Tópicos de programação em Python
Curso Avançado

Tópico 2 - Herança de classe

Prof. Louis Augusto
louis.augusto@ifsc.edu.br

Instituto Federal de Santa Catarina
Campus São José

Índice

Sumário

Caso mais simples de herança

Usamos herança quando queremos expandir as funcionalidades de uma classe. Avisamos ao interpretador que queremos usar uma classe que herda as propriedades da anterior e que pode ser incrementada com mais métodos e mais atributos.

```
#coding: utf-8
class Camera():
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
A = Camera("Sony", "24mp")
A.tirar_foto()
#Herança simples:
class CameraCelular(Camera):
    pass
B = CameraCelular("Olympus", "36mp")
B.tirar_foto()
```

Observe que as instâncias `A` e `B` fazem a mesma coisa. No caso `B` acessa a todas as funcionalidades de `A`. A vantagem é que podemos colocar mais atributos e funcionalidades na classe `CameraCelular` que havia em `Camera`.

Caso mais simples de herança

Usamos herança quando queremos expandir as funcionalidades de uma classe. Avisamos ao interpretador que queremos usar uma classe que herda as propriedades da anterior e que pode ser incrementada com mais métodos e mais atributos.

```
#coding: utf-8
class Camera():
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
A = Camera("Sony", "24mp")
A.tirar_foto()
#Herança simples:
class CameraCelular(Camera):
    pass
B = CameraCelular("Olympus", "36mp")
B.tirar_foto()
```

Observe que as instâncias `A` e `B` fazem a mesma coisa. No caso `B` acessa a todas as funcionalidades de `A`. A vantagem é que podemos colocar mais atributos e funcionalidades na classe `CameraCelular` que havia em `Camera`.

Caso mais simples de herança

Usamos herança quando queremos expandir as funcionalidades de uma classe. Avisamos ao interpretador que queremos usar uma classe que herda as propriedades da anterior e que pode ser incrementada com mais métodos e mais atributos.

```
#coding: utf-8
class Camera():
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
A = Camera("Sony", "24mp")
A.tirar_foto()
#Herança simples:
class CameraCelular(Camera):
    pass
B = CameraCelular("Olympus", "36mp")
B.tirar_foto()
```

Observe que as instâncias `A` e `B` fazem a mesma coisa. No caso `B` acessa a todas as funcionalidades de `A`. A vantagem é que podemos colocar mais atributos e funcionalidades na classe `CameraCelular` que havia em `Camera`.

