11

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

11

Tópicos Avançados de Orientação a Objetos

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo estudaremos um pouco das características funcionais do Python, isto é, como o Python trata os objetos de funções e quais são algumas das vantagens que surgem desse tratamento. Vamos estudar o que são os decoradores, e como podemos criar nossos próprios decoradores. Veremos também o conceito de classes abstratas e como ele pode ser aplicado em Python.

10.1. Decoradores em Python

Em Python, funções são consideradas cidadãs de primeira classe, o que significa que elas podem ser passadas como argumento para outras funções e podem também ser usadas como valor de retorno, e isso nos permite, entre muitas outras coisas, a criação de decoradores como o `property` e `.setter` que usamos até agora.

Para entender um pouco melhor sobre como funcionam os decoradores, vamos antes estudar o que significa uma função ser uma "cidadã de primeira classe" em Python e ver alguns exemplos de situações comuns em que tal característica pode nos ajudar a simplificar a codificação, deixando a mais simples e fácil de ler.

10.1.1. Funções como argumentos para outras funções

Pense no seguinte problema: criar uma função que converta uma lista de números para *string* ou uma lista de *strings* para números. Uma solução inicial que poderíamos implementar para resolver esse problema é mostrada na Codificação 11.1.

Codificação 11.1: Função de conversão (versão 1) usando estruturas de seleção

```
def converte_v1(para, lista):
    nova_lista = []
    for x in lista:
        if para == 'int':
            nova_lista.append(int(x))
        elif para == 'float':
            nova_lista.append(float(x))
        elif para == 'str':
            nova_lista.append(str(x))
    return nova_lista
```

Fonte: do autor, 2021

Essa função já nos ajuda muito a resolver problemas nos quais precisamos converter os dados de uma lista de um tipo a outro, como podemos ver no exemplo da Codificação 11.2.

Codificação 11.2: Utilização da versão 1 da função de conversão

```
>>> s = (0, 1, 2, 3)
>>> converte_v1('str', s)
['0', '1', '2', '3']
>>> converte_v1('float', s)
[0.0, 1.0, 2.0, 3.0]
```

Fonte: do autor, 2021

No entanto, ela ainda é bastante limitada, pois se precisarmos converter os dados para um outro tipo que não está previsto na função, precisamos editá-la. Outra desvantagem é que, conforme a função cresce, torna-se mais difícil saber quais são todos os tipos que ela aceita e manter isso bem documentado pode ser trabalhoso.

Felizmente, temos uma solução que é ao mesmo tempo fácil e elegante para este problema. Basta fazer com que a nossa função de conversão receba a função que irá aplicar como parâmetro e utilize-a dentro de seu código para fazer a conversão de fato. Veja o exemplo da Codificação 11.3.

Codificação 11.3: Função de conversão (versão 2) recebendo a função como parâmetro

```
def converte_v2(f, lista):
    nova_lista = []
    for x in lista:
        nova_lista.append(f(x))
    return nova_lista
```

Fonte: do autor, 2021

Agora, nossa função de conversão não precisa mais saber para qual tipo de dado a conversão será feita e então decidir a partir de uma estrutura fixa no código, qual

função chamar, pois ela já irá receber a função que deverá chamar como parâmetro. Veja o exemplo de uso na Codificação 11.4.

Codificação 11.4: Utilização da versão 2 da função de conversão - exemplo 1

```
>>> s = (0, 1, 2, 3)
>>> converte_v2(str, s)
['0', '1', '2', '3']
>>> converte_v2(float, s)
[0.0, 1.0, 2.0, 3.0]
```

Fonte: do autor, 2021

Repare que a utilização é muito parecida, mas agora nós não passamos uma *string* como argumento, e sim a própria função que desejamos usar na conversão, no caso str e float. Observe que não há aspas, pois não é uma *string* e tampouco colocamos os parênteses após a função, pois não queremos executá-la no momento da chamada da função. O objetivo é passar sua referência para dentro da função `converte_v2`, que ficará então responsável por chamá-la no momento certo.

Isso abre caminho para utilizarmos quaisquer outras funções para converter os dados, sem que seja necessário alterar uma única linha sequer em nossa função de conversão. Veja mais alguns exemplos de uso na Codificação 11.5.

