Ada Tech [DS-PY-002] Programação Orientada a Objetos (PY)

Apresentação

Rudiney Casali:

- Doutorado em Física (UFSC/Univ. Coimbra)
- Especialista em Dados (Mackenzie)
- Lead Senior Data Scientist (Kyndryl)
- linkedin.com/in/rcasali/

Requerimentos

- Todos aqui estão familiarizados com a ferramenta Jupyter Notebook?
- Você pode editar seu JN no browser, e usando anaconda. Usaremos o editor de texto Visual Studio Code (NNF).
- Os encontros com a língua inglesa serão inevitáveis daqui pra frente…

Conteúdos

- Aula 1 (Expositiva e Prática): Introdução, Paradigmas de Programação, Programação Orientada a Objetos (POO);

- Aulas 2 e 3 (Expositiva e Prática): POO -
  - Classes e Objetos em Python,
  - Modulos e Propriedades,
  - Atributos Privados e de classe,
  - Métodos Mágicos,
  - Herança,
  - Polimorfismo;

- Aulas 4 e 5 (Expositiva e Prática): Desenvolvimento de Classes em POO;

- Aulas 6, 7 e 8 (Prática): Prática de POO em grupo;

  Aula 9 (Prática): Apresentação do Trabalho em POO e Rubrica.

Aulas 02 & 03 | Programação Orientada a Objetos

O problema das classes

Imagine que você tem um problema que precisa automatizar e que já tem uma ideia de paradigmas de programação deverá adotar para desenvolver tais processos. Agora você precisa organizar seu código de modo a atacar o problema da forma mais eficiente possível. Você escolhe a linguagem Python, por ser multi-paradigma e precisa arquitetar a forma mais eficiente de desenvolver seu código. Que objetos criar? E como organizá-los em classes, de modo a otimizar o acesso de tais objetos aos métodos definidos pela automatização?

Programação Orientada a Objetos (POO)

Um método de estruturação de programas por meio do agrupamento de propriedades e comportamentos a objetos individuais.

Os objetos são componentes do sistema, eles contêm dados (cru ou processado) ou ações a serem executadas.

O que é programação orientada a objetos em Python?

POO é um paradigma de programação que estrutura o código de modo a atribuir propriedades e ações a objetos criados.

Um objeto que armazena as propriedades de um veículo, como sua cor, peso e ano de fabricação e também sobre suas ações, como frenagem e aceleração.

POO é uma abordagem para a modelização de objetos concretos, assim como suas relações com outros objetos.

O que é programação orientada a objetos em Python?

Faz isso ao modelar entidades reais como objetos de software, que possuem dados associados e que podem performar certas operações.

Objetos estão no centro da POO. Em outros paradigmas de programação, os objetos apenas representam os dados.

Na POO, eles informam adicionalmente a estrutura geral do programa.

Como definir uma classe em Python

Em Python definimos **classes** com a palavra chave **class**, seguida por um nome e dois pontos (**:**). Para declarar quais atributos cada **instância** da **classe** deverá ter, usamos *__init__()*.

```
class Classe:
        def __init__(self, propriedade1, propriedade2):
            self.propriedade1 = propriedade1
            self.propriedade2 = propriedade2
```

Como definir uma classe em Python

Quando um novo objeto de uma classe é criado, Python chama automaticamente o método ***__init__()*** para inicializar os atributos do objeto.

Com linhas sublinhadas duplas, o método ***__init__()*** indica que Python usará o método internamente, não é necessário que o usuário o chame.

Você pode usar o método ***__init__()*** para inicializar os atributos do objeto.

Como definir uma classe em Python

Números, **seqüências de caracteres** (strings) e **listas** são estruturas primitivas de dados, projetadas para representar informações simples.

```
Objeto1 = ['propriedade1_Objeto1','propriedade2_Objeto1', …]
Objeto2 = ['propriedade1_Objeto2','propriedade3_Objeto2', …]
```

Listas podem deixar o processamento lento, e causar problemas de definição/redefinição. E podem introduzir inconsistências quanto a número de elementos contidos.

As **classes** proporcionam códigos mais gerenciáveis e monitoráveis.

Classes e Instâncias

Classes permitem a criação de estruturas de dados definidas pelos usuários, elas definem funções chamadas de **métodos**, que identificam as ações que um objeto criado a partir da **classe** pode realizar com seus dados.

