

FERNANDO FELTRIN

**PROGRAMAÇÃO
ORIENTADA A
OBJETOS
COM**

PHYTON



**1ª EDIÇÃO
2020**

ÍNDICE

ÍNDICE.....	3
Nota do Autor.....	4
Introdução.....	5
Ambiente de Programação	6
JetBrains PyCharm.....	6
Microsoft Visual Studio Code	7
Jupyter Notebook.....	8
Programação Orientada a Objetos.....	11
Objetos e Classes.....	11
Variáveis vs Objetos	12
Criando uma classe vazia	17
Atributos de classe	18
Manipulando atributos de uma classe.....	19
Métodos de classe.....	22
Método construtor de uma classe	24
Escopo / Indentação de uma classe	26
Classe com parâmetros opcionais.....	30
Múltiplos métodos de classe.....	32
Interação entre métodos de classe	34
Estruturas condicionais em métodos de classe	36
Métodos de classe estáticos e dinâmicos	39
Getters e setters.....	42
Encapsulamento.....	46
Associação de classes	53
Agregação e composição de classes	56
Herança Simples.....	58
Cadeia de heranças	60
Herança Múltipla.....	62
Sobreposição de membros.....	64
Classes abstratas	69
Polimorfismo	72
Sobrecarga de operadores	74
Tratando exceções	79
Capítulo Final.....	80

Nota do Autor

Seja muito bem-vindo(a) ao meu livro dedicado à programação orientada a objetos. Neste pequeno livro estaremos abordando de forma prática e descomplicada o paradigma de orientação a objeto, método de programar este que visa abstrair situações de algum problema computacional real de forma a criar estruturas de código mais eficientes para a resolução dos mesmos.

Estaremos usando a linguagem Python 3 para estarmos pondo em prática cada conceito explicado no decorrer desse livro. Importante salientar que não iremos nos ater a explicar os conceitos básicos desta linguagem de programação, mas iremos diretamente ao ponto, já programando cada linha de código em seu formato orientado a objetos com as devidas explicações.

A programação orientada a objetos, independente de qual linguagem de programação estejamos falando, é um campo um pouco mais avançado da programação, logo, se espera que você domine o básico da linguagem Python para que não tenha dificuldade em entender os conceitos aqui utilizados e criar seus códigos. Caso contrário, recomendo fortemente que você estude primeiro e domine o básico, pois a forma como iremos abstrair cada conceito e codifica-lo pode parecer confusa caso você não domine a sintaxe e a lógica da programação em Python.

Por fim, independente de você vem de outra linguagem de programação, a metodologia que estaremos utilizando aqui pode ser aplicada em outras linguagens com suporte a orientação a objetos, desde que se respeite sua sintaxe. Porém acredito que se você buscou por esse livro deva ser mais um apaixonado por Python em busca de aumentar sua bagagem de conhecimento dessa maravilhosa linguagem de programação.

Leia, releia, pratique, e tenho certeza que ao final desse livro você terá condições de desenvolver seus códigos tanto em programação convencional (estruturada) quanto orientada a objetos.

Um forte abraço.

Fernando Feltrin

Introdução

A programação orientada à objetos é uma forma avançada de se programar, onde deixamos um pouco de lado o conceito de uma variável ser apenas um simples espaço de memória alocado onde está atribuído um dado ou valor. Na chamada programação orientada a objetos damos poderes a nossas variáveis de forma que elas podem ter em si toda uma estrutura de dados como suporte a variáveis dentro de variáveis, atribuições complexas, métodos e funções, etc... de modo a poder criar interações mais eficientes dentro de nossos blocos de código.

Dessa forma, conseguimos criar estruturas lógicas que podem se alocadas, instanciadas, utilizadas, reutilizadas e até mesmo descartadas conforme a necessidade, inclusive definindo toda uma estrutura hierárquica e de permissões sobre as mesmas.

Em outras palavras, há um enorme ganho de performance neste tipo de programação uma vez que nem todo o código precisa ser carregado e executado integralmente, conforme gatilhos vão acionando ações e funções dentro do código, os objetos responsáveis são acionados dinamicamente. Este conceito na verdade existe fora da programação orientada a objetos, quando modularizamos nossos códigos, porém, nos moldes de programação orientada a objetos teremos além de um ganho real de performance, um mundo de possibilidades de interações entre blocos de código, entre módulos e pacotes, além das interações convencionais que estamos acostumados em programação convencional estruturada.

Sendo assim, embora inicialmente pareça mais complexo, conseguiremos criar programas mais robustos, onde dependendo das suas funcionalidades, quando programadas de forma orientada a objetos e não da maneira convencional estruturada, teremos vantagens implícitas como tornar mais fácil a manutenção de um código ou a implementação de novas funções no mesmo, haja visto que nesses moldes podemos realizar tais alterações diretamente no objeto correspondente.

Para muitos estudantes de programação existe uma barreira inicial a ser quebrada quando se entra nesta parte, uma vez que sua sintaxe muda de forma que iremos criar objetos e estruturas de dados de maior complexidade. Assim como tudo na programação, tenha em mente que cada tópico, cada conceito abordado deve ser entendido e posto em prática, caso contrário, você não aprenderá de fato este tipo de programação.

Por fim, só resta lhe fazer uma breve pergunta: - Você está pronto para começar?
Se a resposta for sim, boa leitura e nos vemos nos capítulos seguintes.

Ambiente de Programação

Se tratando do ambiente o qual estaremos desenvolvendo nossos códigos, há uma grande variedade de IDEs que, cada uma com suas respectivas particularidades, não necessariamente são umas melhores que outras, mas atendem melhor o gosto do próprio programador.

Eu particularmente recomendo fortemente o uso da PyCharm, ou do Visual Studio Code ou do Jupyter Notebook. Porém se você já está familiarizado a outra IDE pode fazer o uso dela sem problemas, o importante é você ter um ambiente confiável onde consiga criar e rodar seus códigos em tempo real.

Caso você seja usuário do Windows deve saber que o Python não vem nativamente instalado em seu sistema operacional, sendo assim, antes mesmo de instalar uma IDE é preciso instalar a última versão estável do Python (no momento da escrita deste livro, versão 3.8.0).

Disponível em: <https://www.python.org/downloads/>



Uma vez instalado corretamente a linguagem Python por meio de seu instalador, podemos prosseguir com a instalação da IDE ao qual faremos o uso.

JetBrains PyCharm

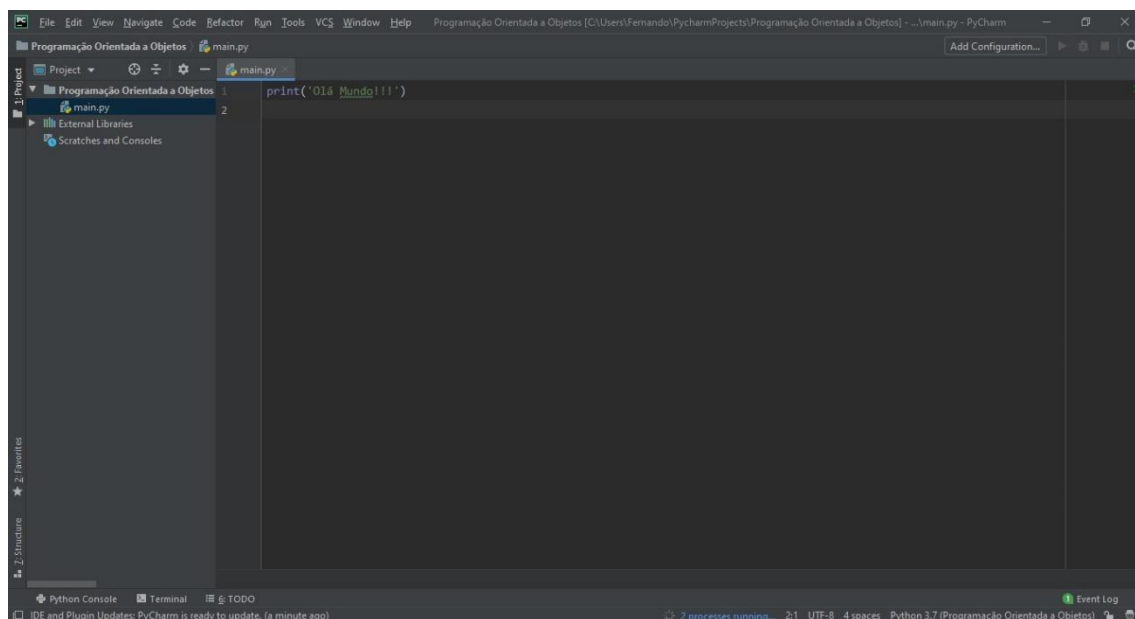
O Pycharm é uma das IDEs mais utilizadas por programadores Python, ela possui uma versão corporativa paga com algumas funcionalidades a mais em relação a sua versão comunitária, que por sua vez é totalmente gratuita. Porém, a versão gratuita da mesma possui todas as funcionalidades que precisará ter à disposição para este tipo de programação, dessa forma, para fins de estudo, é perfeitamente útil a utilização da versão gratuita do PyCharm.

A versão mais recente pode ser baixada em: <https://www.jetbrains.com/pycharm/>

Site



Interface

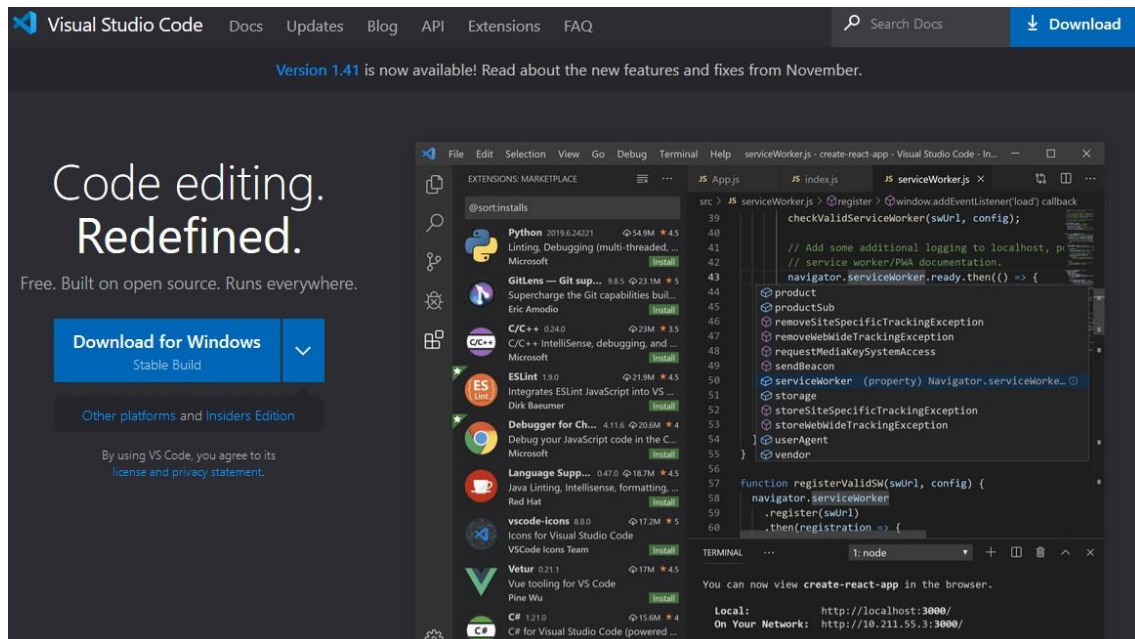


Microsoft Visual Studio Code

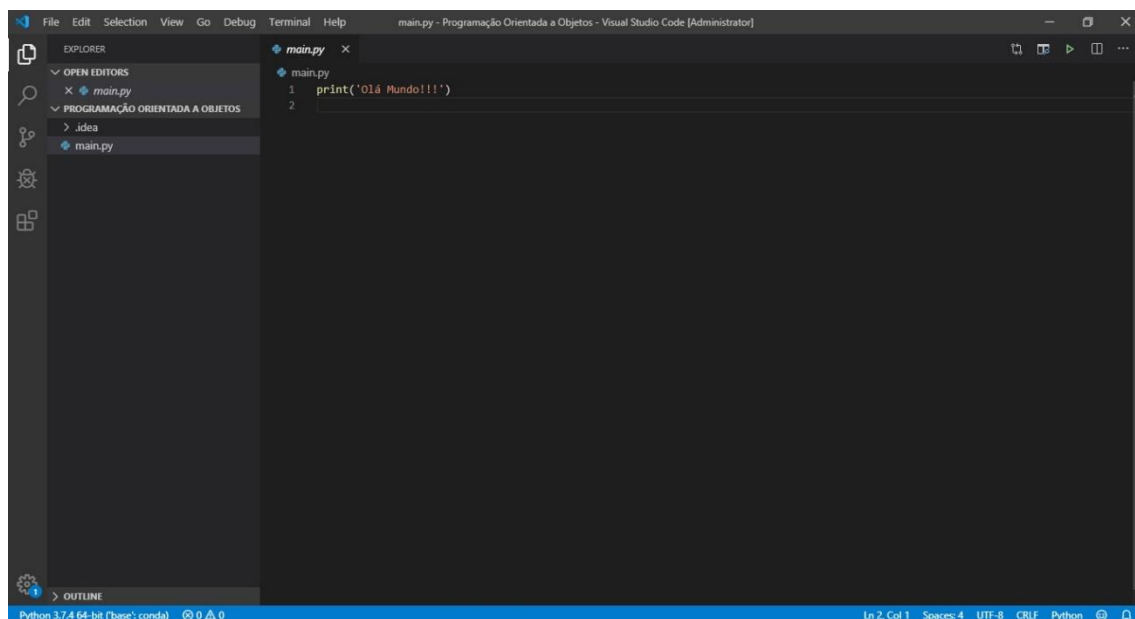
A IDE desenvolvida pela Microsoft não deixa a desejar em nenhum aspecto quando comparada ao PyCharm, inclusive é uma IDE bem aceita pela comunidade e bastante utilizada em função da quantidade de plugins que podem ser instalados na mesma para oferecer outras funcionalidades. É a IDE que estarei utilizando para criar e capturar cada linha de código deste livro, em função do uso de alguns plugins complementares que eu uso normalmente em minhas rotinas.

A versão mais recente pode ser baixada em: <https://code.visualstudio.com/>

Site



Interface



Jupyter Notebook

Esta IDE faz parte da suíte Anaconda, muito utilizado por quem está dando os primeiros passos em redes neurais artificiais e machine learning pela característica de suportar a execução individual de blocos de códigos em tempo real, de modo que funções específicas de redes neurais podem ser implementadas e alteradas em tempo

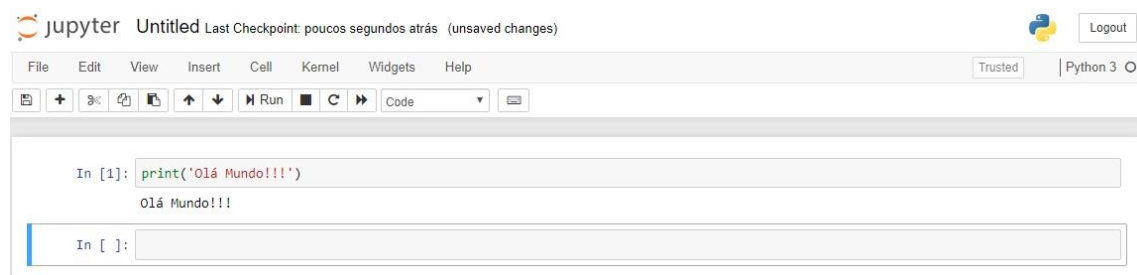
real sem afetar o restante do código pré-processado. Este tipo de notebook interativo também é bastante útil por rodar diretamente no navegador e suportar salvar diferentes estados de execução do código.

A versão mais recente pode ser baixada em: <https://jupyter.org/>

Site



Interface



Como mencionado anteriormente, realmente fica a seu critério usar uma dessas IDEs citadas acima ou outra de sua preferência. Os códigos que iremos criar não requerem a instalação de módulos nem bibliotecas adicionais*, estaremos trabalhando em cima do Python 3 puro, em função disso qualquer IDE é capaz de executá-los normalmente.

*em um dos exemplos que serão abordados posteriormente usaremos de uma biblioteca nativa do Python para criação de classes abstratas, não será usada nenhuma biblioteca ou módulo externo, que tenha que ser instalado ou implementado manualmente no Python de nosso sistema.

Programação Orientada a Objetos

Objetos e Classes

Independente se você está começando do zero com a linguagem Python ou se você já programa nela ou em outras linguagens, você já deve ter ouvido a clássica frase “em Python tudo é objeto” ou “em Python não há distinção entre variável e objeto”. De fato, uma das características fortes da linguagem Python é que uma variável pode ser simplesmente uma variável assim como ela pode “ganhar poderes” e assumir o papel de um objeto sem nem ao menos ser necessário alterar sua sintaxe.

Por convenção, dependendo que tipo de problema computacional que estaremos abstraindo, poderemos diretamente atribuir a uma variável/objeto o que quisermos, independente do tipo de dado, sua complexidade ou seu comportamento dentro do código. O interpretador irá reconhecer normalmente uma variável/objeto conforme seu contexto e será capaz de executá-lo em tempo real.

Outro ponto que precisamos começar a entender, ou ao menos por hora desmistificar, é que na programação orientada a objetos, teremos variáveis/objetos que receberão como atributo uma classe. Raciocine que na programação convencional estruturada você ficava limitado a **variável = atributo**. Supondo que tivéssemos uma variável de nome lista1 e seu atributo fosse uma lista (tipo de dado lista), a mesma estaria explícita como atributo de lista1 e sua funcionalidade estaria ali definida (todas características de uma lista). Agora raciocine que, supondo que houvésssemos um objeto de nome mercado1, onde lista1 fosse apenas uma das características/atributos internas desse objeto, essas demais estruturas estariam dentro de uma classe, que pode inclusive ser encapsulada ou modularizada para separar seu código do restante.

Em outras palavras, apenas tentando exemplificar a teoria por trás de uma classe em Python, uma classe é uma estrutura de dados que pode perfeitamente guardar dentro de si uma infinidade de estruturas de dados, como variáveis e seus respectivos dados/valores/atributos, funções e seus respectivos parâmetros, estruturas lógicas e/ou condicionais, etc... tudo atribuído a um objeto facilmente instanciável, assim como toda estrutura dessa classe podendo ser reutilizada como molde para criação de novos objetos, ou podendo interagir com outras classes.

Como dito anteriormente, não se assuste com estes tópicos iniciais e a teoria envolvida, tudo fará mais sentido a medida que formos exemplificando cada conceito destes por meio de blocos de código.

Variáveis vs Objetos

Partindo para a prática, vamos inicialmente entender no código quais seriam as diferenças em um modelo estruturado e em um modelo orientado a objetos. Para esse exemplo, vamos começar do básico apenas abstraindo uma situação onde simplesmente teríamos de criar uma variável de nome **pessoa1** que recebe como atributos um nome e uma idade.

Inicialmente raciocine que estamos atribuindo mais de um dado (nome e idade) que inclusive serão de tipos diferentes (para nome uma **string** e para idade um **int**). Convencionalmente isto pode ser feito sem problemas por meio de um dicionário, estrutura de dados em Python que permite o armazenamento de dados em forma de chave / valor, aceitando como dado qualquer tipo de dado alfanumérico desde que respeitada sua sintaxe:

```
pessoa1 = {'Nome': 'Fernando', 'Idade': 32}
print(pessoa1)
```

Então, finalmente dando início aos nossos códigos, inicialmente criamos na primeira linha uma variável de nome **pessoa1** que por sua vez, respeitando a sintaxe, recebe um dicionário onde as chaves e valores do mesmo são respectivamente **Nome : Fernando e Idade : 32**. Em seguida usamos a função **print()** para exibir o conteúdo de **pessoa1**.

O retorno será: **Nome : Fernando, Idade : 32**



Como dito anteriormente, nesses moldes, temos uma estrutura estática, onde podemos alterar/adicionar/remover dados manipulando o dicionário, mas nada além das funcionalidades padrão de um dicionário.

Partindo para um modelo orientado a objetos, podemos criar uma classe de nome **Pessoa** onde **Nome** e **Idade** podem simplesmente ser variáveis com seus respectivos atributos guardados lá dentro. E não nos limitamos somente a isso, podendo adicionar manualmente mais características a **pessoa1** conforme nossa necessidade.

```
class Pessoa:
    pass

pessoa1 = Pessoa()
pessoa1.Nome = 'Fernando'
pessoa1.Idade = 32
```

Inicialmente, pela sintaxe Python, para criarmos a estrutura de uma classe de forma que o interpretador faça a sua leitura léxica correta, precisamos escrever **class** seguido do nome da classe, com letra maiúscula apenas para diferenciar do resto. Dentro da classe, aqui nesse exemplo, colocamos apenas o comando **pass**, dessa forma, esta será inicialmente uma classe “em branco”, por enquanto vazia. Na sequência criamos o objeto de nome **pessoa1** que inicializa a classe **Pessoa** sem parâmetros mesmo, apenas a colocando como atributo próprio. Em seguida, por meio do comando **pessoa1.Nome**, estamos criando a variável **Nome** dentro da classe **Pessoa**, que por sua vez recebe ‘**Fernando**’ como atributo, o mesmo é feito adicionando a variável **Idade** com seu respectivo valor.



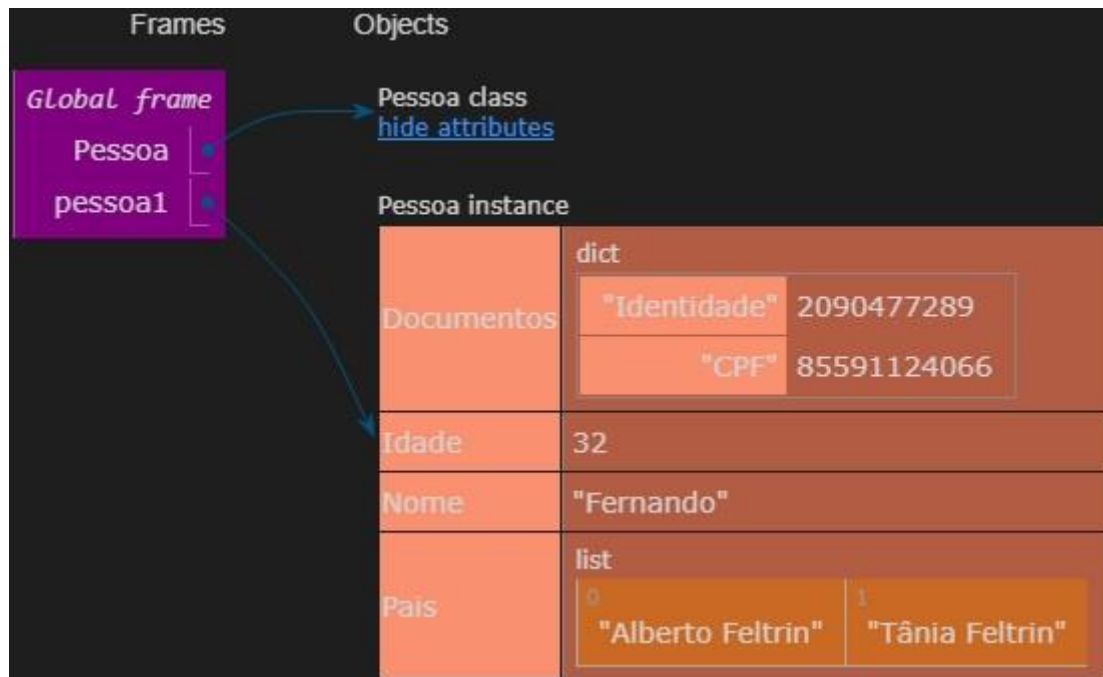
Note que em comparação ao código anterior temos mais linhas de código e de maior complexidade em sua lógica sintática, neste exemplo em particular ocorre isto mesmo, porém você verá posteriormente que programar blocos de código dentro de classes assim como as modularizar tornará as coisas muito mais eficientes, além de reduzir a quantidade de linhas/blocos de código.

