Termo de Cooperação 5900.0117579.21.9: Sistema inteligente para levantamento dos fatores de uso de cargas elétricas nos FPSOs correlacionados às demandas de produção

Friends DEVELOPERS

UNIVERSIDADE FEDERAL FLUMINENSE		
Nome	**Titulação**	**Função**
Vitor Hugo Ferreira	D.Sc.	Coordenador
André Abel Augusto	D.Sc.	Pesquisador
André da Costa Pinho	D.Sc.	Pesquisador
Angelo Cesar Colombini	D.Sc.	Pesquisador
Artur Alves Pessoa	D.Sc.	Pesquisador
Bruno Soares Moreira César Borba	D.Sc.	Pesquisador
Eduardo Uchoa Barboza	D.Sc.	Pesquisador
Márcio Zamboti Fortes	Dr.	Pesquisador

Versão	Data de emissão
INDICAR A VERSÃO DO DOCUMENTO. EM CASO DE NOVA VERSÃO, INCLUIR UMA LINHA A MAIS E DATAR AO LADO	DATA DA EMISSÃO DA RESPECTIVA VERSÃO
Responsável PETROBRAS pela aprovação final	Data de aprovação da versão final
NOME DO PROFISSIONAL PETROBRAS RESPONSÁVEL PELA APROVAÇÃO	DATA DA APROVAÇÃO DA VERSÃO FINAL DO DOCUMENTO

SUMÁRIO

0. Histórico de Versão do Documento

Este documento apresenta a versão 1.0 do documento Friends Develops. Seu objetivo é estabelecer padrões mínimos para as equipes de desenvolvimento de soluções computacionais dentro do contexto do FriendsLab.

Esta é a versão final do documento, aprovado pelo Prof. Vitor Hugo Ferreira responsável pelo Laboratório FrinedsLab na data 21/10/2021.

1. Organização do Documento

Este Documento está organizado da seguinte forma: a seção 2 apresenta as práticas de programação e documentação a serem adotadas ao longo do projeto. Já na seção 3, apresentar-se-á os padrões adotados para construção de classes e módulos.

2. Práticas de Programação

> "*Any fool can write code that a computer can understand. Good programmers write code that humans can understand*."
>
> *Martin Fowler*

Mais importante do que escrever um código-fonte, é ser capaz de compreendê-lo, independentemente daquele que o escreve. Assim sendo, o estabelecimento de práticas de programação e desenvolvimento de *software* é essencial para a criação de *softwares* complexos, envolvendo equipes extensas. Esta seção apresenta de forma resumida os padrões de programação a serem seguidos pela equipe ao longo do projeto.

2.1. Linguagem de Programação

2.1.1. Versão

A Linguagem de programação a ser adota no Projeto é a Linguagem **`Python`**. A versão deve ser aquela estável, oficial e mais recente. Esteja atento se a componentes de terceiros (bibliotecas, pacotes, módulos), mantém sua funcionalidade plena em versões mais recentes, caso contrário, consulte os líderes das equipes para saber como proceder.

2.1.2. Módulos e pacotes

Módulos, pacotes de terceiros, ou da biblioteca padrão do Python, devem ser instalados utilizando a versão mais recente do pacote de instalação **`pip`**. Para mais informações sobre a instalação e uso do **`pip`**, consulte https://pip.pypa.io/en/stable/.

2.2. Ambiente de desenvolvimento

O ambiente computacional onde a solução será desenvolvida deve replicar, na medida do possível, aquele onde ela será utilizada. Isto tem como objetivo evitar problemas de compatibilidade e funcionamento inesperado da aplicação nos computadores do cliente. Caso não seja possível adquirir o ambiente computacional, deve-se construir máquinas virtuais que atendam as especificações. Cuidado ao utilizar plataformas de prototipagem de código na nuvem/Web (como o **`Google Colaboratory`**, **`Jupyter Lab`**, **`Jupyter Notebook`**, **`Rextester`** etc.), a máquina virtual deve replicar o ambiente computacional do cliente.

2.2.1. Sistema Operacional

O sistema operacional no qual os componentes da solução devem ser desenvolvidos e testados é o **`Windows 10`**. Códigos desenvolvidos e testados em outros sistemas operacionais como **`Linux`** e **`iOS`** não serão aceitos como entregáveis.

2.2.2. Ambiente Virtual

Ambiente virtual é um ambiente de execução isolado, executado em um sistema. Nele, usuários e aplicações podem instalar e atualizar pacotes Python sem que esses interfiram no comportamento de outras aplicações Python em execução no próprio sistema. Os desenvolvedores devem criar ambientes virtuais nos quais se instalará única e exclusivamente os componentes empregados na sua solução. Caso opte por trocas de pacotes e bibliotecas (por exemplo: trocar o pacote **`keras`** pelo **`sci-kit learn`**), **desinstale o pacote que deixará de utilizar**. Mais informações podem ser consultadas em https://docs.python.org/pt-br/3/tutorial/venv.html.

2.3. Ferramentas de Programação e Desenvolvimento

Nesta seção serão apresentadas as ferramentas de programação e desenvolvimento a serem adotadas na solução.