Uma **classe** é um modelo de como definir algo, ela não contém nenhum dado. Uma **instância** é um objeto construído a partir de uma **classe** e contém dados reais.

*Se uma **classe** é um formulário, uma **instância** é uma amostra preenchida deste formulário.*

Definição de uma classe

*Todas as classes são inicializadas com a palavra chave **class**, depois disso o nome da classe e dois pontos (:).*

```
class Carro:
    pass
```

Qualquer código indentado abaixo da classe definida será interpretado pelo Python como parte do corpo da **classe**.

O comando ***pass*** indica um espaço reservado.

Definição de uma classe

O **método** (**method**) ***.__init__()*** define as propriedades de cada objeto criado.

```
class Carro:

    def __init__(self, nome, ano_fabricacao):
        self.nome = nome
        self.ano_fabricacao = ano_fabricacao
```

Em cada criação de um novo objeto **Carro** esse método inicializa uma nova instância da classe.

Definição de uma classe

Definimos propriedades para o objeto carro a ser criado.

```
class Carro:

    def __init__(self, nome, ano_fabricacao):
        self.nome = nome
        self.ano_fabricacao = ano_fabricacao
```

No método ***__init__()*** definimos essas propriedades. Toda vez que um novo objeto **Carro** é criado, esse método define o estado inicial do objeto, atribuindo os valores de suas propriedades.

Definição de uma classe

O método ***__init__()*** inicializa as novas instâncias da classe. Ele aceita qualquer número de parâmetros, mas ***self*** deve ser o primeiro.

```
class Carro:

    def __init__(self, nome, ano_fabricacao):
        self.nome = nome
        self.ano_fabricacao = ano_fabricacao
```

Na criação de uma nova instância de classe, o Python passa automaticamente a instância para o parâmetro ***self*** em ***.__init__()*** para que o Python possa definir os novos **atributos** no objeto.

Definição de uma classe

A identação de ***.__init__()*** requer 1 tab/ 4 espaços, e o corpo do método 2 tabs/ 8 espaços.

1 - ***self.nome***: cria um parâmetro de nome e atribui o valor do parâmetro ***nome*** a ele.

2 - ***self.ano_fabricacao***: cria um parâmetro chamado ano de fabricação e atribui o valor do parâmetro ***ano_fabricacao*** a ele.

Definição de uma classe

Atributos criados com ***.__init__()*** são chamados de **atributos de instância** (**instance attributes**).

O valor de um **atributo de instância** é específico para uma uma instância particular da classe:

Todo objeto **carro** tem um ***nome*** e um ***ano_fabricacao***, mas os valores para os atributos ***nome*** e ***ano_fabricacao*** variam dependendo da instância carro.

Definição de uma classe

Por outro lado, **atributos de classe** (**class attributes**) são atributos que possuem o mesmo valor para todas as **instâncias de classe.**

```
class Carro:
    marca = "VW"

    def __init__(self, nome, ano_fabricacao):
        self.nome = nome
        self.ano_fabricacao = ano_fabricacao
```

É possível definir um atributo de classe atribuindo um valor à variável ***marca*** do lado de fora da rotina ***.__init__()***.

Definição de uma classe

Os **atributos de classe** são definidos diretamente abaixo da primeira linha do nome da **classe** e recuados com quatro espaços (1 tab). É necessário atribuí-los um valor inicial.

Quando uma **instância da classe** é criada, o Python cria e atribui automaticamente **atributos de classe** aos seus valores iniciais.

Atributos de classe definem propriedades que devem ter o mesmo valor para cada **instância de classe**.

Para propriedades que variam de uma instância para outra usa-se os **atributos de instância**.

A instanciação de uma classe

Instanciar uma classe é criar um novo objeto dessa classe. Cria-se um novo objeto ao se digitar o nome da classe, seguido pelo abrir e fechar de parênteses '()'.

```
class Carro:
    pass

Carro()
```

A nova instância da classe **Carro** está localizada em um endereço de memória diferente.

A instanciação de uma classe

Uma instância totalmente nova e completamente exclusiva do primeiro objeto Carro foi criada.

```
Carro_1 = Carro()
Carro_2 = Carro()

print('Carro_1 == Carro_2:', Carro_1 == Carro_2)
```

Acima duas instâncias da classe *Carro()* são criadas. A comparação (==) indica qual é a relação de igualdade entre os objetos, nesse caso, dois objetos distintos.

