07

Mãos na massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Nesta aula, teste um pouco uma outra forma de herança, onde você pode reutilizar comportamentos de mais de uma classe mãe. Para isso, comece com este código:

```
class Funcionario:
    def registra_horas(self, horas):
        print('Horas registradas.')

    def mostrar_tarefas(self):
        print('Fez muita coisa...')

class Caelum(Funcionario):
    def mostrar_tarefas(self):
        print('Fez muita coisa, Caelumer')

    def busca_cursos_do_mes(self, mes=None):
        print(f'Mostrando cursos - {mes}' if mes else 'Mostrando cursos desse mês')

class Alura(Funcionario):
    def mostrar_tarefas(self):
        print('Fez muita coisa, Alurete!')

    def busca_perguntas_sem_resposta(self):
        print('Mostrando perguntas não respondidas do fórum')
```

2) Agora, crie as classes que herdam os comportamentos destas classes já criadas. Crie `Junior`, que herda de `Alura`, e `Pleno`, que herda de `Alura e Caelum`, pois trabalha nas duas frentes:

```
class Junior(Alura):
    pass

class Pleno(Alura, Caelum):
    pass
```

3) Para testar estas classes, adicione o código abaixo e execute:

```
jose = Junior()
jose.busca_perguntas_sem_resposta()

luan = Pleno()
luan.busca_perguntas_sem_resposta()
luan.busca_cursos_do_mes()
```

Com este código, dá para ver que foi possível herdar comportamento das duas classes no funcionário `Pleno`.

4) Agora, adicione o seguinte código na última linha:

```
luan.mostrar_tarefas()
```

Executando, você pode entender qual método `mostrar_tarefas` está sendo chamado.

5) Para visualizar de forma diferente, comente o método `mostrar_tarefas` da classe `Alura` :

```
class Alura(Funcionario):
    # def mostrar_tarefas(self):
    #     print('Fez muita coisa, Alurete!')
```

Execute novamente e veja como a ordem muda.

6) Outro uso comum de herança múltipla é quando você tem aspectos não-funcionais usados por múltiplas classes.

Para compartilhar estes comportamentos, você pode usar *Mixins*, que são apenas classes que podem ser herdadas mas que não devem ser instanciadas, já que servem para disponibilizar comportamentos para outras classes.

Crie a classe `Hipster` , abaixo da classe `Alura` :

```
class Hipster:
    def __str__(self):
        return f'Hipster,  {self.nome}'
```

7) Adicione um método `init` na classe `Funcionario` , para ter um nome no funcionário:

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

    def registra_horas(self, horas):
        print('Horas registradas.')

    def mostrar_tarefas(self):
        print('Fez muita coisa...')
```

8) E em seguida, modifique a classe `Pleno` para herdar também de `Hipster` , no fim da lista, e execute para ver o que acontece ao imprimir o objeto da classe `Pleno` :

```
class Pleno(Alura, Caelum, Hipster):
    pass

# restante do código ...


jose = Junior('José')
jose.busca_perguntas_sem_resposta()

luan = Pleno('Luan')
luan.busca_perguntas_sem_resposta()
luan.busca_cursos_do_mes()
```

```
luan.mostrar_tarefas()

print(luan)
```

9) Com esse `print`, dá para concluir que agora que há um `__str__` compartilhado pela classe `Hipster`, qualquer classe pode herdar este comportamento, se você desejar.

Os *mixins* são usados desta forma pois o seu comportamento normalmente não precisa ser de responsabilidade da classe filha, pois esta é mais específica.