Caso mais simples de herança

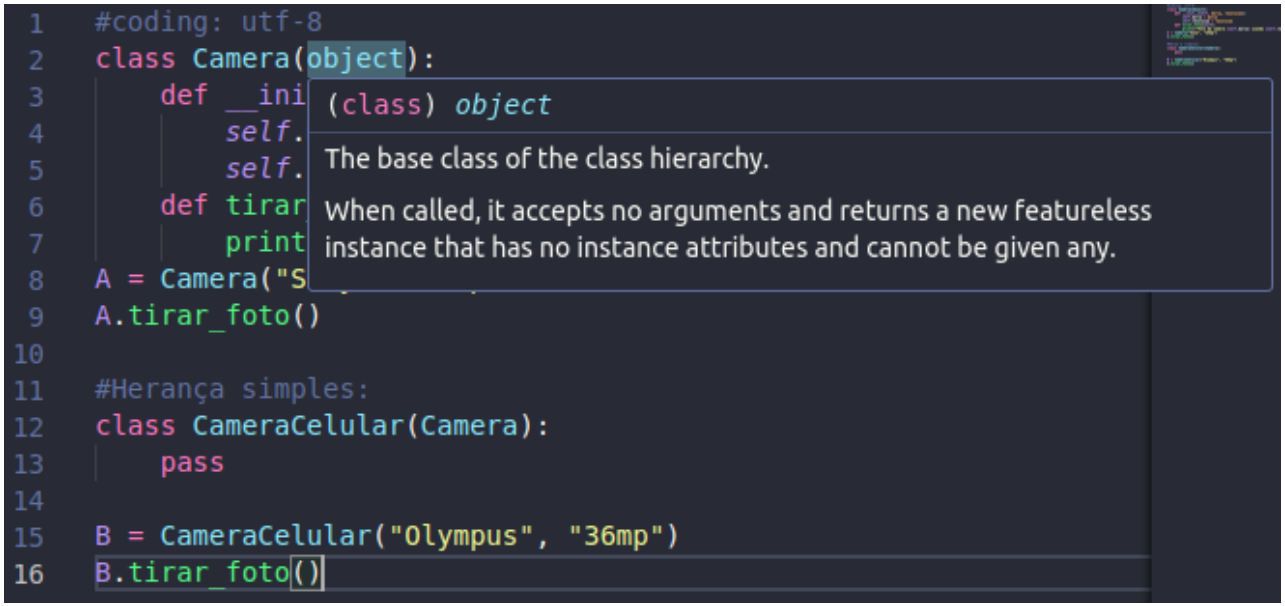
Usamos herança quando queremos expandir as funcionalidades de uma classe. Avisamos ao interpretador que queremos usar uma classe que herda as propriedades da anterior e que pode ser incrementada com mais métodos e mais atributos.

```
#coding: utf-8
class Camera():
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
A = Camera("Sony", "24mp")
A.tirar_foto()
#Herança simples:
class CameraCelular(Camera):
    pass
B = CameraCelular("Olympus", "36mp")
B.tirar_foto()
```

Observe que as instâncias `A` e `B` fazem a mesma coisa. No caso `B` acessa a todas as funcionalidades de `A`. A vantagem é que podemos colocar mais atributos e funcionalidades na classe `CameraCelular` que havia em `Camera`.

class object

Toda classe definida em python recebe uma classe original chamada `object`, que pode ser acessada no *vs code* pressionado `F12` quando o mouse estiver sobre a string:



Pressionando `F12` obtemos acesso a todos as funcionalidades da *class object*. E são muitas.

Herança

Podemos modificar o método `__init__` recebendo mais atributos. Suponha que um celular tenha mais de uma câmera e que queiramos selecionar uma:

```
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
```

Aqui temos uma classe pai (`Camera`) e uma classe filha (`CameraCelular`). Quando inicializamos as variáveis recebidas pela classe filha podemos repetir o que foi feito na classe pai para as variáveis de entrada da classe pai ou usar o identificador `super()`. Sem o identificador `super()` seria necessário repetir o recebimento dos atributos feitos na classe pai:

```
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        self.marca = marca
        self.resolucao = resolucao
        self.qde_cameras = qde_cameras
```

com o mesmo resultado.

Herança

Podemos modificar o método `__init__` recebendo mais atributos. Suponha que um celular tenha mais de uma câmera e que queiramos selecionar uma:

```
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
```

Aqui temos uma classe pai (`Camera`) e uma classe filha (`CameraCelular`). Quando inicializamos as variáveis recebidas pela classe filha podemos repetir o que foi feito na classe pai para as variáveis de entrada da classe pai ou usar o identificador `super()`. Sem o identificador `super()` seria necessário repetir o recebimento dos atributos feitos na classe pai:

```
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        self.marca = marca
        self.resolucao = resolucao
        self.qde_cameras = qde_cameras
```

com o mesmo resultado.

Herança

Podemos modificar o método `__init__` recebendo mais atributos. Suponha que um celular tenha mais de uma câmera e que queiramos selecionar uma:

```
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
```

Aqui temos uma classe pai (`Camera`) e uma classe filha (`CameraCelular`). Quando inicializamos as variáveis recebidas pela classe filha podemos repetir o que foi feito na classe pai para as variáveis de entrada da classe pai ou usar o identificador `super()`. Sem o identificador `super()` seria necessário repetir o recebimento dos atributos feitos na classe pai:

```
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        self.marca = marca
        self.resolucao = resolucao
        self.qde_cameras = qde_cameras
```

com o mesmo resultado.