2.3.1. Diagramas UML

Para a construção de fluxogramas e diagramas UML para desenhar fluxos de processos, modelos de processos, diagramas de casos de uso, diagramas de atividades, diagramas de classes etc. utilizar o **`draw.io`**. Este aplicativo web pode ser acessado via https://app.diagrams.net/.

2.3.2. Ambiente de Desenvolvimento Integrado

O ambiente de desenvolvimento integrado (*Integrated Development Environment* - IDE) que será utilizado no projeto é o **`PyCharm`**. Portanto, *a versão final do código-fonte a constar nos entregáveis de código deve ser inteiramente concebida e testada dentro desse IDE.* O **`PyCharm`** permite a criação de ambientes virtuais. Para saber mais, acesse https://www.jetbrains.com/help/pycharm/creating-virtual-environment.html. Além disso possui uma ferramenta para gerenciamento de pacotes. Mais informações podem ser encontradas em https://www.jetbrains.com/help/pycharm/pipenv.html.

Há também um tutorial disponível no GitHub do FrindsLab.

2.3.3. Anotações e Registro de alterações

Para facilitar o desenvolvimento do código, evitar o retrabalho e fazer o registro da evolução dos módulos, os programadores devem utilizar o **`Notepad++`** para fazer o "diário de bordo", isto é, o registro das atividades, ocorrências, alterações, ideias e comentários durante o desenvolvimento do código. Recomenda-se colocar a cada registro de atividade sua data, horário, nome do autor da modificação, descrição sumária, comentários e código. Esse documento é de grande importância para registro da evolução do código, rastreamento de erros, e documentação do software. *As anotações de registro no Notepad++ constituem parte dos entregáveis de código.*

2.3.4. Ambiente de prototipagem e testagem

Para prototipagem e testagem de partes solução é recomendado o uso de notebooks, em especial o **`Google Colaboratory`** (https://colab.research.google.com/). *É importante destacar que os notebooks não são entregáveis de código*. Portanto, uma vez construído o protótipo e verificado, esse deverá ser reconstruído no PyCharm, respeitando os padrões de programação estabelecidos neste documento. Uma vez testados, deverão ser juntados ao "diário de bordo" e relatório, caso necessário, para que constituam um entregável. Os notebooks devem ser utilizados para teste de ideias, não para a construção da solução – evite retrabalho!

2.4. Requisitos dos entregáveis de código

Os requisitos já foram apresentados ao longo das seções anteriores e são listados resumidamente a seguir:

- Códigos-fonte das classes e módulos contendo a solução do projeto, desenvolvias no **`PyCharm`**, em ambiente virtual específico.
- Documento texto escrito no **`Notepad++`**, contendo o registro do desenvolvimento dos componentes;

2.5. Padrões de Codificação

2.5.1. Padrão PEP-8 Abel aqui cabe ressaltar pontos chaves da PEP8 que potencializam a qualidade do software?

2.5.2. Padrão para cabeçalhos e *docstrings* Abel, existe um documento de referência sobre o uso de *docstrings*. É preciso definir, construir exatamente qual será o padrão para os cabeçalhos e como as *docstrings* devem ser usadas?

2.5.3. Padrão para nome de constantes, variáveis, funções, classes e módulos? Abel aqui é fundamental construir o padrão a ser adotado, como se espera que cada membro do time nomeie variáveis, funções classes etc.

3. Funções

Os sistemas devem fazer uso extensivo de funções, com o objetivo de modularizar e potencializar a manutenibilidade do sistema, gerar documentação mais clara e precisa e prover um maior reaproveitamento do código. Para avaliar se um determinado trecho de código deve ser escrito como uma função pense em: *Don't Repeat Yourself* - DRY (não repita seu código).

Não trataremos de funções aqui, os exemplos abaixo são apenas uma pequena amostra das possibilidades que o uso de funções em Python oferece para tornar um código mais legível e dentro das melhores práticas de programação do mercado.

Em Python uma função é criada:

```
def nome_da_funcao(parametros_de_entrada):
    """
    Esta função objetiva exemplificar o processo de criação de uma função em Python. Note que o nome de uma função
    deve ser escrito com letras minúsculas separadas por underline (Snake Case)
    :param parametros_de_entrada: são opcionais, onde havendo mais de um devem vir separados por vírgula,
    descreva aqui algo como: seu tipo, se existe ou não valor default, qual sua finalidade dentro da função, etc.
    :return: Descreva aqui o valor de retorno da função caso ele exista.
    """
    # corpo da função

    return "opcional"
```

Nota: a palavra reservada ***return*** finaliza a função Python, diferentes ***returns*** são possíveis em uma mesma função, mas apenas um deles será executado;

em Python é possível criar uma variável sendo ela do tipo da função criada (embora possível essa prática deve ser desencorajada), por exemplo:

```
variavel = nome_da_funcao

variavel()
```