Codificação 11.5: Utilização da versão 2 da função de conversão - exemplo 2

```
>>> s = (0, 1, 2, 3)
>>> converte_v2(bool, s)
[False, True, True, True]
>>> def dobro(n):
        return n * 2

>>> converte_v2(dobro, s)
[0, 2, 4, 6]
```

Fonte: do autor, 2021

Isso é tão útil que o próprio Python já possui uma função integrada que faz exatamente isso, a função `map`, que recebe uma função e uma sequência como parâmetros, e retorna uma nova sequência formada aplicando a função recebida a cada um dos itens da sequência.

VAMOS PRATICAR!

Com base no exemplo acima, crie uma função de filtro, que irá receber dois parâmetros: uma função `f` e uma sequência `s`, e deve retornar uma lista contendo apenas os itens da sequência para os quais a função `f` retornar True. A função f deve receber um parâmetro e retornar um verdadeiro ou falso, baseado no parâmetro recebido.

10.1.2. Funções como valor de retorno de outras funções

Em Python, é possível criar uma função dentro do escopo local de outra função, e também retornar essa função como valor de retorno da função principal. Veja na Codificação 11.6 um exemplo ilustrativo dessa funcionalidade, embora sem uso prático.

Codificação 11.6: Função que cria uma função local e retorna sua referência

```
def principal():
    def interna():
        return 'Executando a função interna'
    return interna
```

Fonte: do autor, 2021

Observe que estamos retornando a referência para a função local `interna`, sem chamá-la, pois não queremos executar seu código e então repassar o retorno, queremos retornar a referência para a própria função em si. Veja como podemos usá-la na Codificação 11.7.

Codificação 11.7: Utilização da função local criada internamente à função principal

```
>>> f = principal()
>>> f()
'Executando a função interna'
```

Fonte: do autor, 2021

Ao executarmos a função `principal`, recebemos a referência para a função interna e a atribuímos à variável `f`, que é chamada em seguida.

10.1.3. Decoradores

Com esses dois conceitos, receber funções como parâmetros e retornar funções a partir de outras funções, podemos finalmente entender o que é um decorador em Python. Basicamente um decorador é uma função que recebe como parâmetro uma outra função, cria internamente uma nova função que executa um dado código, chama a função recebida, e então retorna a função criada internamente.

Se esta explicação pareceu confusa, é porque sem um exemplo prático, entender o conceito de um decorador é de fato bastante complicado. Então vamos rever tal explicação a partir do exemplo na Codificação 11.8.

Codificação 11.8: Criação de um decorador simples e uma função de teste

```
def decorador(f):
    def envelope():
        print('código executado antes de chamar f')
        f()
        print('código executado após chamar f')
    return envelope

def ola_mundo():
    print('Olá, mundo!')
```

Fonte: do autor, 2021

Aqui criamos uma função chamada `decorador`, que irá servir para "decorarmos" outras funções, e criamos também a função `ola_mundo`, que será a função que iremos decorar no exemplo da Codificação 11.9.

Codificação 11.9: Utilização do decorador na função de teste

```
>>> ola_mundo()                          # execução da função original
Olá, mundo!
>>> ola_mundo = decorador(ola_mundo)     # decoramos ola_mundo
>>> ola_mundo()                          # execução da nova função
código executado antes de chamar f
Olá, mundo!
código executado após chamar f
```

Fonte: do autor, 2021

A essa função interna, é comum darmos o nome genérico de `envelope`, ou em inglês *wrapper*, pois essa função irá envelopar a chamada da função recebida. Atente que não a chamaremos pelo nome dado, pois no retorno da função principal, o que importa é o valor retornado, no caso a referência para um objeto de função na memória, e não o nome da variável local ao qual ele estava associado.

Resumindo a explicação no começo da seção, decoradores são funções que envelopam, ou envolvem outras funções, modificando seu comportamento, mas sem alterar o código interno da função decorada.

Vejamos outro exemplo, um pouco mais prático. Digamos que eu queira criar um decorador que me permita cronometrar o tempo de execução das minhas funções, para comparar seu desempenho. Eu sei que o Python possui a biblioteca timeit, já pensada para exatamente essa situação, e se estivesse fazendo isso para escrever um artigo científico, usaria a biblioteca do Python, que garante uma avaliação mais robusta. Mas como acabo de aprender sobre decoradores, e estou fazendo isso apenas como um projeto pessoal para aplicar meu conhecimento, decidi implementar minha própria solução para a questão. Veja a Codificação 11.10.

