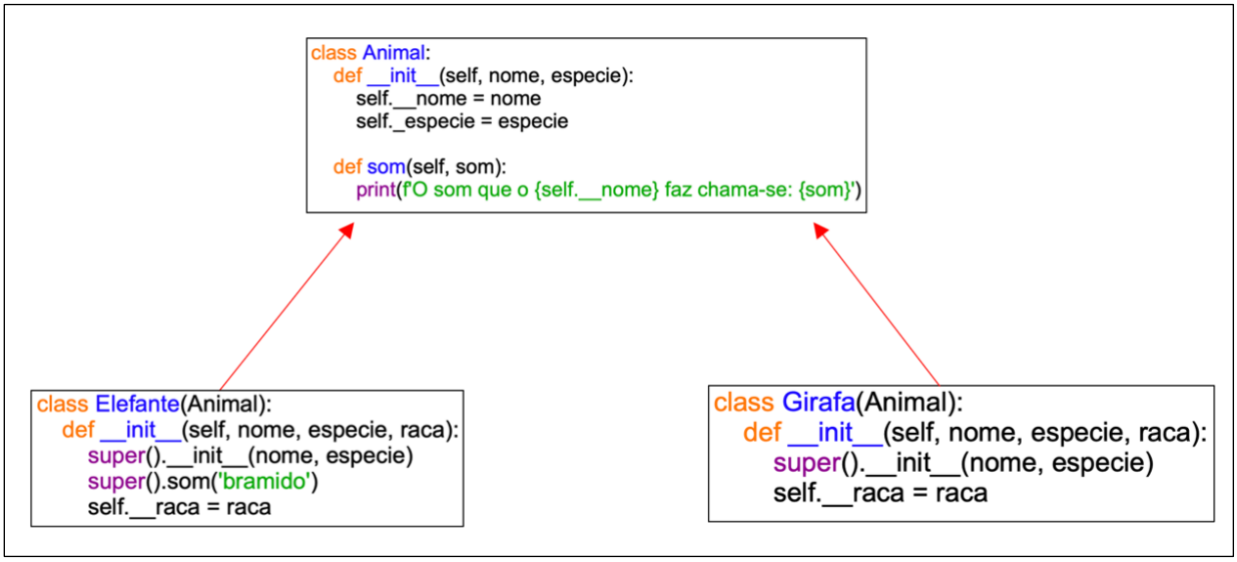
PARADIGMAS E CONCEITOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

MÉTODO super(): se refere à **super classe** que tivemos a oportunidade de conhecer quando estudamos herança. Segundo F.V.C (2020, página 232): "O método super() faz a coleta dos atributos da super classe (classe pai)".

 # 72 - Implementando o método **super**():



 # 73 - Testando o método **super**():

```
dumbo = Elefante('Elefante','Africano', 'Loxodonta africana')

gisela = Girafa('Girafa','Africana',' Giraffidae')

gisela.som('Zumbido')
```

```
O som que o Elefante faz chama-se: bramido

O som que o Girafa faz chama-se: Zumbido
```

Note: Os dois objetos foram instanciados da classe **super**(). Porém, o acesso aos atributos foi realizado de duas formas diferentes. Com o método **super**() podemos fazer acesso a qualquer elemento da classe pai.

PARADIGMAS E CONCEITOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

HERANÇA MÚLTIPLA: Em Python herança é uma maneira de gerar novas classes utilizando outras classes, ela ocorre quando uma classe (**filha**) herda características e métodos de outra classe (**pai**), mas não impede que a classe filha **possua seus próprios** métodos e atributos. A principal vantagem aqui é poder reutilizar o código e reduz a complexidade do módulo. (F.V.C. 2019).

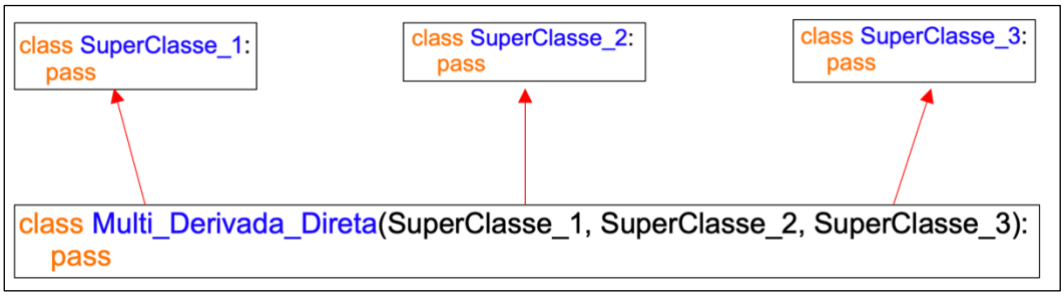
Herança Múltipla: possibilita que a classe **filha** herde todos os atributos e métodos de todas as classes herdadas. Ela pode ser implementada através da **multiderivação.**