Recomendação: sempre que possível trabalhe com parâmetros ***defult*** eles permitem elevar a flexibilidade do código, evita erros com parâmetros incorretos e melhora a legibilidade do código.

```
def param_defaut(nome='FrindsLab', sobrenome=False):
    """
    Esta função é uma amostra de boas práticas no uso de parâmetros em função
    :param nome: valor default == FriendsLab → deve receber o parâmetro nome
    :param sobrenome: valor default == True → deve receber um valor booleano
    :return: valor de retorno será uma função dos parâmetros de entrada
    """
    if nome == 'FriendLab' and sobrenome:
        return 'Bem vindo ao FriendsLab'
    elif nome == 'Vitor':
        return 'Bem vindo ao Jogo'
    return f'Ola seja bem vindo {nome}'
```

Entendendo ***args**: não é obrigatório, pode ser utilizado com parâmetros obrigatórios sem qualquer problema

```
def soma_numeros(*args):
    """
    Função simples com o propósito de gerar entendimento sobre o uso do parâmetro args
    :param args: Absorverá todos os parâmtros de entrada não importa seu número
    :return: o valor de retorno será a soma dos parâmetros de entrada
    """
    return f"Entendendo *args: {sum(args)}"
```

```
def verifica_info(*args):
    """
    Exemplo de aplicação de *args → objetivo mostrar versatilidade
    :param args:
    :return: Valor de retorno será uma função da condição
    """
    if 'Aluno' in args and 'FriendsLab' in args:
        return f'Bem vindo Aluno ao FriendsLab!!!'
    return f'Eu não estou certo de quem você seja ...'
```

Entendendo ****kwargs**: é mais um parâmetro, mas diferente do ***args** que coloca os valores extras em uma tupla, o ****kwargs** exige que utilizemos parâmetros nomeados, e transforma estes parâmetros extras em um dicionário

```python
def cumprimento(**kwargs):
    if 'friendslab' in kwargs and kwargs['friendslab'] == 'Machinelearning':
        return 'Receba um cumprimento ML especial você que é filiado ao friendslab!!!'
    elif 'friendslab' in kwargs:
        return f"{kwargs['friendslab']} Machinelearning!"
    return 'Não tenho certeza de quem é você ... !!!'
```

Atenção:

a. se a sua função realiza várias tarefas convém avaliar a possibilidade de quebrar sua função em outras funções menores, procure em seu código ser minimalista e muito claro;

b. com relação ao escopo de variáveis, recomenda-se fortemente que **NÃO** trabalhe com variáveis **GLOBAIS**;

c. o parâmetro ***args** coloca os valores extras informados como entrada em uma tupla (tuplas para Python são imutáveis);

d. o asterisco * serve para dizer ao Python que estamos passando como argumento uma coleção de dados. Desta forma ele saberá que precisará antes de qualquer ação desempacotar estes dados;

e. o duplo asterisco ** serve para dizer ao Python que estamos passando como argumento um dicionário, logo o uso de ****kwargs** exigira que se trabalhe com parâmetro nomeados, podendo ser aplicado tudo o que se conhece de dicionários;

f. A ordem dos parâmetros em uma função Python é essa:
   a. parâmetros obrigatórios;
   b. *args
   c. parâmetros default (não obrigatórios);
   d. **kwargs

g. o asterisco ** serve para dizer ao Python que estamos passando como argumento um dicionário de dados. Desta forma ele saberá que precisará antes de qualquer ação desempacotar estes dados;

4. Módulos

Em Python módulos nada mais são do que outros arquivos escritos em Python, são úteis para deixar o código mais simples e facilitar o reaproveitamento do código. Deve-se gerir com propriedade os importes de módulos, pois ao fazê-lo Python deixará todos os recursos existentes no módulo disponíveis na memória, concorrendo desta forma pelos recursos da máquina. Recomenda-se que procure identificar exatamente quais os métodos que realmente irá utilizar do módulo e fazer o importante de forma específica. Recomenda-se recorrer quando necessário à: https://docs.python.org/3/py-modindex.html. Estes módulos, já estão instalados com sua linguagem Python (*builtin*), mas o acesso a eles em muitos casos requer o *import*, isso ocorre, por motivos de gestão dos recursos computacionais

a) *import random as rdm*– neste importe todos os recursos do módulo estarão disponíveis, note o uso do alias para facilitar o processo de digitação – evite essa abordagem
b) *from random import random as rdm*– neste caso importa-se apenas o método *random()* do módulo *random* essa deve ser a forma adotada no projeto, note que aqui também usamos um alias, neste caso o apelido não se aplica ao módulo, mas sim à função
c) Quando for preciso realizar múltiplos *imports* de um mesmo módulo, usar ***tuple***, como:

```
from random import (
        random,
        randint,
        shuffle,
        choice
)
```

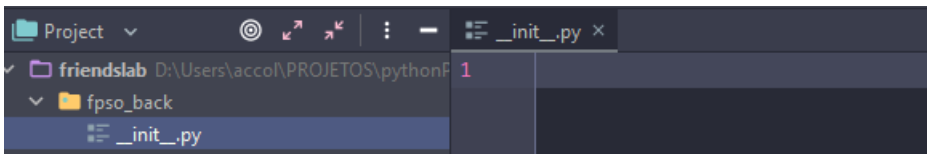
5. Pacotes

Quando falamos de módulo nos referimos a um arquivo Python que pode ter diversas funções que podemos utilizar.