Codificação 11.10: Criação de um decorador para cronometrar a execução de funções

```
from time import perf_counter

def cronometro(f):
    def envelope():
        t1 = perf_counter()
        f()
        t2 = perf_counter()
        print(f'{f.__name__} executada em {t2-t1:.4e} segundos')
    return envelope
```

Fonte: do autor, 2021

Agora, caso precise contar o tempo de execução de uma função qualquer, basta decorá-la com meu cronômetro e executá-la, que o tempo em segundos será exibido na tela, usando notação científica[1], com duas casas decimais. Um exemplo de utilização deste decorador é dado na Codificação 11.11.

Codificação 11.11: Utilização do decorador para cronometrar a execução de uma função

```
>>> from time import sleep
>>> def funcao_exemplo():
        print('esperando 2 segundos')
        sleep(2)

>>> funcao_exemplo()                # execução da função original
esperando 2 segundos
>>> funcao_exemplo = cronometro(funcao_exemplo)
>>> funcao_exemplo                  # execução da função decorada
esperando 2 segundos
funcao_exemplo executada em 2.008e+00 segundos
```

Fonte: do autor, 2021

Neste exemplo, podemos ver que o Python levou 2.008 segundos para executar a nossa função de exemplo, que apenas espera por dois segundos. O tempo extra se deve ao fato que essa função precisa fazer outras tarefas além de esperar os 2 segundos, como exibir uma mensagem na tela, e o tempo total vai sempre depender das condições do processador e memória disponível no momento da execução. Em outros momentos, essa diferença poderia ser menor ou maior, mas o importante é que conseguimos criar uma forma de medir isso com um simples decorador.

Agora digamos que eu queira criar um arquivo com algumas funções para praticar a resolução de algoritmos e queira sempre cronometrar tais funções. Nessa situação, existe uma forma mais simples de aplicarmos o decorador que já conhecemos:

[1] Isso é especialmente útil quando precisamos visualizar números muito pequenos ou muito grandes, por exemplo, 0.00000000000356 vira 3.56e-12 e 25800000 vira 2.58e+7. Assim não precisamos nos preocupar em contar todo esse monte de zeros.

a utilização da notação com o @ no momento de criação da função (ou método). Esta forma é apenas um açúcar sintático que facilita a aplicação de um decorador, veja o exemplo da Codificação 10.12, assumindo que ela esteja no mesmo arquivo do nosso decorador criado na Codificação 10.10.

Codificação 11.12: Aplicação de um decorador com usando o @

```
def cronometro():
    ...  # código do decorador omitido aqui

@cronometro
def minha_funcao():
    ... # código interno da minha função
```

Fonte: do autor, 2021

Com a notação acima, podemos mais facilmente decorar nossas funções, mas esse decorador ainda é muito simples, pois só conseguimos decorar com ele funções que não recebem nenhum parâmetro, e não possuem nenhum retorno. No caso do retorno, funcionaria caso a função retornasse algo, mas esse retorno seria perdido, pois não o estamos capturando dentro da nossa função envelope.

Poderíamos adicionar um parâmetro à nossa função envelope, repassar esse parâmetro para a chamada interna, da função `f`, capturar o retorno e isso serviria para funções com um parâmetro, mas não com zero ou dois. Então, para deixarmos nosso cronômetro realmente genérico, podemos utilizar o que vou chamar aqui de parâmetros estrelados do Python, comumente conhecidos como `*args` e `**kwargs`[2].

O asterisco simples faz o desempacotamento de listas e tuplas a variáveis, e quando usado nos parâmetros de uma função, captura um número qualquer de argumentos posicionais. É assim que a função integrada `print` é capaz de funcionar recebendo um número qualquer de argumentos.

O asterisco duplo faz o desempacotamento de dicionários para funções, passando seu conteúdo como argumentos nomeados para a função, sendo a chave do dicionário o nome do parâmetro e o valor do dicionário o valor passado como argumento para aquele parâmetro. Veja o exemplo da Codificação 11.13.

[2] O termo args vem do inglês *arguments*, que podemos traduzir para argumentos, e kwargs vem de *keyword arguments*, que pode ser traduzido para argumentos por palavra-chave, que chamamos aqui de argumentos nomeados.

Codificação 11.13: Exemplo de desempacotamento de um dicionário para os parâmetros de uma função

```
>>> def emite_senha(nome, senha):
        return f'Olá {nome}, sua senha é {senha}.'

>>> d = {'nome': 'Megan', 'senha': 23}
>>> emite_senha(**d)
'Olá Megan, sua senha é 23.'
```

Fonte: do autor, 2021

Quando usamos o asterisco duplo nos parâmetros de uma função, ele irá capturar todos os argumentos passados de maneira nomeada para a função, para os quais não existam parâmetros explicitamente definidos. Dessa forma, podemos combinar esses argumentos estrelados com a nossa função envelope para simplesmente capturar todos os argumentos passados, sejam eles posicionais ou nomeados, e repassá-los para nossa função decorada, de modo que nosso decorador passa a funcionar com qualquer função, independentemente do número de parâmetros que ela possua e se são nomeados ou posicionais. Veja o código final na Codificação 11.14.