Multiderivação: na língua portuguesa é uma "Derivação parassintética – a palavra é formada pelo acréscimo simultâneo de um prefixo e de um sufixo ao radical da palavra primitiva. Exs.: amadurecer, desalmado, entardecer". Mais informações acesse: http://proedu.rnp.br/bitstream/handle/123456789/602/Aula_08.pdf?sequence=9&isAllowed=y

Em nosso caso, na linguagem Python a **multiderivação** é formada pela palavra multi + derivação, pois dizemos que uma classe deriva de outra classe quando ela deriva de outra classe. Dessa maneira, quando nos referimos a multiderivação estamos falando de **herança múltipla**.

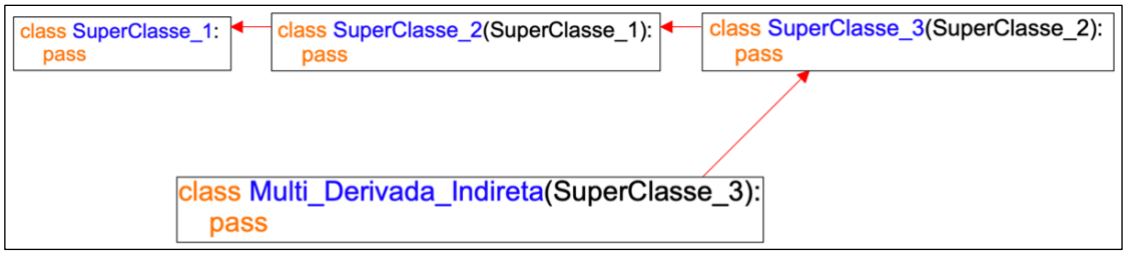
A herança múltipla em Python pode ocorrer de duas formas:

- **Por multiderivação direta**:



Acima a classe Multi_Derivada_Direta() está herdando diretamente os atributos e métodos das SuperClasse_1(), SuperClasse_2(), SuperClasse_3().