Já o conceito de pacote diz respeito a um diretório que contenha uma coleção de módulos Python. Esse é conceito de trabalho recomendado para os projetos no contexto do FriendsLab.

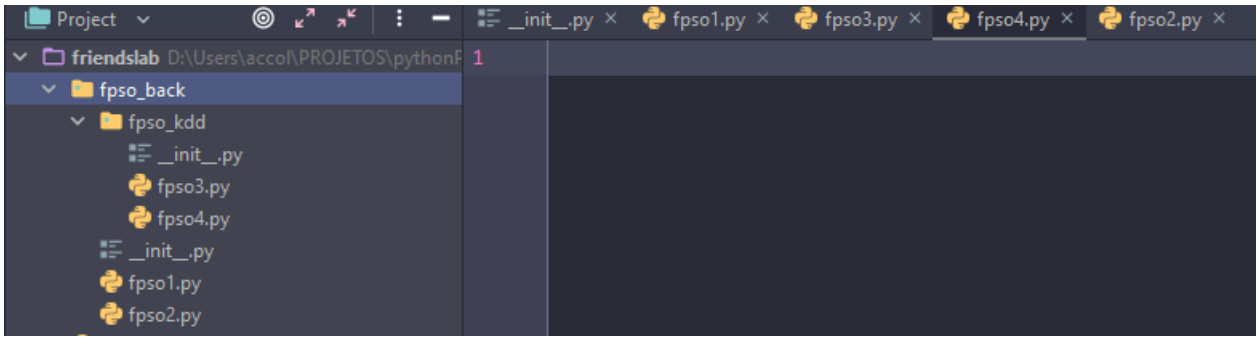
Atenção: nas versões mais antigas do Python tipo 2.x, um diretório definido com um pacote deveria conter um arquivo chamado __init__.py. Este requisito não é mais necessário, mas por questões de compatibilidade seu uso será mantido, a fim de, assegurar uma maior escalabilidade do código produzido. O próprio PyCharm realiza estas atividades para nós, observe as imagens a seguir:

Uma vez criado o diretório (fpso_back) o arquivo __init__.py já estará definido, note que o arquivo é um arquivo vazio (simplesmente seu papel é assegurar a compatibilidade com versões mais antigas do Python), observe:



Dentro deste diretório (fpso_back) todos os módulos Python do projeto serão criados, abaixo um exemplo só para ilustrar o processo. Note que foi criada uma árvore de diretórios

hierarquizando o projeto (isto é apenas um exemplo), veja a estrutura montada, note a presença do arquivo __init__.py nos dois diretórios.



Nota: recomenda-se não ultrapassar o limite de um nível hierárquico (embora perfeitamente possível), a fim de, manter o código o mais ***clean*** possível. Embora só estes exemplos trabalham apenas com funções, ele se aplica para classes e métodos.

A Seguir uma ilustração do processo de codificação a partir de um processo modular.

```
"""
Dando sequência às boas práticas de programação, neste laboratório exploramos o conceito de módulos
Quando falamos de módulo nos referimos a um arquivo Python que pode ter diversas funções que podemos utilizar.
Já o conceito de pacote diz respeito a um diretório que contenha uma coleção de módulos Python. Esse é conceito de
trabalho recomendado para os projetos no contexto do FriendsLab.

Nota: Recomenda-se não ultrapassar o limite de um nível hierárquico nos módulos, a fim de, manter o código o mais
cleam possível

"""

from fpso_back import fpso1, fpso2  # módulos no mesmo nível
from fpso_back.fpso_kdd import fpso3, fpso4  # forma de acesso aos submódulos
from fpso_back.fpso1 import funcao1 as fc1  # importando uma função específica em um módulo e usando aliás
from fpso_back.fpso_kdd.fpso4 import funcao4 as fc4  # importando uma função específica a partir de um submódulo


print(f"Imprimindo a constante pi a partir de fpso1:\n{fpso1.pi}")
print(f"Imprimindo o resultado da operação de soma a partir de fpso1:\n{fpso1.funcao1(2, 5)}")
print(f"Imprimindo a constante string a partir de fpso2:\n{fpso2.projeto}")
print(f"Imprimindo uma string a partir de fpso2:\n{fpso2.funcao2()}")
print(f"Imprimento a partir do submódulo fpso_kdd:\n{fpso3.funcao3()}")
print(f"Imprimento a partir do submódulo fpso_kdd:\n{fpso4.funcao4()}")
print(f"Imprimindo o resultado de soma através do import da funcao1:\n{fc1(4, 10)}")
print(f"Imprimindo uma string a partir a partir de fpso4 da funcao1:\n{fc4()}")
```

6. Dunder Main e Dunder Name

Dunder significa *doble under*, sendo *dunder main* - **__main__** e *dunder name* **__name__.** Em Python emprega-se *dunder* para criar atributos, funções, métodos, etc. que não gere conflito com os nomes desses elementos na programação (uma forma de diferenciar a nomenclatura).