Tratando os dados como instâncias de uma classe, temos liberdade para adicionar o que quisermos dentro da mesma, independente do tipo de dado e de sua função no contexto do código.

```
class Pessoa:
    pass

pessoa1 = Pessoa()
pessoa1.Nome = 'Fernando'
pessoa1.Idade = 32
pessoa1.Documentos = {'Identidade':2090477289, 'CPF':85591124066}
pessoa1.Pais = ['Alberto Feltrin', 'Tânia Feltrin']
```

Repare que seguindo com o exemplo, adicionalmente inserimos um dicionário de nome **Documentos** com valores de **Identidade** e **CPF**, assim como uma lista de nome **Pais** que tem atribuída a si **Alberto Feltrin** e **Tânia Feltrin**.



Toda essa estrutura, guardada na classe **Pessoa**, se fosse ser feita de forma convencional estruturada demandaria a criação de cada variável em meio ao corpo do código geral, dependendo o contexto, poluindo bastante o mesmo e acarretando desempenho inferior em função da leitura léxica do interpretador da IDE.

Sendo assim, aqui inicialmente apenas mostrando de forma visual os primeiros conceitos, podemos começar a entender como se programa de forma orientada a objetos, paradigma este que você notará que será muito útil para determinados tipos de códigos / determinados tipos de problemas computacionais. Nos capítulos subsequentes, gradualmente vamos começar a entender as funcionalidades que podem ser atribuídas a uma classe de um determinado objeto, assim como a forma como instanciaremos dados internos desse objeto para interação com outras partes do código geral.

Outro ponto a levar em consideração neste momento inicial, pode estar parecendo um pouco confuso de entender a real distinção entre se trabalhar com uma variável (programação estruturada) e um objeto (programação orientada a objetos). Para que isso fique bem claro, relembre do básico da programação em Python, uma variável é um espaço de memória alocado onde guardamos algum dado ou valor, de forma estática. Já na programação orientada a objetos, uma variável ganha "super poderes" a partir do momento em que ela possui atribuída a si uma classe, dessa forma ela pode inserir e utilizar qualquer tipo de dado que esteja situado dentro da classe, de forma dinâmica.

Supondo que pela modularização tenhamos um arquivo de nome **pessoa.py**, dentro de si o seguinte bloco de código, referente a uma classe vazia.

```
class Pessoa:
    pass
```

Agora, em nosso arquivo principal de nome **main.py** realizamos a importação dessa classe **Pessoa** e a atribuímos a diferentes variáveis/objetos.

```
from pessoa import Pessoa

pessoa1 = Pessoa()
pessoa2 = Pessoa()
pessoa3 = Pessoa()

print(pessoa1)
print(pessoa2)
print(pessoa3)
```

Diferentemente da programação estruturada, cada variável pode usar a classe **Pessoa** inserindo diferentes dados/valores/atributos a mesma. Neste contexto, a classe **Pessoa** serve como “molde” para criação de outros objetos, sendo assim, neste exemplo, cada variável que inicializa **Pessoa()** está criando toda uma estrutura de dados única para si, usando tudo o que existe dentro dessa classe (neste caso não há nada mesmo, porém poderia ter perfeitamente blocos e mais blocos de códigos...) sem modificar a estrutura de dados da classe **Pessoa**.

```
<pessoa.Pessoa object at 0x000001DAA659AA08>
<pessoa.Pessoa object at 0x000001DAA6565048>
<pessoa.Pessoa object at 0x000001DAA84E2208>
PS C:\Users\Fernando\PycharmProjects\Programação Orientada a Objetos>
```

Por meio da função **print()** podemos notar que cada variável **pessoa** está alocada em um espaço de memória diferente, podendo assim usar apenas uma estrutura de código (a classe **Pessoa**) para atribuir dados/valores a cada variável (ou para mais de uma variável).



Raciocine inicialmente que, a vantagem de se trabalhar com classes, enquanto as mesmas tem função inicial de servir de “molde” para criação de objetos, é a maneira como podemos criar estruturas de código reutilizáveis. Para o exemplo anterior, imagine uma grande estrutura de código onde uma pessoa ou um item teria uma série de características (muitas mesmo), ao invés de criar cada característica manualmente para cada variável, é mais interessante se criar um molde com todas estas características e aplicar para cada nova variável/objeto que necessite destes dados.

Em outras palavras, vamos supor que estamos a criar um pequeno banco de dados onde cada pessoa cadastrada tem como atributos um nome, uma idade, uma nacionalidade, um telefone, um endereço, um nome de pai, um nome de mãe, uma estimativa de renda, etc... Cada dado/valor destes teria de ser criado manualmente por meio de variáveis, e ser recriado para cada nova pessoa cadastrada neste banco de dados... além de tornar o corpo do código principal imenso. É logicamente muito mais eficiente criar uma estrutura de código com todos estes parâmetros que simplesmente possa ser reutilizado a cada novo cadastro, ilimitadamente, simplesmente atribuindo o mesmo a uma variável/objeto, e isto pode ser feito perfeitamente por meio do uso de classes (estrutura de dados de classe / estrutura de dados orientada a objeto).

Entendidas as diferenças básicas internas entre programação estruturada convencional e programação orientada a objetos, hora de começarmos a de fato criar nossas primeiras classes e suas respectivas usabilidades, uma vez que como dito anteriormente, este tipo de estrutura de dados irá incorporar em si uma série de dados/valores/variáveis/objetos/funções/etc... de forma a serem usados, reutilizados, instanciados e ter interações com diferentes estruturas de blocos de código.

Criando uma classe vazia

Como dito anteriormente, uma classe é uma estrutura de código que permite associar qualquer tipo de atribuição a uma variável/objeto. Também é preciso que se entenda que tudo o que será codificado dentro de uma classe funcionará como um molde para que se criem novos objetos ou a interação entre os mesmos.

Dessa forma, começando a entender o básico das funcionalidades de uma função, primeira coisa a se entender é que uma classe tem uma sintaxe própria que a define, internamente ela pode ser vazia, conter conteúdo próprio previamente criado ou inserir conteúdo por meio da manipulação da variável/objeto ao qual foi atribuída. Além disso, uma classe pode ou não retornar algum dado ou valor de acordo com seu uso em meio ao código.

```
class Pessoa:  
    pass
```

Começando do início, a palavra reservada ao sistema **class** indica ao interpretador que logicamente estamos trabalhando com uma classe. A nomenclatura usual de uma classe deve ser o nome da mesma iniciando com letra maiúscula. Por fim, especificado que se trata de uma classe, atribuído um nome a mesma, é necessário colocar dois pontos para que seja feita sua indentação, muito parecida com uma função comum em Python.

```
class Pessoa:  
    pass  
  
pessoa1 = Pessoa()  
pessoa1.nome = 'Fernando'  
  
print(pessoa1.nome)
```

Dando sequência, após criada a classe **Pessoa**, podemos começar a manipular a mesma. Neste exemplo, em seguida é criada no corpo do código geral a variável **pessoa1** que inicializa a classe **Pessoa()**, atribuindo a si todo e qualquer conteúdo que estiver inserido nessa classe. É necessário colocar o nome da classe seguido de parênteses pois posteriormente você verá que é possível passar parâmetros neste espaço. Na sequência é dado o comando **pessoa1.nome = 'Fernando'**, o que pode ser entendido que, será criada dentro da classe uma variável de nome **nome** que recebe como dado a **string** **'Fernando'**. Executando a função **print()** e passando como parâmetro **pessoa1.nome**, o retorno será **Fernando**.



Atributos de classe

Dando mais um passo no entendimento lógico de uma classe, hora de começarmos a falar sobre os atributos de classe. No exemplo anterior criamos uma classe vazia e criamos de fora da classe uma variável de nome **nome** que recebia **'Fernando'** como atributo. Justamente, um atributo de classe nada mais é do que uma variável criada/declarada dentro de uma classe, onde de acordo com a sintaxe Python, pode receber qualquer dado ou valor.

```
class Pessoa:
    nome = 'Fernando'

pessoa1 = Pessoa()

print(pessoa1.nome)
```

Dessa vez criamos uma classe de nome **Pessoa**, internamente criamos uma variável de nome **nome** que recebe como atributo **'Fernando'**. Em seguida, fora da classe, criamos uma variável de nome **pessoa1** que recebe como atributo a classe **Pessoa**, a partir desse momento, todo conteúdo de **Pessoa** estará atribuído a variável **pessoa1**. Dessa forma, por meio da função **print()** passando como parâmetro **pessoa1.nome**, o resultado será: **Fernando**, já que a variável interna a classe **Pessoa** agora pertence a **pessoa1**.



O que fizemos, iniciando nossos estudos, foi começar a entender algumas possibilidades, como: A partir do momento que uma variável no corpo geral do código recebe como atributo uma classe, ela passa a ser um objeto, uma super variável, uma vez que a partir deste momento ela consegue ter atribuída a si uma enorme gama de possibilidades, desde armazenar variáveis dentro de variáveis até ter inúmeros tipos de dados dentro de si. Também vimos que é perfeitamente possível ao criar uma classe, criar variáveis dentro dela que podem ser instanciadas de fora da classe, por uma variável/objeto qualquer. E ainda neste contexto, é importante que esteja bem entendido que, a classe de nosso exemplo **Pessoa**, dentro de si tem uma variável **nome** com um atributo próprio, uma vez que esta classe seja instanciada por outra variável, que seja usada como molde para outra variável, toda sua estrutura interna seria transferida também para o novo objeto.

Manipulando atributos de uma classe

Da mesma forma que na programação convencional estruturada, existe a chamada leitura léxica do interpretador, e ela se dá garantindo que o interpretador fará a leitura dos dados em uma certa ordem e sequência (sempre da esquerda para a direita e linha após linha descendente). Seguindo essa lógica, um bloco de código declarado posteriormente pode alterar ou “atualizar” um dado ou valor em função de que na ordem de interpretação a última informação é a que vale. Aqui não há necessidade de nos estendermos muito, mas apenas raciocine que é perfeitamente normal e possível manipular dados de dentro de uma classe de fora da mesma. Porém é importante salientar que quando estamos trabalhando com uma classe, ocorre a atualização a nível da variável que está instanciando a classe, a estrutura interna da classe continua intacta. Por exemplo:

```
class Pessoa:
    nome = 'Padrão'

pessoa1 = Pessoa()

print(pessoa1.nome)
```

Aproveitando o exemplo anterior, note que está declarada dentro da classe **Pessoa** uma variável de nome **nome** que tem **'Padrão'** como atributo. Logo após criamos uma variável de nome **pessoa1** que inicializa a classe e toma posse de seu conteúdo. Neste momento, por meio da função **print()** passando como parâmetro **pessoa1.nome** o retorno é: **Padrão**

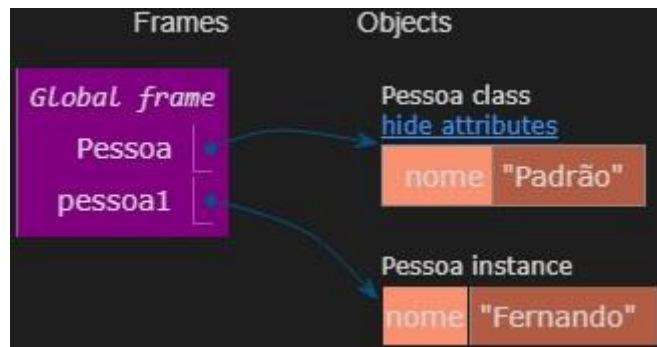


```
class Pessoa:
    nome = 'Padrão'

pessoa1 = Pessoa()
pessoa1.nome = 'Fernando'

print(pessoa1.nome)
```

Agora após criar a variável **pessoa1** e atribuir a mesma a classe **Pessoa**, no momento que realizamos a alteração **pessoa1.nome = 'Fernando'**, esta alteração se dará somente para a variável **pessoa1**. Por meio da função **print()** é possível ver que **pessoa1.nome** agora é **Fernando**.



Note que a estrutura da classe, “o molde” da mesma se manteve, para a classe **Pessoa**, nome ainda é **‘Padrão’**, para a variável **pessoa1**, que instanciou **Pessoa** e sobrescreveu nome, nome é **‘Fernando’**. Como dito anteriormente, uma classe pode servir de molde para criação de vários objetos, sem perder sua estrutura a não ser que as alterações sejam feitas dentro dela, e não pelas variáveis/objetos que a instanciam.

Importante salientar que é perfeitamente possível fazer a manipulação de dados dentro da classe operando diretamente sobre a mesma, porém, muito cuidado pois dessa forma, como a classe serve como “molde”, uma alteração diretamente na classe afeta todas suas instâncias. Em outras palavras, diferentes variáveis/objetos podem instanciar uma classe e modificar o que quiser a partir disto, a estrutura da classe permanecerá intacta. Porém quando é feita uma alteração diretamente na classe, essa alteração é aplicada para todas variáveis/objetos que a instancia.

```
class Pessoa:
    nome = 'Padrão'

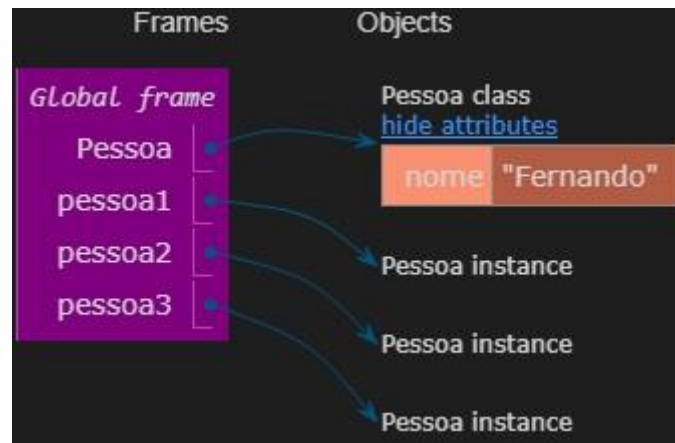
pessoa1 = Pessoa()
pessoa2 = Pessoa()
pessoa3 = Pessoa()

Pessoa.nome = 'Fernando'

print(pessoa1.nome)
print(pessoa2.nome)
print(pessoa3.nome)
```

Aqui seguindo com o mesmo exemplo, porém criando 3 variáveis **pessoa** que recebem como atributo a classe **Pessoa**, a partir do momento que é lida a linha de código **Pessoa.nome = ‘Fernando’** pelo interpretador, a alteração da variável **nome** de **‘Padrão’** para **‘Fernando’** é aplicada para todas as instâncias. Sendo assim, neste caso o retorno será:

Fernando
Fernando
Fernando



Em resumo, alterar um atributo de classe via instancia altera o valor somente para a instância em questão (somente para a variável/objeto que instanciou), alterar via classe altera diretamente para todas as instâncias que a usar a partir daquele momento.

Métodos de classe

Uma prática bastante comum quando se trabalha com orientação a objetos é criar funções dentro de uma classe, essas funções por sua vez recebem a nomenclatura de métodos de classe. Raciocine que nada disto seria possível na programação estruturada convencional, aqui, como dito anteriormente, você pode criar absolutamente qualquer coisa dentro de uma classe, ilimitadamente. Apenas como exemplo, vamos criar uma simples função (um método de classe) dentro de uma classe, que pode ser executada por meio de uma variável/objeto.

```
class Pessoa:
    def acao1(self):
        print('Ação 1 sendo executada...')
```

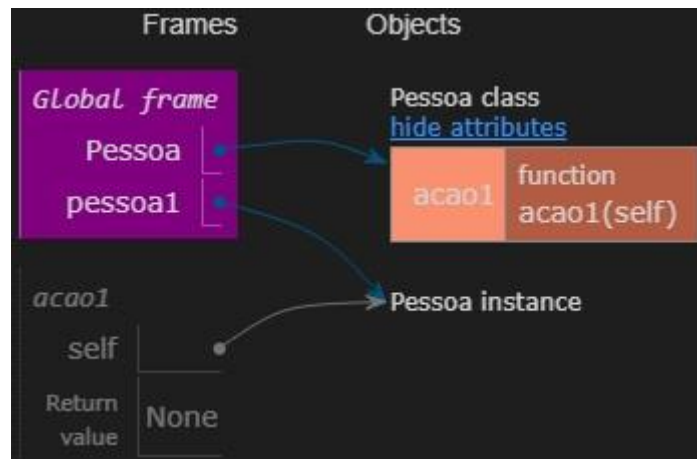
Todo processo se inicia com a criação da classe **Pessoa**, dentro dela, simplesmente criamos uma função (método de classe) de nome **acao1** que tem como parâmetro **self**. Vamos entender esse contexto, a criação da função, no que diz respeito a sintaxe, é exatamente igual ao que você está acostumado, porém este parâmetro **self** é característico de algo sendo executado dentro da classe. Uma classe implicitamente dá o comando ao interpretador pra que seus blocos de código sejam executados apenas dentro de si, apenas quando instanciados, por uma questão de performance. Posteriormente haverá situações diferentes, mas por hora vamos nos ater a isto. Note que a função **acao1** por sua vez simplesmente exibe uma **string** com uma mensagem pré-programada por meio da função **print()**.

```
class Pessoa:
    def acao1(self):
        print('Ação 1 sendo executada...')

pessoa1 = Pessoa()

print(pessoa1.acao1())
```

Da mesma forma do exemplo anterior, é criada a variável **pessoa1** que recebe como atributo a classe **Pessoa()**, em seguida, por meio da função **print()** passamos como parâmetro a função **acao1**, sem parâmetros mesmo, por meio de **pessoa1.acao1()**. Neste caso o retorno será: **Ação 1 sendo executada...**



Método construtor de uma classe

Quando estamos trabalhando com classes simples, como as vistas anteriormente, não há uma real necessidade de se criar um construtor para as mesmas. Na verdade, internamente esta estrutura é automaticamente gerada, o chamado construtor é criado manualmente quando necessitamos parametrizá-lo manualmente de alguma forma, criando estruturas que podem ser instanciadas e receber atribuições de fora da classe.

Para alguns autores, esse processo é chamado de método construtor, para outros de método inicializador, independente do nome, este é um método (uma função interna) que recebe parâmetros (atributos de classe) que se tornarão variáveis internas desta função, guardando dados/valores a serem passados pelo usuário por meio da variável que instanciar essa classe. Em outras palavras, nesse exemplo, do corpo do código, uma variável/objeto que instanciar a classe **Pessoa** terá de passar os dados referentes a nome, idade, sexo e altura, já que o método construtor dessa classe está esperando que estes dados sejam fornecidos.

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

Ainda trabalhando sobre o exemplo da classe **Pessoa**, note que agora criamos uma função interna de nome **__init__(self)**, palavra reservada ao sistema para uma função que inicializa dentro da classe algum tipo de função. Em sua IDE, você notará que ao digitar **__init__** automaticamente ele criará o parâmetro (**self**), isto se dá porque tudo o que estiver indentado a essa função será executado internamente e associado a variável que é a instância desta classe (a variável do corpo geral do código que tem esta classe como atributo).

Da mesma forma que uma função convencional, os nomes inseridos como parâmetro serão variáveis temporárias que receberão algum dado ou valor fornecido pelo usuário, normalmente chamados de atributos de classe. Repare que neste exemplo estão sendo criadas variáveis (atributos de classe) para armazenar o **nome**, a **idade**, o **sexo** e a **altura**. Internamente, para que isto se torne variáveis instanciáveis, é criada uma estrutura **self.nomedavariavel** que recebe como atributo o dado ou valor inserido pelo usuário, respeitando inclusive, a ordem em que foram parametrizados.

