Funções Especiais em Python

Este documento lista 27 funções especiais (métodos mágicos) em Python, com exemplos de código e a saída esperada. Use o índice abaixo para navegar rapidamente para cada função.

Índice

__init__

Inicializa uma nova instância de uma classe.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

obj = MinhaClasse(10)
print(obj.valor)
```

```
Saída:
10
```

Exemplos Reais do Método __init__

1. Classe Básica com Atributos Simples

```
class Carro:
    def __init__(self, marca, modelo, ano):
        self.marca = marca
        self.modelo = modelo
        self.ano = ano

    def exibir_informacoes(self):
        return f"{self.ano} {self.marca} {self.modelo}"

# Criando uma instância da classe Carro
carro1 = Carro("Toyota", "Corolla", 2020)
print(carro1.exibir_informacoes())  # Saída: 2020 Toyota Corolla
```

2. Classe com Atributos Padrão

```
class Pessoa:
    def __init__(self, nome, idade=30):
        self.nome = nome
        self.idade = idade

    def apresentar(self):
        return f"Olá, meu nome é {self.nome} e eu tenho {self.idade}
```

```
anos."

# Criando instâncias da classe Pessoa
pessoa1 = Pessoa("Ana")
pessoa2 = Pessoa("Carlos", 25)

print(pessoa1.apresentar())  # Saída: Olá, meu nome é Ana e eu tenho 30
anos.
print(pessoa2.apresentar())  # Saída: Olá, meu nome é Carlos e eu tenho 25
anos.
```

3. Classe com Atributos Calculados

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

    def area(self):
        return self.largura * self.altura

# Criando uma instância da classe Retangulo
retangulo = Retangulo(10, 5)
print(f"Área do retângulo: {retangulo.area()}")  # Saída: Área do
retângulo: 50
```

4. Classe com Inicialização Complexa

```
class ContaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.titular = titular
        self.saldo = saldo_inicial

    def depositar(self, valor):
        self.saldo += valor

    def sacar(self, valor):
        if valor <= self.saldo:
            self.saldo -= valor
```

```
        else:
            print("Saldo insuficiente!")

    def mostrar_saldo(self):
        return f"Saldo atual: R${self.saldo:.2f}"

# Criando uma instância da classe ContaBancaria
conta = ContaBancaria("João", 1000)
conta.depositar(500)
conta.sacar(200)
print(conta.mostrar_saldo())  # Saída: Saldo atual: R$1300.00
```

5. Classe com Dependências Externas

```
class Livro:
    def __init__(self, titulo, autor, ano_publicacao):
        self.titulo = titulo
        self.autor = autor
        self.ano_publicacao = ano_publicacao

    def idade_livro(self):
        from datetime import datetime
        ano_atual = datetime.now().year
        return ano_atual - self.ano_publicacao

# Criando uma instância da classe Livro
livro = Livro("1984", "George Orwell", 1949)
print(f"Idade do livro: {livro.idade_livro()} anos")  # Saída: Idade do
livro: 75 anos (considerando o ano atual como 2024)
```

__str__

Retorna uma string representativa de uma instância para o usuário final.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor
```

```
    def __str__(self):
        return f'MinhaClasse com valor {self.valor}'

obj = MinhaClasse(10)
print(obj)
```

```
 Saída:
MinhaClasse com valor 10
```

__repr__

Retorna uma string representativa de uma instância para desenvolvedores, geralmente incluindo informações que podem ser usadas para recriar o objeto.

```
 class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __repr__(self):
        return f'MinhaClasse({self.valor})'

obj = MinhaClasse(10)
print(repr(obj))
```

```
 Saída:
MinhaClasse(10)
```

__len__

Retorna o comprimento de um objeto.

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __len__(self):
        return len(self.itens)

obj = MinhaClasse([1, 2, 3, 4])
print(len(obj))
```

```
Saída:
4
```

__getitem__

Permite o acesso a elementos usando a notação de colchetes (indexação).

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __getitem__(self, index):
        return self.itens[index]

obj = MinhaClasse([1, 2, 3, 4])
print(obj[2])
```

```
Saída:
3
```

__setitem__

Permite a modificação de elementos usando a notação de colchetes (indexação).

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __setitem__(self, index, valor):
        self.itens[index] = valor

obj = MinhaClasse([1, 2, 3, 4])
obj[2] = 10
print(obj.itens)
```

```
Saída:
[1, 2, 10, 4]
```

__delitem__

Permite a exclusão de elementos usando a notação de colchetes (indexação).