Codificação 11.14: Generalização do decorador para funcionar com funções que recebam um número qualquer de parâmetros

```
from time import perf_counter

def cronometro(f):
    def envelope(*args, **kwargs):
        t1 = perf_counter()
        r = f(*args, **kwargs)
        t2 = perf_counter()
        print(f'{f.__name__} executada em {t2-t1:.4e} segundos')
        return r
    return envelope
```

Fonte: do autor, 2021

10.1.4. Decorador property

Vejamos como funciona o decorador que usamos até agora em diversas situações: a `property`. Ela é um decorador especial, bem mais complexo do que os que vimos neste capítulo, e vamos apenas ilustrar seu funcionamento.

Você deve ter reparado que não é preciso importar a `property` para utilizá-la, portanto podemos simplesmente digitar `help(property)` na Shell do Python, e isso nos trará algumas informações do seu uso. Esse decorador é uma classe que define diversos métodos especiais, mas estamos interessados agora na sua assinatura quando é chamado, mostrada na Codificação 11.15.

Codificação 11.15: Docstring do decorador *property*, obtido com `help(property)`

```
class property(object)
 |  property(fget=None, fset=None, fdel=None, doc=None)
 |
 |  Property attribute.
 |
 |    fget
 |      function to be used for getting an attribute value
 |    fset
 |      function to be used for setting an attribute value
 |    fdel
 |      function to be used for del'ing an attribute
 |    doc
 |      docstring
```

Fonte: do autor, 2021

Podemos notar que ela pode ser chamada com até 4 argumentos:

- Uma função para definir o acessor[3] de um atributo;
- Uma função para definir o modificador de um atributo;
- Uma função para definir como o atributo deve ser excluído; e
- Uma *string* de documentação.

Comumente usamos apenas os dois primeiros argumentos, e quando o argumento da *string* de documentação é omitido, automaticamente a *string* de documentação da função passada em `fget` será utilizada, caso exista.

Então, podemos criar uma property/setter definindo métodos não públicos que serão o nosso *getter* e *setter*, e então passar esses métodos para o decorador da property, e atribuir o retorno ao nome que pretendemos expor publicamente em nossa classe. Veja o exemplo na Codificação 11.16.

Codificação 11.16: Utilização do decorador property sem o uso do @

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

    def _get_nome(self):
        return self._nome

    def _set_nome(self, novo_nome):
        self._nome = novo_nome

    nome = property(_get_nome, _set_nome)
```

Fonte: do autor, 2021

[3] Aqui escrevemos acessor com C pois é um termo que deriva do verbo acessar, de dar acesso.

Nesse exemplo, estamos criando a nossa *property* como um atributo de classe, que fará internamente a atribuição e a leitura dos valores corretos de `self._nome` para cada objeto criado a partir desta classe. E estamos fazendo uso dessa *property* já no método `__init__`, que é executado no momento que o objeto é instanciado. Vale lembrar, que neste momento a classe já existe, então todos os demais métodos e a property já existem na memória (internos ao objeto[4] da classe na memória do Python).

O código mostrado na Codificação 11.16 tem exatamente o mesmo resultado que o código dado na Codificação 11.17, na qual o decorador é aplicado com o símbolo de @, forma mais recomendada pois facilita tanto a escrita quanto a leitura do código.

Codificação 11.17: Utilização do decorador property com o uso do @

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

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

    @nome.setter
    def nome(self, novo_nome):
        self._nome = novo_nome
```

Fonte: do autor, 2021

Para conferir, crie as duas classes no Python, em arquivos diferentes para não haver conflito de nomes, crie instâncias de ambas e compare os dicionários de atributos de cada instância, que podem ser obtidos com a função integrada `vars`.

10.2. Classes abstratas

Classes abstratas são classes cujo objetivo é servir de base para outras classes, definindo desta forma um conjunto de atributos e métodos que deverão ser comuns a todas as classes que a estenderem. Ou seja, uma classe abstrata não terá nenhum objeto instanciado a partir dela.