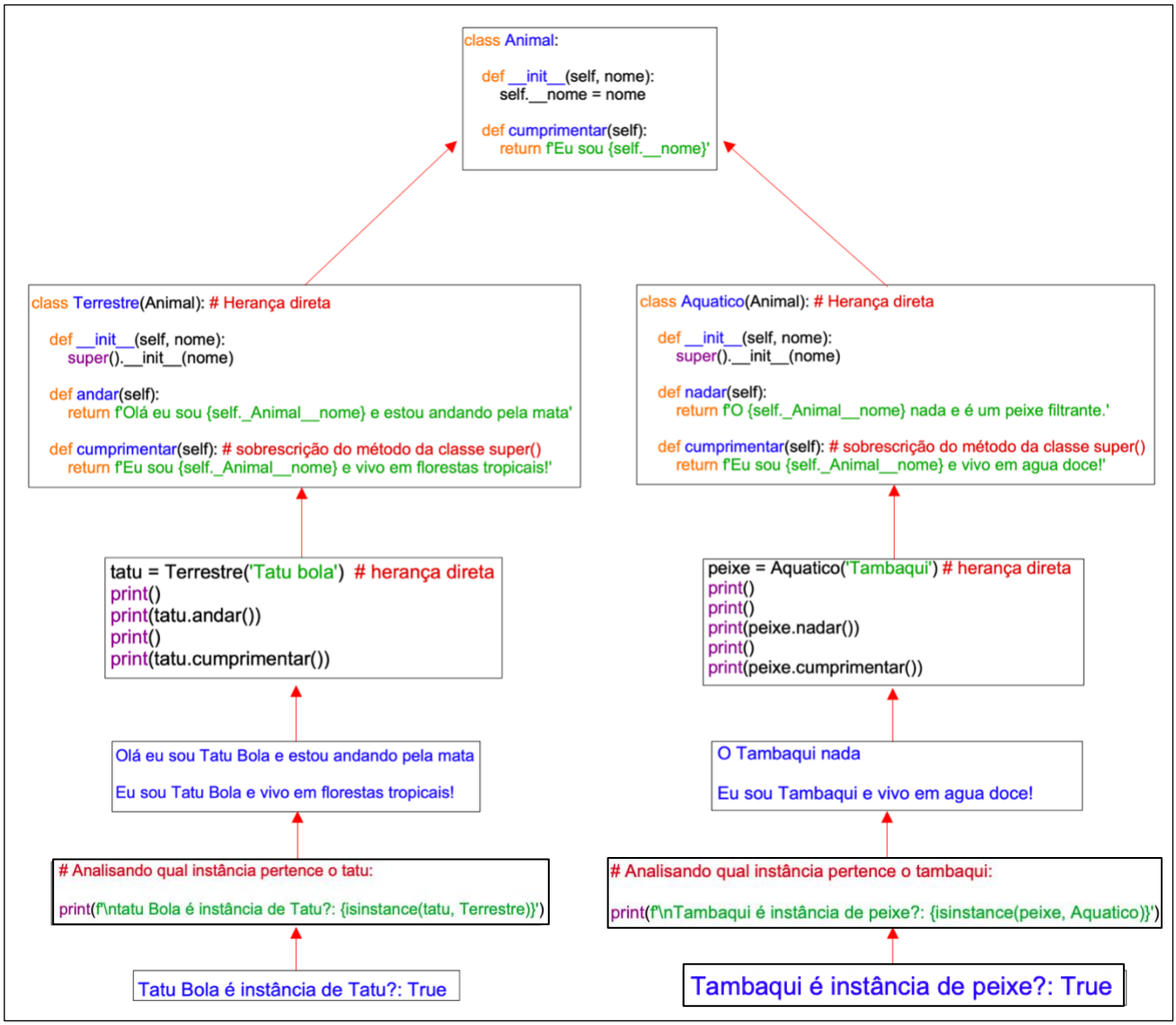
- **Por multiderivação indireta**:



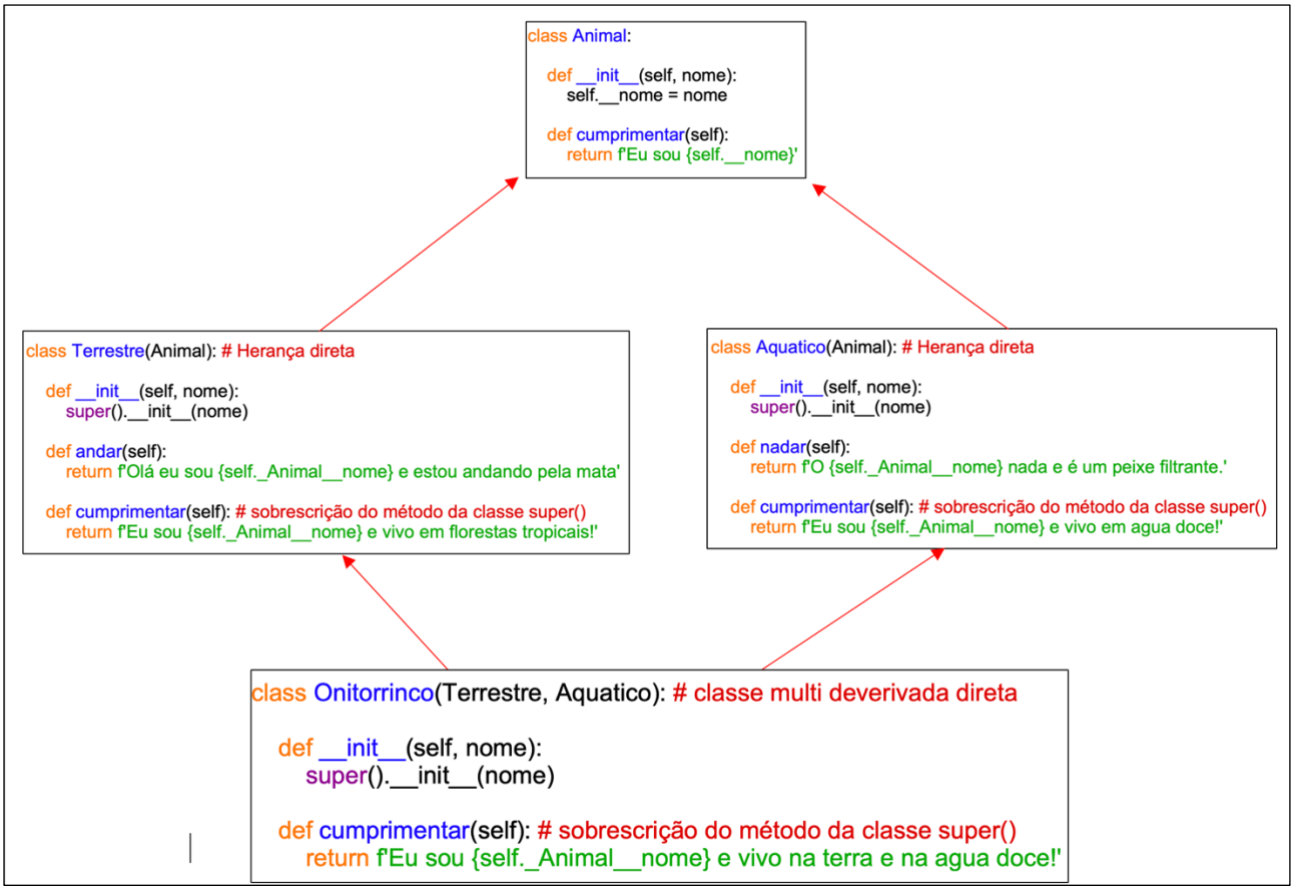
- A acima a classe Multi_Derivada_Indireta() está herdando diretamente os atributos e métodos da SuperClasse_3(), A SuperClasse_3() herda os atributos e métodos da SuperClasse_2() e a SuperClasse_2() herda os atributos da SuperClasse_1().