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __delitem__(self, index):
        del self.itens[index]

obj = MinhaClasse([1, 2, 3, 4])
del obj[2]
print(obj.itens)
```

```
Saída:
[1, 2, 4]
```

__iter__

Retorna um iterador para o objeto.

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __iter__(self):
        return iter(self.itens)

obj = MinhaClasse([1, 2, 3])
for item in obj:
    print(item)
```

```
Saída:
1
2
3
```

__next__

Retorna o próximo item do iterador. Levanta uma exceção StopIteration quando não há mais itens.

```
class MinhaClasse:
    def __init__(self):
        self.n = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= 3:
            result = self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

```
obj = MinhaClasse()
for item in obj:
    print(item)
```

```
Saída:
1
2
3
```

__call__

Permite que uma instância de classe seja chamada como uma função.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __call__(self, valor):
        return self.valor + valor

obj = MinhaClasse(10)
print(obj(5))
```

```
Saída:
15
```

__contains__

Verifica se um item está contido em um objeto.

```
class MinhaClasse:
    def __init__(self, itens):
```

```
        self.itens = itens

    def __contains__(self, item):
        return item in self.itens

obj = MinhaClasse([1, 2, 3])
print(2 in obj)
```

```
Saída:
True
```

__add__

Define o comportamento do operador de adição (+).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __add__(self, outro):
        return MinhaClasse(self.valor + outro.valor)

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(20)
resultado = obj1 + obj2
print(resultado.valor)
```

```
Saída:
30
```

__sub__

Define o comportamento do operador de subtração (-).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __sub__(self, outro):
        return MinhaClasse(self.valor - outro.valor)

obj1 = MinhaClasse(20)
obj2 = MinhaClasse(10)
resultado = obj1 - obj2
print(resultado.valor)
```

```
Saída:
10
```

__mul__

Define o comportamento do operador de multiplicação (*).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __mul__(self, outro):
        return MinhaClasse(self.valor * outro.valor)

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(5)
resultado = obj1 * obj2
print(resultado.valor)
```

```
Saída:
50
```

__truediv__

Define o comportamento do operador de divisão (/).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __truediv__(self, outro):
        return MinhaClasse(self.valor / outro.valor)

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(2)
resultado = obj1 / obj2
print(resultado.valor)
```

```
Saída:
5.0
```

__floordiv__

Define o comportamento do operador de divisão inteira (//).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __floordiv__(self, outro):
        return MinhaClasse(self.valor // outro.valor)

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(3)
resultado = obj1 // obj2
print(resultado.valor)
```

```
Saída:
3
```

__mod__

Define o comportamento do operador módulo (%).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __mod__(self, outro):
        return MinhaClasse(self.valor % outro.valor)

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(3)
resultado = obj1 % obj2
print(resultado.valor)
```

```
Saída:
1
```

__pow__

Define o comportamento do operador de exponenciação (**).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __pow__(self, outro):
        return MinhaClasse(self.valor ** outro.valor)
```

```
obj1 = MinhaClasse(2)
obj2 = MinhaClasse(3)
resultado = obj1 ** obj2
print(resultado.valor)
```

```
Saída:
8
```

__eq__

Define o comportamento do operador de igualdade (==).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __eq__(self, outro):
        return self.valor == outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 == obj2)
```

```
Saída:
True
```

__ne__

Define o comportamento do operador de desigualdade (!=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor
```

```
    def __ne__(self, outro):
        return self.valor != outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(20)
print(obj1 != obj2)
```

```
Saída:
True
```

__lt__

Define o comportamento do operador menor que (<).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __lt__(self, outro):
        return self.valor < outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(20)
print(obj1 < obj2)
```

```
Saída:
True
```

__le__

Define o comportamento do operador menor ou igual (<=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __le__(self, outro):
        return self.valor <= outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 <= obj2)
```