Herança

Podemos modificar o método `__init__` recebendo mais atributos. Suponha que um celular tenha mais de uma câmera e que queiramos selecionar uma:

```
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
```

Aqui temos uma classe pai (`Camera`) e uma classe filha (`CameraCelular`). Quando inicializamos as variáveis recebidas pela classe filha podemos repetir o que foi feito na classe pai para as variáveis de entrada da classe pai ou usar o identificador `super()`. Sem o identificador `super()` seria necessário repetir o recebimento dos atributos feitos na classe pai:

```
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        self.marca = marca
        self.resolucao = resolucao
        self.qde_cameras = qde_cameras
```

com o mesmo resultado.

Métodos em classes com herança

Devemos ter em mente que qualquer método da classe pai pode ser usada na classe filho.

```
#coding: utf-8
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
    def Aplicar_filtro(self, filtro):
        print(f'Aplicando o filtro {filtro}')
CameraCel = CameraCelular('Sony', '25mp', 4)    #instancia a classe
CameraCel.Aplicar_filtro("Azul")    #Usa função da classe filho
CameraCel.tirar_foto()  #Usa função da classe pai
```

Vale salientar que há uma função em python que verifica se a herança de uma classe foi feita corretamente: `issubclass(filho, pai)`, que retorna `True` ou `False`. Para projetos pequenos isto é irrelevante, mas projetos grandes com módulos e submódulos poupa muito tempo.

Métodos em classes com herança

Devemos ter em mente que qualquer método da classe pai pode ser usada na classe filho.

```
#coding: utf-8
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
    def Aplicar_filtro(self, filtro):
        print(f'Aplicando o filtro {filtro}')
CameraCel = CameraCelular('Sony', '25mp', 4)    #instancia a classe
CameraCel.Aplicar_filtro("Azul")    #Usa função da classe filho
CameraCel.tirar_foto()  #Usa função da classe pai
```

Vale salientar que há uma função em python que verifica se a herança de uma classe foi feita corretamente: `issubclass(filho, pai)`, que retorna `True` ou `False`. Para projetos pequenos isto é irrelevante, mas projetos grandes com módulos e submódulos poupa muito tempo.

Métodos em classes com herança

Devemos ter em mente que qualquer método da classe pai pode ser usada na classe filho.

```
#coding: utf-8
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
    def Aplicar_filtro(self, filtro):
        print(f'Aplicando o filtro {filtro}')
CameraCel = CameraCelular('Sony', '25mp', 4)    #instancia a classe
CameraCel.Aplicar_filtro("Azul")    #Usa função da classe filho
CameraCel.tirar_foto()  #Usa função da classe pai
```

Vale salientar que há uma função em python que verifica se a herança de uma classe foi feita corretamente: `issubclass(filho, pai)`, que retorna `True` ou `False`. Para projetos pequenos isto é irrelevante, mas projetos grandes com módulos e submódulos poupa muito tempo.

Métodos em classes com herança

Devemos ter em mente que qualquer método da classe pai pode ser usada na classe filho.

```
#coding: utf-8
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")
class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
    def Aplicar_filtro(self, filtro):
        print(f'Aplicando o filtro {filtro}')
CameraCel = CameraCelular('Sony', '25mp', 4)    #instancia a classe
CameraCel.Aplicar_filtro("Azul")    #Usa função da classe filho
CameraCel.tirar_foto()  #Usa função da classe pai
```

Vale salientar que há uma função em python que verifica se a herança de uma classe foi feita corretamente: `issubclass(filho, pai)`, que retorna `True` ou `False`. Para projetos pequenos isto é irrelevante, mas projetos grandes com módulos e submódulos poupa muito tempo.