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



```
pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)
```

Dessa forma, é criada a variável **pessoa1** que recebe como atributo a classe **Pessoa**, que por sua vez tem os parâmetros **'Fernando', 32, 'M', 1.90**. Estes parâmetros serão substituídos internamente, ordenadamente, pelos campos referentes aos mesmos. O **self** nunca receberá atribuição ou será substituído, ele é apenas a referência de que estes dados serão guardados dentro e para a classe, mas para **nome** será atribuído **'Fernando'**, para **idade** será atribuído **32**, para o **sexo** será atribuído **M** e por fim para a **altura** será atribuído o valor de **1.90**.

```
pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)
```

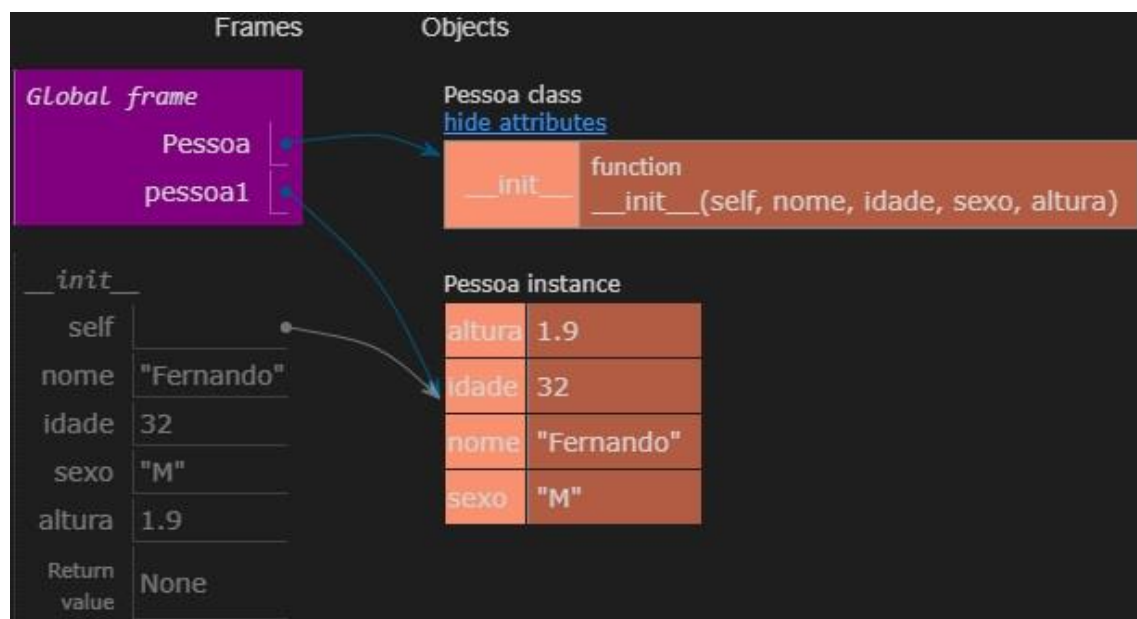
```
print(pessoa1.nome, pessoa1.idade)
```

A partir deste momento, podemos normalmente instanciar qualquer dado ou valor que está guardado na classe **Pessoa**. Por meio da função **print()** passando como parâmetros **pessoa1.nome** e **pessoa1.idade**, o retorno será: **Fernando, 32**.

```
print(f'Bem vindo {pessoa1.nome}, parabéns pelos seus {pessoa1.idade} anos!!!')
```

Apenas a nível de curiosidade, a partir do momento que temos variáveis internas a uma classe, com seus devidos atributos, podemos usá-los normalmente de fora da classe, no exemplo acima, criando uma mensagem usando de **f strings** e máscaras de substituição normalmente. Neste caso o retorno será: **Bem vindo Fernando, parabéns pelos seus 32 anos!!!**

A questão da declaração de parâmetros, que internamente farão a composição das variáveis homônimas, devem respeitar a estrutura **self.nomedavariavel**, porém dentro do mesmo escopo, da mesma indentação, é perfeitamente normal criar variáveis independentes para guardar algum dado ou valor dentro de si.



Escopo / Indentação de uma classe

Já que uma classe é uma estrutura que pode guardar qualquer tipo de bloco de código dentro de si, é muito importante respeitarmos a indentação correta desses blocos de código, assim como entender que, exatamente como funciona na programação convencional estruturada, existem escopos os quais tornam itens existentes dentro do mesmo bloco de código que podem ou não estar acessíveis para uso em funções, seja dentro ou fora da classe. Ex:

```
class Pessoa:
    administrador = 'Admin'

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

        msg = 'Classe Pessoa em execução.'
        print(msg)

    def metodo1(self):
        print(msg)
        pass

var1 = Pessoa('Fernando')
```

Neste momento foque apenas no entendimento do escopo, o bloco de código acima tem estruturas que estaremos entendendo a lógica nos capítulos seguintes.

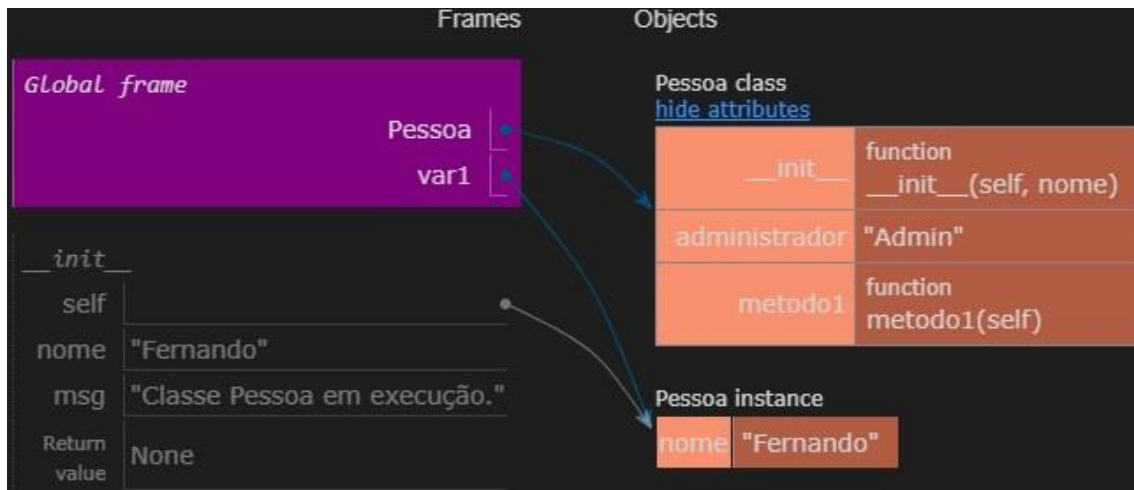
Inicialmente é criada uma classe de nome **Pessoa**, dentro dela existe a variável de nome **administrador** com sua respectiva atribuição **'Admin'**, esta variável de acordo com sua localização indentada estará disponível para uso de qualquer função dentro ou fora dessa classe. Em seguida temos o método construtor que simplesmente recebe um nome atribuído pelo usuário e exibe uma mensagem padrão conforme declarado pela variável **msg**. Na sequência é criada uma função chamada **método1** que dentro de si apenas possui o comando para exibir o conteúdo de **msg**, porém, note que **msg** é uma variável dentro do escopo do método construtor, e em função disso ela só pode ser acessada dentro deste bloco de código. Por fim, fora da classe é criada uma variável de nome **var1** que instancia a classe **Pessoa** passando **'Fernando'** como o parâmetro a ser substituído em **self.nome**.

```
print(var1)
```

Por meio da função **print()** passando como parâmetro **var1**, o retorno será a exibição de **msg**, uma vez que é a única linha de código que irá retornar algo ao usuário. Sendo assim o retorno será: **Classe Pessoa em execução.**

```
print(var1.metodo1())
```

Já ao tentarmos executar **msg** dentro da função **metodo1**, o retorno será **NameError: name 'msg' is not defined**, porque de fato, **msg** é uma variável interna de **__init__**, inacessível para outras funções da mesma classe.



Apenas concluindo o raciocínio, a variável **msg**, “solta” dentro do bloco de código construtor, somente é acessível e instanciável neste mesmo bloco de código. Porém, caso quiséssemos tornar a mesma acessível para outras funções dentro da mesma classe ou até mesmo fora dela, bastaria reformular o código e indexa-la ao método de classe por meio da sintaxe **self.msg**, por exemplo.

```
class Pessoa:
    administrador = 'Admin'

    def __init__(self, nome, msg):
        self.nome = nome
        self.msg = msg
        print(msg)

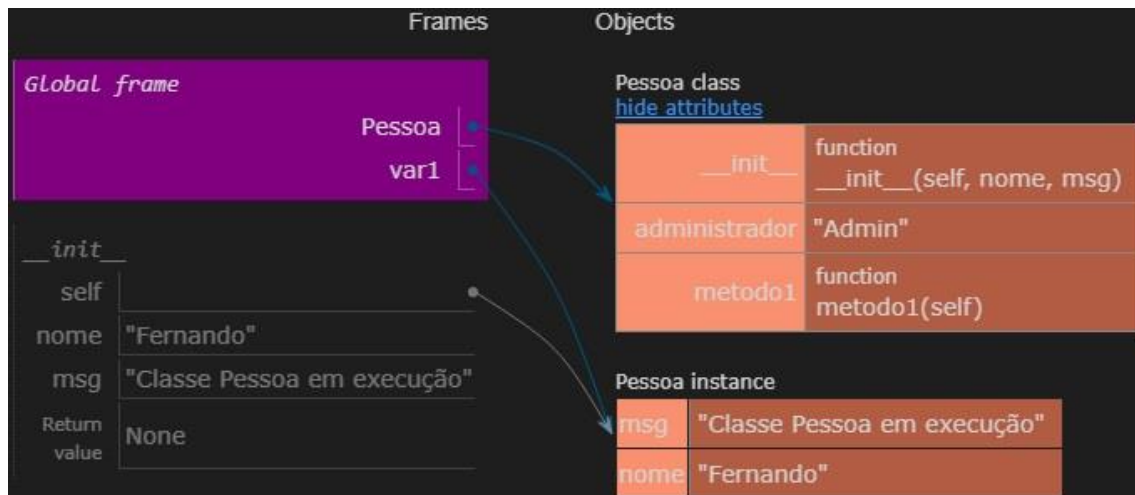
    def metodo1(self):
        print(msg)
        pass

var1 = Pessoa('Fernando', 'Classe Pessoa em execução')
```

Note que agora **msg** é um dos parâmetros a ser repassados pelo usuário no momento da criação da variável que instancia a classe **Pessoa**.

```
print(var1.metodo1)
```

Dessa forma, por meio da função **print()** fora da classe conseguimos perfeitamente instanciar tal variável uma vez que agora internamente a classe existe a comunicação do método construtor e da função **metodo1**. De imediato isto pode parecer bastante confuso porém por hora o mais importante é começar a entender as possibilidades que temos quando se trabalha fazendo o uso de classes, posteriormente estaremos praticando mais tais conceitos e dessa forma entendendo definitivamente sua lógica.



Apenas finalizando o entendimento de escopo, caso ainda não esteja bem entendido, raciocine da seguinte forma:

```
class Pessoa:
    pessoa1 = 'Admin'
    #pessoa1 faz parte do escopo global da classe
    #pessoa1 é instanciável por qualquer método

    def __init__(self, pessoa2):
        self.pessoa2 = pessoa2
        #pessoa2 faz parte do escopo do método construtor
        #pessoa2 é acessível dentro e fora deste método
        #pessoa2 pode ser instanciada de fora da classe

        pessoa3 = 'DefaultUser'
        #pessoa3 faz parte do escopo do método construtor
        #pessoa3 é acessível somente dentro deste método
```

Por fim somente fazendo um adendo, é muito importante você evitar o uso de variáveis/objetos de classe de mesmo nome, ou ao menos ter bem claro qual é qual de acordo com seu escopo.

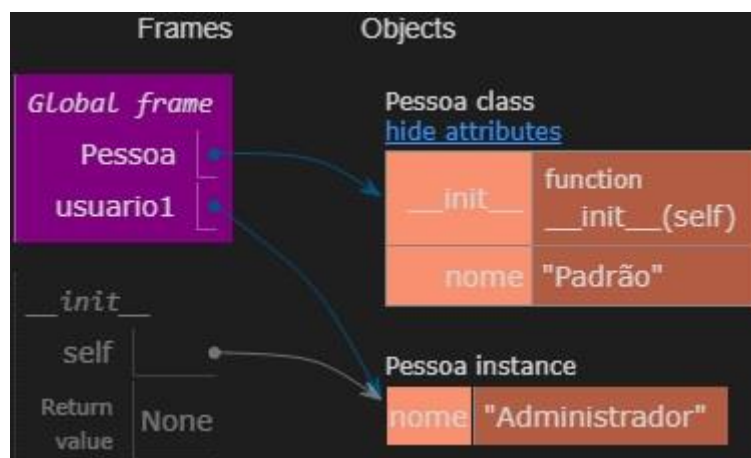
```
class Pessoa:
    nome = 'Padrão'

    def __init__(self):
        self.nome = 'Administrador'

usuario1 = Pessoa()

print(usuario1.nome)
print(Pessoa.nome)
```

Apenas simulando uma situação dessas, note que dentro da classe **Pessoa** existe o objeto de classe **nome**, de atributo '**Padrão**', em seguida existe um método construtor que dentro de si também possui um objeto de classe **nome**, agora de atributo '**Administrador**'. Por mais estranho que pareça sempre que há um método construtor dentro de uma classe, esse é lido e interpretado por primeiro, ignorando até mesmo linhas de código anteriores a ele. Pela leitura léxica do interpretador, é feita a leitura linha após linha, porém a prioridade é processar o método construtor dentro da estrutura da classe. Na sequência é criada uma variável que instancia a classe **Pessoa**. Por meio da nossa função **print()**, passando como parâmetro **usuario1.nome** o retorno será **Administrador** (instância do método construtor), passando como parâmetro **Pessoa.nome**, o retorno será **Padrão** (objeto de classe). Então, para evitar confusão é interessante simplesmente evitar criar objetos de classe de mesmo nome, ou ao menos não confundir o uso dos mesmos na hora de incorporá-los ao restante do código.



Classe com parâmetros opcionais

Outra situação bastante comum é a de definirmos parâmetros (atributos de classe) que serão obrigatoriamente substituídos e em contraponto parâmetros que opcionalmente serão substituídos. Como dito anteriormente, uma classe pode servir de “molde” para criar diferentes objetos a partir dela, e não necessariamente todos esses objetos terão as mesmas características. Sendo assim, podemos criar parâmetros que opcionalmente farão parte de um objeto enquanto não farão parte de outro por meio de não atribuição de dados ou valores para esse parâmetro. Basicamente isto é feito simplesmente declarando que os parâmetros que serão opcionais possuem valor inicial definido como **False**. Ex:

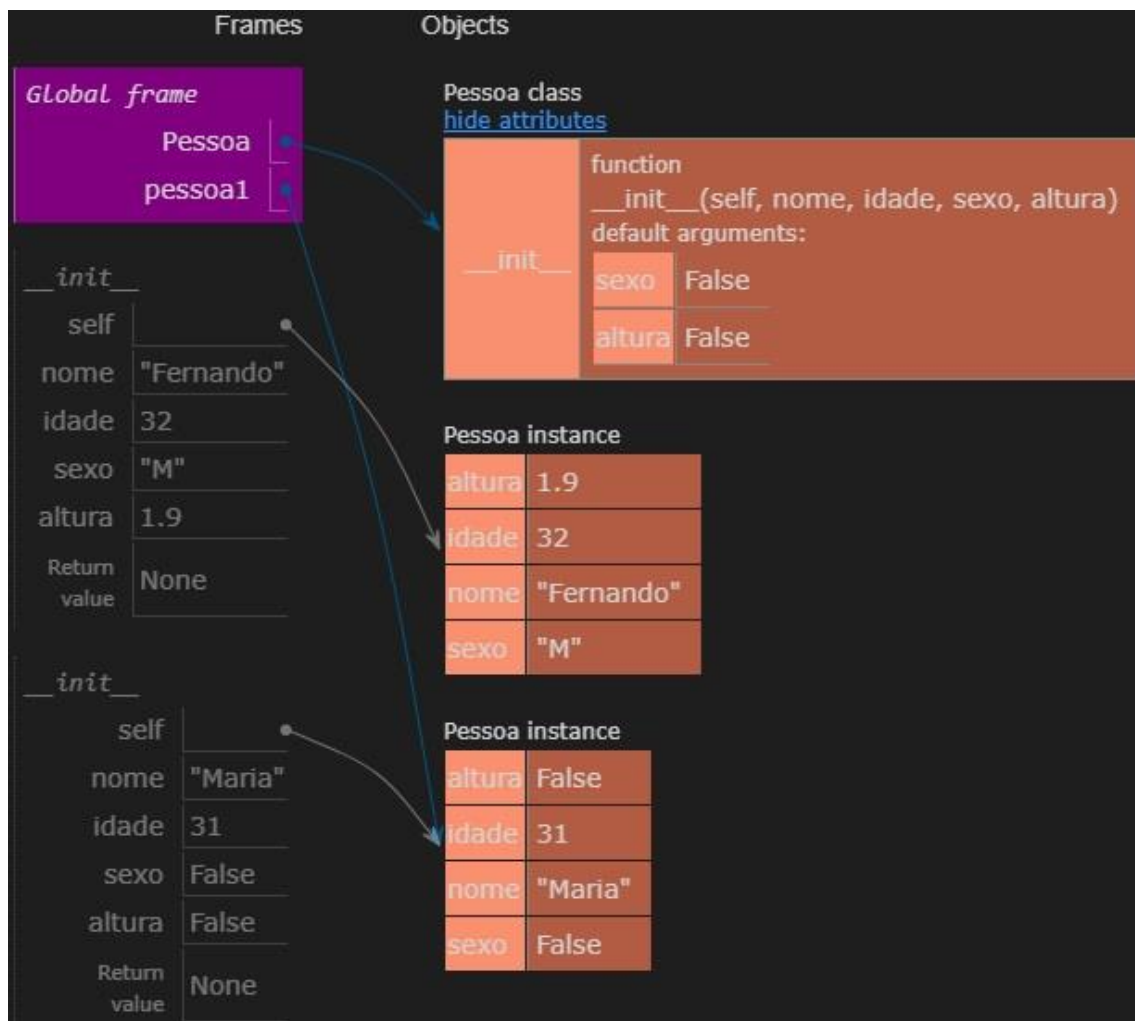
```
class Pessoa:
    def __init__(self, nome, idade, sexo=False, altura=False):
        self.nome = nome
        self.idade = idade
        self.sexo = sexo
        self.altura = altura
```

Repare que estamos na mesma estrutura de código do exemplo anterior, porém dessa vez, declaramos que **sexo=False** assim como **altura=False**, o que faz com que esses parâmetros sejam inicializados como **None**. Se o usuário quiser atribuir dados ou valores a estes dados, simplesmente **False** será substituído pelo referente dado/valor, uma vez que **False** é uma palavra reservada ao sistema indicando que neste estado o interpretador ignore essa variável.

```
pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)
pessoa2 = Pessoa('Maria', 31)

print(pessoa1.nome, pessoa1.altura)
print(pessoa2.nome, pessoa2.idade)
```

Dessa forma, continuamos trabalhando normalmente atribuindo ou não dados/valores sem que haja erro de interpretador. Neste caso o retorno será: **Fernando, 1.90. Maria, 31.**



Múltiplos métodos de classe

Uma prática comum é termos dentro de uma classe diversas variáveis assim como mais de uma função (métodos de classe), tudo isso atribuído a um objeto que é “dono” dessa classe. Dessa forma, podemos criar também interações entre variáveis com variáveis e entre variáveis com funções, simplesmente respeitando a indentação dos mesmos dentro da hierarquia estrutural da classe. Por exemplo:

```
class Pessoa:
    ano_atual = 2019

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

Inicialmente criamos a classe **Pessoa**, dentro dela, criamos uma variável de nome **ano_atual** que recebe como atributo o valor fixo **2019**, **ano_atual** está no escopo geral da classe, sendo acessível a qualquer função interna da mesma. Em seguida, é criado um construtor onde simplesmente é definido que haverão parâmetros para **nome** e **idade** fornecidos pelo usuário, estes, serão atribuídos a variáveis homônimas internas desta função.

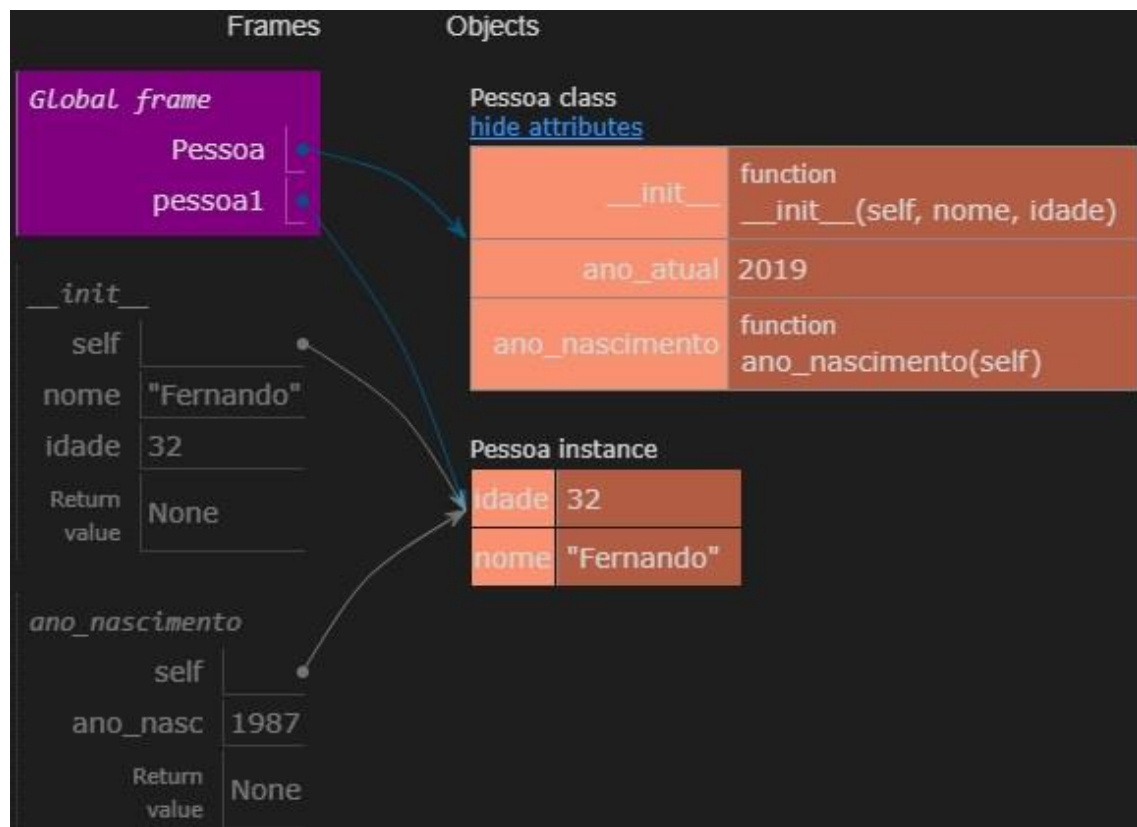
```
def __init__(self, nome, idade):
    self.nome = nome
    self.idade = idade

def ano_nascimento(self):
    ano_nasc = self.ano_atual - self.idade
    print(f'Seu ano de nascimento é {ano_nasc}')
```

Em seguida é criada a função **ano_nascimento**, dentro de si é criada a variável **ano_nasc** que por sua vez faz uma operação instanciando e subtraindo os valores de **ano_atual** e **idade**. Note que esta operação está instanciando a variável **ano_atual** que não faz parte do escopo nem do construtor nem desta função, isto é possível porque **ano_atual** é integrante do escopo geral da classe e está disponível para qualquer operação dentro dela.