```
Saída:
True
```

__gt__

Define o comportamento do operador maior que (>).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __gt__(self, outro):
        return self.valor > outro.valor

obj1 = MinhaClasse(20)
obj2 = MinhaClasse(10)
print(obj1 > obj2)
```

```
Saída:
True
```

__ge__

Define o comportamento do operador maior ou igual (>=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __ge__(self, outro):
        return self.valor >= outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 >= obj2)
```

```
Saída:
True
```

__radd__

Define o comportamento do operador de adição (+) quando o objeto é o segundo operando.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __radd__(self, outro):
        return MinhaClasse(outro + self.valor)

obj1 = MinhaClasse(10)
resultado = 20 + obj1
print(resultado.valor)
```

```
Saída:
30
```

__iadd__

Define o comportamento do operador de adição (+) para operações in-place.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __iadd__(self, outro):
        self.valor += outro.valor
        return self

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(5)
obj1 += obj2
print(obj1.valor)
```

```
Saída:
15
```

__rsub__

Define o comportamento do operador de subtração (-) quando o objeto é o segundo operando.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __rsub__(self, outro):
        return MinhaClasse(outro - self.valor)

obj1 = MinhaClasse(10)
resultado = 20 - obj1
print(resultado.valor)
```

```
Saída:
10
```

__rmul__

Define o comportamento do operador de multiplicação (*) quando o objeto é o segundo operando.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __rmul__(self, outro):
        return MinhaClasse(outro * self.valor)

obj1 = MinhaClasse(10)
resultado = 5 * obj1
print(resultado.valor)
```

```
Saída:
50
```

__rtruediv__

Define o comportamento do operador de divisão (/) quando o objeto é o segundo operando.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __rtruediv__(self, outro):
        return MinhaClasse(outro / self.valor)
```

```
obj1 = MinhaClasse(2)
resultado = 10 / obj1
print(resultado.valor)
```

```
Saída:
5.0
```

__rfloordiv__

Define o comportamento do operador de divisão inteira (//) quando o objeto é o segundo operando.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __rfloordiv__(self, outro):
        return MinhaClasse(outro // self.valor)

obj1 = MinhaClasse(3)
resultado = 10 // obj1
print(resultado.valor)
```

```
Saída:
3
```

__rmod__

Define o comportamento do operador módulo (%) quando o objeto é o segundo operando.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor
```

```
    def __rmod__(self, outro):
        return MinhaClasse(outro % self.valor)

obj1 = MinhaClasse(3)
resultado = 10 % obj1
print(resultado.valor)
```

```
Saída:
1
```

__rpow__

Define o comportamento do operador de exponenciação (**) quando o objeto é o segundo operando.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __rpow__(self, outro):
        return MinhaClasse(outro ** self.valor)

obj1 = MinhaClasse(3)
resultado = 2 ** obj1
print(resultado.valor)
```

```
Saída:
8
```

__getitem__

Define o comportamento de acesso a itens com a notação de colchetes ([]).

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __getitem__(self, index):
        return self.itens[index]

obj = MinhaClasse([1, 2, 3])
print(obj[1])
```

```
Saída:
2
```

__setitem__

Define o comportamento da atribuição de itens com a notação de colchetes ([]).

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __setitem__(self, index, valor):
        self.itens[index] = valor

obj = MinhaClasse([1, 2, 3])
obj[1] = 20
print(obj.itens)
```

```
Saída:
[1, 20, 3]
```

__delitem__

Define o comportamento da exclusão de itens com a notação de colchetes ([]).

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __delitem__(self, index):
        del self.itens[index]

obj = MinhaClasse([1, 2, 3])
del obj[1]
print(obj.itens)
```

```
Saída:
[1, 3]
```

__iter__

Define o comportamento do objeto iterável.