Sumário

Sobrescrevendo método da classe pai

Há também a possibilidade de sobrescrever método da classe pai, reescrevendo-a com o mesmo nome. Deve-se ter em mente que em cada classe o método com mesmo nome terá funcionalidades diferentes.

```
class Camera(object):
    def __init__(self, marca, resolucao):
        self.marca = marca
        self.resolucao = resolucao
    def tirar_foto(self):
        print(f"Foto da camera {self.marca} usando {self.resolucao} megapixels")

class CameraCelular(Camera):
    def __init__(self, marca, resolucao, qde_cameras):
        super().__init__(marca, resolucao)
        self.qde_cameras = qde_cameras
    def tirar_foto(self, num_cameras):
        print(f"Marca {self.marca} com {self.megapixels} MP e {num} câmeras
         de {self.QdeCameras}.")
```

Observe os instanciamentos:

```
A = Camera("Canon", 64)
B = CameraCelular("Sony", 32, 3)
C = Camera("Olympus", 128)
A.tirar_foto()
B.tirar_foto(2)
C.tirar_foto()
```

E as saídas:

```
Foto camera Canon com 64 MP
Marca Sony com 32 MP e 2 câmeras de 3.
Foto camera Olympus com 128 MP
```

Sumário

Evitando códigos espagueti

Um código espagueti é do tipo que quando se puxa uma informação, esta está relacionada a outra, que pode estar relacionada a outra, e mexer em uma pode arruinar completamente a construção.

Veja um exemplo:

```
class Veiculo():
    def __init__(self, marca, potencia):
        self.marca = marca
        self.potencia= potencia
class VeiculoRodoviario(Veiculo):
    def __init__(self, marca, potencia, NumPortas):
        super().__init__(marca, potencia)
        #Veiculo.__init__(self, marca, potencia) #Equivalente à linha de cima.
        self.NumPortas = NumPortas
class VeiculoMaritimo(Veiculo):
    def __init__(self, marca, potencia, tipo):
        super().__init__(marca, potencia)
        self.tipo = tipo #Vela ou motor
class VeiculoEletrico(VeiculoRodoviario):
    def __init__(self, marca, potencia, NumPortas, AutonomiaBateria):
        super().__init__(marca, potencia, NumPortas)
        self.AutonomiaBateria = AutonomiaBateria

Carro1 = VeiculoEletrico("BYD", "140", "4", "400")
print("Potencia = ", Carro1.potencia)
print("Qde portas: ", Carro1.NumPortas)
```

Evitando códigos espagueti

Um código espagueti é do tipo que quando se puxa uma informação, esta está relacionada a outra, que pode estar relacionada a outra, e mexer em uma pode arruinar completamente a construção.

Veja um exemplo:

```
class Veiculo():
    def __init__(self, marca, potencia):
        self.marca = marca
        self.potencia= potencia
class VeiculoRodoviario(Veiculo):
    def __init__(self, marca, potencia, NumPortas):
        super().__init__(marca, potencia)
        #Veiculo.__init__(self, marca, potencia) #Equivalente à linha de cima.
        self.NumPortas = NumPortas
class VeiculoMaritimo(Veiculo):
    def __init__(self, marca, potencia, tipo):
        super().__init__(marca, potencia)
        self.tipo = tipo #Vela ou motor
class VeiculoEletrico(VeiculoRodoviario):
    def __init__(self, marca, potencia, NumPortas, AutonomiaBateria):
        super().__init__(marca, potencia, NumPortas)
        self.AutonomiaBateria = AutonomiaBateria

Carro1 = VeiculoEletrico("BYD", "140", "4", "400")
print("Potencia = ", Carro1.potencia)
print("Qde portas: ", Carro1.NumPortas)
```

Evitando códigos espagueti

Um código espagueti é do tipo que quando se puxa uma informação, esta está relacionada a outra, que pode estar relacionada a outra, e mexer em uma pode arruinar completamente a construção.