```
pessoa1 = Pessoa('Fernando', 32)
print(pessoa1.ano_nascimento())
```

Na sequência é criada a variável **pessoa1** que por sua vez inicializa a classe **Pessoa** passando como parâmetros **‘Fernando’**, **32**. Por fim simplesmente pela função **print()** é exibida a mensagem resultante do cruzamento desses dados. Nesse caso o retorno será: **Seu ano de nascimento é 1987**.



Interação entre métodos de classe

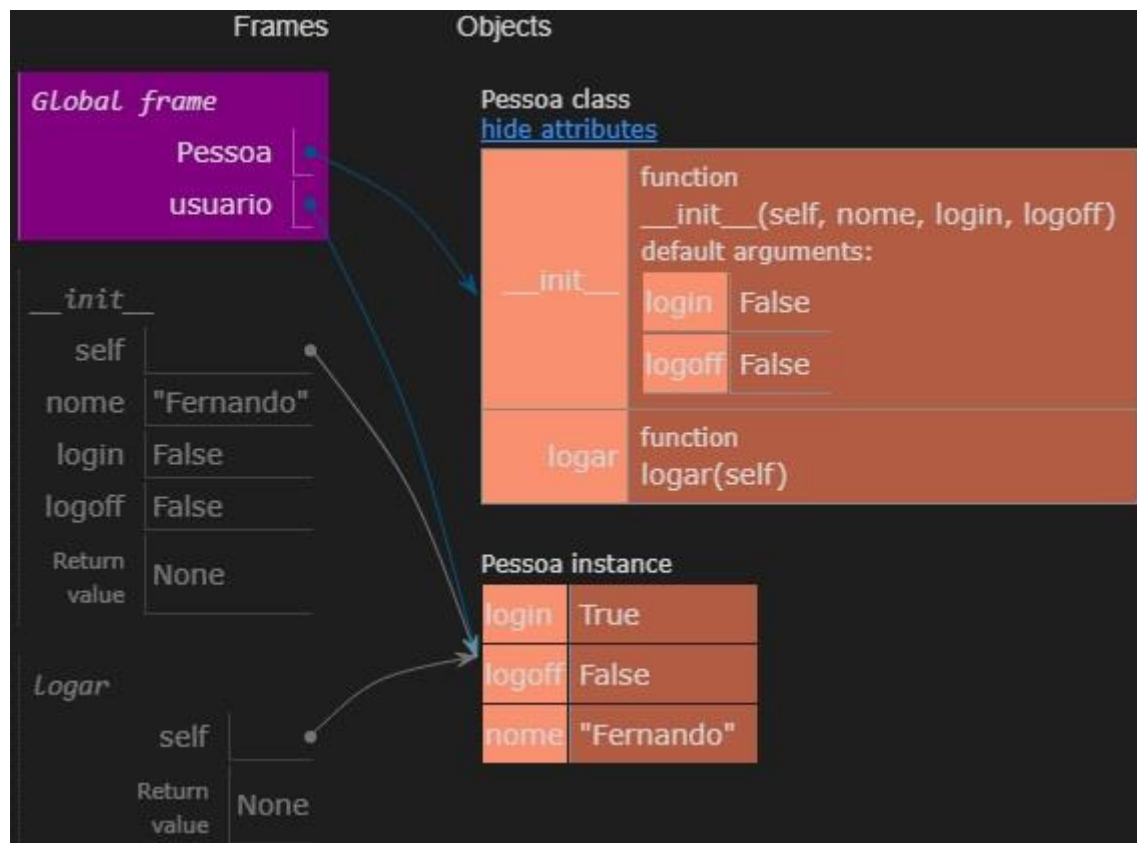
Aprofundando um pouco nossos estudos, podemos criar um exemplo que engloba praticamente tudo o que foi visto até então. Criaremos uma classe, com parâmetros opcionais alteráveis, que internamente realiza a interação entre métodos/funções para retornar algo ao usuário.

```
class Pessoa:
    def __init__(self, nome, login=False, logoff=False):
        self.nome = nome
        self.login = login
        self.logoff = logoff

    def logar(self):
        print(f'Bem vindo {self.nome}, você está logado no sistema.')
        self.login = True

usuario = Pessoa('Fernando')
usuario.logar()
```

Inicialmente criamos a classe **Pessoa**, em seguida criamos o método construtor da mesma **__init__** que recebe um **nome** atribuído pelo usuário e possui variáveis temporárias reservadas para **login** e **logoff**, por hora desabilitadas. Em seguida criamos as respectivas variáveis internas instanciadas para o construtor por meio da sintaxe **self.nomedavariavel**. Na sequência criamos uma função **logar** que simplesmente exibe em tela uma **f string** com máscara de substituição onde consta o nome atribuído a variável **nome** do método construtor, assim como atualiza o status de **self.login** de **False** para **True**. Fora da classe criamos uma variável chamada **usuario** que instancia a classe **Pessoa** passando como parâmetro **'Fernando'**, nome a ser substituído na variável homônima. Por fim, dando o comando de **usuario.logar()**, chamando a função interna **logar** da classe **Pessoa**, o retorno será: **Bem vindo Fernando, você está logado no sistema.**



Estruturas condicionais em métodos de classe

Apenas um adendo a este tópico, pelo fato de que é bastante comum quando estamos aprendendo este tipo de abstração em programação esquecermos de criar estruturas condicionais ou validações em nosso código. Pegando como exemplo o código acima, supondo que fossem dados vários comandos `print()` sob a mesma função, ocorreria a repetição pura da mesma, o que normalmente não fará muito sentido dependendo da aplicação deste código. Então, lembrando do que foi comentado anteriormente, dentro de uma classe podemos fazer o uso de qualquer estrutura de código, inclusive estruturas condicionais, laços de repetição e validações. Por exemplo:

```
class Pessoa:
    def __init__(self, nome, login=False, logoff=False):
        self.nome = nome
        self.login = login
        self.logoff = logoff

    def logar(self):
        print(f'Bem vindo {self.nome}, você está logado no sistema.')
        self.login = True

usuario = Pessoa('Fernando')
usuario.logar()
usuario.logar()
```

O retorno será: **Bem vindo Fernando, você está logado no sistema.**

Bem vindo Fernando, você está logado no sistema.

Porém podemos criar uma simples estrutura que evita este tipo de erro, por meio de uma estrutura condicional que por sua vez para a execução do código quando seu objetivo for alcançado.

```
class Pessoa:
    def __init__(self, nome, login=False, logoff=False):
        self.nome = nome
        self.login = login
        self.logoff = logoff

    def logar(self):
        if self.login:
            print(f'{self.nome} já está logado no sistema')
            return

        print(f'Bem vindo {self.nome}, você está logado no sistema.')
        self.login = True

usuario = Pessoa('Fernando')
```

```
usuario.logar()  
usuario.logar()
```

Note que apenas foi criado uma condição dentro do método/função **logar** que, se **self.login** tiver o status como **True**, é exibida a respectiva mensagem e a execução do código para a partir daquele ponto. Então, simulando um erro de lógica por parte do usuário, pedindo para que usuário faça duas vezes a mesma coisa, o retorno será:

Bem vindo Fernando, você está logado no sistema.
Fernando já está logado no sistema

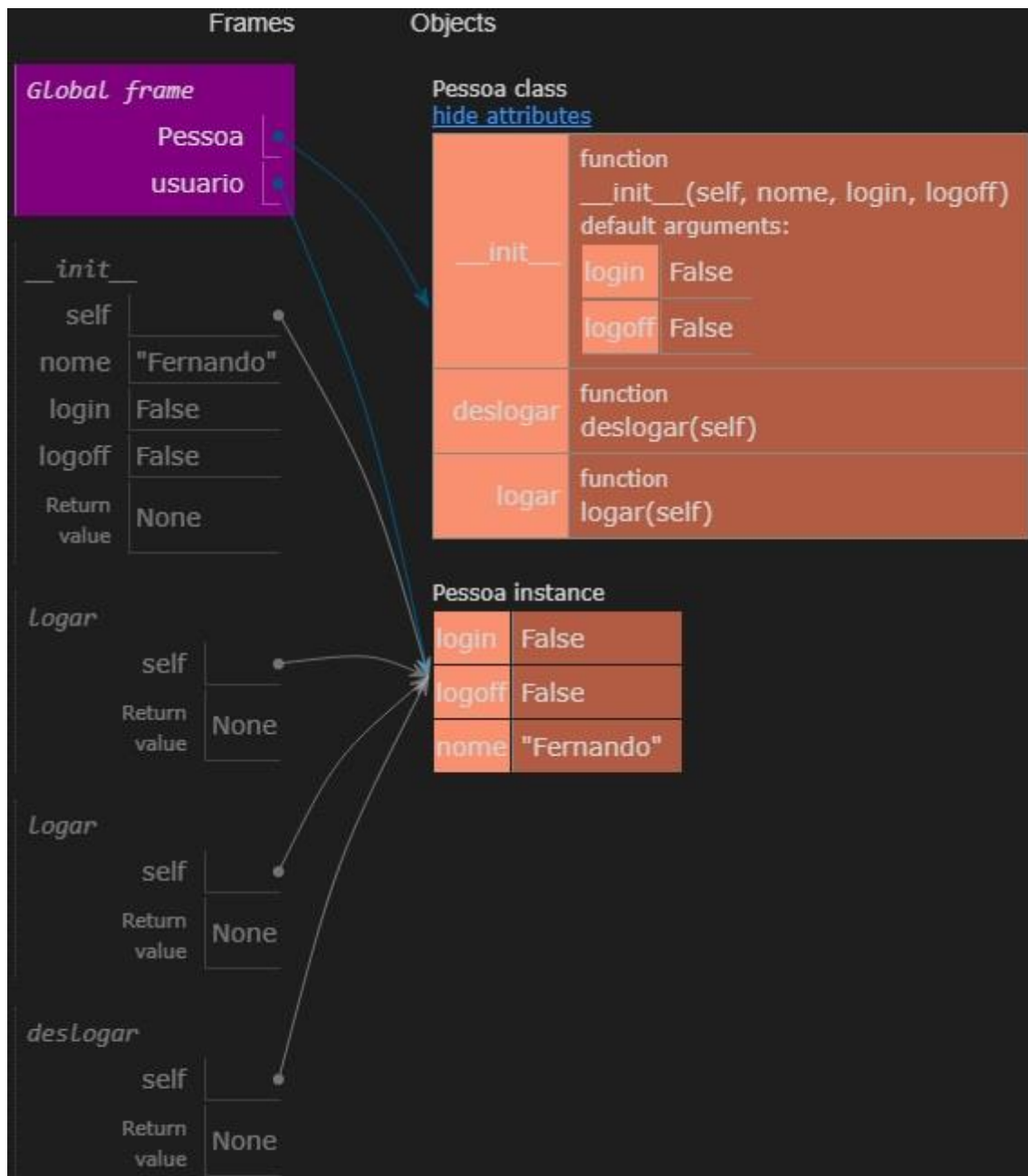
```
class Pessoa:  
    def __init__(self, nome, login=False, logoff=False):  
        self.nome = nome  
        self.login = login  
        self.logoff = logoff  
  
    def logar(self):  
        if self.login:  
            print(f'{self.nome} já está logado no sistema')  
            return  
  
        print(f'Bem vindo {self.nome}, você está logado no siste  
ma.')  
        self.login = True  
  
    def deslogar(self):  
        if not self.login:  
            print(f'{self.nome} não está logado no sistema')  
            return  
        print(f'{self.nome} foi deslogado do sistema')  
        self.login = False
```

Apenas finalizando nossa linha de raciocínio para este tipo de código, anteriormente havíamos criado o atributo de classe **logoff** porém ainda não havíamos lhe dado um uso em nosso código. Agora simulando que existe uma função específica para deslogar o sistema, novamente o que fizemos foi criar uma função **deslogar** onde consta uma estrutura condicional de duas validações, primeiro ela chega se o usuário não está logado no sistema, e a segunda, caso o usuário esteja logado, o desloga atualizando o status da variável **self.login** e exibe a mensagem correspondente.

```
usuario = Pessoa('Fernando')  
usuario.logar()  
usuario.logar()  
usuario.deslogar()
```

Novamente simulando o erro, o retorno será:

Bem vindo Fernando, você está logado no sistema.
Fernando já está logado no sistema
Fernando foi deslogado do sistema



Métodos de classe estáticos e dinâmicos

Outra aplicação muito usada, embora grande parte das vezes implícita ao código, é a definição manual de um método de classe quanto a sua visibilidade dentro do escopo da classe. Já vimos parte disso em tópicos anteriores, onde aprendemos que dependendo da indentação do código é criada uma hierarquia, onde determinadas variáveis dentro de funções/métodos de classe podem ser acessíveis e instanciáveis entre funções ou não, dependendo onde a mesma está declarada.

Em certas aplicações, é possível definir esse status de forma manual, e por meio deste tipo de declaração manual é possível fixar o escopo onde o método de classe irá retornar algum dado ou valor. Em resumo, você pode definir manualmente se você estará criando objetos que utilizam de tudo o que está no escopo da classe, ou se o mesmo terá suas próprias atribuições isoladamente.

```
class Pessoa:
    ano_atual = 2019

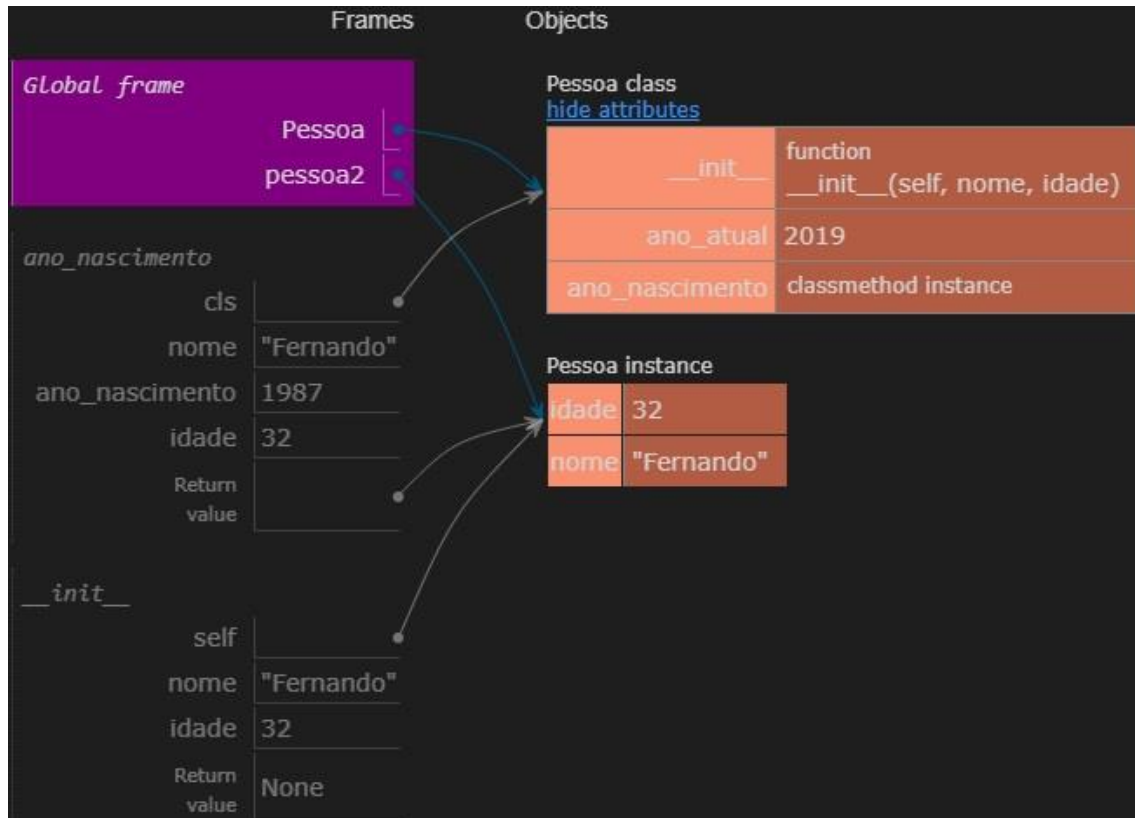
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    @classmethod
    def ano_nascimento(cls, nome, ano_nascimento):
        idade = cls.ano_atual - ano_nascimento
        return cls(nome, idade)
```

Da mesma forma que as outras vezes, todo o processo se inicia com a criação da classe, note que estamos reutilizando o exemplo onde calculamos a idade do usuário visto anteriormente. Inicialmente criada a classe **Pessoa**, dentro dela é criada a variável **ano_atual** com **2019** como valor atribuído, logo após, criamos o método construtor da classe que receberá do usuário um **nome** e uma **idade**. Em seguida declaramos manualmente que a classe a seguir é dinâmica, por meio de **@classmethod** (sintaxe igual a de um decorador), o que significa que dentro desse método/função, será realizada uma determinada operação lógica e o retorno da mesma é de escopo global, acessível e utilizável por qualquer bloco de código dentro da classe Pessoa. Sendo assim, simplesmente criamos o método/função **ano_nascimento** que recebe como parâmetro **cls**, **nome**, **ano_nascimento**. Note que pela primeira vez, apenas como exemplo, não estamos criando uma função em **self**, aqui é criada uma variável a critério do programador, independente, que será instanciável no escopo global e internamente será sempre interpretada pelo interpretador como **self**. Desculpe a redundância, mas apenas citei este exemplo para demonstrar que **self**, o primeiro parâmetro de qualquer classe, pode receber qualquer nomenclatura.

```
pessoa2 = Pessoa.ano_nascimento('Fernando', 1987)
print(pessoa2.idade)
```

Por fim, repare que é criada uma variável de nome **pessoa2** que instancia **Pessoa** e repassa atributos diretamente para o método **ano_nascimento**. Se não houver nenhum erro de sintaxe ocorrerão internamente o processamento das devidas funções assim como a atualização dos dados em todos métodos de classe. Neste caso, por meio da função **print()** que recebe como parâmetro **pessoa2.idade**, o retorno será: **32**



Entendido o conceito lógico de um método dinâmico (também chamado método de classe), hora de entender o que de fato é um método estático. Raciocine que todos métodos de classe / funções criadas até agora, faziam referência ao seu escopo (**self**) ou instância, assim como reservavam espaços para variáveis com atributos fornecidos pelo usuário. Ao contrário disso, se declararmos um método que não possui instâncias como atributos, o mesmo passa a ser um método estático, como uma simples função dentro da classe, acessível e instanciável por qualquer objeto. Um método estático, por sua vez, pode ser declarado manualmente por meio da sintaxe **@staticmethod** uma linha antes do método propriamente dito.

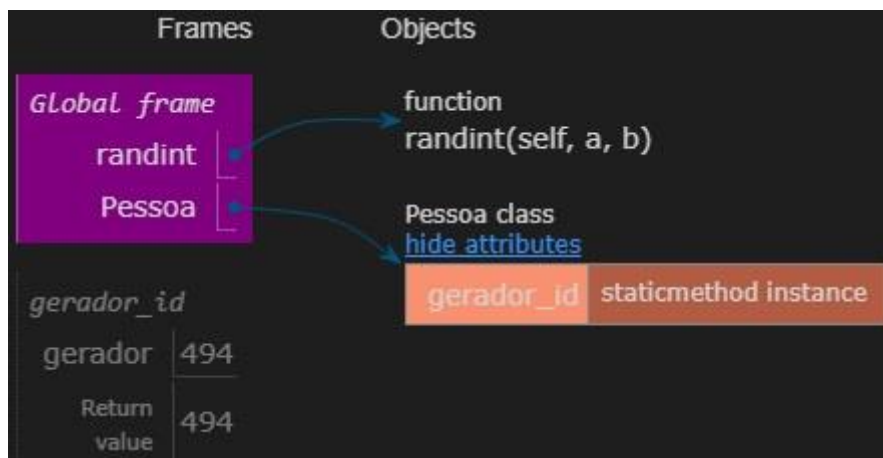
```
from random import randint

class Pessoa:
    @staticmethod
    def gerador_id():
        gerador = randint(100, 999)
        return gerador
```



```
print(Pessoa.gerador_id())
```

Mudando um pouco de exemplo, inicialmente importamos da biblioteca **random** o módulo **randint**, muito usado quando é necessário se gerar números inteiros aleatórios. Em seguida é criada a classe **Pessoa**, é declarada que o método **gerador_id()** dentro dela é um método estático pelo **@staticmethod**. O método **gerador_id()** por sua vez simplesmente possui uma variável de nome **gerador** que inicializa o módulo **randint** especificando que seja gerado um número aleatório entre 100 e 999, retornando esse valor para **gerador**. Por fim, por meio da função **print()** podemos diretamente instanciar como parâmetro a classe **Pessoa** executando seu método estático **gerador_id()**. Neste caso o retorno será um número gerado aleatoriamente entre 100 e 999.



Getters e setters

Anteriormente vimos rapidamente que é perfeitamente possível criarmos estruturas condicionais em nossos métodos de classe, para que dessa forma tenhamos estruturas de código que tentam contornar as possíveis exceções cometidas pelo usuário (normalmente essas exceções são inserções de tipos de dados não esperados pelo interpretador). Aprofundando um pouco mais dentro desse conceito, podemos criar estruturas de validação que contornem erros em atributos de uma classe. Pela nomenclatura usual, um **Getter** obtém um determinado dado/valor e um **Setter** configura um determinado dado/valor que substituirá o primeiro.

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

produto1 = Produto('Processador', 370)

print(produto1.preco)
```

Como sempre, o processo se inicia com a criação de nossa classe, nesse caso, de nome **Produto**, dentro dela criamos um construtor onde basicamente iremos criar objetos fornecendo o **nome** e o **preço** do mesmo para suas respectivas variáveis. De fora da classe, criamos uma variável de nome **produto1** que recebe como atributo a classe **Produto** passando como atributos **'Processador', 370**. Por meio do comando **print()** podemos exibir por exemplo, o preço de **produto1**, por meio de **produto1.preco**. Neste caso o retorno será **370**

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

    def desconto(self, percentual):
        self.preco = self.preco - (self.preco*(percentual/100))

produto1 = Produto('Processador', 370)
produto1.desconto(15)

produto2 = Produto('Placa Mãe', 'R$280')
produto2.desconto(20)

print(produto1.preco)
print(produto2.preco)
```

Na sequência adicionamos um método de classe de nome **desconto** responsável por aplicar um desconto com base no percentual definido pelo usuário. Também cadastramos um segundo produto e note que aqui estamos simulando uma exceção no

valor atribuído ao preço do mesmo. Já que será executada uma operação matemática para aplicar desconto sobre os valores originais, o interpretador espera que todos dados a serem processados sejam do tipo **int** ou **float** (numéricos). Ao tentar executar tal função ocorrerá um erro: **TypeError: can't multiply sequence by non-int of type 'float'**.

Em tradução livre: Erro de tipo: Não se pode multiplicar uma sequência de não-inteiros pelo tipo 'float' (número com casas decimais).

Sendo assim, teremos de criar uma estrutura de validação que irá prever esse tipo de exceção e contornar a mesma, como estamos trabalhando com orientação a objetos, mais especificamente tendo que criar um validador dentro de uma classe, isto se dará por meio de **Getters** e **Setters**.

