucao = 'FullHD'

tv_1 = TV()
tv_2 = TV()
print('fim')
```

Em seguida salve o arquivo, adicione pontos de parada na primeira e na última linha e aperte F5 para executá-lo no modo de depuração. O resultado é mostrado na Figura 5.2.

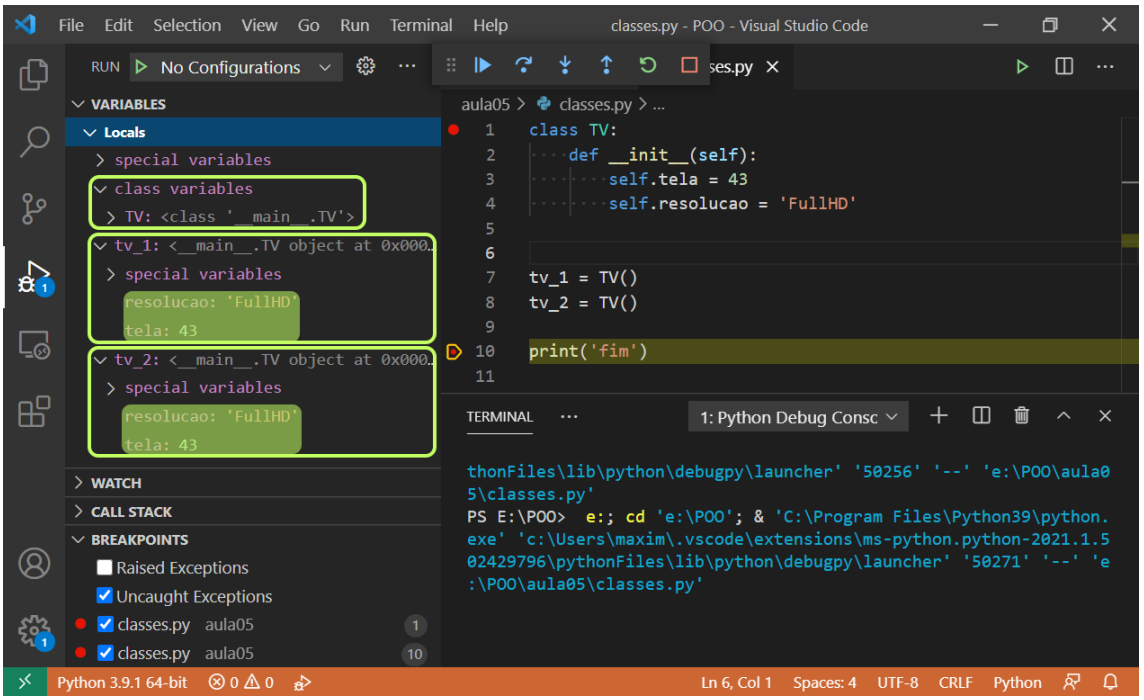
Figura 5.2: Início da execução no modo de depuração.



Fonte: do autor, 2021.

Agora avance na execução passo a passo e observe o que ocorre na memória com a execução de cada instrução deste código. A Figura 5.3 mostra o estado da memória após a criação dos dois objetos de TV.

Figura 5.3: Estado da memória após a execução do código.



Fonte: do autor, 2021.

Podemos observar que ambos objetos foram criados com os mesmos valores para os atributos. Isso acontece pois deixamos os valores fixos no código, mas, antes de ver como podemos alterar esse comportamento, precisamos entender o que é esse parâmetro `self` que colocamos em nossos métodos.

5.3.4. Entendendo o parâmetro *self*

Quando comparamos os paradigmas procedural com orientado a objetos, vimos que em POO, cada objeto tem consciência sobre si mesmo. Isso se dá através desse parâmetro *self*, que pode ser traduzido para "próprio" ou "si mesmo". Por padrão, o Python injeta uma referência para o objeto em questão como primeiro argumento de todos os métodos em uma classe.

Para confirmar isso, podemos usar a função `id`, que nos retorna o número de identificação do objeto, que é único para cada objeto criado na memória. No exemplo inicial na *Shell*, podemos ver que os dois objetos criados possuem identidades diferentes, pois são objetos diferentes na memória.

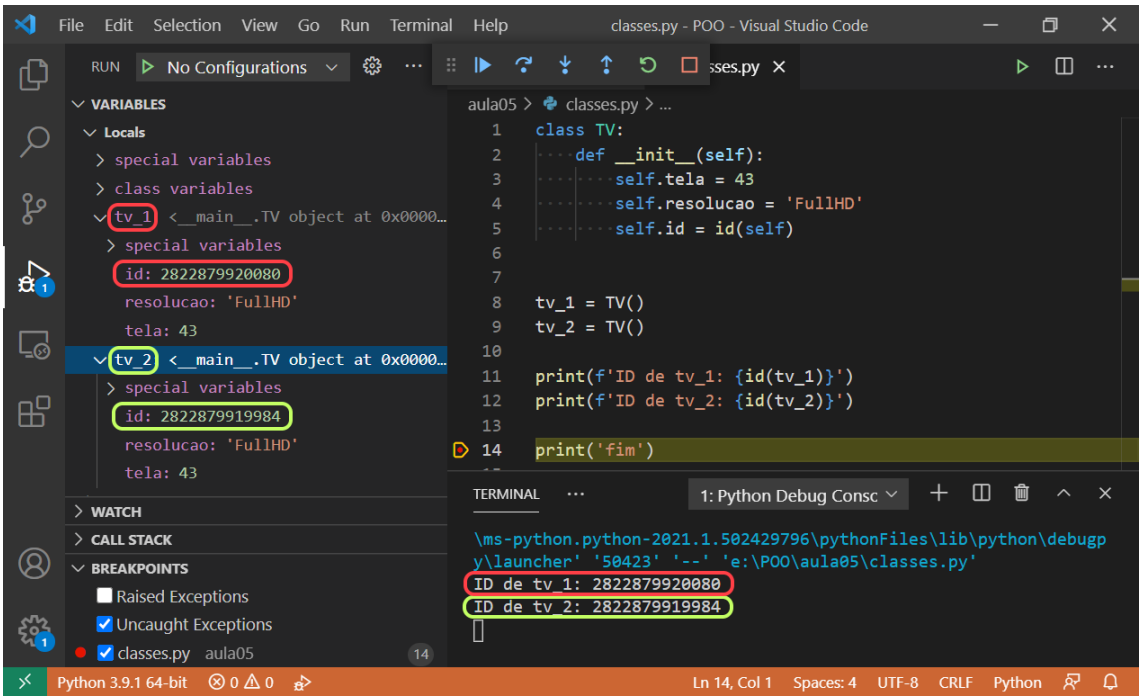
```
>>> id(tv_1)
2280534403472
>>> id(tv_2)
2280534403440
```

Vamos agora adicionar um atributo para guardar a identidade do objeto referenciado pelo `self`, e depois compará-la com a identidade de cada objeto de TV instanciado. Altere o código do arquivo "classes.py" para:

```
class TV:
    def __init__(self):
        self.tela = 43
        self.resolucao = 'FullHD'
        self.id = id(self)

tv_1 = TV()
tv_2 = TV()
print(f'ID de tv_1: {id(tv_1)}')
print(f'ID de tv_2: {id(tv_2)}')
```

Veja na Figura 5.4 o resultado da execução desse código no modo de depuração.

Figura 5.4: Comparação das identidades dos objetos com o valor do parâmetro *self*.



Fonte: do autor, 2021.

Como podemos ver, efetivamente o parâmetro `self` do método `__init__` recebe uma cópia da referência para o objeto a partir do qual é chamado. Isto é válido tanto para os

métodos com nomes especiais quanto para os demais métodos que criamos, com exceção dos métodos de classe e métodos estáticos, que veremos mais adiante no curso.

5.3.5. Personalizando a inicialização dos objetos em Python e incluindo novos métodos

Vamos então fazer a implementação completa da classe TV que vimos no diagrama da Figura 5.1. Como você talvez já tenha deduzido, para adicionar um novo método à classe basta definir uma função dentro do bloco de código da classe, que deverá obrigatoriamente ter o `self` como primeiro parâmetro.

Podemos ainda receber quantos argumentos forem necessários para o método adicionando mais parâmetros após o `self`, então para inicializar o objeto basta simplesmente adicionar mais parâmetros ao método especial `__init__`.

Agora, no momento de instanciar um objeto, podemos passar os valores que queremos atribuir para aquele objeto, e o Python ao criar o objeto, automaticamente executa o método `__init__`, repassando-lhe os valores dos argumentos que passamos à classe. Veja o código a seguir:

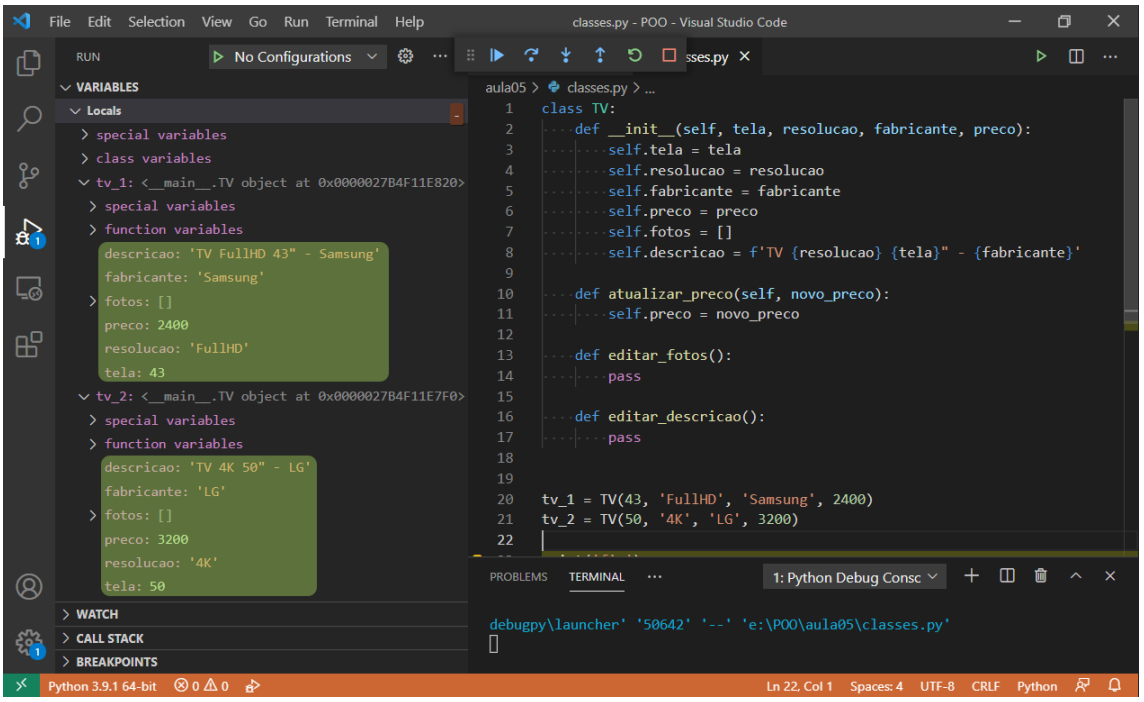
```
class TV:
    def __init__(self, tela, resolucao, fabricante, preco):
        self.tela = tela
        self.resolucao = resolucao
        self.fabricante = fabricante
        self.preco = preco
        self.fotos = []
        self.descricao = f'TV {resolucao} {tela}" - {fabricante}'

    def atualizar_preco(self, novo_preco):
        self.preco = novo_preco

    def editar_fotos(self):
        pass

    def editar_descricao(self):
        pass

tv_1 = TV(43, 'FullHD', 'Samsung', 2400)
tv_2 = TV(50, '4K', 'LG', 3200)