Veja um exemplo:

```
class Veiculo():
    def __init__(self, marca, potencia):
        self.marca = marca
        self.potencia= potencia
class VeiculoRodoviario(Veiculo):
    def __init__(self, marca, potencia, NumPortas):
        super().__init__(marca, potencia)
        #Veiculo.__init__(self, marca, potencia) #Equivalente à linha de cima.
        self.NumPortas = NumPortas
class VeiculoMaritimo(Veiculo):
    def __init__(self, marca, potencia, tipo):
        super().__init__(marca, potencia)
        self.tipo = tipo #Vela ou motor
class VeiculoEletrico(VeiculoRodoviario):
    def __init__(self, marca, potencia, NumPortas, AutonomiaBateria):
        super().__init__(marca, potencia, NumPortas)
        self.AutonomiaBateria = AutonomiaBateria

Carro1 = VeiculoEletrico("BYD", "140", "4", "400")
print("Potencia = ", Carro1.potencia)
print("Qde portas: ", Carro1.NumPortas)
```

Evitando códigos espagueti

O código anterior funciona. A princípio não há muito problema:

Herança simples: As classes `VeiculoRodoviario` e `VeiculoMaritimo` herdam a classe `Veiculo`.

Herança multinível: a classe `VeiculoEletrico` herda `VeiculoRodoviario` que herda `Veiculo`.

Principalmente em projetos grandes isto deve ser evitado porque qualquer mudança em classe anterior afeta o funcionamento da classe posterior, e acaba por criar erros difíceis de serem encontrados e corrigidos.

Não se deve, de qualquer forma, adicionar mais de dois níveis de herança em um código, apesar de ser possível.

É recomendado organizar um código com listas de classes por exemplo, ou uma classe que alguns de seus atributos sejam outras classes.

Evitando códigos espagueti

O código anterior funciona. A princípio não há muito problema:

Herança simples: As classes `VeiculoRodoviario` e `VeiculoMaritimo` herdam a classe `Veiculo`.

Herança multinível: a classe `VeiculoEletrico` herda `VeiculoRodoviario` que herda `Veiculo`.

Principalmente em projetos grandes isto deve ser evitado porque qualquer mudança em classe anterior afeta o funcionamento da classe posterior, e acaba por criar erros difíceis de serem encontrados e corrigidos.

Não se deve, de qualquer forma, adicionar mais de dois níveis de herança em um código, apesar de ser possível.

É recomendado organizar um código com listas de classes por exemplo, ou uma classe que alguns de seus atributos sejam outras classes.

Evitando códigos espagueti

O código anterior funciona. A princípio não há muito problema:

Herança simples: As classes `VeiculoRodoviario` e `VeiculoMaritimo` herdam a classe `Veiculo`.

Herança multinível: a classe `VeiculoEletrico` herda `VeiculoRodoviario` que herda `Veiculo`.

Principalmente em projetos grandes isto deve ser evitado porque qualquer mudança em classe anterior afeta o funcionamento da classe posterior, e acaba por criar erros difíceis de serem encontrados e corrigidos.

Não se deve, de qualquer forma, adicionar mais de dois níveis de herança em um código, apesar de ser possível.

É recomendado organizar um código com listas de classes por exemplo, ou uma classe que alguns de seus atributos sejam outras classes.

Evitando códigos espagueti

O código anterior funciona. A princípio não há muito problema:

Herança simples: As classes `VeiculoRodoviario` e `VeiculoMaritimo` herdam a classe `Veiculo`.

Herança multinível: a classe `VeiculoEletrico` herda `VeiculoRodoviario` que herda `Veiculo`.

Principalmente em projetos grandes isto deve ser evitado porque qualquer mudança em classe anterior afeta o funcionamento da classe posterior, e acaba por criar erros difíceis de serem encontrados e corrigidos.