- Note que, tanto na derivação direta ou indireta, a classe que realizar a herança herdará todos os atributos e métodos das **SuperClasses**.

74 - A herança múltipla - implementação



- A herança múltipla - implementação:

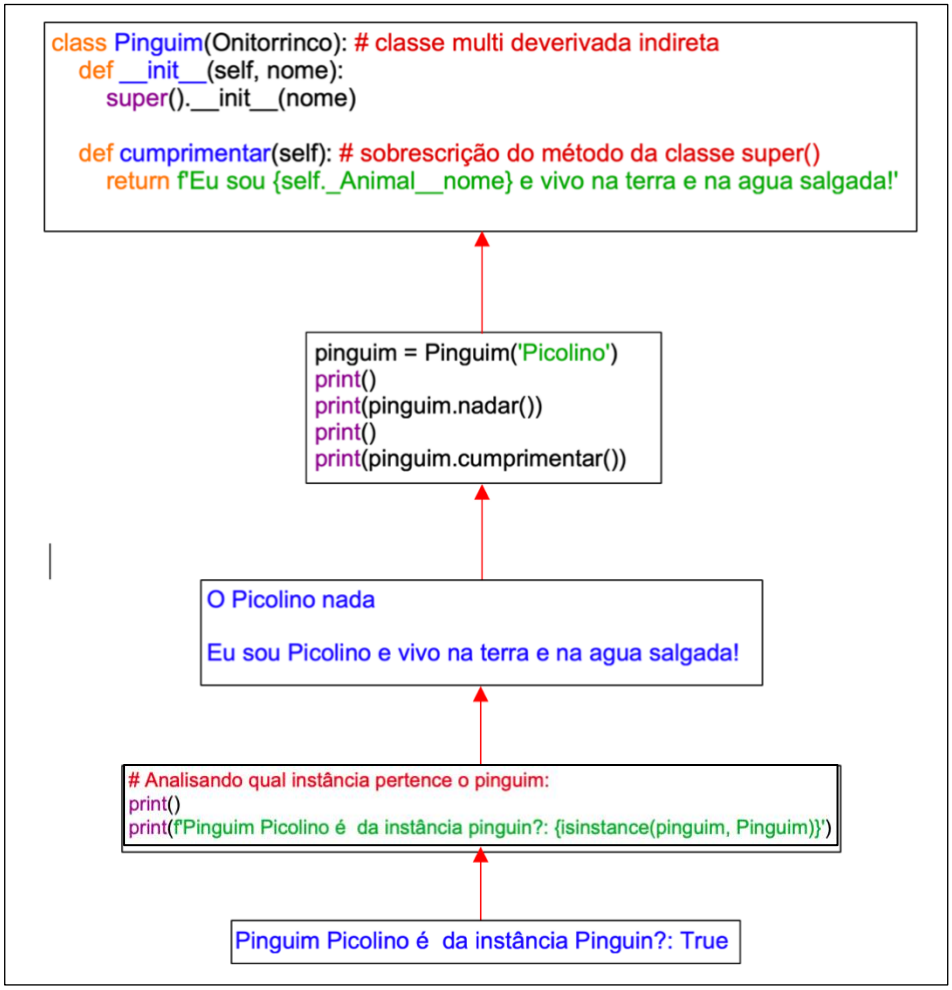


75 - testando a classe multi derivada direta

```
perry = Ornitorrinco('Perry')

print('\n',perry.andar())

print('\n',perry.nadar())

print('\n',perry.cumprimentar())
```

```
Olá eu sou Perry e estou andando pela mata

O Perry nada

Eu sou Perry e vivo na terra e na agua doce!
```

76 - Classe multi derivada indireta:



77 - Testando classe multi derivada indireta

```
pinguim = Pinguim('Picolino')
print()
print(pinguim.nadar())
print()
print(pinguim.cumprimentar())
```

- Analisando qual instância pertence o pinguim

```
print()
print(f'Pinguim Picolino é  da instância pinguin?: {isinstance(pinguim, Pinguim)}')
```

PARADIGMAS E CONCEITOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

- **Method Resolution Order** – MRO: define a ordem de execução dos métodos. Para verificar essa ordem o desenvolvedor pode utilizar uma das três técnicas abaixo. Para dar forma vamos utilizar a classe ornitorrinco:

- Entendendo a ordem de execução de uma classe multi derivada
Method Resolution Order - MRO

a) - ornitorrinco herda das classes Terrestre e Aquático:

```
class Ornitorrinco(Terrestre, Aquatico):

    def __init__(self, nome):
        super().__init__(nome)

    def cumprimentar(self): # sobrescrição do método da classe super()
        return f'Eu sou {self._Animal__nome} e vivo na terra e na agua doce!'
```

```
print()
perry = Ornitorrinco('Perry')
print(perry.cumprimentar())
```

```
Eu sou Perry e vivo na terra e na agua doce!
```

b) - **Se não houver** o método cumprimentar dentro da classe, aí entra o MRO

```
class Ornitorrinco(Terrestre, Aquatico): # classe multi deverivada direta

    def __init__(self, nome):
        super().__init__(nome)
```

```
print()
perry = Ornitorrinco('Perry')
print(perry.cumprimentar()) #
```

```
Eu sou Perry e vivo em florestas tropicais!
```

c) - Se a ordem da classe for **alterada**, o resultado será diferente:

```
class Ornitorrinco(Aquatico, Terrestre):

    def __init__(self, nome):
        super().__init__(nome)
```

```
print()
perry = Ornitorrinco('Perry')
print(perry.cumprimentar())
```

```
Eu sou Perry e vivo em agua doce!
```

- Nos exemplos fica claro a importância de as classes estarem em uma sequencia lógica quanto temos herança multi derivada sem sobreposição de métodos.

78 - Entendendo a ordem de execução de uma classe multi derivada

```
class Ornitorrinco(Aquatico, Terrestre): # classe multi deverivada direta

    def __init__(self, nome):
        super().__init__(nome)

    def cumprimentar(self): # sobrescrição do método da classe super()
        return f'Eu sou {self._Animal__nome} e vivo na terra e na agua doce!'
```

```
# - POO - Method Resolution Order - MRO
print()
perry = Ornitorrinco('Perry')
print(perry.cumprimentar())  # -  Metodo Resolution Order ou MRO

print(help(Ornitorrinco))
```

```
Eu sou Perry e vivo na terra e na agua doce!
Help on class Ornitorrinco in module __main__:

class Ornitorrinco(Aquatico, Terrestre)
 |  Ornitorrinco(nome)
 |
 |  Method resolution order:
 |      Ornitorrinco
 |      Aquatico
 |      Terrestre
 |      Animal
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, nome)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  cumprimentar(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Aquatico:
 |
 |  nadar(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Terrestre:
 |
 |  andar(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Animal:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

PARADIGMAS E CONCEITOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

- **Polimorfismo**: Segundo F. V. C (2019, páginas 201-202), “O polimorfismo permite que nossos objetos que herdam características possam alterar seu funcionamento interno a partir de métodos herdados de um objeto pai”.

 # 79 - O ***overrinding*** é a melhor representação do **polimorfismo**