print('fim')
```

Aqui omitimos o código dos métodos de editar fotos e descrição, pois o objetivo é ver como funciona a criação dos métodos em uma classe. A palavra chave `pass` indica ao Python que nada deve ser feito ali. Podemos usá-la quando queremos criar primeiro a estrutura de uma classe, método ou função, e testar parcialmente seu funcionamento, antes de implementá-la completamente, evitando que haja um erro de sintaxe ao deixar um bloco de código vazio.

Caso seja passado um número diferente de argumentos para a classe TV no momento de criação de um novo objeto, isso irá gerar um erro de execução, pois em Python não podemos chamar uma função ou método com o número incorreto de parâmetros obrigatórios. Portanto, podemos agora instanciar quantas TVs forem necessárias de maneira prática e cada uma já irá possuir os valores adequados aos seus atributos.

A Figura 5.5 mostra o resultado da execução deste código no modo de depuração.

Figura 5.5: Visualização dos objetos criados e inicializados com valores personalizados.



Fonte: do autor, 2021.

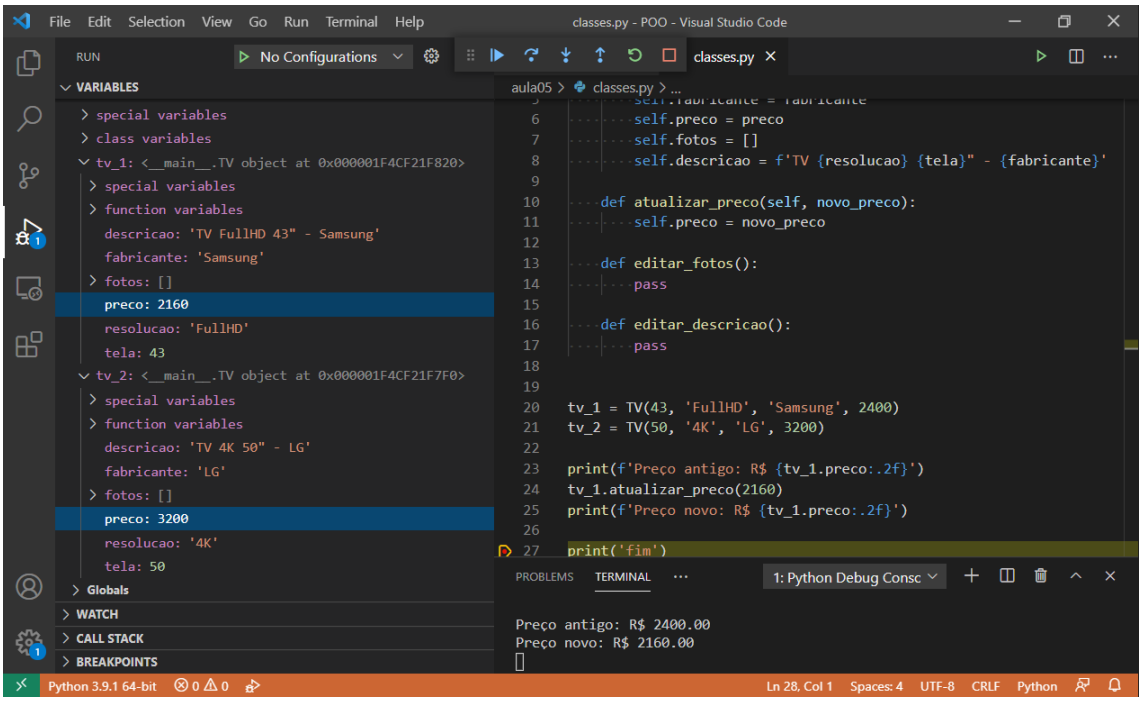
Vamos ver agora como podemos usar um dos métodos de um objeto. Nesse nosso exemplo, o único método funcional é o de atualizar o preço, os outros dois foram colocados apenas como exemplo e não fazem nada. Vamos então alterar o preço da primeira TV aplicando um desconto de 10%.

Com a implementação atual, precisamos informar diretamente o novo preço final, que com o desconto será de R$ 2160,00. Para isso, adicione as seguintes linhas ao final do código:

```
print(f'Preço antigo: R$ {tv_1.preco:.2f}')
tv_1.atualizar_preco(2160)
print(f'Preço novo: R$ {tv_1.preco:.2f}')
```

Na Figura 5.6 podemos ver que o preço do primeiro objeto foi alterado, mas não o do segundo, pois cada objeto só é capaz de alterar seus próprios atributos.

Figura 5.6: Visualização da execução de um método em um dos objetos criados.



Fonte: do autor, 2021.

VAMOS PRATICAR!

1) Crie um método aplicar_desconto, que deverá receber um número inteiro representando o desconto percentual a ser aplicado e então deve internamente ler o valor do preço atual, calcular o novo preço e atualizar o atributo com este novo valor.

2) Faça também um exercício de abstração, pense que outros métodos ou atributos poderiam ser interessantes para os objetos de TV no contexto que vimos nesse capítulo, como um produto de um e-commerce. Faça uma lista e discuta com seus colegas os motivos para cada método ou atributo que pensou.

5.3.6. Encapsulamento

A ideia principal do encapsulamento é ocultar atributos e métodos que não devem ser acessíveis a partes externas da aplicação. Dentre as razões para fazermos isso, podemos citar:

- Possibilita a inclusão de regras de negócio para validar ou preparar os dados antes de fazermos a alteração do valor de um atributo;
- Aumenta a segurança do código contra bugs ou erros inesperados, pois limita quem pode alterar os atributos de um objeto;
- Facilita a manutenção do código, pois caso seja preciso alterar um método interno, só precisamos alterar o código da própria classe e o restante da aplicação ou programa não é afetado por esta alteração.

Quando falamos de encapsulamento, precisamos entender três conceitos importantes de POO, e como esses conceitos são aplicados no Python. Em POO podemos ter atributos e métodos de um objeto classificados de acordo com sua visibilidade e acessibilidade em:

- *Públicos*: são aqueles que podem ser acessados diretamente por qualquer parte da aplicação com acesso ao objeto em si. No diagrama de classes da UML, são marcados precedidos do sinal `+`.
- *Protegidos*: são aqueles que podem ser acessados apenas pelo próprio objeto e seus descendentes, isto é, objetos de classes que tenham estendido a classe original, através de herança. São indicados na UML pelo sinal de `#` e falaremos mais a respeito deles no capítulo sobre herança.
- *Privados*: são aqueles que só podem ser acessados pelo próprio objeto e por mais nenhuma outra parte da aplicação ou programa. Indicados na UML por `-`.