O uso de __name__ e __main__ são fortemente recomendados para que não incorra em duplo processamento ao trabalhar-se com módulos. O fato é que ao trabalhar com módulos deve se evitar a execução local, mas nem sempre é possível. Sendo assim, use o recurso:

if __name__ == __main__:

Aqui todo corpo de código executável do módulo – veja o exemplo:

```
def soma_impares(numeros):
    total = 0
    for num in numeros:
        if num % 2 != 0:
            total += num
    return total


# Para eliminar o problema de dupla execução em processos modulares, utiliza-se:
# Essa prática pode auxiliar nos casos em que testes se fazem necessários

if __name__ == '__main__':
    lista = [1, 2, 3, 4, 5, 6, 7]
    print(f"\nO valor da soma lista é: {soma_impares(lista)}")
    tupla = 3, 7, 9, 11, 15
    print(f"\nO valor da soma tupla é: {soma_impares(tupla)}")
    print(f"Visualizando a variável __name__. Seu valor será: {__name__}")
```

No módulo principal, fica assim:

```
"""
Neste laboratório vamos trabalhar com o conceito de Dunder → duplo unde

Especificamente falaremos de dois deles que atuam em conjunto com algumas propriedades relevantes para o
desenvolvimento de sistemas otimizados, customizados e escaláveis. São eles:

    - Dunder Name → __name__
    - Dunder Main → __main__

Em Python, são utilizados dunder para criar funções, métodos, atributos, etc para não gerar conflito com os nomes
desses elementos na programação.

    Nota: em Python, se executarmos um módulo Python diretamente na linha de comando, internamente o Python atribuirá à
    variável __name__ o valor __main__ indicando que que este módulo é o módulo de execução principal.

"""

from fpso_back.funcoes_parametros import soma_impares as si

print(f"\nImprimindo soma_impares: {si([3, 4, 5, 6, 7, 8, 9])}")
```

Nota: em Python, se executarmos um módulo Python diretamente na linha de comando, internamente o Python atribuirá à variável __name__ o valor __main__ indicando que que este módulo é o módulo de execução principal.

Nesta linha cabe ressaltar que estes testes se aplicam a módulo que foram desenvolvidos para serem executados diretamente e que dada sua relevância podem ser importados e utilizados em outras partes do código. Na dúvida deixe sempre preparado para possíveis importes.

7. Orientação a objetos

Orientação a objetos é o paradigma definido no momento para o desenvolvimento de projetos no contexto do FriendsLab.

```
Conceitualmente POO é um paradigma de programação que utiliza de mapeamento de objetos
do mundo real para modelos
computacionais.

Paradigma de programação é a forma/metodologia para pensar/desenvolver sistemas

|> Principais elementos de orientação a objetos:
    - Classe -> modelo do objeto do mundo real sendo representado computacionalmente
    - Atributo -> característica do objeto
    - Método -> comportamento do objeto (funções)
    - Construtor -> método especial utilizado para criar os objetos
    - Objeto -> instância da classe
```

7.1. Classes

Classes podem conter:

- **Atributos** - que representam as características do objeto. Ou seja, pelos atributos conseguimos representar computacionalmente o comportamento de um objeto.
- **Métodos** (funções) - representam o comportamento do objeto, ou seja, as ações que este objeto pode realizar no seu sistema.

- **Notas**:
  - Em Python uma classe é iniciada com a palavra reservada **class;**
  - O nome de uma classe segundo a Pep8 segue o modelo **CamelCase** (válido para classes que nós criamos);
  - Quando estamos planejando um sistema e definimos quais classes teremos que ter no sistema, chamamos estes objetos que serão mapeados para classes de **entidade**.

Obs.: diferentemente de outras linguagens Python permite a criação de múltiplas classes em um mesmo bloco de código.

7.2. Atributos

Atributos representam as características do objeto, já sabemos disso, é importante reforçar que através do(s) atributo(s) conseguimos representar computacionalmente os estados do objeto. O que é importante frisar neste momento é que um bom projeto passa por uma adequada modelagem

de atributos, dados os objetivos pretendidos, por exemplo, pense nos atributos possíveis para se representar uma lâmpada, observe que, nem sempre todos os atributos possíveis são necessários, mas existem aqueles que atendem de forma precisa os requisitos do sistema. Recomenda-se fortemente uma adequada modelagem dos objetos. No caso da lâmpada, pode ser que o projeto só esteja interessado em saber se ela está ligada ou desligada.

O que significa **ATRIBUTO DE INSTÂNCIA**: significa que ao criarmos instâncias/objetos de uma classe, todas as instâncias terão o mesmo conjunto de atributos.

ATRIBUTOS DE CLASSE: são aqueles declarados diretamente na classe, ou seja, fora do construtor. Geralmente já inicializamos um valor que é compartilhado com todas as instâncias da classe. Ou seja, ao invés de cada instância da classe ter seus próprios valores como é o caso dos atributos de instâncias, com os atributos de classe, todas as classes terão o mesmo valor para este atributo.