```
#Getter
@property
def preco(self):
    return self.preco_valido

#Setter
@preco.setter
def preco(self, valor):
    if isinstance(valor, str):
        valor = float(valor.replace('R$', ''))
    self.preco_valido = valor
```

Prosseguindo com nosso código, indentado como método de nossa classe **Pessoa**, iremos incorporar ao código a estrutura acima. Inicialmente criamos o que será nosso **Getter**, um decorador (que por sua vez terá prioridade 1 na leitura do interpretador) que cria uma função **preco**, simplesmente obtendo o valor atribuído a **preco** e o replicando em uma nova variável de nome **preco_valido**. Feito isso, hora de criarmos nosso **Setter**, um pouco mais complexo, porém um tipo de validador funcional. Inicialmente criamos um decorador de nome **@preco.setter** (esse decorador deverá ter o nome da variável em questão, assim como a palavra reservada **.setter**). Então criamos a função **preco** que recebe um **valor**, em seguida é criada uma estrutura condicional onde, se o **valor** da instância for do tipo **str**, valor será convertido para **float** assim como os caracteres desnecessários serão removidos. Por fim, **preco_valido** é atualizado com o valor de **valor**.

Código completo:

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

    def desconto(self, percentual):
        self.preco = self.preco - (self.preco*(percentual/100))
```

```

@property
def preco(self):
    return self.preco_valido

@preco.setter
def preco(self, valor):
    if isinstance(valor, str):
        valor = float(valor.replace('R$', ''))
    self.preco_valido = valor

produto1 = Produto('Processador', 370)
produto1.desconto(15)
produto2 = Produto('Placa Mãe', 'R$280')
produto2.desconto(20)

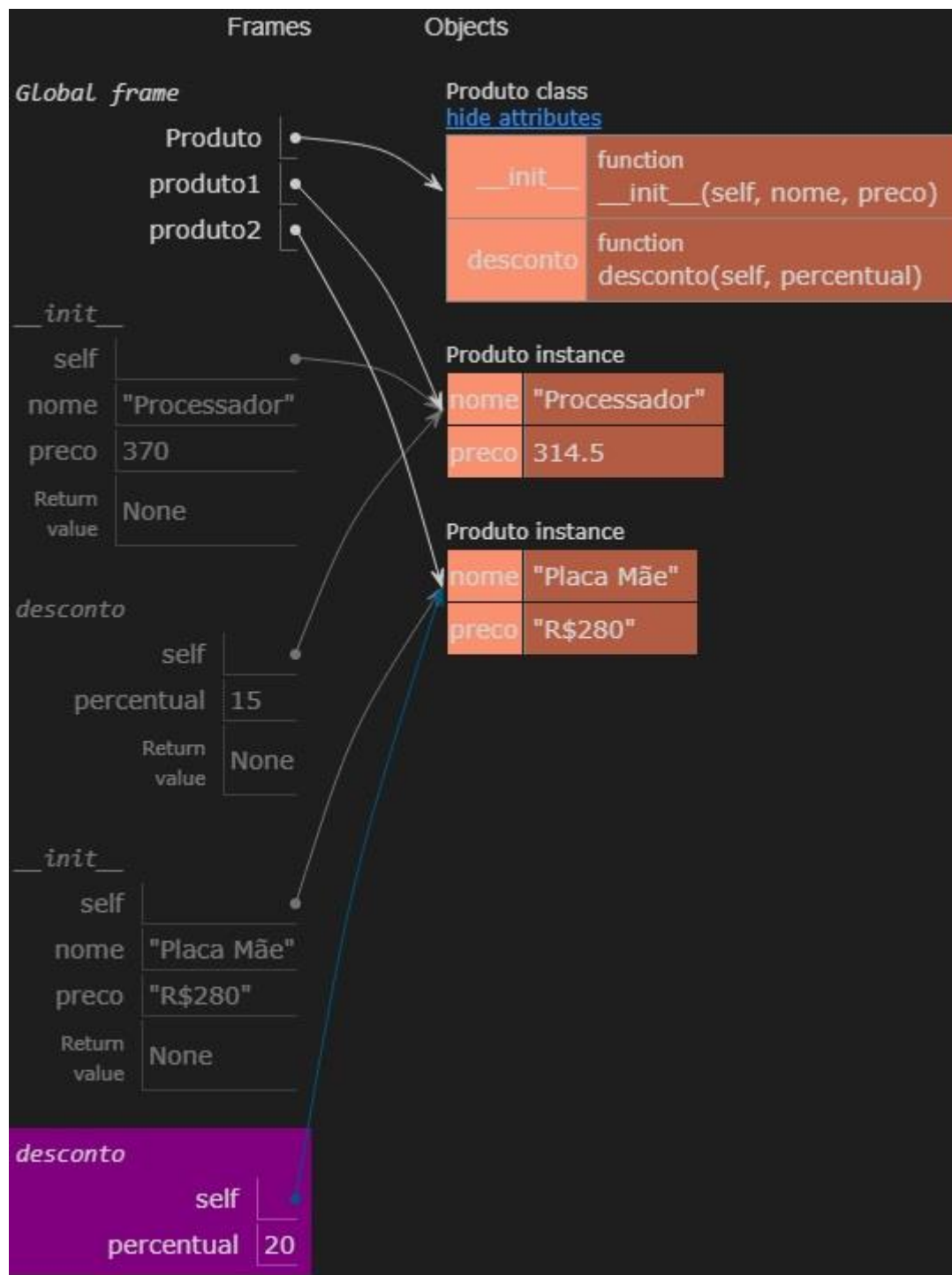
print(produto1.preco)
print(produto2.preco)

```

Realizadas as devidas correções e validações, é aplicado sobre o preço de **produto1** um desconto de 15%, e sobre o preço de **produto2** 20%. Desta forma, o retorno será de:

314.5

224.0



Encapsulamento

Se você já programa em outras linguagens está habituado a declarar manualmente em todo início de código, se o bloco de código em questão é de acesso público ou privado. Esta definição em Python é implícita, podendo ser definida manualmente caso haja necessidade. Por hora, se tratando de programação orientada a objetos, dependendo da finalidade de cada bloco de código, é interessante você definir as permissões de acesso aos mesmos no que diz respeito a sua leitura e escrita.

Digamos que existam determinadas classes acessíveis e mutáveis de acordo com a finalidade e com sua interação com o usuário, da mesma forma existirão classes com seus respectivos métodos que devem ser protegidos para que o usuário não tenha acesso total, ou ao menos, não tenha como modifica-la. Raciocine que classes e funções importantes podem e devem ser modularizadas, assim como encapsuladas de acordo com sua finalidade, e na prática isto é feito de forma mais simples do que você imagina, vamos ao exemplo:

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

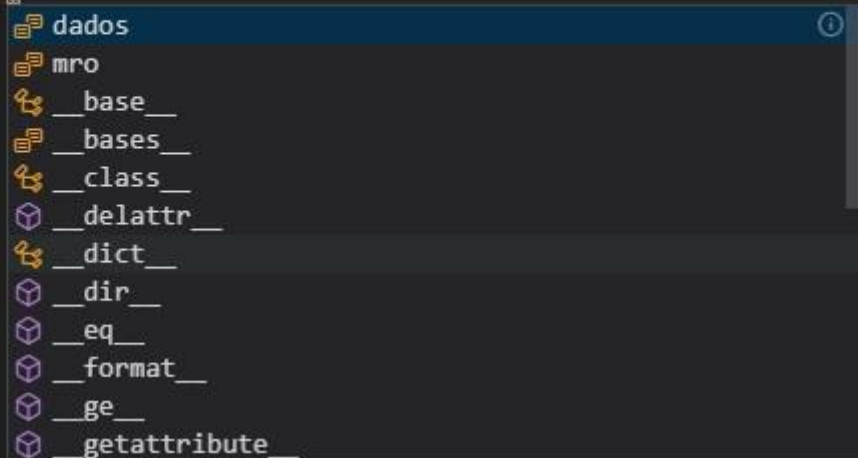
base = BaseDeDados()
print(base.dados)
```

Repare que aqui, inicialmente, estamos criando uma classe como já estamos habituados, neste caso, uma classe chamada **BaseDeDados**, dentro da mesma há um construtor onde dentro dele simplesmente existe a criação de uma variável na própria instância que pela sintaxe receberá um dicionário, apenas como exemplo mesmo. Fora da classe é criada uma variável de nome **base** que inicializa a classe **BaseDeDados** assim como um comando para exibir o conteúdo de dados.

O ponto chave aqui é, **self.dados** é uma variável, um atributo de classe que quando declarado sob esta nomenclatura, é visível e acessível tanto dentro dessa instância quanto de fora dela.

```
self.dados = {}

base = BaseDeDados()
print(base.)
```

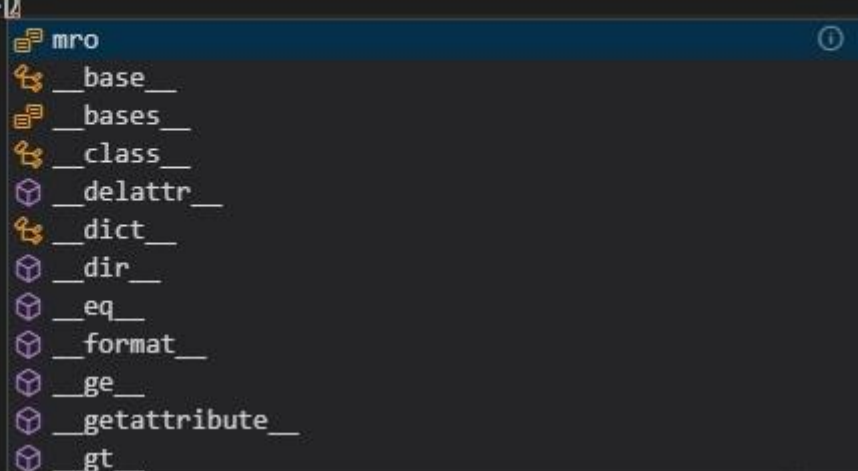


Note que em nossa função **print()** ao passar como parâmetro **base**, sua variável **dados** está disponível para ser instanciada.

A partir do momento que declaramos a variável **dados** pela sintaxe **_dados**, a mesma passa a ser uma variável protegida, não visível, ainda acessível de fora da classe, porém implícita.

```
self._dados = {}

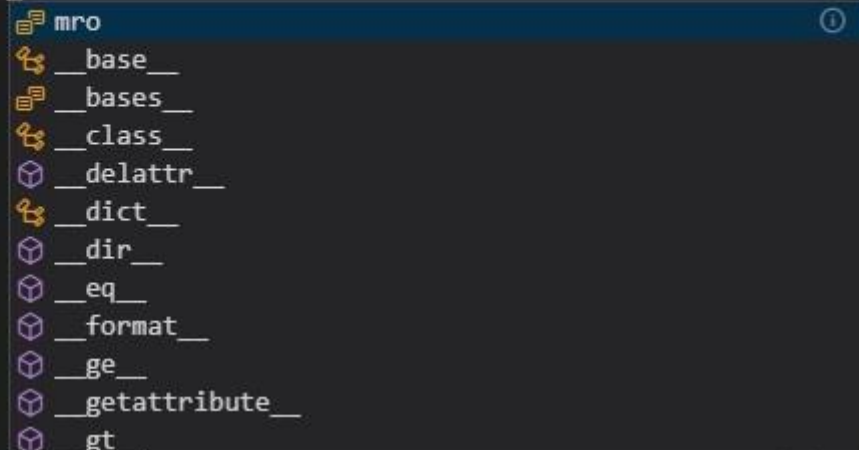
base = BaseDeDados()
print(base.)
```



Repare que ao passar **base** como parâmetro, **dados** não está mais visível, ela pode ser declarada manualmente normalmente, porém supondo que este é um código compartilhado, onde os desenvolvedores tem níveis de acesso diferentes ao código, neste exemplo, **dados** não estaria visível para todos, logo, é um atributo de classe protegido restrito somente a quem sabe sua instância.

```
self.__dados = {}

base = BaseDeDados()
print(base.)
```



Por fim, declarando a variável dados como `__dados`, a mesma passa a ser privada, e dessa forma, a mesma é inacessível e imutável de fora da classe. Lembre-se que toda palavra em Python, com prefixo `__` (underline duplo) é uma palavra reservada ao sistema, e neste caso não é diferente. Em resumo, você pode definir a visibilidade de uma variável por meio da forma com que declara, uma underline torna a mesma protegida, underline duplo a torna privada e imutável. Por exemplo:

```
class BaseDeDados:
    def __init__(self):
        self.base = {}

    def inserir(self, nome, fone):
        if 'clientes' not in self.base:
            self.base['clientes'] = {nome:fone}
        else:
            self.base['clientes'].update({nome:fone})

    def listar(self):
        for nome, fone in self.base['clientes'].items():
            print(nome, fone)

    def apagar(self, nome):
        del self.base['clientes'][nome]
```

Apenas pondo em prática o conceito explicado anteriormente, supondo que tenhamos um sistema comercial onde sua programação foi feita sem as devidas proteções de código realizadas via encapsulamento. Inicialmente é criada uma classe **BaseDeDados** com um método construtor assim como métodos de classe para **inserir**, **listar** e **apagar** clientes de um dicionário, por meio de seus respectivos **nomes** e **telefones**. Note que nenhum encapsulamento foi realizado (assim como nesse caso, apenas como exemplo, esta base de dados não está modularizada), toda e qualquer parte do código neste momento é acessível e alterável por qualquer desenvolvedor.

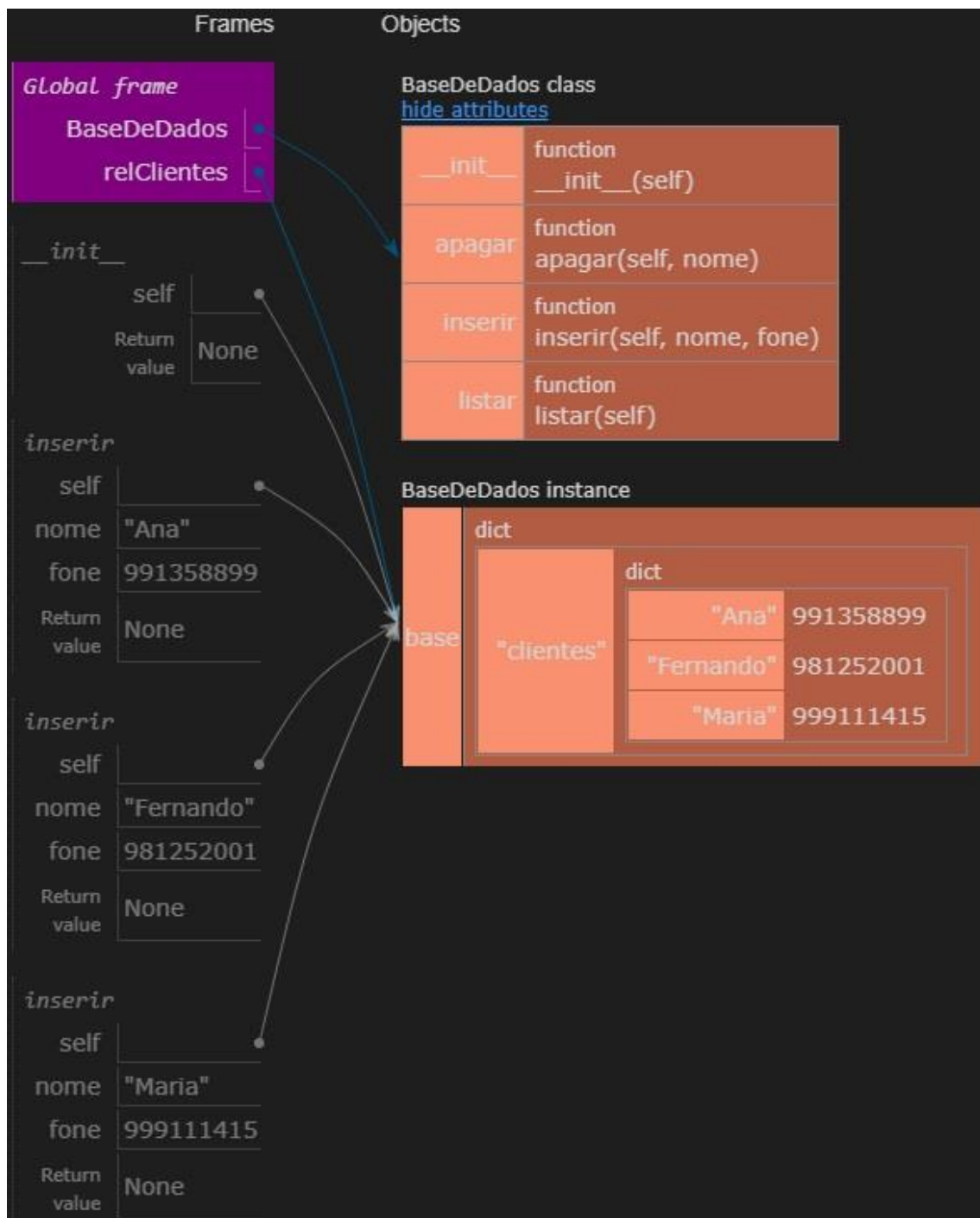

```
relClientes = BaseDeDados()

relClientes.inserir('Ana', 991358899)
relClientes.inserir('Fernando', 981252001)
relClientes.inserir('Maria', 999111415)

relClientes.listar()
```

Em seguida é criada uma variável que instancia **BaseDeDados()**. Dessa forma, é possível por meio dessa variável usar os métodos internos da classe para suas referidas funções, nesse caso, adicionamos 3 clientes a base de dados e em seguida simplesmente pedimos a listagem dos mesmos por meio do método de classe listar. O retorno será:

Ana 991358899
Fernando 981252001
Maria 999111415



Agora simulando uma exceção cometida por um desenvolvedor, supondo que o mesmo pegue como base este código, se em algum momento ele fizer qualquer nova atribuição para a variável **relClientes** o que ocorrerá é que a mesma atualizará toda a classe quebrando assim todo código interno pronto anteriormente.

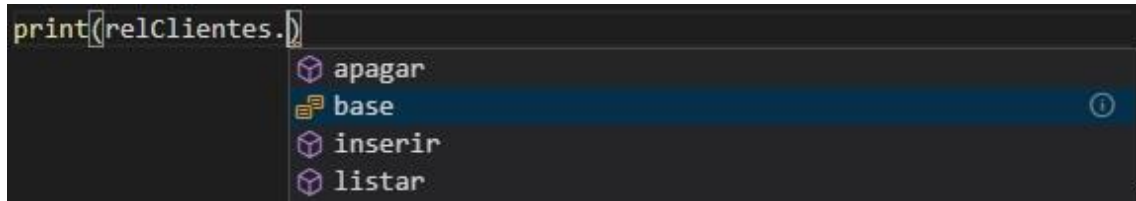
```
relClientes.base = 'Novo Banco de Dados'
relClientes.listar()
```

O retorno será:

main.py", line 12, in listar for nome, fone in self.base['clientes'].items():
TypeError: string indices must be integers

Isto se deu pelo fato de que, como não havia o devido encapsulamento para proteger a integridade da base de dados, a mesma foi simplesmente substituída pelo novo atributo e a partir desse momento, nenhum bloco de código dos métodos de classe criados anteriormente serão utilizados, uma vez que todos eles dependem do núcleo de **self.base** que consta no método construtor da classe.

```
print(relClientes.)
```




Note que até esse momento ao tentar instanciar qualquer coisa de **relClientes**, entre os métodos aparece inclusive a **base**, que deveria ser protegida.

```
class BaseDeDados:
    def __init__(self):
        self.__base = {}
```

Realizando a devida modularização, renomeando todas as referências a **base** para **__base**, a partir deste momento a mesma passa a ser um objeto de classe com as devidas proteções de visualização e contra modificação.

```
print(relClientes.)
```



Executando o mesmo comando **print()** como anteriormente, repare que agora **base** não aparece mais como um objeto instanciável para um usuário desse nível. Ou que ao menos não deveria ser instanciável. Porém, ainda assim é possível aplicar mudanças sobre este objeto, mas como ele está definido de forma a ser imutável, o que acontecerá é que o interpretador irá fazer uma espécie de backup do conteúdo antigo, com núcleo protegido e imutável, que pode ser restaurado a qualquer momento.

```
relClientes.__base = 'Novo Banco de Dados'
print(relClientes.__base)
```

O interpretador verá que **base** é uma variável protegida, e que manualmente você ainda assim está forçando atribuir a mesma um novo dado/valor. Sendo assim, evitando uma quebra de código, ele irá executar esta função. Neste caso, o retorno será **Novo Banco de Dados**.

```
print(relClientes._BaseDeDados__base)
```

Por meio da função **print()**, passando como parâmetro **_BaseDeDados__base** você tem acesso ao núcleo original, salvo pelo interpretador. Dessa forma o resultado será: **{'clientes': {'Ana': 991358899, 'Fernando': 981252001, 'Maria': 999111415}}**

BaseDeDados class									
hide attributes									
__init__	function __init__(self)								
apagar	function apagar(self, nome)								
inserir	function inserir(self, nome, fone)								
listar	function listar(self)								
BaseDeDados instance									
__BaseDeDados__base	dict <table> <tr> <td>"clientes"</td><td>dict <table> <tr> <td>"Ana"</td><td>991358899</td></tr> <tr> <td>"Fernando"</td><td>981252001</td></tr> <tr> <td>"Maria"</td><td>999111415</td></tr> </table> </td></tr> </table>	"clientes"	dict <table> <tr> <td>"Ana"</td><td>991358899</td></tr> <tr> <td>"Fernando"</td><td>981252001</td></tr> <tr> <td>"Maria"</td><td>999111415</td></tr> </table>	"Ana"	991358899	"Fernando"	981252001	"Maria"	999111415
"clientes"	dict <table> <tr> <td>"Ana"</td><td>991358899</td></tr> <tr> <td>"Fernando"</td><td>981252001</td></tr> <tr> <td>"Maria"</td><td>999111415</td></tr> </table>	"Ana"	991358899	"Fernando"	981252001	"Maria"	999111415		
"Ana"	991358899								
"Fernando"	981252001								
"Maria"	999111415								
__base	"Novo Banco de Dados"								

Este é um ponto que gera bastante confusão quando estamos aprendendo sobre encapsulamento de estruturas de uma classe. Visando por convenção facilitar a vida de quem programa, é sempre interessante quando for necessário fazer este tipo de alteração no código deixar as alterações devidamente comentadas, assim como ter o discernimento de que qualquer estrutura de código prefixada com `_` deve ser tratada como um código protegido, e somente como última alternativa realizar este tipo de alterações.

Associação de classes

Associação de classes, ou entre classes, é uma prática bastante comum uma vez que nossos programas dependendo sua complexidade não se restringirão a poucas funções básicas, dependendo do que o programa oferece ao usuário uma grande variedade de funções internas são executadas ou ao menos estão à disposição do usuário para que sejam executadas. Internamente estas funções podem se comunicar e até mesmo trabalhar em conjunto, assim como independentes elas não dependem uma da outra.

Raciocine que esta prática é muito utilizada pois quando es está compondo o código de um determinado programa é natural criar uma função para o mesmo e, depois de testada e pronta, esta pode ser modularizada, ou ser separada em um pacote, etc... de forma que aquela estrutura de código, embora parte do programa, independente para que se faça a manutenção da mesma assim como novas implementações de recursos.

É normal, dependendo a complexidade do programa, que cada “parte” de código seja na estrutura dos arquivos individualizada de forma que o arquivo que guarda a mesma quando corrompido, apenas em último caso faça com que o programa pare de funcionar. Se você já explorou as pastas dos arquivos que compõe qualquer programa deve ter se deparado com milhares de arquivos dependendo o programa. A manutenção dos mesmos se dá diretamente sobre os arquivos que apresentam algum problema.