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __iter__(self):
        return iter(self.itens)

obj = MinhaClasse([1, 2, 3])
for item in obj:
    print(item)
```

```
Saída:
1
2
3
```

__next__

Define o comportamento do próximo item em um iterador.

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.itens):
            resultado = self.itens[self.index]
            self.index += 1
            return resultado
        else:
            raise StopIteration

obj = MinhaClasse([1, 2, 3])
for item in obj:
    print(item)
```

```
Saída:
1
2
3
```

__len__

Define o comportamento da função `len()` .

```
class MinhaClasse:
    def __init__(self, itens):
```

```
        self.itens = itens

    def __len__(self):
        return len(self.itens)

obj = MinhaClasse([1, 2, 3])
print(len(obj))
```

```
Saída:
3
```

__repr__

Define a representação do objeto para debugging.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __repr__(self):
        return f'MinhaClasse({self.valor!r})'

obj = MinhaClasse(10)
print(repr(obj))
```

```
Saída:
MinhaClasse(10)
```

__str__

Define a representação do objeto para usuários.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __str__(self):
        return f'Valor: {self.valor}'

obj = MinhaClasse(10)
print(str(obj))
```

```
Saída:
Valor: 10
```

__call__

Define o comportamento quando o objeto é chamado como uma função.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __call__(self, valor):
        return self.valor + valor

obj = MinhaClasse(10)
print(obj(5))
```

```
Saída:
15
```

__contains__

Define o comportamento do operador `in` .

```
class MinhaClasse:
    def __init__(self, itens):
        self.itens = itens

    def __contains__(self, item):
        return item in self.itens

obj = MinhaClasse([1, 2, 3])
print(2 in obj)
```

```
Saída:
True
```

__eq__

Define o comportamento do operador de igualdade (==).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __eq__(self, outro):
        return self.valor == outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 == obj2)
```

```
Saída:
True
```

__ne__

Define o comportamento do operador de desigualdade (!=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __ne__(self, outro):
        return self.valor != outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(20)
print(obj1 != obj2)
```

```
Saída:
True
```

__lt__

Define o comportamento do operador de menor que (<).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __lt__(self, outro):
        return self.valor < outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(20)
print(obj1 < obj2)
```

```
Saída:
True
```

__le__

Define o comportamento do operador de menor ou igual (<=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __le__(self, outro):
        return self.valor <= outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 <= obj2)
```

```
Saída:
True
```

__gt__

Define o comportamento do operador de maior que (>).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __gt__(self, outro):
        return self.valor > outro.valor

obj1 = MinhaClasse(20)
obj2 = MinhaClasse(10)
print(obj1 > obj2)
```

```
Saída:
True
```

__ge__

Define o comportamento do operador de maior ou igual (>=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __ge__(self, outro):
        return self.valor >= outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 >= obj2)
```

```
Saída:
True
```

__copy__

Define o comportamento da cópia rasa (shallow copy) do objeto.

```
import copy

class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __copy__(self):
        return MinhaClasse(self.valor)

obj1 = MinhaClasse(10)
obj2 = copy.copy(obj1)
print(obj2.valor)
```

```
Saída:
10
```

__deepcopy__

Define o comportamento da cópia profunda (deep copy) do objeto.

```
import copy

class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __deepcopy__(self, memo):
        return MinhaClasse(copy.deepcopy(self.valor, memo))

obj1 = MinhaClasse([1, 2, 3])
obj2 = copy.deepcopy(obj1)
print(obj2.valor)
```

```
Saída:
[1, 2, 3]
```

__hash__

Define o comportamento da função `hash()`.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __hash__(self):
        return hash(self.valor)
```

```
obj = MinhaClasse(10)
print(hash(obj))
```

```
Saída:
10
```

__eq__

Define o comportamento do operador de igualdade (==).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __eq__(self, outro):
        return self.valor == outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 == obj2)
```

```
Saída:
True
```

__ne__

Define o comportamento do operador de desigualdade (!=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor
```

```
    def __ne__(self, outro):
        return self.valor != outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(20)
print(obj1 != obj2)
```

```
Saída:
True
```

__lt__

Define o comportamento do operador de menor que (<).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __lt__(self, outro):
        return self.valor < outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(20)
print(obj1 < obj2)
```

```
Saída:
True
```

__le__

Define o comportamento do operador de menor ou igual (<=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __le__(self, outro):
        return self.valor <= outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 <= obj2)
```

```
Saída:
True
```

__gt__

Define o comportamento do operador de maior que (>).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __gt__(self, outro):
        return self.valor > outro.valor

obj1 = MinhaClasse(20)
obj2 = MinhaClasse(10)
print(obj1 > obj2)
```

```
Saída:
True
```

__ge__

Define o comportamento do operador de maior ou igual (>=).

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __ge__(self, outro):
        return self.valor >= outro.valor

obj1 = MinhaClasse(10)
obj2 = MinhaClasse(10)
print(obj1 >= obj2)
```

```
Saída:
True
```