Nota: não é preciso criar uma instância de uma classe para ter acesso a um de seus atributos de classe. Para isto, basta usar o nome da classe (Produto, por exemplo) seguida do operador (**.**) (**Produto.imposto**) fornece acesso ao atributo imposto da classe Produto.

Importante: os atributos de classe contribuem para uma gestão otimizada dos recursos de memória, pois apenas um endereço de memória será reservado para estes atributos independentemente do número de instâncias criadas.

ATRIBUTOS DINÂMICOS: nada mais é que um atributo de instância que pode ser criado em tempo de execução.

Notas

- O atributo dinâmico será exclusivo da instância que o criou;

- O parâmetro self presente no construtor e antes do nome das variáveis, representa o próprio objeto que está executando o método;

- O método **__init__** é disparado quando se instância um objeto com o nome da classe e abre e fecha parênteses. Por exemplo: prod1 = **Produto();**

- Atributos de instâncias podem ser públicos ou privados, o atributo privado recebe um duplo **under** antes de seu nome, por exemplo, **__voltagem**, já o atributo público não recebe o duplo **under** antes de seu nomo, exemplo, **valor**. A diferença entre eles (público e privado) é a forma de acesso, por exemplo, um atributo privado só pode ser acessado dentro de sua própria classe.

Importante: em Python por convenção, foi estabelecido que todo atributo de uma classe é público, sendo assim, ele pode ser acessado em todo projeto. Caso o projeto exija que atributos sejam privados (por questões de segurança, por exemplo), deve se tratar essa questão utilizando outros recursos computacionais. Resumindo, mesmo que seu atributo esteja definido como privado sempre será possível acessá-lo.

Recomendação: embora possível essa prática deve ser evitada (acesso indevido).

A seguir alguns exemplos para ilustrar os conceitos apresentados nesta proposta de trabalho:

Classe com atributos de instância privados:

```python
# Classe com atributos privados


class Lampada:

    def __init__(self, voltagem, cor):  # Este é um clássico exemplo de método construtor em Python
        self.__voltagem = voltagem  # Atributo de instância
        self.__cor = cor  # Atributo de instância
        self.__ligada = False  # Atributo de instância

    @property
    def voltagem(self):
        return self.__voltagem

    @property
    def cor(self):
        return self.__cor

    @property
    def ligada(self):
        return self.__ligada
```

Classe com atributos de instância públicos:

```python
# Classes com atributos de instância públicos


class ContaCorrente:

    def __init__(self, numero, limite, saldo):
        self.numero = numero
        self.limite = limite
        self.saldo = saldo
```

Importante: você sempre terá meios de acessar um atributo privado em Python, portanto, fique atento quando o sistema demandar segurança. Uma das formas de proteger minimamente seu sistema é criptografar informações que deseje manter de forma confidencial. Exemplo de aplicação será tratado aqui, embora, não faça parte do contexto desse projeto.

7.3. Métodos

Métodos representam os comportamentos do objeto. Ou seja, as ações que este objeto pode realizar no sistema. Em Python os métodos são divididos em dois grupos:

- Métodos de instância;
- Métodos de classe.

MÉTODOS DE INSTÂNCIA: são aqueles que estão no contexto da classe (vinculados às instâncias da classe – são métodos com o objetivo de acesso aos atributos de instância), sendo assim, possuem acesso direto aos atributos de instância, sendo acessados (métodos de instância) diretamente pela instância (Objeto) criada e do operador ponto (.).

MÉTODOS DE CLASSE TAMBÉM CHAMADOS DE MÉTODOS ESTÁTICOS: são aqueles que não estão vinculados a nenhuma instância de classe, mas somente a ela (classe), em outras palavras, são métodos com o objetivo de acesso aos atributos de classe. O diferencial aqui é a necessidade do uso de um **decorator** na sua declaração (**@classmetod**). Outra diferença é que o parâmetro gerado neste método é o **cls** e não mais o **self**, **cls** representa a própria classe, assim como o **self** representa o objeto.

MÉTODOS PRIVADOS: assim como nos atributos, poderá haver a necessidade de construir métodos privados, métodos com ação no escopo da classe, emprega duplo **under** antes do nome do método (isso é uma convenção não uma obrigatoriedade) (__nome_metodo). Recomenda-se seguir a convenção.

MÉTODO ESTÁTICO: Python possui os métodos estáticos, apesar de, os métodos de classe também receberem esse nome. A diferença aqui está no **decorator** que neste caso passa a ser **(@staticmethod**). Neste método não temos acesso nem à instância e nem a classe, não há o **cls** e nem o **self**.