Atributos de Classe e Instância

Voltamos a definir a classe *Carro()*:

```
class Carro:
    marca = "VW"

    def __init__(self, nome, ano_fabricacao):
        self.nome = nome
        self.ano_fabricacao = ano_fabricacao
```

Ao instanciar um novo objeto dessa classe, é necessário providenciar nome e ano de fabricação.

Atributos de Classe e Instância

Os valores dos parâmetros são passados para a classe pelos parênteses:

```
Passat = Carro(nome = 'Passat', ano_fabricacao = 2005)
Golf = Carro('Golf', 2015)
```

Ao criamos uma instâncias da classe *Carro()*, ela é recebida pelo primeiro parâmetro do método *.__init__()* (*self*), que atribui os valores de nome e ano de fabricação.

Atributos de Classe e Instância

A notação com pontos (.) é utilizada para acessar os atributos de uma instância:

```
print('Golf.nome:', Golf.nome)
print('Golf.ano_fabricacao:', Golf.ano_fabricacao)
```

E para acessar os atributos de uma classe:

```
print('Golf.marca:', Golf.marca)
```

Atributos de Classe e Instância

Os valores desses atributos podem ser alterados dinamicamente:

```
Passat.ano_fabricacao = 2008
```

Um objeto é mutável se pode ser alterado dinamicamente, como listas e dicionários e diferente das *strings* e tuplas.

Objetos personalizados são mutáveis por padrão.

Métodos de Instâncias

Métodos de instâncias são funções definidas dentro de uma classe e que só podem chamar apenas uma instância dessa classe.

```
class Carro:
    marca = "VW"
    ano_atual = 2024

    def __init__(self, nome, ano_fabricacao):
        self.nome = nome
        self.ano_fabricacao = ano_fabricacao

    def idade(self):
        return f"{self.nome} tem {self.ano_atual - self.ano_fabricacao} anos"

    def ruido(self, som):
        return f"{self.nome} faz {som}"
```

Assim como *.__init__()*, um método de instância sempre toma *self* como seu primeiro parâmetro.

Métodos de Instâncias

A classe *Carros()* tem dois métodos:

.idade(): que retorna um inteiro, a diferença entre o ano atual e o ano de fabricação

.ruido(): com o parâmetro *som*, que retorna uma *string* com o ruído que o carro produz.

Métodos de Instâncias

Os métodos como *.__init__()* e *.__str__()* são chamados de métodos dunder por terem dois pares de sublinhados.

```
class Carro:
    marca = "VW"
    ano_atual = 2024

    def __init__(self, nome, ano_fabricacao):
        self.nome = nome
        self.ano_fabricacao = ano_fabricacao

    def __str__(self):
        return f"{self.nome} tem {self.ano_atual - self.ano_fabricacao} anos"

    def ruido(self, som):
        return f"{self.nome} faz {som}"
```

Há outros métodos dunder, usados para personalização.

Exercício proposto:

Vamos ao JN com os exercícios propostos.

Você pode também criar uma classe de seu gosto e conte para a turma no que pensou.

A herança de outra classe

Herança é o processo pelo qual uma classe assume os atributos e métodos de outra.

```
class Pai:
    cor_cabelo = 'castanho'

class Filho(Pai):
    pass
```

Novas classes são filhas, e as classes das quais derivam, são chamadas de classes pai. A herança implica na criação de uma nova classe com o nome da classe pai entre parênteses.

A herança de outra classe

As classes filhas podem substituir ou estender os atributos e métodos das classes pai.

```
class Pai:
    cor_cabelo = 'castanho'

class Filho(Pai):
    cor_cabelo = 'moreno'
```

Elas herdam todos os atributos e métodos dos pais, mas também podem especificar atributos e métodos exclusivos.

A herança de outra classe

É possível usar o atributo *append()* para estender um atributo da classe pai.

```
class Pai:
    lingua = ['Inglês']

class Filho(Pai):
    def __init__(self):
        super().__init__()
        self.lingua.append('Francês')
```

Você pode acessar a classe pai de dentro de um método de uma classe filha usando o método *super()*.