Cada linguagem implementa estes conceitos de uma maneira diferente, havendo pontos positivos e negativos em todas elas. No caso do Python, tais conceitos não são impostos pela linguagem, isto é, todos os atributos e métodos de um objeto estão sempre visíveis e acessíveis. Mas isso não significa que não seja possível utilizar tais conceitos a nosso favor em Python.

A PEP8 não utiliza a nomenclatura padrão vista acima, classificando os atributos e métodos apenas em públicos e não-públicos, mas isso não significa que os conceitos não possam ser aplicados ao design das nossas classes. Portanto a recomendação para nomear os atributos e métodos é:

- *Públicos*: seguem a mesma regra para nomenclatura de variáveis e funções `letras_minusculas_separadas_por_sublinhado`.

- *Não-públicos*, quando tratados como:
  - *Protegidos*: devem ser precedidos por um sublinhado `_atributo_protegido`.
  - *Privados*: devem ser precedidos por dois sublinhados `__atributo_privado`.

Lembrando que os nomes especiais, como o `__init__`, são precedidos e sucedidos por dois sublinhados, e não podem ser inventados, isto é, só podemos usar os nomes especiais definidos na documentação do Python.

Quando criamos um atributo que queremos tratar como privado, precedendo-o com dois sublinhados, o Python realiza o que chamamos de "desfiguração de nomes", alterando o identificador (nome do atributo ou método) para incluir o nome da classe: um atributo `__atributo_privado` que seja definido em qualquer parte da classe `MinhaClasse` terá seu identificador transformado pelo Python em `_MinhaClasse__atributo_privado` (PSF, 2021c).

Sendo assim, quando estamos escrevendo uma classe em Python, sabemos que os clientes ou consumidores desta classe, isto é, qualquer parte da aplicação ou programa que faça uso da nossa classe, deverão utilizar apenas os métodos e atributos que sejam públicos. Seguindo esta recomendação, sabemos que no futuro, podemos alterar os nomes e a implementação de qualquer método ou atributo não-público sem nos preocupar em quebrar o código que faz uso atualmente da nossa classe.

5.3.6.1. Trabalhando com atributos não-públicos em Python

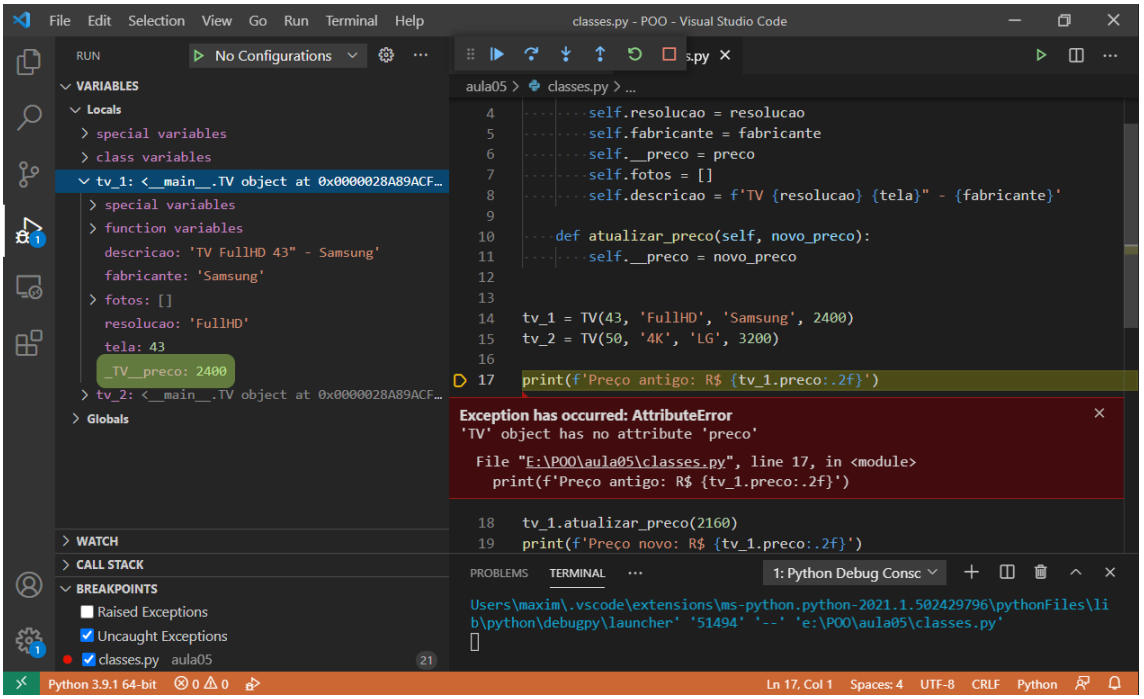
Vamos ver como isso funciona na prática. No exemplo da TV, o preço é um atributo público, assim como todo o resto, então qualquer parte do nosso programa pode acessá-lo e alterá-lo com facilidade, basta atribuir um novo valor ao atributo:

```
tv_1.preco = 300
```

Que o atributo `preco` terá seu valor alterado no objeto. Portanto, podemos transformá-lo em um atributo não-público, de modo que o único responsável por alterá-lo seja o próprio objeto. O mais comum é usar apenas um sublinhado para atributos não-públicos, deixando o uso de dois sublinhados para casos específicos. Falaremos mais a respeito ao estudar herança, por hora, vamos fazer um exemplo com o uso de dois sublinhados para visualizarmos o funcionamento da "desfiguração de nomes" do Python.

Para isso altere o nome do atributo de `self.preco` para `self.__preco`, em todas as suas ocorrências, e vamos executar novamente nosso código. Veja a Figura 5.7.

Figura 5.7: Visualização da execução após tornar o atributo *preco* não-público.



Fonte: do autor, 2021.

Como podemos ver, quebramos a aplicação pois agora não podemos mais acessar o atributo `tv_1.preco`, ele não existe! O nome deste atributo agora é `_TV__preco`, e poderíamos fazer `tv_1._TV__preco`[3] que isso funcionaria. Mas ao fazer isso estamos violando a privacidade deste atributo, pois ele não é mais público e só deve ser acessado diretamente no interior do objeto.

Então, ao criar um atributo não-público, podemos escolher se queremos que ele seja 100% não-público, aberto para leitura ou aberto para leitura e escrita, e fazemos isso através dos métodos conhecidos por *getters* e *setters*. Para abrir o atributo para leitura, definimos um método *getter*, que ficará responsável por recuperar e retornar o valor do atributo; e para abrir o atributo para escrita, definimos um método *setter*, que ficará responsável por atribuir um novo valor ao atributo. No exemplo que fizemos, o método para atualizar o preço já está fazendo o papel de um *setter*.

Tradicionalmente em POO, estes métodos devem ser nomeados com o nome do atributo precedido de da palavra *get* ou *set*. Portanto, faça uma cópia do arquivo "classes.py" e renomeie-o "classes_getters_e_setters.py", e vamos alterar neste novo arquivo o nome do método `atualizar_preco` para `set_preco` e criar outro método chamado `get_preco`.

[3] Atenção: Não faça isso!

O código dos métodos deverá ficar assim:

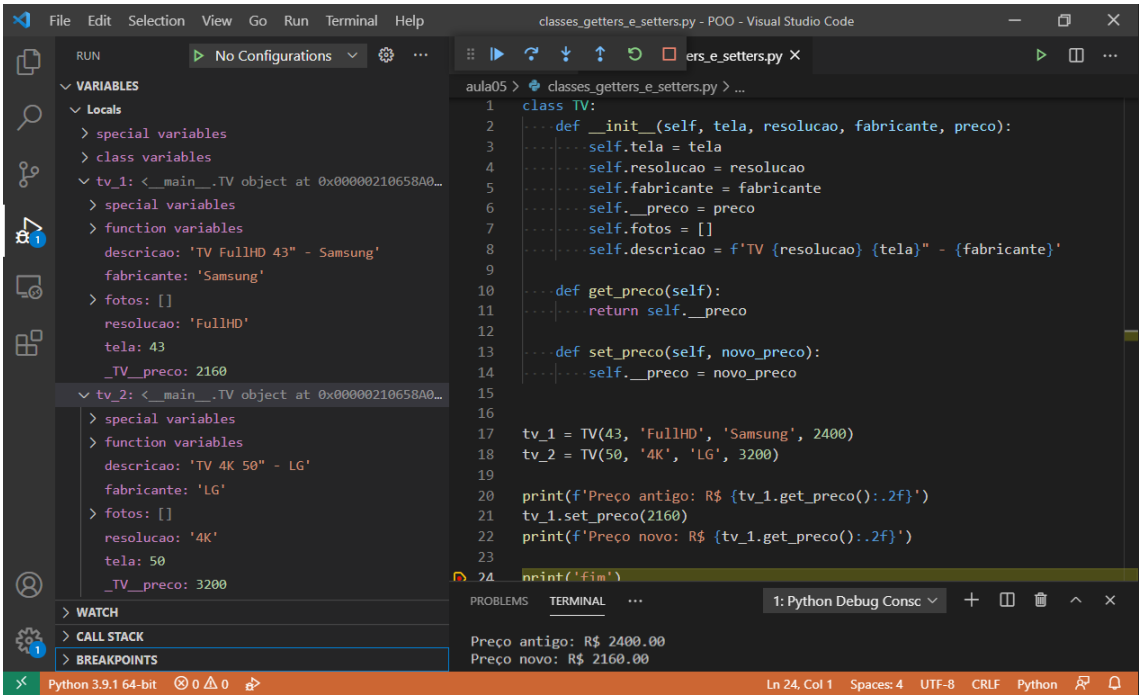
```
class TV:
    def __init__(self, tela, resolucao, fabricante, preco):
        self.__preco = preco
        ...  # demais atributos não são alterados