A seguir alguns exemplos de métodos para ilustrar as recomendações deste documento:

```python
class Produto:
    # Atributo de Classe
    imposto = 1.05  # Atributo de classe que atribui a todas as instâncias a taxa de 0.05% de imposto
    contador = 0  # Atributo de classe que atribui às instâncias o código identificador

    def __init__(self, nome, descricao, valor):
        self.__id = Produto.contador + 1
        self.__nome = nome
        self.__descricao = descricao
        self.__valor = valor * Produto.imposto
        Produto.contador = self.__id

    def desconto(self, porcentagem):  # Exemplo de método de instância
        """Retorna o valor do produto com o desconto negociado"""
        return (self.__valor * (100 - porcentagem)) / 100
```

```python
class Usuario:
    contador = 0

    @classmethod
    def conta_usuarios(cls):  # cls é a própria classe, assim como self é o próprio objeto
        print(f"\nTemos até o momento {cls.contador} usuari(os) cadastrado(s) no sistema!")

    @staticmethod  # Exemplo de mátodo estático
    def definicao():
        return 'ACC_33.#$'

    def __init__(self, nome, sobrenome, email, senha):
        self.__id = Usuario.contador + 1
        self.__nome = nome
        self.__sobrenome = sobrenome
        self.__email = email
        self.__senha = cryp.hash(senha, rounds=200000, salt_size=16)  # Atenção não se preocupe com isso,
        # não faz parte do escopo deste documento
        Usuario.contador = self.__id
        print(f"\nUsuário criado ⇒ {self.__gera_usuario()}")

    def nome_completo(self):
        return f"\n{self.__nome} {self.__sobrenome}"

    def checa_senha(self, senha):
        if cryp.verify(senha, self.__senha):
            return True
        return False

    def __gera_usuario(self):  # Exemplo de método privado
        return self.__email.split('@')[0]
```

```python
p1 = Produto('Play Station 4', 'Video game', 2300)

print(f"\nQual será o valor final do produto após um desconto de 20% ⇒ {p1.desconto(20)}\nEste é um exemplo de "
      f"acesso a um método de instância")
```

```python
# Métodos de classe - início dos testes

print(f"\nAcesso a contador antes de instanciar um objeto ⇒ {Usuario.contador}")

print(f"\nAcesso ao método estático ⇒ {Usuario.definicao()}")

user = Usuario('Artur', 'Dutra', 'dutra@gmail.com', '654321')

print(f"\nAcesso a contador após instanciar um objeto ⇒ {Usuario.contador}")

print(f"\nAcesso ao método estático após instanciar o objeto ⇒ {Usuario.definicao()}")

# Acesso ao método de classe

Usuario.conta_usuarios()  # Forma correta de acesso
user.conta_usuarios()  # Acesso possível, mas deve ser desencorajado
```

```
user1 = Usuario('Vitor', 'Ferreira', 'vhf@gmail.com', '123456')
user2 = Usuario('Abel', 'Augusto', 'aagusto@gmail.com', '654321')

print(f"\nTestando o método de instância nome_completo: {user1.nome_completo()}")
print(f"\nTestando o método de instância nome_completo: {user2.nome_completo()}")

# NÃO FAÇA ISSO, vale apenas para ilustrar fragilidade de segurança, ok

print(f"\nSenha dos usuários expostas:\nUsuário 1 ⇒ {user1._Usuario__senha}\nUsuário 2 ⇒ {user2._Usuario__senha}")

# A seguir um exemplo muito simples de uma aplicação com um pouco mais de segurança

nome = input('Informe o nome: ')
sobrenome = input('Informe o seobrnome: ')
email = input('Informe o e-mail: ')
senha = input('Informe a senha: ')
confirma_senha = input('Confirme a senha: ')

if senha == confirma_senha:
    user = Usuario(nome, sobrenome, email, senha)
    print(f"\nUsuário criado com sucesso. Bem vindo!!!")
else:
    print(f"\nA senha não confere - Usuário não cadastrado!!!")
    exit(33)  # Apenas um escape no caso de problemas com a senha

senha = input('Informe a senha para acesso: ')

if user.checa_senha(senha):
    print(f"\nAcesso permitido!")
else:
    print(f"\nAcesso negado!")
    exit(33)

print(f"\nVamos ver a senha criptografada - não faça isso: {user._Usuario__senha}")
```

7.4. Objetos

Objetos são instâncias da classe. Ou seja, após o mapeamento do objeto do mundo real para sua representação computacional, devemos poder criar quantos objetos forem necessários. Podemos pensar nos objetos/instâncias de uma classe como variáveis do tipo definido na classe. É muito importante pensar no projeto e na forma mais adequada para o uso dos objetos. Deve-se ter particular atenção quando um objeto for passado como parâmetro, algo perfeitamente possível, no entanto, poderá fragilizar a segurança do sistema, se esse for um requisito do projeto, deve-se evitar essa prática.