A herança de outra classe

De forma mais geral, todos os objetos criados a partir de uma classe filha são instâncias da classe pai, embora possam não ser instâncias de outras classes filhas.

Funcionalidades Extensivas da Classe Pai

Podemos definir valores específicos a atributos de diferentes objetos.

```
class Sedan(Carro):
    def ruido(self, som = 'Vroom Vroom'):
        return f"{self.nome} faz {som}"
```

Substituímos um método definido na classe pai, ao definirmos um método com o mesmo nome na classe filha.

```
Passat = Sedan(nome = 'Passat', ano_fabricacao = 2005)
print('Passat.ruido():', Passat.ruido())
```

Funcionalidades Extensivas da Classe Pai

alterações na classe pai são propagadas automaticamente para classes filhas.

Desde que o atributo ou método que está sendo alterado não seja substituído na classe filha.

Funcionalidades Extensivas da Classe Pai

Podemos acessar a classe pai de dentro de um método de uma classe filha usando *super()*:

```
class Sedan(Carro):
    def ruido(self, som = 'Vroom Vroom'):
        return super().ruido(som)
```

Ao ser chamarmos o método *super()* dentro de uma classe filha, Python procura na classe pai um método homônimo e o chama com o parâmetro definido na classe filha.

```
Passat = Sedan(nome = 'Passat', ano_fabricacao = 2005)
print('Passat.ruido():', Passat.ruido())
```

Funcionalidades Extensivas da Classe Pai

Uma classe filha pode ter uma única classe pai, ou várias.

O método *super()* pesquisa um método ou atributo na classe pai, percorrendo toda a hierarquia de classes procurando um método ou atributo correspondente.

Exercício proposto:

Vamos ao JN com os exercícios propostos.

Voc6e também pode criar uma classe pai e uma classe filho a seu gosto, contendo uma hierarquia de herança a seu gosto e conte para a turma no que pensou.

Polimorfismo em Python

Polimorfismo é a condição de ocorrência de uma entidade em diferentes formas, um conceito importante em programação.

```
num1 = 1
num2 = 2
print('num1 + num2:', num1 + num2)
```

Refere-se ao uso de uma entidade de tipo único (método, operador ou objeto) para representar diferentes tipos em diferentes cenários.

Polimorfismo em Python

Para tipos de dados inteiros, o operador (+) é usado para realizar operações de adição aritmética.

```
str1 = "Python"
str2 = "Programming"

print("str1 + ' ' + str2:", str1 + ' ' + str2)
```

Da mesma forma, para tipos de dados *string*, o operador (+) é usado para realizar a concatenação.

Polimorfismo de Função em Python

Há funções em Python que são compatíveis com múltiplos tipos de dados.

```
print(len("Programing"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Polimorfismo de Classe em Python

Ao criar métodos de classe, Python permite que diferentes classes tenham métodos com o mesmo nome.

```
class Gato:

    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def info(self):
        print(f"Eu sou um gato, meu nome é {self.nome}. Tenho {self.idade} anos de idade.")

    def make_sound(self):
        som = 'Miau'
        print(som)
```

Polimorfismo de Classe em Python

Pode-se generalizar a chamada de métodos, independentemente do objeto chamado.

```
class Rato:

    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def info(self):
        print(f"Eu sou um rato, meu nome é {self.nome}. Tenho {self.idade} anos de idade.")

    def make_sound(self):
        som = 'Som de rato'
        print(som)
```

Polimorfismo e Herança

Classes filhas em Python herdam métodos e atributos da classe pai. Pode-se re-definir certos métodos e atributos especificamente para se adequarem à classe filha.

```
class Forma:
    def __init__(self, name):
        self.name = name

    def area(self):
        pass

    def info(self):
        return print('tenho uma forma bi-dimensional.')

    def __str__(self):
        return self.name
```

Polimorfismo e Herança

Podemos acessar esses métodos e atributos substituídos que possuem o mesmo nome da classe pai.

```
class Quadrado(Forma):
    def __init__(self, length):
        super().__init__("Quadrado")
        self.length = length

    def area(self):
        return self.length**2

    def info(self):
        return print('Quadrados têm ângulos de 90 graus.')
```

Vemos que métodos como *__str__()*, que não foram substituídos nas classes filhas, são usados na classe pai.

Polimorfismo e Herança

O interpretador reconhece que o método *info()* foi sobre-escrito na classe *Quadrado()*, usando a definição da classe filha.