```
class Animal(object):

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

    def emite_som(self):
        raise NotImplementedError('A classe filha precisa implementar esse método')

    def come(self):
        print(f'{self.__nome} está comendo')


class Cachorro(Animal):

    def __init__(self, nome):
        super().__init__(nome)

    def emite_som(self):
        print(f'{self._Animal__nome} fala wau wau')


class Gato(Animal):

    def __init__(self, nome):
        super().__init__(nome)

    def emite_som(self):
        print(f'{self._Animal__nome} fala miau miau')


print()
feliz = Gato('Feliz')
feliz.come()
feliz.emite_som()

print()
gerivaldo = Cachorro('Gerivaldo')
gerivaldo.come()
gerivaldo.emite_som()
```

```
Feliz está comendo
Feliz fala miau miau

Gerivaldo está comendo
Gerivaldo fala wau wau
```

Colocamos o ***objetc*** dentro da classe Animal, mas isso não é necessário, pois, por *default* toda classe em Python herda de ***objetc***

PARADIGMAS E CONCEITOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

- **MÉTODOS MÁGICOS**: são todos os métodos que utilizam Dunder.

dunder init: __init__ ==> método construtor, esses métodos utilizam double underscore.

 # 80 - Aplicando métodos mágicos

```
class Livros:

    def __init__(self, titulo, autor, paginas):
        self.titulo = titulo
        self.autor = autor
        self.paginas = paginas
```

 # 81 - Testando

```
livro_1 = Livros('Python: Guia Prático do Básico ao Avançado', 'Rafael F. V. C. Santos', 225)
livro_2 = Livros('Programação Funcional para Leigos', ' Jhon Paul Mueller', 481)

print(f'\n Livro: {livro_1}')
print(f'\n Livro: {livro_2}')
```

```
Livro: <__main__.Livros object at 0x10632e7d0>

Livro: <__main__.Livros object at 0x10632ef20>
```

Endereços de memória onde estão armazenados os dados

A saída de dados acima não representa algo para um usuário.

- Tornando a informação representativa - __repr__(self):

```
def __repr__(self):
    return f'{self.__titulo} escrito por {self.__autor}'
```

O comando __repr__(self) torna as informações representativas.

 # 82 – Refatorando e testando a classe Livros com __repr__(self)

```
class Livros:

    def __init__(self, titulo, autor, paginas):
        self.__titulo = titulo
        self.__autor = autor
        self.__paginas = paginas

    def __repr__(self):
        return f'{self.__titulo} escrito por {self.__autor} - nº páginas: {self.__paginas}'


# - Testando
livro_1 = Livros('Python: Guia Prático do Básico ao Avançado', 'Rafael F. V. C. Santos', 225)
livro_2 = Livros('Programação Funcional para Leigos', ' Jhon Paul Mueller', 481)

print(f'\n Livro: {livro_1}')
print(f'\n Livro: {livro_2}')
```

```
Livro: Python: Guia Prático do Básico ao Avançado escrito por Rafael F. V. C. Santos - nº páginas: 225

Livro: Programação Funcional para Leigos escrito por  Jhon Paul Mueller - nº páginas: 481
```

__repr__(self) ==> representação do objeto. Esse tipo de representação normalmente não é feita para o usuário final e sim para o desenvolver entender como o módulo está rodando.

83 - Tornando a informação representativa para o usuário final - __str__(self)

```
class Livros:

    def __init__(self, titulo, autor, paginas):
        self.__titulo = titulo
        self.__autor = autor
        self.__paginas = paginas

    def __str__(self):
        return f'{self.__titulo} escrito por {self.__autor} - nº páginas: {self.__paginas}'


# - Testando
livro_1 = Livros('Python: Guia Prático do Básico ao Avançado', 'Rafael F. V. C. Santos', 225)
livro_2 = Livros('Programação Funcional para Leigos', ' Jhon Paul Mueller', 481)

print(f'\n Livro: {livro_1}')
print(f'\n Livro: {livro_2}')
```

```
Livro: Python: Guia Prático do Básico ao Avançado escrito por Rafael F. V. C. Santos - nº páginas: 225