Não se deve, de qualquer forma, adicionar mais de dois níveis de herança em um código, apesar de ser possível.

É recomendado organizar um código com listas de classes por exemplo, ou uma classe que alguns de seus atributos sejam outras classes.

Evitando códigos espagueti

O código anterior funciona. A princípio não há muito problema:

Herança simples: As classes `VeiculoRodoviario` e `VeiculoMaritimo` herdam a classe `Veiculo`.

Herança multinível: a classe `VeiculoEletrico` herda `VeiculoRodoviario` que herda `Veiculo`.

Principalmente em projetos grandes isto deve ser evitado porque qualquer mudança em classe anterior afeta o funcionamento da classe posterior, e acaba por criar erros difíceis de serem encontrados e corrigidos.

Não se deve, de qualquer forma, adicionar mais de dois níveis de herança em um código, apesar de ser possível.

É recomendado organizar um código com listas de classes por exemplo, ou uma classe que alguns de seus atributos sejam outras classes.

Evitando códigos espagueti

O código anterior funciona. A princípio não há muito problema:

Herança simples: As classes `VeiculoRodoviario` e `VeiculoMaritimo` herdam a classe `Veiculo`.

Herança multinível: a classe `VeiculoEletrico` herda `VeiculoRodoviario` que herda `Veiculo`.

Principalmente em projetos grandes isto deve ser evitado porque qualquer mudança em classe anterior afeta o funcionamento da classe posterior, e acaba por criar erros difíceis de serem encontrados e corrigidos.

Não se deve, de qualquer forma, adicionar mais de dois níveis de herança em um código, apesar de ser possível.

É recomendado organizar um código com listas de classes por exemplo, ou uma classe que alguns de seus atributos sejam outras classes.

Evitando códigos espagueti

O código anterior funciona. A princípio não há muito problema:

Herança simples: As classes `VeiculoRodoviario` e `VeiculoMaritimo` herdam a classe `Veiculo`.

Herança multinível: a classe `VeiculoEletrico` herda `VeiculoRodoviario` que herda `Veiculo`.

Principalmente em projetos grandes isto deve ser evitado porque qualquer mudança em classe anterior afeta o funcionamento da classe posterior, e acaba por criar erros difíceis de serem encontrados e corrigidos.

Não se deve, de qualquer forma, adicionar mais de dois níveis de herança em um código, apesar de ser possível.

É recomendado organizar um código com listas de classes por exemplo, ou uma classe que alguns de seus atributos sejam outras classes.

Sumário

Quando uma classe tem mais de um pai

Uma herança múltipla é aquela que herda propriedades de mais de uma classe. Vamos a um exemplo:

```
class Pessoa():
    def __init__(self, nome):
        self.nome = nome
    def convidar(self):
        print(f"Pessoa {self.nome} foi convidada")
class Colaborador():
    def __init__(self, profissao):
        self.profissao = profissao
    def convidar(self):
        print(f"Class Colaborador, Profissional {self.profissao} foi convidado")
class Atleta():
    def __init__(self, esporte):
        self.esporte = esporte
    def convidar(self):
        print(f"Class Atleta, Atleta de {self.esporte} foi convidado")
class Patrocinado(Pessoa, Colaborador, Atleta):
    def __init__(self, nome, profissao, esporte):
        #Quando há mais de um usa-se o nome da classe
        Pessoa.__init__(self, nome)
        Colaborador.__init__(self, profissao)
        Atleta.__init__(self, esporte)
    def convidar(self):
        print(f"Atleta {self.nome}, {self.profissao}, {self.esporte} convidado")
Pessoa1 = Patrocinado("Augusto", "professor", "futebol")
Pessoa1.convidar()
```

Quando uma classe tem mais de um pai

Compreender este código é importante.