```
class Círculo(Forma):
    def __init__(self, radius):
        super().__init__("Círculo")
        self.radius = radius

    def area(self):
        return pi * self.radius ** 2
```

Por outro lado, como o método info() não é substituído na classe *Circulo()*, ele mantém o valor da classe Parente *Forma()*.

Métodos Especiais e Protocolos

Um método de forma *__nome__()*, chamado implicitamente pelo Python para executar uma determinada operação em um tipo, como adição. Esses métodos têm nomes começando e terminando com sublinhados duplos.

Python oferece suporte ao que chama de métodos especiais, normalmente métodos de instância, também conhecidos como métodos *dunder* ou mágicos. Python os chama automaticamente em resposta a operações específicas.

Métodos Especiais e Protocolos

Informal string representation *.__str__()*: retorna uma *string* que representa o objeto de maneira amigável.

Formal string representation *.__repr__()*: retorna uma *string* que permite recriar o objeto, se possível.

```
class Carro:

    def __init__(self, fabricante, modelo, cor, ano):
        self.fabricante = fabricante
        self.modelo = modelo
        self.cor = cor
        self.ano = ano

    def __str__(self):
        return f"{self.fabricante}, {self.modelo}, {self.cor}: ({self.ano})"

    def __repr__(self):
        return (
            f"{type(self).__name__}"
            f'(fabricante : "{self.fabricante}", '
            f'modelo : "{self.modelo}", '
            f"ano : {self.ano}, "
            f'cor : "{self.cor}")'
        )
```

Métodos Especiais e Protocolos

Os protocolos consistem em um ou mais métodos especiais que suportam um determinado recurso ou funcionalidade:

Protocolo	Recurso fornecido	Métodos Especiais
Iterator	Permite criar objetos iteradores	**.iter()** e **.next()**
Iterable	Torna seus objetos iteráveis	**.iter()**
Descriptor	Permite escrever atributos gerenciados	**.get()** e opcionalmente **.set()**, **.delete()**, e **.set_name()**
Context manager	Permite que um objeto trabalhe com instruções	**.enter()** e **.exit()**

Métodos Especiais e Protocolos

A classe abaixo recebe três argumentos que representam as coordenadas espaciais de um determinado ponto.

```
class Pontos_3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def __iter__(self):
        yield from (self.x, self.y, self.z)
```

Métodos Especiais e Protocolos

O método *.__iter__()* é uma função geradora que retorna um iterador. O iterador resultante produz as coordenadas de *Pontos_3D* sob demanda.

O comando yield (rendimento) controla o fluxo de uma função geradora de maneira semelhante às instruções de *return* (retorno).

Métodos de Classe

Um método de classe (*@classmethod*) é um método que toma o objeto da classe como seu primeiro argumento, em vez de tomar *self*.

```
class Pontos_3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def __iter__(self):
        yield from (self.x, self.y, self.z)
    @classmethod
    def from_sequence(cls, sequencia):
        return cls(*sequencia)

    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

Métodos de Classe

Nesse caso, a convenção vigente é chamar esse argumento de *cls*.

Métodos estáticos (@staticmethod) Funções regulares definidas dentro de uma classe. Podem ser definidas também fora da classe, como uma função independente.

Métodos Estáticos

Chamar uma função é um pouco diferente de chamar um método. Para chamar um método, você precisa especificar uma classe ou objeto que o forneça.

```
class Pontos_3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def __iter__(self):
        yield from (self.x, self.y, self.z)
    @classmethod
    def from_sequence(cls, sequencia):
        return cls(*sequencia)
    @staticmethod
    def show_intro_message(name):
        print(f"Olá {name}! Este é seu ponto em 3D!")
    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

Instância de um Método, Métodos Estáticos e Método de Classe

Uma **instância de método** carrega implicitamente o parâmetro *self* como seu primeiro argumento.

Um **método estático** não carrega nenhum parâmetro implícito como primeiro argumento. São incomuns e podem ser definidos fora da classe.

Um **método de classe** carrega implicitamente um parâmetro da classe como primeiro argumento. De uso comun, criam uma instância usando um conjunto diferente de argumentos. De nomes normalmente pre fixados (*from_*, *make_*, *create_*)

Módulos em Python

Códigos podem se tornar longos e complexos, os módulos servem para separar o código em arquivos separados, de acordo com sua funcionalidade.

Módulos oferecem organização e fácil manutenção aos códigos longos.

Módulos em Python

Um módulo é um arquivo que contém um código que executa uma tarefa específica. Ele pode conter variáveis, funções, classes, etc.