    def get_preco(self):
        return self.__preco

    def set_preco(self, novo_preco):
        self.__preco = novo_preco
```

Agora, não precisamos mais acessar o atributo não-público diretamente, e podemos então alterar as linhas que estão utilizando nossos objetos de TV para usar o *getter* e o *setter*:

```
print(f'Preço antigo: R$ {tv_1.get_preco():.2f}')
tv_1.set_preco(2160)
print(f'Preço novo: R$ {tv_1.get_preco():.2f}')
```

Veja na Figura 5.8 a execução do arquivo “classes_getters_e_setters.py”.

Figura 5.8: Visualização da execução do arquivo com *getters* e *setters*.



Fonte: do autor, 2021.

A principal vantagem de se fazer isso é que agora o acesso, tanto para escrita quanto para leitura, é feito por um método, no qual podemos introduzir qualquer lógica que seja necessária para validar as ações sendo feitas. Por exemplo, em `set_preco`, poderíamos antes de alterar o atributo de preço, verificar se o funcionário que está logado no sistema atualmente tem a permissão para fazer isso, ou então pedir que seja entrada uma senha para continuar com a operação. Um exemplo de código ilustrativo de tal verificação é:

```
from modulo_validacao import valida_autorizacao

class TV:
    # restante do código sem alterações

    def set_preco(self, novo_preco):
        senha = input('Digite a senha de autorização: ')
        if not valida_autorizacao(senha):
            return
        self.__preco = novo_preco
```

Dessa forma, estamos colocando uma camada extra de proteção na alteração do preço. Agora apenas os funcionários que possuírem uma senha de autorização para editar os preços das TVs poderão fazê-lo. Esse é o conceito de encapsulamento, o acesso a um atributo está encapsulado dentro de métodos que o protegem.

5.3.6.2. Utilizando os decoradores *@property* e *@property.setter*

Na criação dos *getters* e *setters* tradicionais, como vimos acima, precisamos alterar um código que já fazia uso do nosso atributo público `preco` para uma chamada de método. No entanto, o Python possui uma forma mais natural de criarmos *getters* e *setters*, que facilitam o acesso dos atributos pelos clientes da nossa classe ao mesmo tempo que permitem as verificações que fizemos ao criar os métodos tradicionais.

Isto é feito através de um decorador chamado *property*. Veremos em mais detalhes o que são decoradores mais pra frente no curso, por hora é suficiente entender que são funções especiais que podemos usar para decorar nossos métodos, fornecendo-lhes uma funcionalidade extra. E aplicamos um decorador colocando-o, precedido do símbolo `@`, na linha imediatamente anterior à definição do método.

Ao decorarmos um método com o decorador *property*, estamos criando um *getter*. O Python irá criar um identificador público com o mesmo nome do método, que funcionará como um atributo padrão para o mundo exterior ao objeto, e toda vez que o atributo público for acessado, por baixo dos panos o Python irá chamar o método associado a ele pelo decorador *property*.

Para criar um *setter*, devemos decorar o método do *setter* com o decorador *<nome>.setter*, onde *<nome>* deve ser o nome público criado pelo decorador *property*. Com isso, toda vez que um valor for atribuído ao atributo público criado pelo Python, ele irá repassar esse valor para o método associado ao *setter*.

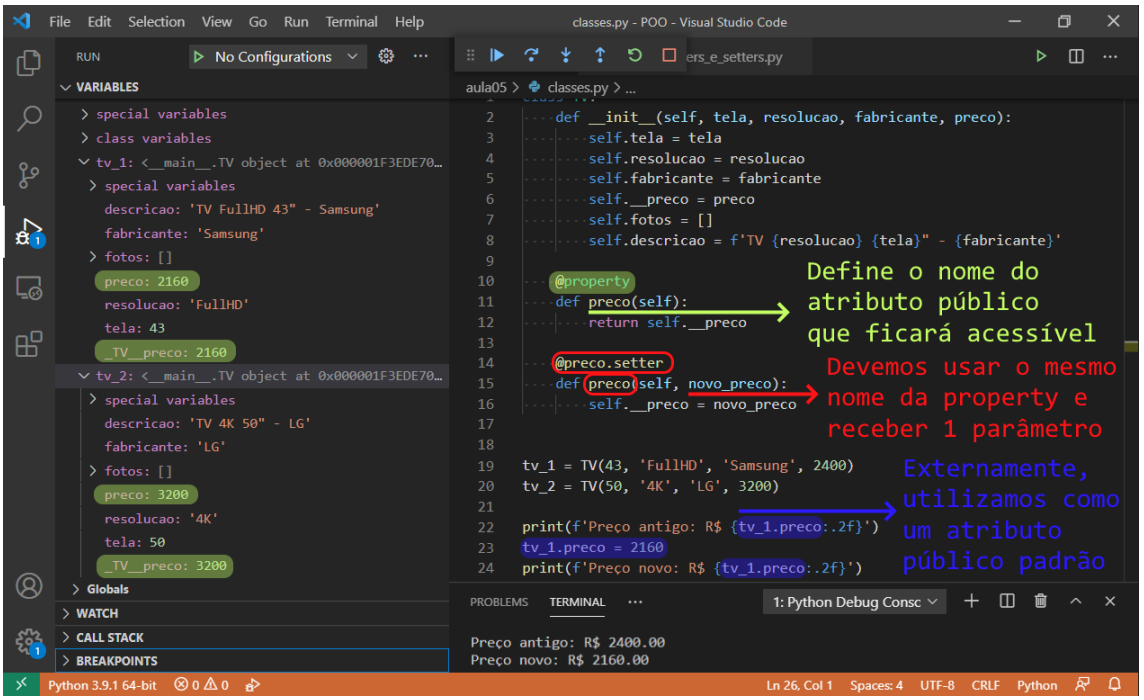
> Importante: para criar um *setter*, é obrigatório antes criar uma *property*. Mas é possível criar apenas a *property*, sem criar o *setter* associado a ela.

Vamos ver agora como ficaria o exemplo do preço utilizando estes decoradores. Para isso vamos voltar para o arquivo "classes.py" e alterar os métodos para:

```
class TV:
    def __init__(self, tela, resolucao, fabricante, preco):
        self.__preco = preco
        ...  # demais atributos não são alterados

    @property
    def preco(self):
        return self.__preco

    @preco.setter
    def preco(self, novo_preco):
        self.__preco = novo_preco
```

Veja na Figura 5.9 o resultado da execução deste código no modo de depuração.

Figura 5.9: Visualização do uso dos decoradores *property* e *property.setter*.



Fonte: do autor, 2021.

Pontos importantes a se observar:

- Na criação da *property*:
  - Podemos escolher o nome que quisermos para o método, e ele será adotado como o nome do atributo que será exposto publicamente;
  - No interior do método decorado com *@property*, não podemos jamais acessar o atributo público que ela cria, pois isso irá criar uma recursão infinita, e o programa não irá funcionar. No exemplo acima, no interior do método `preco()` definido na linha 11, é proibido acessar o atributo `self.preco`;
  - Esse método pode conter verificações, se necessário, mas o mais comum é apenas retornar o valor do respectivo atributo não-público.
- Na criação do *setter*, que é opcional, devemos:
  - Sempre usar o mesmo nome criado pela property;
  - Sempre receber um único parâmetro, além do `self` que o Python automaticamente injeta em todos os métodos. O nome desse parâmetro não importa, basta usar o mesmo nome dentro do método, portanto escolha um nome representativo;
  - Em geral, esse método não possui retorno de valor, e é comum realizarmos verificações antes de alterarmos de fato o valor do respectivo

atributo não-público, sendo também comum haver um retorno antecipado (vazio) quando alguma validação falha.

- Este atributo criado pela *property* será usado pelos clientes da classe (demais módulos da nossa aplicação) como se fosse um atributo padrão, sem que eles tenham conhecimento da implementação por trás. Portanto, não devemos implementar métodos que tenham um alto custo computacional ou que levem muito tempo para serem processados. Se for este o caso, evite usar uma *property*/*setter* e crie um método tradicional para alterar o atributo não-público, pois ao utilizar o *setter*, os clientes da classe estarão esperando interagir com um atributo de dados, cujo acesso é extremamente rápido e de baixo custo computacional.

Referências

PSF. **Style Guide for Python Code: Class Names**. 2021a. Disponível em: <https://www.python.org/dev/peps/pep-0008/#class-names>. Acesso em: 14 fev. 2021.

PSF. **Data Model: Special Method Names**. 2021b. Disponível em: <https://docs.python.org/3/reference/datamodel.html#special-method-names>. Acesso em: 14 fev. 2021.

PSF. **Classes: Private Variables**. 2021c. Disponível em: <https://docs.python.org/3/tutorial/classes.html#private-variables>. Acesso em: 14 fev. 2021.