```
class Usuario:
    def __init__(self, nome):
        self.__nome = nome
        self.__logar = None
    @property
    def nome(self):
        return self.__nome
    @property
    def logar(self):
        return self.__logar
    @logar.setter
    def logar(self, logar):
        self.__logar = logar
```

Partindo para a prática, note que em primeiro lugar criamos nossa classe **Usuario**, a mesma possui um método construtor que recebe um **nome** (com variável homônima declarada como privada), em seguida precisamos realizar a criação de um **getter** e **setter** para que possamos instanciar os referentes métodos de fora dessa classe. Basicamente, pondo em prática tudo o que já vimos até então, esta seria a forma adequada de permitir acesso aos métodos de classe de **Usuario**, quando as mesmas forem privadas ou protegidas.

```
class Identificador:
    def __init__(self, numero):
```

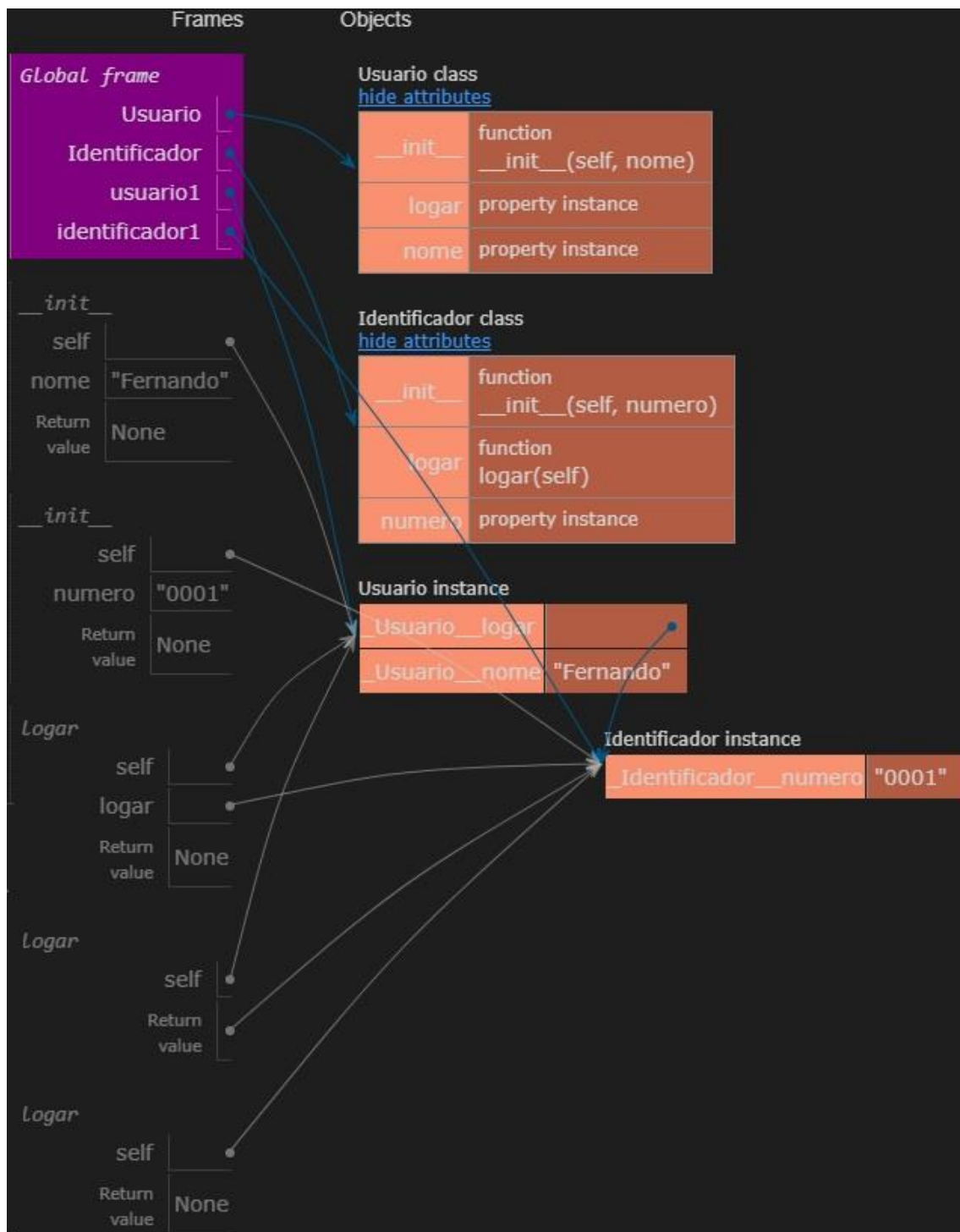
```
        self.__numero = numero
    @property
    def numero(self):
        return self.__numero
    def logar(self):
        print('Logando no sistema...')
```

Na sequência criamos uma nova classe de nome **Identificador** que aqui, apenas como exemplo, seria o gatilho para um sistema de autenticação de usuário em um determinado sistema. Logo, a mesma também possui um método construtor que recebe um número para identificação do usuário, novamente, é criado um **getter** e um **setter** para tornar o núcleo desse código instanciável.

```
usuario1 = Usuario('Fernando')
identificador1 = Identificador('0001')

usuario1.logar = identificador1
usuario1.logar.logar()
```

Por fim, é criado um objeto no corpo de nosso código de nome **usuario1** que instancia a classe **Usuario**, lhe passando como atributo **'Fernando'**. O mesmo é feito para o objeto **identificador1**, que instancia a classe **Identificador** passando como parâmetro **'0001'**. Agora vem a parte onde estaremos de fato associando essas classes e fazendo as mesmas trabalhar em conjunto. Para isso por meio de **usuario.logar = identificador** associamos **0001** a **Fernando**, e supondo que essa é a validação necessária para esse sistema de identificação (ou pelo menos uma das etapas), por meio da função **usuario1.logar.logar()** o retorno é: **Logando no sistema...**



Raciocine que este exemplo básico é apenas para entender a lógica de associação, obviamente um sistema de autenticação iria exigir mais dados como senha, etc... assim como confrontar esses dados com um banco de dados... Também é importante salientar que quanto mais associações houver em nosso código, de forma geral mais eficiente é a execução do mesmo ao invés de blocos e mais blocos de códigos independentes e repetidos para cada situação, porém, quanto mais robusto foi o código, maior também será a suscetibilidade a erro, sendo assim, é interessante testar o código a cada bloco criado para verificar se não há nenhum erro de lógica ou sintaxe.

Agregação e composição de classes

Começando o entendimento pela sua lógica, anteriormente vimos o que basicamente é a associação de classes, onde temos mais de uma classe, elas podem compartilhar métodos e atributos entre si ao serem instanciadas, porém uma pode ser executada independentemente da outra. Já quando falamos em agregação e composição, o laço entre essas classes e suas estruturas passa a ser tão forte que uma depende da outra. Raciocine que quando estamos realizando a composição de uma classe, uma classe tomará posse dos objetos das outras classes, de modo que se algo corromper essa classe principal, as outras param de funcionar também.

```
class Cliente:
    def __init__(self, nome, idade, fone=None):
        self.nome = nome
        self.idade = idade
        self.fone = []

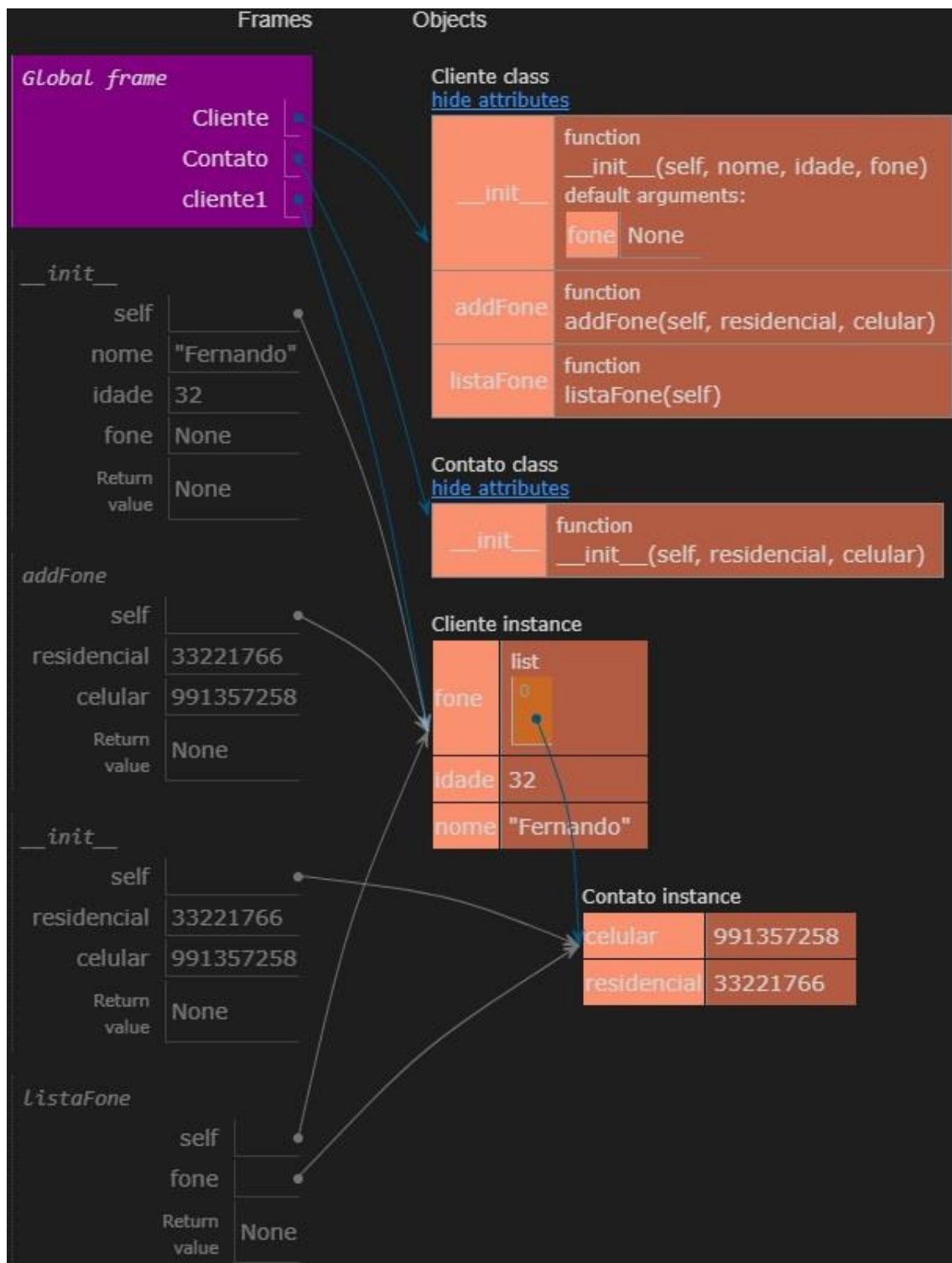
    def addFone(self, residencial, celular):
        self.fone.append(Contato(residencial, celular))
    def listaFone(self):
        for fone in self.fone:
            print(fone.residencial, fone.celular)
```

Para esse exemplo, inicialmente criamos uma classe de nome **Cliente**, ela possui seu método construtor que recebe um **nome**, uma **idade** e um ou mais telefones para contato, isto porque como objeto de classe **fone** inicialmente recebe uma lista em branco. Em seguida é criado um método de classe responsável por alimentar essa lista com números de telefone. Porém, repare no diferencial, em **self.fone**, estamos adicionando dados instanciando como parâmetro outra classe, nesse caso a classe **Contato** com toda sua estrutura. Como dito anteriormente, este laço, instanciando uma classe dentro de outra classe é um laço forte onde, se houver qualquer exceção no código, tudo irá parar de funcionar uma vez que tudo está interligado operando em conjunto.

```
cliente1 = Cliente('Fernando', 32)
cliente1.addFone(33221766, 991357258)
print(cliente1.nome)
print(cliente1.listaFone())
```

Seguindo com o exemplo, é criada uma variável de nome **cliente1** que instancia **Cliente** passando como atributos de classe **Fernando**, **32**. Em seguida é chamada a função **addFone()** passando como parâmetros **33221766** e **991357258**. Por fim, por meio do comando **print()** pedimos que seja exibidos os dados de **cliente1.nome** de **cliente1.listaFone()**. Nesse caso o retorno será:

Fernando
33221766 991357258



Herança Simples

Na estrutura de código de um determinado programa todo orientado a objetos, é bastante comum que algumas classes em teoria possuam exatamente as mesmas características que outras, porém isso não é nada eficiente no que diz respeito a quantidade de linhas de código uma vez que esses blocos, já que são idênticos, estão repetidos no código. Por Exemplo:

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

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

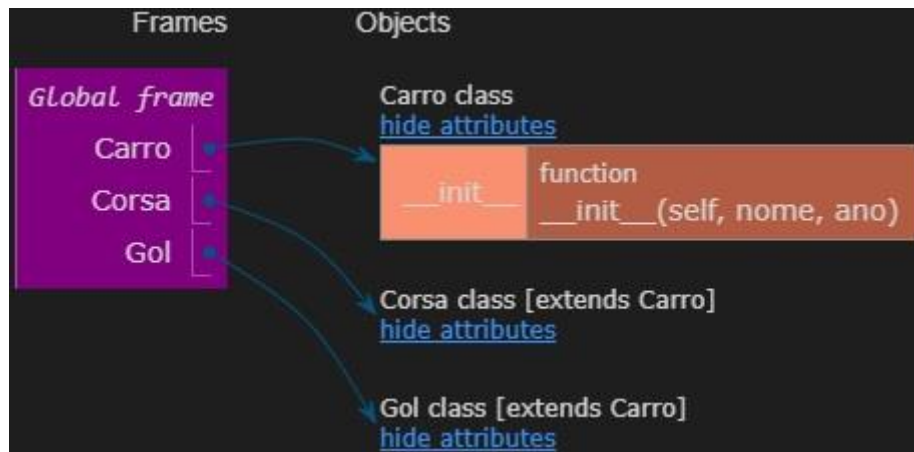
Dado o exemplo acima, note que temos duas classes para dois carros diferentes, **Corsa** e **Gol**, e repare que as características que irão identificar os mesmos, aqui nesse exemplo bastante básico, serão um **nome** e o **ano** do veículo. Duas classes com a mesma estrutura de código repetida duas vezes. Podemos otimizar esse processo criando uma classe molde de onde serão lidas e interpretadas todas essas características de forma otimizada por nosso interpretador. Vale lembrar que neste exemplo são apenas duas classes, não impactando na performance, mas numa aplicação real pode haver dezenas ou centenas de classes idênticas em estrutura, o que acarreta em lentidão se lidas uma a uma de forma léxica.

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

class Corsa(Carro):
    pass

class Gol(Carro):
    pass
```

Reformulando o código, é criada uma classe de nome **Carro** que servirá de molde para as outras, dentro de si ela tem um método construtor assim como recebe como atributos de classe um **nome** e um **ano**. A partir de agora as classes **Corsa** e **Gol** simplesmente recebem como parâmetro a classe **Carro**, recebendo toda sua estrutura. Internamente o interpretador não fará distinção e assumirá que cada classe é uma classe normal, independente, inclusive com alocações de memória separadas para cada classe, porém a estrutura do código dessa forma é muito mais enxuta.



Na literatura, principalmente autores que escreveram por versões anteriores do Python costumavam chamar esta estrutura de classes e subclasses, apenas por convenção, para que seja facilitada a visualização da hierarquia das mesmas.

Vale lembrar também que uma “subclasse” pode ter mais métodos e atributos particulares a si, sem problema nenhum, a herança ocorre normalmente a tudo o que estiver codificado na classe mãe, porém as classes filhas desta tem flexibilidade para ter mais atributos e funções conforme necessidade. Ainda sob o exemplo anterior, a subclasse Corsa, por exemplo, poderia ser mais especializada tendo maiores atributos dentro de si além do nome e ano herdado de Carro.

Cadeia de heranças

Como visto no tópico anterior, uma classe pode herdar outra sem problema nenhum, desde que a lógica estrutural e sintática esteja correta na hora de definir as mesmas. Extrapolando um pouco, vale salientar que essa herança pode ocorrer em diversos níveis, na verdade, não há um limite para isto, porém é preciso tomar bastante cuidado para não herdar características desnecessárias que possam tornar o código ineficiente. Ex:

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

class Gasolina(Carro):
    def __init__(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

class Jeep(Gasolina):
    pass
```

Repare que nesse exemplo inicialmente é criada uma classe **Carro**, dentro de si ela possui um método construtor onde é repassado como atributo de classe um **nome** e um **ano**. Em seguida é criada uma classe de nome **Gasolina** que herda toda estrutura de **Carro** e dentro de si define que o veículo desta categoria será do tipo gasolina. Por fim é criada uma última classe se referindo a um carro marca **Jeep**, que herda toda estrutura de **Gasolina** e de **Carro** pela hierarquia entre classe e subclasses. Seria o mesmo que:

```
class Jeep:
    def carro():
        def __init__(self, nome, ano):
            self.nome = nome
            self.ano = ano

    def gasolina(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

jeep = Jeep()
```

Código usual, porém ineficiente. Assim como estático, de modo que quando houvesse a criação de outros veículos não seria possível reaproveitar nada de dentro dessa Classe a não ser por instâncias de fora da mesma, o que gera muitos caminhos de leitura para o interpretador, tornando a performance do código ruim.

Lembrando que a vantagem pontual que temos ao realizar heranças entre classes é a reutilização de seus componentes internos no lugar de blocos e mais blocos de código repetidos.

```

class Carro:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

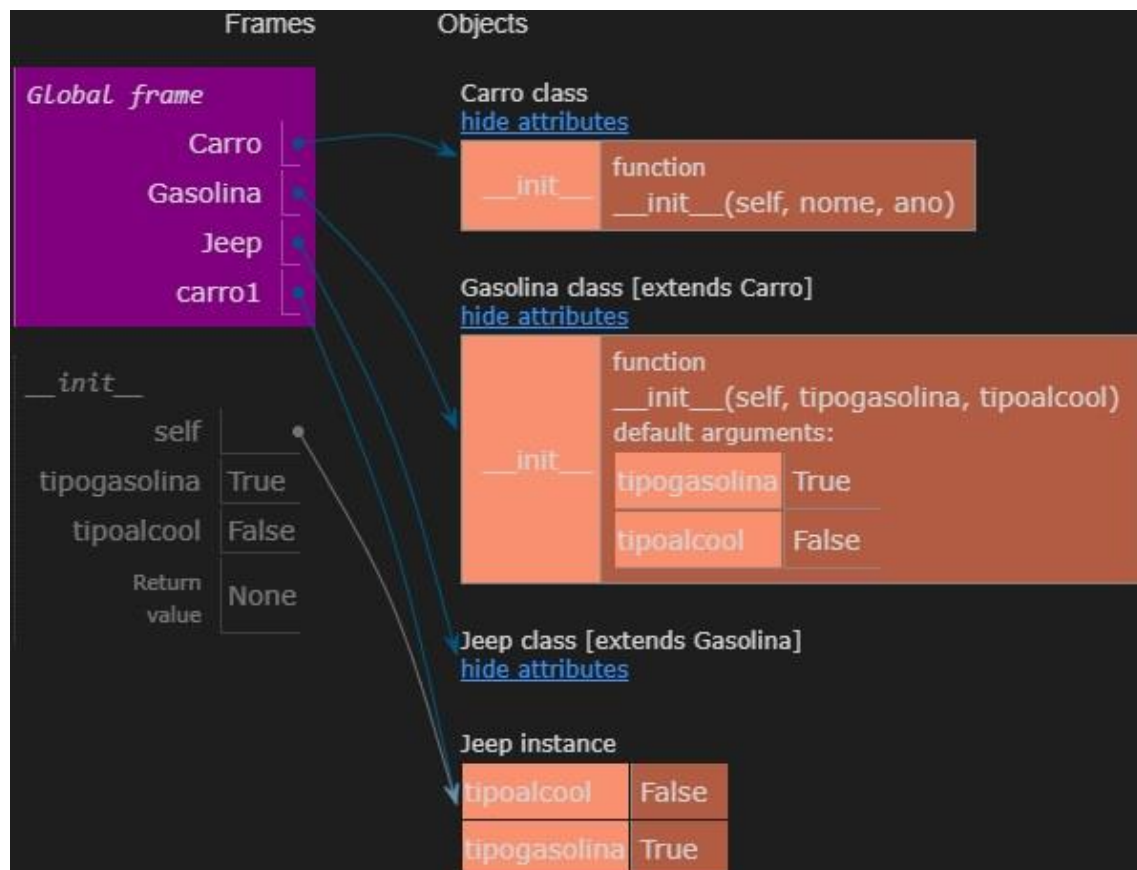
class Gasolina(Carro):
    def __init__(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

class Jeep(Gasolina):
    pass

carro1 = Jeep()
print(carro1.tipogasolina)

```

Código otimizado via herança e cadeia de hierarquia simples. Executando nossa função `print()` nesse caso o retorno será: **True**



Herança Múltipla

Entendida a lógica de como uma classe herdar toda a estrutura de outra classe, podemos avançar um pouco mais fazendo o que é comumente chamado de herança múltipla. Raciocine que numa herança simples, criávamos uma determinada classe assim como todos seus métodos e atributos de classe, de forma que podíamos instanciar / passar ela como parâmetro de uma outra classe. Respeitando a sintaxe, é possível realizar essa herança vinda de diferentes classes, importando dessa forma seus conteúdos. Por exemplo:

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

class Carnes(Mercadoria):
    def __init__(self, tipo, peso):
        self.tipo = tipo
        self.peso = peso

class Utensilios:
    def __init__(self, espetos, carvao):
        self.espetos = espetos
        self.carvao = carvao

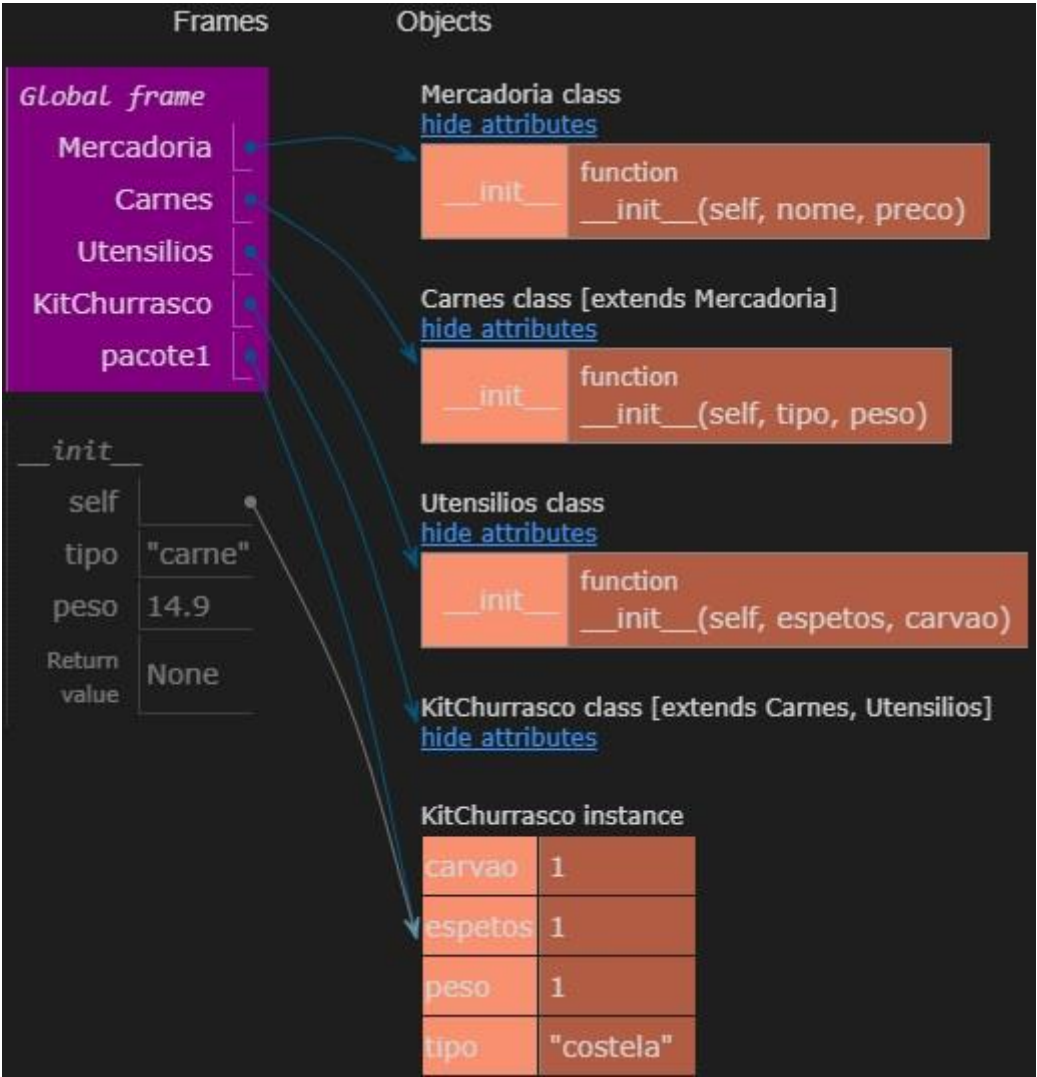
class KitChurrasco(Carnes, Utensilios):
    pass
```