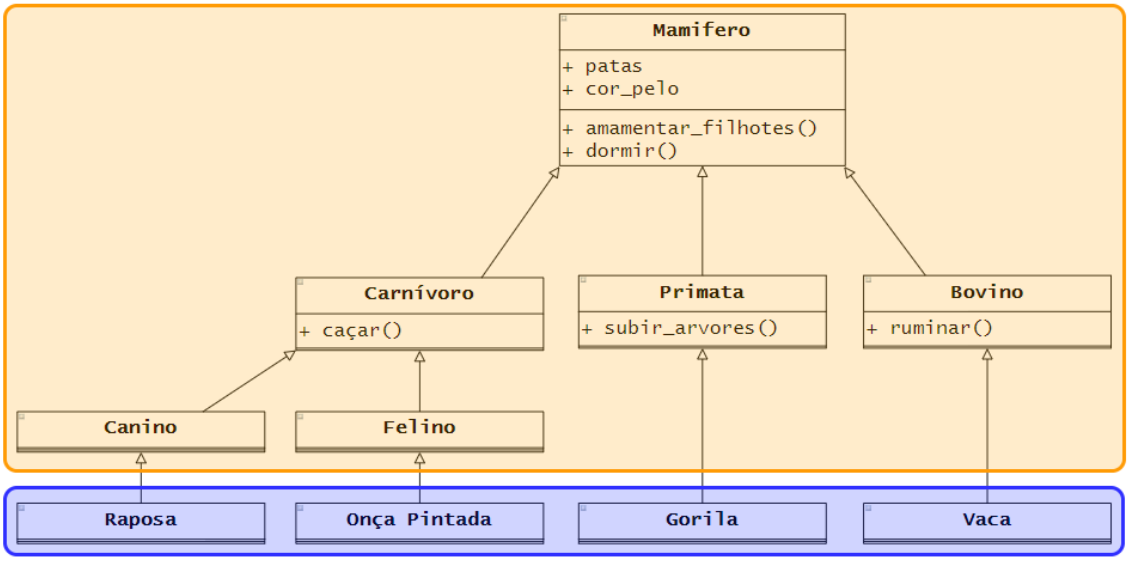
Se pensarmos em modelar classes para representar animais, poderíamos seguir uma classificação parecida com a definida por biólogos, para agrupar comportamentos e características comuns em classes mais genéricas, e criar uma hierarquia de herança que representasse a especialização dos animais em questão.

Por exemplo, para modelar os quatro animais a seguir: gorila, vaca, onça pintada e raposa, poderíamos criar diretamente uma classe para cada um, mas com certeza haveria código muito semelhante em todas elas, pois os quatro são mamíferos terrestres,

[4] Você leu corretamente, as classes em Python também são objetos (tudo é objeto), a diferença é que elas são objetos capazes de criarem instâncias do seu tipo. Uma classe Aluno é um objeto na memória capaz de criar outros objetos que são vistos como sendo "do tipo Aluno".

dois são carnívoros e dois são herbívoros, etc. Isto é, eles possuem características e comportamentos em comum, que podem ser agrupados em classes mais genéricas, mas também possuem comportamentos e características exclusivas a cada espécie. Uma possível forma de representar tais animais pode ser vista na Figura 11.1.

Figura 11.1: Exemplo ilustrativo de classes abstratas e concretas



Fonte: do autor, 2021

Na Figura 11.1, apenas as classes ressaltadas em azul teriam objetos instanciados a partir delas, pois não vemos na natureza nenhum exemplar que seja uma "instância" direta de `Primata`, `Carnivoro` ou `Mamifero`. Essas classes, em laranja, são uma invenção nossa para nos ajudar a compreender melhor a natureza à nossa volta, e no exemplo aqui, evitar duplicidade de código ao agrupar comportamentos semelhantes em classes mais genéricas através da herança de POO.

Dizemos então que as classes em laranja são abstratas e as classes em azul são concretas. Mas é importante notar que essa classificação entre abstrata e concreta não se faz em função de uma classe ser mãe ou filha, e sim de acordo com a existência ou não de objetos daquela classe no contexto em que está inserida. Poderíamos ter uma nova classe `RaposaDoArtico`, que herda de `Raposa`, e ambas seriam classes concretas.

Em Python, a possibilidade de se criar classes abstratas, isto é, a introdução de um mecanismo na própria linguagem que nos permita identificar se uma classe é ou não abstrata, além de coibir que objetos sejam instanciados a partir dela, foi introduzida com a PEP 3119 (ROSSUM, G. V., VIRIDIA, T., 2007). A documentação oficial (PSF, 2021) traz as definições de sintaxe e exemplos de uso para criação de classes abstratas.