Livro: Programação Funcional para Leigos escrito por  Jhon Paul Mueller - nº páginas: 481
```

84 – Verificando o tamanho de um atributo usando - __len__(self)

```
class Livros:

    def __init__(self, titulo, autor, paginas):
        self.__titulo = titulo
        self.__autor = autor
        self.__paginas = paginas

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


# - Testando
livro_1 = Livros('Python: Guia Prático do Básico ao Avançado', 'Rafael F. V. C. Santos', 225)
livro_2 = Livros('Programação Funcional para Leigos', ' Jhon Paul Mueller', 481)

print('\nO tamanho do título do  Livro_1 é:', livro_1)
print('\nO tamanho do título do  Livro_2 é:', len(livro_2))
```

```
O tamanho do título do  Livro_1 é: <__main__.Livros object at 0x10b426f50>

O tamanho do título do  Livro_2 é: 33
```

Quando trabalhamos orientação a objetos e pretendemos descobrir quantos caracteres um determinado atributo tem aplicamos a função __len__(self).

- Apagando objetos da memória. Considerando de guardamos na memória RAM os objetos livro_1 e livro_2, para apagá-los basta:

85 – Apagando objetos da memória

```
del livro_1

del livro_2
```

O comando acima apaga o objeto e não retorna nenhum aviso.

86 – Apagando objetos da memória e retornando uma mensagem - __del__(self).

```
class Livros:

    def __init__(self, titulo, autor, paginas):
        self.__titulo = titulo
        self.__autor = autor
        self.__paginas = paginas

    def __del__(self):
        print(f'\nO objeto do tipo livro foi apagado da memória')


# - Testando
livro_1 = Livros('Python: Guia Prático do Básico ao Avançado', 'Rafael F. V. C. Santos', 225)
livro_2 = Livros('Programação Funcional para Leigos', ' Jhon Paul Mueller', 481)

del livro_1

del livro_2
```

```
O objeto do tipo livro foi apagado da memória

O objeto do tipo livro foi apagado da memória
```

- Concatenando objetos. Considerando instanciamos objetos livro_1 e livro_2, para concatena-los utilizamos __add__(self).

87 - Concatenando objetos __add__(self, segundo_objeto)

```
class Livros:

    def __init__(self, titulo, autor, paginas):
        self.__titulo = titulo
        self.__autor = autor
        self.__paginas = paginas

    def __str__(self):
        return self.__titulo

    def __add__(self, segundo_objeto):
        return f'Título livro_1: {self} - Título livro_2: {segundo_objeto}'


# - Testando

livro_1 = Livros('Python: Guia Prático do Básico ao Avançado', 'Rafael F. V. C. Santos', 225)
livro_2 = Livros('Programação Funcional para Leigos', ' Jhon Paul Mueller', 481)

print(livro_1 + livro_2)
```

```
Título livro_1: Python: Guia Prático do Básico ao Avançado - Título livro_2: Programação Funcional para Leigos
```

- Multiplicando valor do atributos dos objetos. Considerando instanciamos objetos livro_1 e livro_2 __mul__(self, outro).

88 - Multiplicando o valor do atributo de um objeto por um número

```
class Livros:

    def __init__(self, titulo, autor, paginas):
        self.__titulo = titulo
        self.__autor = autor
        self.__paginas = paginas

    def __str__(self):
        return self.__titulo

    def __mul__(self, numero):
        if isinstance(numero, int):
            mensagem = ' '
            for n in range(numero):
                mensagem += ' - ' + str(self)
            return mensagem
        return 'Não foi possível multiplicar'


# - Testando

livro_1 = Livros('Python: Guia Prático do Básico ao Avançado', 'Rafael F. V. C. Santos', 225)

print('\n', livro_1 * 3)

print('\n', livro_1 * 'abc')
```

```
 - Python: Guia Prático do Básico ao Avançado - Python: Guia Prático do Básico ao Avançado - Python: Guia Prático do Básico ao Avançado

 Não foi possível multiplicar
```

Note que estamos utilizando métodos builtins do Python para implementar funções em nossas classes. Observe também que nos exemplos acima utilizamos ***overrinding*** e ***Polimorfismo***.