Para este exemplo, vamos supor que esse seria o background de um simples sistema de inventário de um mercado, onde as categorias de produtos possuem estrutura em classes distintas, porém que podem ser combinadas para montar “kits” de produtos. Então note que inicialmente é criada uma classe de nome **Mercadoria**, que por sua vez possui um método construtor assim como **nome** e **preco** das mercadorias como atributos de classe. Em seguida é criada uma classe de nome **Carnes** que herda toda estrutura de **Mercadoria** e adiciona os atributos de classe referentes ao **tipo** de carne e seu **peso**. Na sequência é criada uma classe de nome **Utensilios** que não herda nada das classes anteriores, e dentro de si possui estrutura de método construtor assim como seus atributos de classe. Por fim, é criada uma classe de nome **KitChurrasco**, que por sua vez herda **Carnes** (que herda **Mercadoria**) e **Utensilios** com todo seu conteúdo.

```
pacote1 = KitChurrasco('carne', 14.90)
pacote1.tipo = 'costela'
pacote1.peso = 1
pacote1.espetos = 1
pacote1.carvao = 1
```

A partir disso, como nos exemplos anteriores, é possível instanciar objetos e definir dados/valores para os mesmos normalmente, uma vez que **KitChurrasco** agora, devido

as características que herdou, possui campos para **nome**, **preco**, **tipo**, **peso**, **espetos** e **carvao** dentro de si.



Sobreposição de membros

Um dos cuidados que devemos ter ao manipular nossas classes em suas hierarquias e heranças é o fato de apenas realizarmos sobreposições quando de fato queremos alterar/sobrescrever um determinado dado/valor/método dentro da mesma. Em outras palavras, anteriormente vimos que o interpretador do Python realiza a chamada leitura léxica do código, ou seja, ele lê linha por linha (de cima para baixo) e linha por linha (da esquerda para direita), dessa forma, podemos reprogramar alguma linha ou bloco de código para que na sequência de sua leitura/interpretação o código seja alterado. Por exemplo:

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

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    pass
```

Inicialmente criamos a classe **Pessoa**, dentro de si um método construtor que recebe um nome como atributo de classe. Também é definida uma **Acao1** que por sua vez por meio da função **print()** exibe uma mensagem. Em seguida é criada uma classe de nome **Jogador1** que herda tudo de **Pessoa** e por sua vez, apenas tem um método **Acao2** que exibe também uma determinada mensagem. Por fim é criado uma classe **SaveJogador1** que herda tudo de **Jogador1** e de **Pessoa**.

```
p1 = SaveJogador1('Fernando')
print(p1.nome)
```

Trabalhando sobre essa cadeia de classes, criamos uma variável de nome **p1** que instancia **SaveJogador1** e passa como parâmetro **'Fernando'**, pela hierarquia destas classes, esse parâmetro alimentará **self.nome** de **Pessoa**. Por meio da função **print()** passando como parâmetros **p1.nome** o retorno é: **Fernando**

```
p1.Acao1()
p1.Acao2()
```

Da mesma forma, instanciando qualquer coisa de dentro de qualquer classe da hierarquia, se não houver nenhum erro de sintaxe tudo será executado normalmente.

Neste caso o retorno será:

Fernando está dormindo
Fernando está comendo


```

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

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    def Acao1(self):
        print(f'{self.nome} está acordado')

```

Por fim, partindo para uma sobreposição de fato, note que agora em **SaveJogador1** criamos um método de classe de nome **Acao1**, dentro de si uma função para exibir uma determinada mensagem. Repare que **Acao1** já existia em **Pessoa**, mas o que ocorre aqui é que pela cadeia de heranças **SaveJogador1** criando um método **Acao1** irá sobrepor esse método já criado anteriormente. Em outras palavras, dada a hierarquia entre essas classes, o interpretador irá considerar pela sua leitura léxica a última alteração das mesmas, **Acao1** presente em **SaveJogador1** é a última modificação desse método de classe, logo, a função interna do mesmo irá ser executada no lugar de **Acao1** de **Pessoa**, que será ignorada.

```

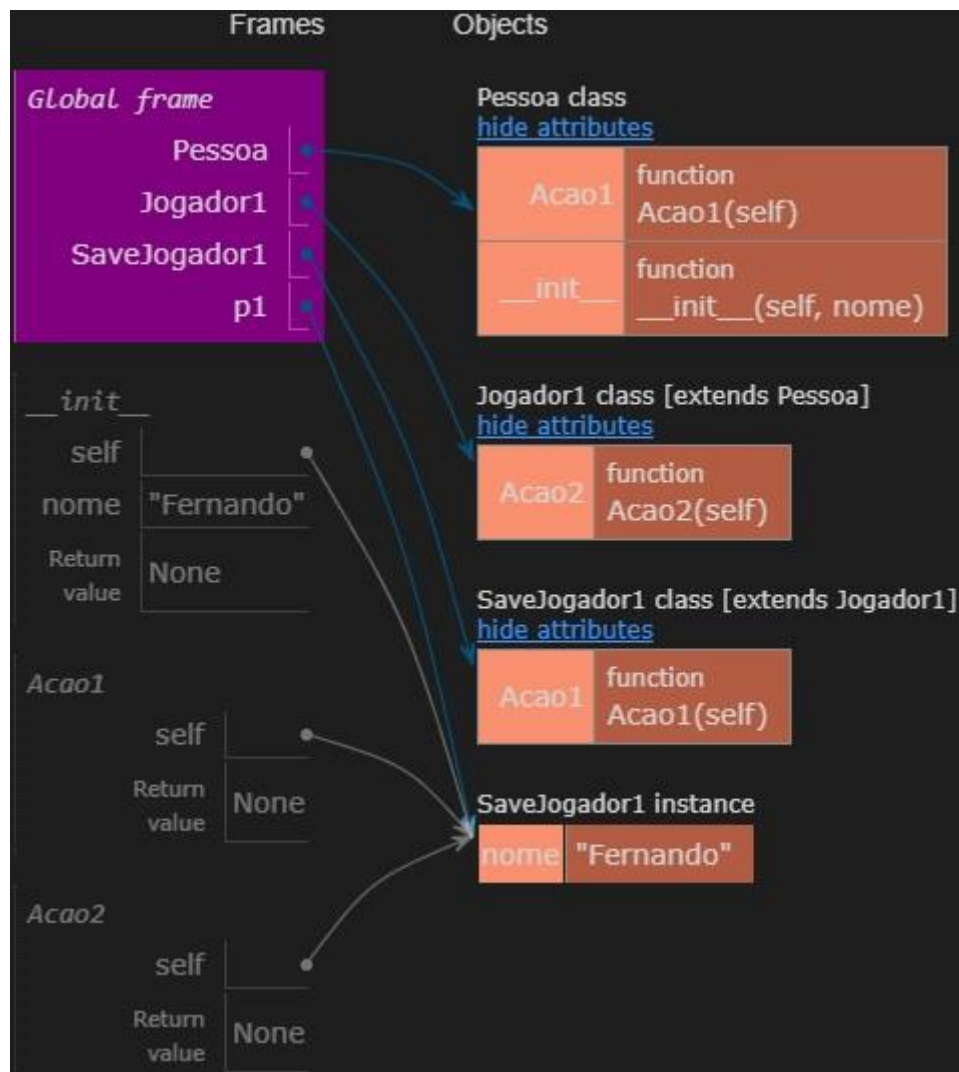
p1 = SaveJogador1('Fernando')
p1.Acao1()
p1.Acao2()

```

Nesse caso o retorno será:

Fernando está acordado (a última instrução atribuída a Acao1)

Fernando está comendo



Outra funcionalidade que eventualmente pode ser utilizada é a de, justamente executar uma determinada ação da classe mãe e posteriormente a sobrescrever, isso é possível por meio da função reservada **super()**. Esta função em orientação a objetos faz com que seja respeitada a hierarquia de classe mãe / super classe em primeiro lugar e posteriormente as classes filhas / sub classes.

```
class SaveJogador1(Jogador1):
    def Acao1(self):
        super().Acao1()
        print(f'{self.nome} está acordado')
```

Neste caso, primeiro será executada a **Acao1** de **Pessoa** e em seguida ela será sobrescrita por **Acao1** de **SaveJogador1**. Nesse caso o retorno será:

Fernando está dormindo (Acao1 de Pessoa)
Fernando está acordado (Acao1 de SaveJogador1)
Fernando está comendo

Avançando com nossa linha de raciocínio, ao se trabalhar com heranças deve-se tomar muito cuidado com as hierarquias entre as classes dentro de sua ordem de leitura léxica. Raciocine que o mesmo padrão usado no exemplo anterior vale para qualquer coisa, até mesmo um método construtor pode ser sobrescrito, logo, devemos tomar os devidos cuidados na hora de usar os mesmos.

Também é importante lembrar que por parte de sobreposições, quando se trata de heranças múltiplas, a ordem como as classes são passadas como parâmetro influencia nos resultados. Ex:

```
class Pessoa:
    def acao(self):
        print('Inicializando o sistema')

class Acao1(Pessoa):
    def acao(self):
        print('Sistema pronto para uso')

class Acao2(Pessoa):
    def acao(self):
        print('Desligando o sistema')

class SaveJogador1(Acao1, Acao2):
    pass

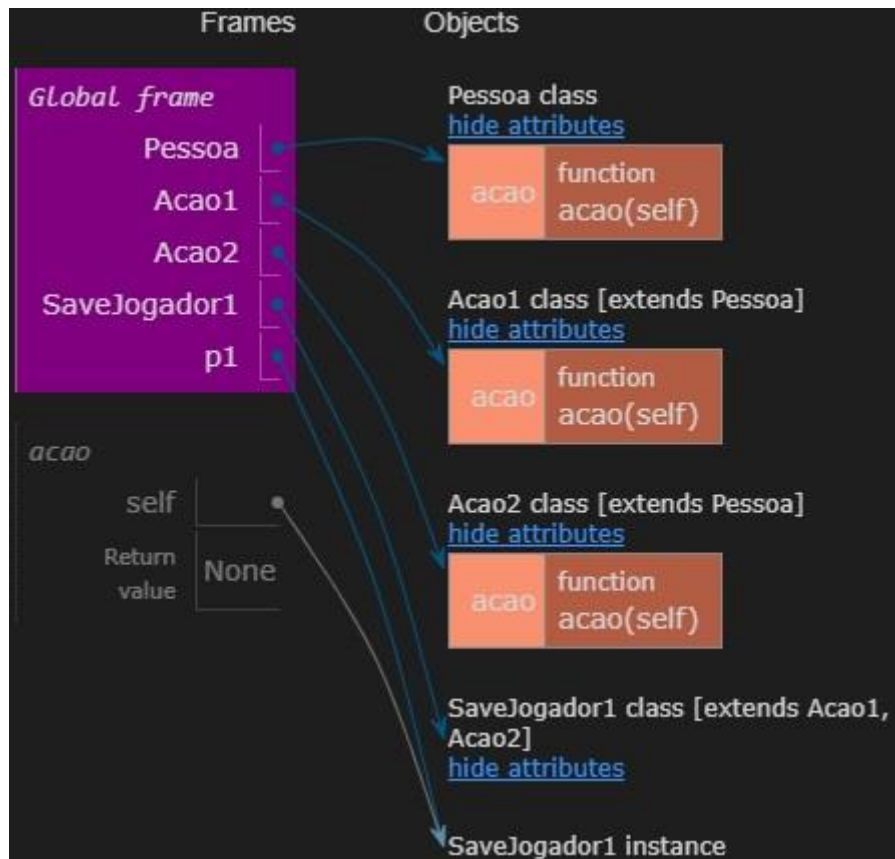
p1 = SaveJogador1()
p1.acao()
```

Apenas como exemplo, repare que foram criadas 4 classes, **Pessoa**, **Acao1**, **Acao2** e **SaveJogador1** respectivamente, dentro de si apenas existe um método de classe designado a exibir uma mensagem. Para esse exemplo, o que importa é a forma como **SaveJogador1** está herdando características de **Acao1** e **Acao2**, como dito anteriormente, essa ordem ao qual as classes são passadas como parâmetro aqui influenciará os resultados da execução do código.

Nesse exemplo, **SaveJogador1(Acao1, Acao2)**, o retorno será: **Sistema pronto para uso**

```
class SaveJogador1(Acao2, Acao1):
    pass
```

Exatamente a mesma estrutura anterior, porém com ordem inversa, **SaveJogador1(Acao2, Acao1)**, o retorno será: **Desligando o sistema**



Apenas fazendo um adendo, caso estes tópicos ainda gerem alguma dúvida para você. Uma forma de diferenciarmos a forma como tratamos as interações entre classes dentro do Python é a seguinte:

Associação: Se dá quando uma classe usa outro ou algum objeto de outra.

Agregação: Se dá quando um ou mais objetos de classe é compartilhado, usado por duas ou mais classes (para que não seja necessário programar esse atributo para cada uma delas).

Composição: Se dá quando uma classe é dona de outra pela interligação de seus atributos e a forma sequencial como uns dependem dos outros.

Herança: Se dá quando estruturalmente um objeto é outro objeto, no sentido de que ele literalmente herda todas características e funcionalidades do outro, para que o código fique mais enxuto.

Classes abstratas

Se você chegou até este tópico do livro certamente não terá nenhum problema para entender este modelo de classe, uma vez que já o utilizamos implicitamente em outros exemplos anteriores. Mas o fato é que, como dito lá no início, para alguns autores uma classe seria um molde a ser utilizado para se guardar todo e qualquer tipo de dado associado a um objeto de forma que possa ser utilizado livremente de forma eficiente ao longo do código. Sendo assim, quando criamos uma classe vazia, ou bastante básica, que serve como base estrutural para outras classes, já estamos trabalhando com uma classe abstrata. O que faremos agora é definir manualmente este tipo de classe, de forma que sua forma estrutural irá forçar as classes subsequentes a criar suas especializações a partir dela. Pode parecer um tanto confuso, mas com o exemplo tudo ficará mais claro.

Como dito anteriormente, na verdade já trabalhamos com este tipo de classe mas de forma implícita e assim o interpretador do Python não gerava nenhuma barreira quanto a execução de uma classe quando a mesma era abstrata. Agora, usaremos para este exemplo, um módulo do Python que nos obrigará a abstrair manualmente as classes a partir de uma sintaxe própria.

Sendo assim, inicialmente precisamos importar os módulos **ABC** e **abstractmethod** da biblioteca **abc**.

```
from abc import ABC, abstractclassmethod
```

Realizadas as devidas importações, podemos prosseguir com o exemplo:

```
class Pessoa(ABC):
    @abstractclassmethod
    def logar(self):
        pass

class Usuario(Pessoa):
    def logar(self):
        print('Usuario logado no sistema')
```

Inicialmente criamos uma classe **Pessoa**, que já herda **ABC** em sua declaração, dentro de si, note que é criado um decorador **@abstractclassmethod**, que por sua vez irá sinalizar ao interpretador que todos os blocos de código que vierem na sequência dessa classe, devem ser sobrescritas em suas respectivas classes filhas dessa hierarquia. Em outras palavras, **Pessoa** no momento dessa declaração, possui um método chamado **logar** que obrigatoriamente deverá ser criado em uma nova classe herdeira, para que possa ser instanciada e realizar suas devidas funções. Dando sequência, repare que em seguida criamos uma classe **Usuario**, que herda **Pessoa**, e dentro de si cria o método **logar** para sobrepor/sobreescrever o método **logar** de **Pessoa**. Este, por sua vez, exibe uma mensagem pré-definida por meio da função **print()**.

```
user1 = Pessoa()
user1.logar()
```

A partir do momento que **Pessoa** é identificada como uma classe abstrata por nosso decorador, a mesma passa a ser literalmente apenas um molde. Seguindo com o exemplo, tentando fazer a associação que estamos acostumados a fazer, atribuindo essa classe a um objeto qualquer, ao tentar executar a mesma será gerado um erro de interpretação.

Traceback (most recent call last):

File "c:/Users/Fernando/Documents/001.py", line 12, in <module>

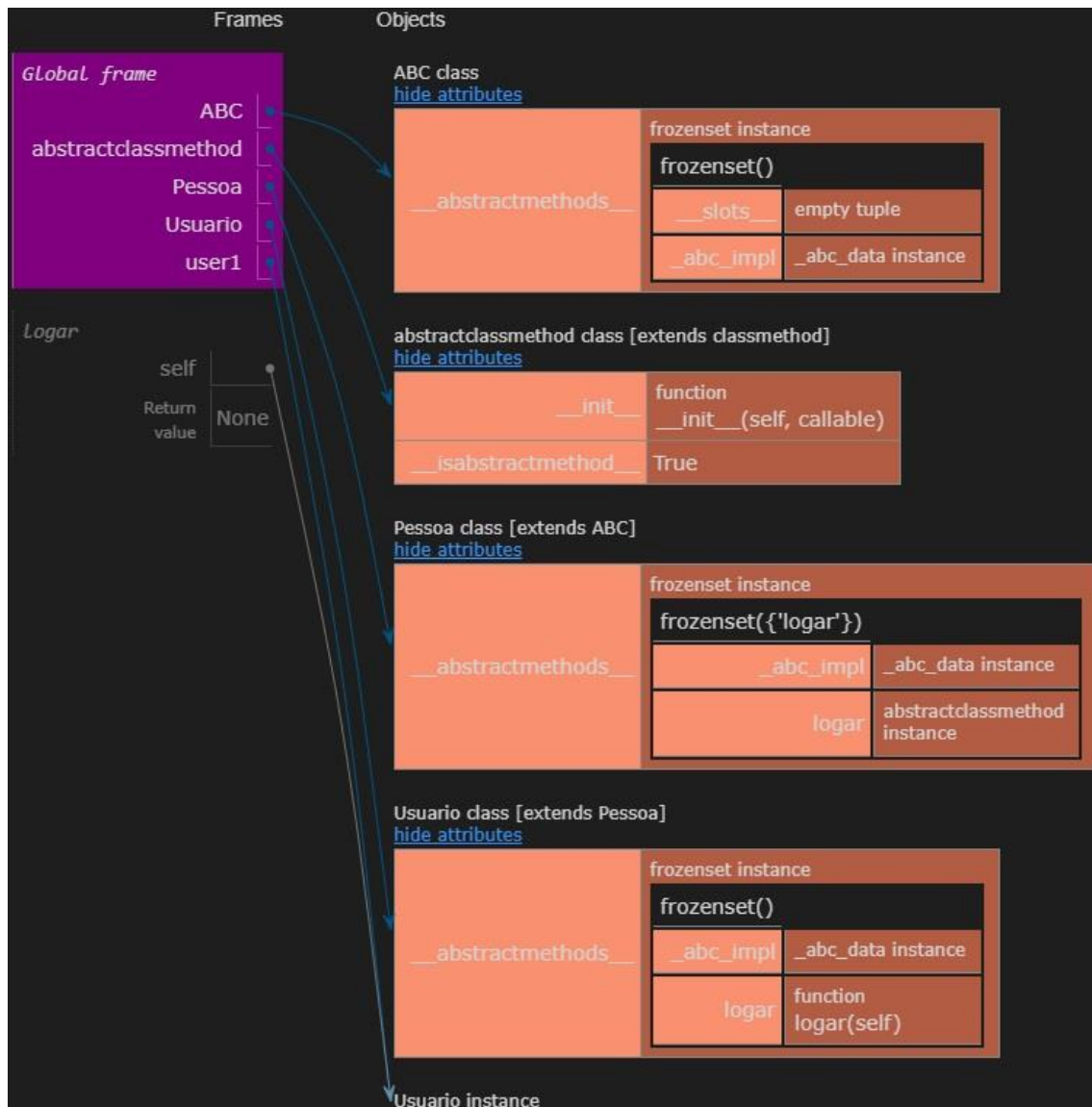
user1 = Pessoa()

TypeError: Can't instantiate abstract class Pessoa with abstract methods logar

Em tradução livre: Não é possível inicializar a classe abstrata Pessoa com o método abstrato logar.