- Observe que não há chamada a `super()`, este somente pode ser usado se a herança for simples.
- Observe que cada uma das classes (Pessoa, Colaborador, Atleta e Patrocinado) possui um método chamado `convidar()`, porém somente o da classe `Patrocinado` foi usado. Isto ocorre quando há o mesmo nome de atributo ou método em classes diferentes.

Para saber qual a ordem de precedência fazemos uma chamada à ***MRO*** *(Method Resolution Order)* da classe. O resultado de MRO é o que vai orientar a ordem de prioridade das classes.

Inserindo no código `print(Patrocinado.mro())`, ou seja, `print(Nome_da_classe.mro())`, imprime a sequência de classes que a classe `Patrocinado` utiliza, com prioridade decrescente.

Quando uma classe tem mais de um pai

Compreender este código é importante.

- Observe que não há chamada a `super()`, este somente pode ser usado se a herança for simples.
- Observe que cada uma das classes (Pessoa, Colaborador, Atleta e Patrocinado) possui um método chamado `convidar()`, porém somente o da classe `Patrocinado` foi usado. Isto ocorre quando há o mesmo nome de atributo ou método em classes diferentes.

Para saber qual a ordem de precedência fazemos uma chamada à ***MRO*** *(Method Resolution Order)* da classe. O resultado de MRO é o que vai orientar a ordem de prioridade das classes.

Inserindo no código `print(Patrocinado.mro())`, ou seja, `print(Nome_da_classe.mro())`, imprime a sequência de classes que a classe `Patrocinado` utiliza, com prioridade decrescente.

Quando uma classe tem mais de um pai

Compreender este código é importante.

- Observe que não há chamada a `super()`, este somente pode ser usado se a herança for simples.
- Observe que cada uma das classes (Pessoa, Colaborador, Atleta e Patrocinado) possui um método chamado `convidar()`, porém somente o da classe `Patrocinado` foi usado. Isto ocorre quando há o mesmo nome de atributo ou método em classes diferentes.

Para saber qual a ordem de precedência fazemos uma chamada à ***MRO*** *(Method Resolution Order)* da classe. O resultado de MRO é o que vai orientar a ordem de prioridade das classes.

Inserindo no código `print(Patrocinado.mro())`, ou seja, `print(Nome_da_classe.mro())`, imprime a sequência de classes que a classe `Patrocinado` utiliza, com prioridade decrescente.

Quando uma classe tem mais de um pai

Compreender este código é importante.

- Observe que não há chamada a `super()`, este somente pode ser usado se a herança for simples.
- Observe que cada uma das classes (Pessoa, Colaborador, Atleta e Patrocinado) possui um método chamado `convidar()`, porém somente o da classe `Patrocinado` foi usado. Isto ocorre quando há o mesmo nome de atributo ou método em classes diferentes.

Para saber qual a ordem de precedência fazemos uma chamada à ***MRO*** *(Method Resolution Order)* da classe. O resultado de MRO é o que vai orientar a ordem de prioridade das classes.

Inserindo no código `print(Patrocinado.mro())`, ou seja, `print(Nome_da_classe.mro())`, imprime a sequência de classes que a classe `Patrocinado` utiliza, com prioridade decrescente.

Quando uma classe tem mais de um pai

Compreender este código é importante.

- Observe que não há chamada a `super()`, este somente pode ser usado se a herança for simples.
- Observe que cada uma das classes (Pessoa, Colaborador, Atleta e Patrocinado) possui um método chamado `convidar()`, porém somente o da classe `Patrocinado` foi usado. Isto ocorre quando há o mesmo nome de atributo ou método em classes diferentes.

Para saber qual a ordem de precedência fazemos uma chamada à ***MRO*** *(Method Resolution Order)* da classe. O resultado de MRO é o que vai orientar a ordem de prioridade das classes.

Inserindo no código `print(Patrocinado.mro())`, ou seja, `print(Nome_da_classe.mro())`, imprime a sequência de classes que a classe `Patrocinado` utiliza, com prioridade decrescente.