```
def add(a, b):

    result = a + b
    return result
Here, we have def
```

O módulo *aritmetica.py* contém as quatro operações fundamentais, entre dois números *a* e *b*.

Importando módulos em Python

Importamos as definições de um módulo para outro módulo, ou para o interpretador interativo do Python.

```
import aritmetica
```

O comando *import*, seguido pelo nome do módulo é utilizado para importar um módulo definido anteriormente.

Importando módulos em Python

É possível acessar uma função com o nome do módulo, seguido por um ponto (.)

```
aritmetica.add(4, 5)
```

Importando módulos em Python

Um módulo é simplesmente um arquivo que contém definições e declarações de funções, classes e assim por diante.

Chama-se "pacote regular" a um diretório contendo módulos e um arquivo *__init__.py* que, pode estar vazio, e é executado quando um pacote é importado.

Embora não sejam estritamente definidos, pacotes e módulos às vezes são chamados de bibliotecas.

Importando módulos em Python

O arquivo especial *__init__.py* é para definir pacotes e inicializar seus *namespaces*, estruturas usadas para organizar os nomes simbólicos atribuídos a objetos.

Pode conter um código de inicialização executado quando o pacote é importado.

Sem este arquivo, o Python não reconhecerá um diretório como um pacote.

Importando módulos em Python

__init__.py marca o diretório como um pacote Python para que o intérpretador possa encontrar seus módulos.

Pode conter código de inicialização do pacote, como importação de submódulos, definição de variáveis ou execução de outro código.

Embora não sejam estritamente definidos, pacotes e módulos às vezes são chamados de bibliotecas.

Importando módulos padrão em Python

A biblioteca padrão do Python contém mais de 200 módulos.

```
import math

print('math.pi:', math.pi)
```

Pode-se importar um módulo de acordo as necessidades do código. E também é possível renomeá-lo.

```
import math as m

print('m.pi:', m.pi)
```

Importando módulos padrão em Python

Pode-se importar atributos específicos de um módulo sem importar o módulo como um todo, usando o comando *from*.

```
from math import pi

print('pi:', pi)
```

O asterisco implica na importação de todos os atributos do módulo, com exceção dos privados, que iniciam com uma linha sublinhada.

```
from math import *

print("The value of pi is", pi)
```

Importando módulos padrão em Python

O uso do asterisco (*) não é considerado uma boa prática, pois pode levar a definições duplicadas para um identificador, além de dificultar a legibilidade do código.

O método dir() acessa uma lista de atributos válidos do objeto passado. Ele aceita tuplas, listas, conjuntos ou dicionários e retorna.

```
print('dir(aritmetica):', dir(aritmetica))
```

Exercício proposto:

Vamos ao JN com os exercícios propostos.

Crie um ou mais módulos para realizar alguma tarefa. Depois conte-nos sobre o que você fez e quais paradigmas de programação você aplicou e porquê.

Referências

- Python __init__
- __init__ in Python
- Object-Oriented Programming (OOP) in Python 3
- Variables in Python
- Python Classes: The Power of Object-Oriented Programming
- Single and Double Underscores in Python Names
- Inheritance and Composition: A Python OOP Guide
- Inheritance in Python
- Python's .append(): Add Items to Your Lists in Place
- Supercharge Your Classes With Python super()

Referências

- Python Modules
- 6. Modules
- How to import modules, packages, functions, etc. in Python
- What is __Init__.Py File in Python?
- Namespaces and Scope in Python
- Python dir()
- Special Method
- Python Classes: The Power of Object-Oriented Programming
- Every dunder method in Python
- Python's Magic Methods: Leverage Their Power in Your Classes
- When Should You Use .__repr__() vs .__str__() in Python?

Referências

- Python __iter__() and __next__() | Converting an object into an iterator
- Built-in Functions: @staticmethod

Vamos ao Notebook