```
user1 = Usuario()  
user1.logar()
```

Porém instanciando **Usuario**, que herda **Pessoa**, ao tentar executar o método de classe logar que consta no corpo do mesmo, este funciona perfeitamente. Nesse caso o retorno será: **Usuario logado no sistema**



Sendo assim, o que precisa ficar bem claro, inicialmente, é que uma das formas que temos de proteger uma classe que apenas nos serve de molde, para que essa seja herdada, mas não sobrescrita, é definindo a mesma manualmente como uma classe abstrata.

Polimorfismo

Avançando mais um pouco com nossos estudos, hora de abordar um princípio da programação orientada a objetos chamada polimorfismo. Como o próprio nome já sugere, polimorfismo significa que algo possui muitas formas (ou ao menos mais que duas) em sua estrutura. Raciocine que é perfeitamente possível termos classes derivadas de uma determinada classe mãe / super classe que possuem métodos iguais porém comportamentos diferentes dependendo sua aplicação no código.

Alguns autores costumam chamar esta característica de classe de “assinatura”, quando se refere a mesma quantidade e tipos de parâmetros, porém com fins diferentes. Último ponto a destacar antes de partirmos para o código é que, polimorfismo normalmente está fortemente ligado a classes abstratas, sendo assim, é de suma importância que a lógica deste tópico visto no capítulo anterior esteja bem clara e entendida, caso contrário poderá haver confusão na hora de identificar tais estruturas de código para criar suas devidas associações.

Entenda que uma classe abstrata, em polimorfismo, obrigatoriamente será um molde para criação das demais classes onde será obrigatório realizar a sobreposição do(s) método(s) de classe que a classe mãe possuir.

```
from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self, chaveseguranca):
        pass

class Usuario(Pessoa):
    def logar(self, chaveseguranca):
        print('Usuario logado no sistema')

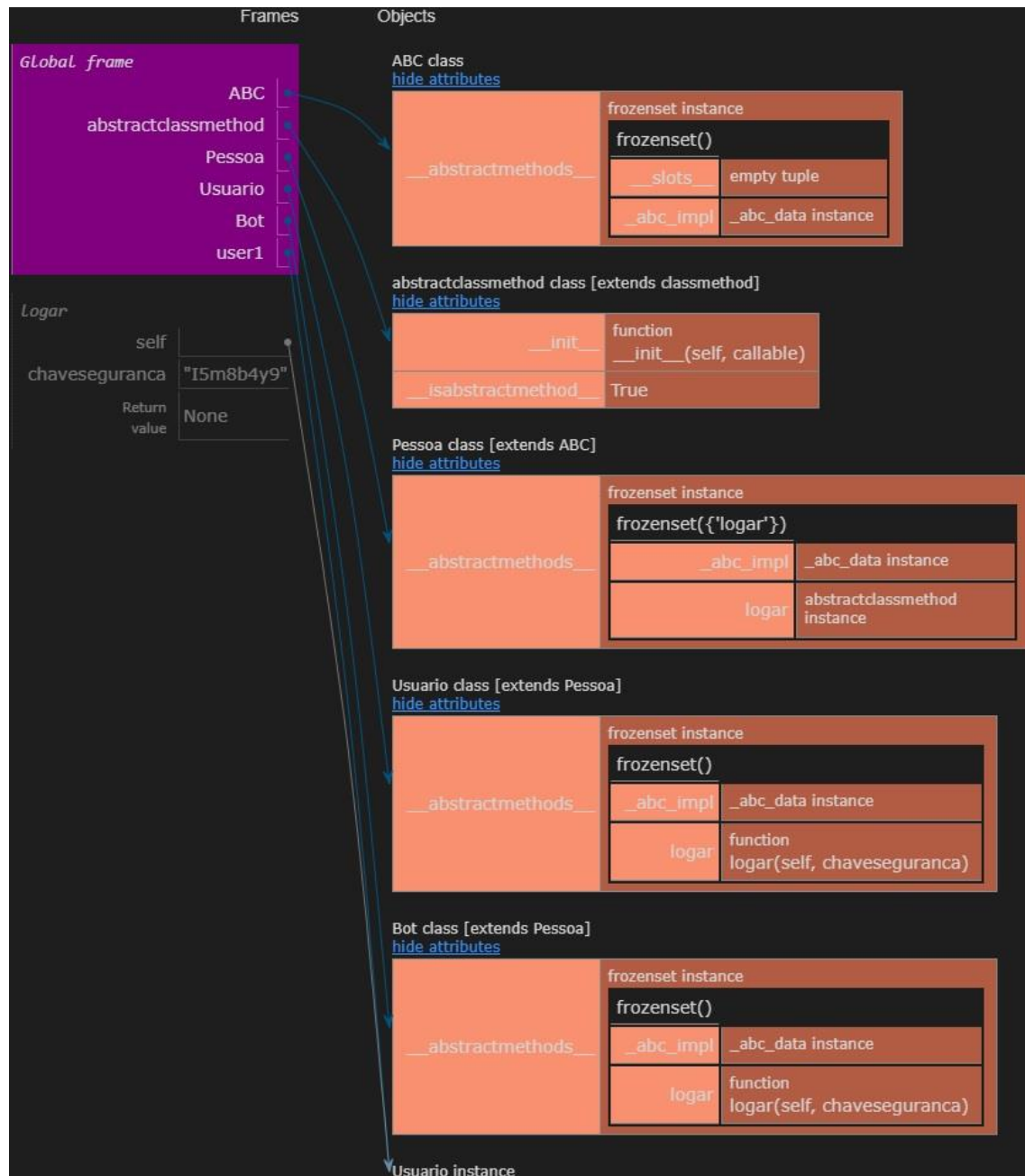
class Bot(Pessoa):
    def logar(self, chaveseguranca):
        print('Sistema rodando em segundo plano')
```

Seguindo com um exemplo muito próximo ao anterior, apenas para melhor identificação, note que inicialmente são feitas as importações dos módulos **ABC** e **abstractclassmethod** da biblioteca **abc**. Em seguida é criada uma classe de nome **Pessoa** que herda **ABC**, dentro de si existe um decorador e um método de classe que a caracteriza como uma classe abstrata. Em outras palavras, toda classe que herdar a estrutura de **Pessoa** terá de redefinir o método **logar**, fornecendo uma **chavedeseguranca** que aqui não está associada a nada, mas na prática seria parte de um sistema de login. Na sequência é criada a classe **Usuário** que herda **Pessoa**, e pela sua estrutura, repare que ela é polimórfica, possuindo a mesma assinatura de objetos instanciados ao método **logar**, porém há uma função personalizada que está programada para exibir uma mensagem. O mesmo é feito com uma terceira classe

chamada **Bot** que herda **Pessoa**, que por sua vez herda **ABC**, tudo sob a mesma “assinatura”.

```
user1 = Usuario()  
user1.logar('I5m8b4y9')
```

Instanciando **Usuario** a um objeto qualquer e aplicando sobre si parâmetros (neste caso, fornecendo uma senha para **chaveseguranca**), ocorrerá a execução da respectiva função. Neste caso o retorno será: **Usuario logado no sistema**



Sobrecarga de operadores

Quando damos nossos primeiros passos no aprendizado do Python, em sua forma básica, programando de forma estruturada, um dos primeiros tópicos que nos é apresentado são os operadores. Estes por sua vez são símbolos reservados ao sistema que servem para fazer operações lógicas ou aritméticas, de forma que não precisamos programar do zero tais ações, pela sintaxe, basta usar o operador com os tipos de dados aos quais queremos a interação e o interpretador fará a leitura dos mesmos normalmente. Em outras palavras, se declaramos duas variáveis com números do tipo inteiro, podemos por exemplo, via operador de soma + realizar a soma destes valores, sem a necessidade de programar do zero o que seria uma função que soma dois valores.

Se tratando da programação orientada a objetos, o que ocorre é que quando criamos uma classe com variáveis/objetos dentro dela, estes dados passam a ser um tipo de dado novo, independente, apenas reconhecido pelo interpretador como um objeto. Para ficar mais claro, raciocine que uma variável normal, com valor atribuído de 4 por exemplo, é automaticamente interpretado por nosso interpretador como int (número inteiro, sem casas decimais), porém a mesma variável, exatamente igual, mas declarada dentro de uma classe, passa a ser apenas um objeto geral para o interpretador.

Números inteiros podem ser somados, subtraídos, multiplicados, elevados a uma determinada potência, etc... um objeto dentro de uma classe não possui essas funcionalidades a não ser que manualmente realizemos a chamada sobrecarga de operadores. Por exemplo:

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

caixa1 = Caixa(10,10)
caixa2 = Caixa(10,20)
```

Aqui propositalmente simulando este erro lógico, inicialmente criamos uma classe de nome **Caixa**, dentro dela um método construtor que recebe um valor para largura e um para altura, atribuindo esses valores a objetos internos. No corpo de nosso código, criando variáveis que instanciam nossa classe **Caixa** e atribuem valores para **largura** e **altura**, seria normal raciocinar que seria possível, por exemplo, realizar a soma destes valores.

```
print(caixa1 + caixa2)
```

Porém o que ocorre ao tentar realizar uma simples operação de soma entre tais valores o retorno que temos é um traceback.

TypeError: unsupported operand type(s) for +: 'Caixa' and 'Caixa'

Em tradução livre: Erro de Tipo: tipo de operador não suportado para +: Caixa e Caixa.

Repare que o erro em si é que o operador de soma não foi reconhecido como tal, pois esse símbolo de + está fora de contexto uma vez que estamos trabalhando com orientação a objetos. Sendo assim, o que teremos de fazer é simplesmente buscar os operadores que sejam reconhecidos e processados neste tipo de programação.

Segue uma lista com os operadores mais utilizados, assim como o seu método correspondente para uso em programação orientada a objetos.

Operador	Método	Operação
+	<code>__add__</code>	Adição
-	<code>__sub__</code>	Subtração
*	<code>__mul__</code>	Multiplicação
/	<code>__div__</code>	Divisão
//	<code>__floordiv__</code>	Divisão inteira
%	<code>__mod__</code>	Módulo
**	<code>__pow__</code>	Potência
<	<code>__lt__</code>	Menor que
>	<code>__gt__</code>	Maior que
<=	<code>__le__</code>	Menor ou igual a
>=	<code>__ge__</code>	Maior ou igual a
==	<code>__eq__</code>	Igual a
!=	<code>__ne__</code>	Diferente de

Dando continuidade a nosso entendimento, uma vez que descobrimos que para cada tipo de operador lógico/aritmético temos um método de classe correspondente, tudo o que teremos de fazer é de fato criar este método dentro de nossa classe, para forçar o interpretador a reconhecer que, nesse exemplo, é possível realizar a soma dos dados/valores atribuídos aos objetos ali instanciados.

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

    def __add__(self, other):
        pass
```

Retornando ao código, note que simplesmente é criado um método de classe de nome `__add__` que recebe como atributos um valor para si e um valor a ser somado neste caso. `__add__` é uma palavra reservada ao sistema, logo, o interpretador sabe sua função e a executará normalmente dentro desse bloco de código.

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

    def __add__(self, other):
        largura1 = self.largura + other.largura
```

```
altura1 = self.altura + other.altura
return Caixa(largura1, altura1)

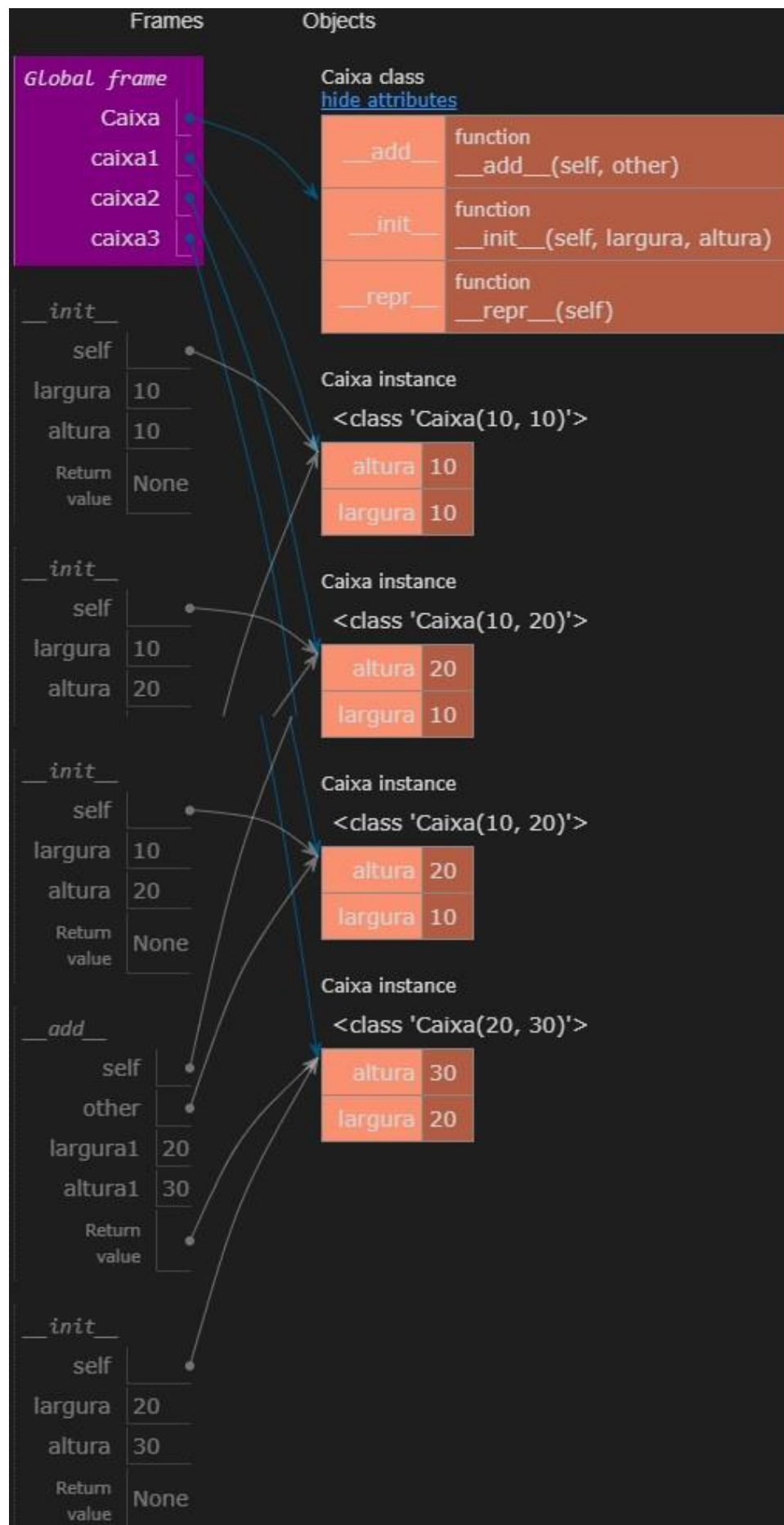
def __repr__(self):
    return f"<class 'Caixa({self.largura}, {self.altura})'>"
```

Dessa forma, podemos criar normalmente uma função de soma personalizada, note que no método `__add__` foi criada uma variável de nome **largura1** que guardará o valor da soma entre as larguras dos objetos. Da mesma forma, **altura1** receberá o valor da soma das alturas, tudo isto retornando para a classe tais valores. Na sequência, apenas para facilitar a visualização, foi criada um método de classe `__repr__` (também reservado ao sistema) que irá retornar os valores das somas explicitamente ao usuário (lembre-se de que como estamos trabalhando dentro de uma classe, por padrão esta soma seria uma mecanismo interno, se você tentar printar esses valores sem esta última função, você terá como retorno apenas o objeto, e não seu valor).

```
caixa1 = Caixa(10,10)
caixa2 = Caixa(10,20)
caixa3 = caixa1 + caixa2

print(caixa3)
```

Por fim, agora instanciando a classe, atribuindo seus respectivos valores, finalmente é possível realizar normalmente a operação de soma entre as variáveis que instanciam essa classe. Nesse exemplo, criando uma variável de nome **caixa3** que recebe como atributo a soma dos valores de **largura** e **altura** de **caixa1** e **caixa2**, o retorno será: `<class 'Caixa(20, 30)'`



Encerrando, apenas tenha em mente que, de acordo com a funcionalidade que quer usar em seu código, deve procurar o método equivalente e o declarar dentro da classe como um método de classe qualquer, para “substituir” o operador lógico/aritmético de forma a ser lido e interpretado por nosso interpretador.

Tratando exceções

Tanto na programação comum estruturada quanto na orientada a objetos, uma prática comum é, na medida do possível, tentar prever os erros que o usuário pode cometer ao interagir com nosso programa e tratar isso como exceções, de forma que o programa não trave, nem feche inesperadamente, mas acuse o erro que o usuário está cometendo para que o mesmo não o repita, ou ao menos, fique ciente que aquela funcionalidade não está disponível. Na programação estruturada temos aquela estrutura básica de declarar uma exceção e por meio dela, via **try** e **except**, tentar fazer com que o interpretador redirecione as tentativas sem sucesso de executar uma determinada ação para alguma mensagem de erro ao usuário ou para alguma alternativa que retorne o programa ao seu estado inicial. Na programação orientada a objetos esta lógica e sintaxe pode ser perfeitamente usada da mesma forma. Ex:

```
class Erro001(Exception):  
    pass  
  
def erro():  
    raise Erro001('Ação não permitida!')  
  
try:  
    erro()  
except Erro001 as msgerro:  
    print(f'ERRO: {msgerro}')
```

Repare na estrutura, é normalmente suportado pelo Python, que exceções sejam criadas também como classe. Inicialmente criamos uma classe de nome **Erro001** que herda a função reservada ao sistema **Exception**. A partir disto, criamos uma função que instancia uma variável para **Erro001** atribuindo uma **string** onde consta uma mensagem de erro pré-programada ao usuário (lembrando que a ideia é justamente criar mensagens de erro que orientem o usuário sobre o mesmo, e não as mensagens padrão de erro que o interpretador gera em função de lógica ou sintaxe errada). Em seguida, exatamente igual ao modelo estruturado, criamos as funções **try** e **except**, onde realizamos as devidas associações para que caso o usuário cometa um determinado erro seja chamada a mensagem de exceção personalizada. Neste caso o retorno será: **ERRO: Ação não permitida!**

Capítulo Final

Tudo o que tem um começo... tem um fim, e assim encerramos este pequeno livro. Espero que ao chegar nestas linhas finais você tenha realmente conseguido entender os conceitos abordados ao longo destas páginas. Programação é uma área fascinante, complexa, em constante evolução, e tenho certeza que as primeiras vezes que você teve contato com programação orientada a objetos deve ter ficado um pouco assustado(a) porque é um choque de realidade onde quando achávamos que dominávamos o essencial sobre uma determinada linguagem de programação, nos era mostrado que aquilo era apenas a ponta do iceberg.

Agora entendido o essencial de programação orientada a objetos em Python, hoje de tentar dar passos maiores e enfrentar desafios maiores, já que ainda há um mundo de possibilidades a serem descobertas.

Nunca se esqueça que programação é prática, prática, muito estudo e mais prática... e mesmo assim sempre terão situações que teremos de recorrer a documentação ou a fóruns especializados para buscar a resolução de algum problema. Importante salientar também que em Python, dependendo o propósito, existe uma infinidade de bibliotecas e módulos desenvolvidos a fim de automatizar ou facilitar certos processos, todas elas, com suas particularidades, também são estudadas as vezes do zero...

Sendo assim, apenas quero que fique bem claro que ao final deste livro, se você praticou em cima dos exemplos, se entendeu toda a lógica por traz de cada tópico abordado, você sim deu um grande passo em seu aprendizado de Python. Com esta bagagem de conhecimento é sim possível criar um universo programado, mas espero que este seja apenas mais um degrau alcançado em busca de objetivos maiores.

Como sempre digo ao final de meus livros, espero que a leitura deste tenha sido tão prazerosa quando foi para mim escrevê-lo, e espero que tenha sido de grande valia para seu aprendizado.

Agradeço a compra deste material e lhe desejo muito sucesso em sua carreira profissional, e quem sabe um dia nos cruzamos por aí em algum bloco de código de algum projeto...

Um forte abraço.

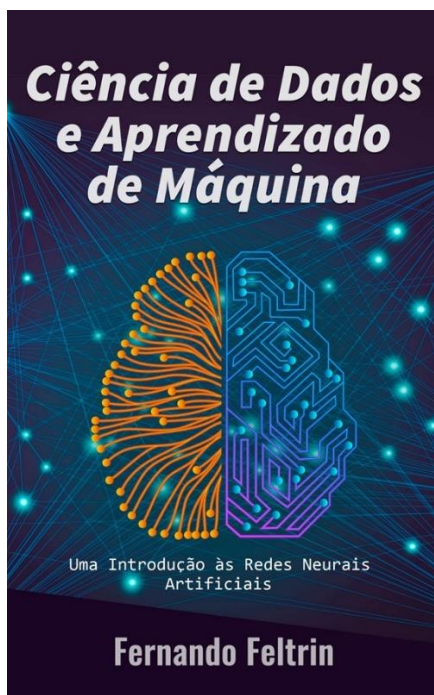
Fernando Feltrin.



Python do ZERO à Programação Orientada a Objetos

Aprenda Python 3 de forma prática e descomplicada. Este livro irá lhe oferecer linguagem clara junto de uma metodologia autodidata que permitirá que você aprenda a linguagem Python do zero absoluto e ao final do mesmo consiga ter o conhecimento necessário para programar desde scripts e softwares básicos até a sistemas mais robustos que envolvem programação orientada a objetos. Obra com abordagem autodidata prática, totalmente exemplificado e com explicações para cada linha de código.

<https://www.amazon.com.br/dp/B07P2VJVW5>



Ciência de Dados e Aprendizado de Máquina: Uma abordagem prática as redes neurais artificiais

Aprenda de forma prática e descomplicada o que são redes neurais artificiais, assim como seus modelos e aplicações. Ao longo desse livro você aprenderá de forma simples e prática de como criar suas próprias redes neurais artificiais, independente do tipo, características e propósito, com modelos baseados em aplicações reais e com as devidas explicações para cada linha de código.

<https://www.amazon.com.br/dp/B07X1TVLKW>