__copy__

Define o comportamento da cópia rasa (shallow copy) do objeto.

```
import copy

class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __copy__(self):
        return MinhaClasse(self.valor)

obj1 = MinhaClasse(10)
obj2 = copy.copy(obj1)
print(obj2.valor)
```

```
Saída:
10
```

__deepcopy__

Define o comportamento da cópia profunda (deep copy) do objeto.

```
import copy

class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __deepcopy__(self, memo):
        return MinhaClasse(copy.deepcopy(self.valor, memo))

obj1 = MinhaClasse([1, 2, 3])
obj2 = copy.deepcopy(obj1)
print(obj2.valor)
```

```
Saída:
[1, 2, 3]
```

__hash__

Define o comportamento da função `hash()`.

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def __hash__(self):
        return hash(self.valor)

obj = MinhaClasse(10)
print(hash(obj))
```

```
Saída:
10
```

Porque importar **Módulos** e usar **Funções Especiais**

Importar Módulos

Reutilizar código de outros arquivos ou bibliotecas, como módulos de terceiros: Como módulos de terceiros (por exemplo, `math`, `numpy`) ou módulos que você mesmo criou.

Organização: Para organizar seu código em vários arquivos e mantê-lo modular e gerenciável. Você pode criar módulos e pacotes e importá-los conforme necessário.

Funcionalidade Adicional: Para acessar funções, classes e variáveis que não são parte da biblioteca padrão, como bibliotecas de terceiros (por exemplo, `requests`, `pandas`).

Exemplo:

```
import math

print(math.sqrt(16))  # Saída: 4.0
```

Funções Especiais

__init__

Inicializa um novo objeto: Usado ao criar uma nova instância de uma classe para inicializar o estado do objeto.

```
class Carro:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
```

__str__ e __repr__

Definem como o objeto será representado como string: `__str__` para uma representação amigável ao usuário e `__repr__` para uma representação que pode ser usada para recriar o objeto.

```
class Pessoa:
    def __str__(self):
        return f"{self.nome}, {self.idade} anos"

    def __repr__(self):
        return f"Pessoa(nome='{self.nome}', idade={self.idade})"
```

__len__

Define o comportamento da função `len()`: Usado para definir como o comprimento de um objeto é calculado.

```
class ListaCustomizada:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)
```

__getitem__, __setitem__, __delitem__

Definem o comportamento de acesso, modificação e exclusão de itens em uma coleção: Usado para acessar, alterar e remover itens em uma coleção, como listas ou dicionários.

```
class MeuDicionario:
    def __init__(self):
        self.dados = {}

    def __getitem__(self, chave):
        return self.dados[chave]

    def __setitem__(self, chave, valor):
        self.dados[chave] = valor

    def __delitem__(self, chave):
        del self.dados[chave]
```

__iter__ e __next__

Tornam a classe iterável: `__iter__` retorna o iterador e `__next__` fornece o próximo item na iteração.

```
class Contador:
    def __init__(self, limite):
        self.limite = limite
        self.contador = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.contador >= self.limite:
            raise StopIteration
        self.contador += 1
        return self.contador - 1
```

__call__

Permite que uma instância de classe seja chamada como uma função: Usado para definir o comportamento de chamadas de instância de classe.

```
class Multiplicador:
    def __init__(self, fator):
        self.fator = fator

    def __call__(self, x):
        return x * self.fator
```

Métodos de Comparação (__eq__, __ne__, __lt__, __le__, __gt__, __ge__)

Definem o comportamento dos operadores de comparação: Usado para definir como os operadores de comparação (==, !=, <, <=, >, >=) funcionam com instâncias da classe.

```
class Ponto:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __eq__(self, outro):
        return self.x == outro.x and self.y == outro.y
```

Métodos Aritméticos (__add__, __sub__, __mul__, etc.)

Definem o comportamento dos operadores aritméticos: Usado para definir como os operadores aritméticos (+, -, *, etc.) funcionam com instâncias da classe.

```
class Vetor:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, outro):
        return Vetor(self.x + outro.x, self.y + outro.y)
```