Para criar uma classe abstrata em Python, devemos importar a classe `ABC`[5], do módulo `abc`, e então fazer com que nossa classe, que queremos marcar como abstrata, herde da classe `ABC`, como pode ser visto na Codificação 11.18.

[5] Do inglês *Abstract Base Classes*, que pode ser traduzido para Classes Base Abstratas.

Codificação 11.18: Criação de classes abstratas

```
from abc import ABC

class MinhaClasseAbstrata(ABC):
    pass
```

Fonte: do autor, 2021

Mas é importante observar que, assim como diversas outras características da linguagem, a simples marcação de uma classe como abstrata a partir da herança de ABC não impede que instâncias sejam criadas, isto é, não é da natureza da linguagem forçar o uso dessa característica e deve ser responsabilidade do programador estar atento a isso.

Isso se deve ao fato de que a razão de existir de uma classe abstrata é definir métodos que seus descendentes obrigatoriamente precisam possuir, sem no entanto precisar definir uma implementação para eles. Em geral, tal implementação sofrerá variações de uma subclasse para outra, de acordo com as características específicas de cada uma, então não faz sentido realizá-las na classe mãe.

A estes métodos, damos o nome de métodos abstratos. Caso uma classe possua métodos abstratos, então aí sim o Python irá forçar que não seja possível instanciar objetos a partir dela. Para marcar métodos como abstratos, usamos o decorador `abstractmethod`, também importado do módulo `abc`. Veja a Codificação 11.19.

Codificação 11.19: Criação de classes abstratas com métodos abstratos

```
from abc import ABC, abstractmethod

class MinhaClasseAbstrata(ABC):
    @abstractmethod
    def metodo_abstrato(self):
        pass
```

Fonte: do autor, 2021

Agora, ao tentarmos instanciar um objeto de `MinhaClasseAbstrata`, o Python irá lançar um erro de tipo, informando que não é possível realizar a operação:

```
TypeError: Can't instantiate abstract class MinhaClasseAbstrata with
abstract method metodo_abstrato
```

Ao criarmos subclasses da classe `MinhaClasseAbstrata`, caso o método `metodo_abstrato` não seja sobrescrito na subclasse, ela automaticamente se torna uma classe abstrata, mesmo sem usar a herança de `ABC`. Isso garante que todo objeto criado a partir das subclasses concretas de `MinhaClasseAbstrata` terá obrigatoriamente uma implementação para todos os métodos marcados como abstratos.

É possível também criar uma `property`/`setter` abstrato, bastando para isso marcá-los primeiro com o decorador `abstractmethod`. Ao adicionarmos mais de um decorador em uma função ou método, eles são executados de baixo para cima, isto é, o

primeiro decorador a ser aplicado será aquele mais próximo a linha de definição do método ou função. Veja o exemplo na Codificação 11.20.

Codificação 11.20: Criação de classes abstratas com property abstrata

```
from abc import ABC, abstractmethod

class MinhaClasseAbstrata(ABC):
    @property
    @abstractmethod
    def minha_property_abstrata(self):
        pass

    @minha_property_abstrata.setter
    @abstractmethod
    def minha_property_abstrata(self, valor):
        pass
```

Fonte: do autor, 2021

Além disso, é possível também criar métodos estáticos e métodos de classe abstratos, como mostra a Codificação 11.21.

Codificação 11.20: Criação de classes abstratas com métodos estáticos e métodos de classe também abstratos

```
from abc import ABC, abstractmethod

class MinhaClasseAbstrata(ABC):
    @staticmethod
    @abstractmethod
    def meu_metodo_estatico_abstrato():
        pass

    @classmethod
    @abstractmethod
    def meu_metodo_de_classe_abstrato(cls):
        pass
```

Fonte: do autor, 2021

Bibliografia

HJELLE, G. A., **Real Python**: primer on python decorators. 2015. Disponível em: <https://realpython.com/primer-on-python-decorators/>. Acesso em: 25 abr. 2021.

PSF. **Abc** - abstract base classes. 2021. Disponível em: <https://docs.python.org/3/library/abc.html>. Acesso em: 25 abr. 2021.

ROSSUM, G. V.; VIRIDIA, T. **PEP 3119** - introducing abstract base classes. 2007. Disponível em: <https://www.python.org/dev/peps/pep-3119/>. Acesso em: 25 abr. 2021.