A seguir alguns exemplos de objetos bem simples:

```python
class Lampada:

    def __init__(self, voltagem, cor, luminosidade):  # Este é um clássico exemplo de método construtor em Python
        self.__voltagem = voltagem  # Atributo de instância
        self.__cor = cor  # Atributo de instância
        self.__luminosidade = luminosidade
        self.__ligada = False  # Atributo de instância

    def checa_lampada(self):
        return self.__ligada

    def ligar_desligar(self):
        if self.__ligada:
            self.__ligada = False
        else:
            self.__ligada = True
```

```python
# Instanciando objetos

lamp1 = Lampada('127', 'cor', '100')

lamp1.ligar_desligar()

print(f"\nA lampada lamp1 está ligada? ⇒ {lamp1.checa_lampada()}")
```

```python
class Cliente:

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

    def diz(self):
        print(f"\nOcliente {self.__nome} diz Oi!!")


class ContaCorrente:

    contador = 4999

    def __init__(self, limite, saldo, cliente):
        self.__numero = ContaCorrente.contador + 1
        self.__limite = limite
        self.__saldo = saldo
        self.__cliente = cliente
        ContaCorrente.contador = self.__numero

    def mostra_cliente(self):
        print(f"\nObserve o que estamos demonstrando aqui. Queremos o nome do cliente que está na classe Cliente\n"
              f"\n{self.__cliente.__Cliente__nome}")
```

```
# Instanciando objetos

# cc1 = ContaCorrente('5000', '20000')

user1 = Usuario('Artur', 'Dutra', 'dutra@gmail.com', '123456')

cli1 = Cliente('Amadeu Bueno', '123.456.789-55')

cc = ContaCorrente('5000', '10000', cli1)  # Observe que estamos aqui passando um objeto como parâmetro. A partir daí
# temos acesso a todos os métodos e atributos da classe. Cabe sempre se preocupar com o melhor projeto.

print(f"\n Testando acesso Cliente através do objeto cc")
cc.mostra_cliente()

cc._ContaCorrente__cliente.diz()  # Atenção essa não é a forma correta, apeanas exibindo falhas de segurança.
```

7.5. Abstração e encapsulamento

O grande objetivo da POO é **ENCAPSULAR** o código dentro de um grupo lógico utilizando classes (encapsula os atributos e métodos).

ABSTRAÇÃO: em POO é o ato de expor apenas dados relevantes de uma classe, escondendo atributos e métodos privados de usuário. Assim, todo processo deve ser oculto/transparente para o usuário.

NOTA: questões simples de segurança estão sendo levantadas, pois estás estão nas mãos do desenvolvedor, no final ele deverá responder por falhas de segurança.

Alguns exemplos de Abstração e encapsulamento:

```python
class Conta:
    contador = 400

    def __init__(self, titular, saldo, limite):
        self.__numero = Conta.contador
        self.__titular = titular
        self.__saldo = saldo
        self.__limite = limite
        Conta.contador += 1

    def extrato(self):
        print(f"\nSaldo de {self.__saldo} do titular {self.__titular} com limite de {self.__limite}")

    def depositar(self, valor):
        if valor > 0:
            self.__saldo += valor
            print(f"\nOperação realizada com sucesso!!")
        else:
            print(f"\nTransação não efetuada o valor deve ser positivo")

    def sacar(self, valor):
        if valor > 0:
            if self.__saldo >= valor:
                self.__saldo -= valor
                print(f"\nOperação realizada com sucesso!!")
            else:
                print(f"\nSaldo insuficiente. Operação não realizada")
        else:
            print(f"\nO valor deve ser positivo!!")

    def transferir(self, valor, conta_destino):
        # 1 - Sacar valor da conta de origem (self)
        self.__saldo -= valor
        self.__saldo -= 20.00  # Taxa de transferência entre contas

        # 2 - Depositar na conta conta_destino
        conta_destino.__saldo += valor
```

```python
# Testando a aplicação → observe alguns problemas que podem passar, pois tudo parece funcionar ok

cc1 = Conta('Marcio', 250.00, 5000.00)
cc2 = Conta('Victoria', 500.00, 10000.00)
```

```python
# Trabalhando com o método transferir

cc1.extrato()
cc2.extrato()
cc2.transferir(100, cc1)
cc1.extrato()
cc2.extrato()
```

8. Herança

POO - Herança - *(Inheritance)*

A ideia de herança é a de reaproveitamento de código. Mas não é só isso, podemos também estender nossas classes.

Com a herança, a partir de uma classe existente, nós estendemos outra classe que passa a herdar atributos e métodos da classe herdada.

Imagine um projeto simples onde temos o seguinte:

Cliente
- nome
- sobrenome
- cpf
- renda

Funcionario
- nome
- sobrenome
- cpf
- matricula

Importante - em um projeto procure sempre responder a seguinte pergunta:
Existe alguma entidade genérica o suficiente para encapsular os atributos e métodos comuns a outras entidades?
A resposta a esta pergunta é a base para se compreender o conceito de herança.

Note que em nosso projeto os atributos nome, sobrenome, cpf e o **método** nome_completo são comuns tanto a **Cliente** como para **Funcionario**. Logo, isso pode sugerir a criação de uma classe que incorpore todos esses atributos e métodos, por exemplo, podemos pensar em criar uma classe **Pessoa**, assim, podemos pensar que um **Cliente** é antes de mais nada uma pessoa e um **Funcionario** é uma pessoa.

Obs.: quando uma classe herda de outra classe ela herda TODOS os atributos e métodos da classe herdada.
Importante: quando uma classe herda de outra classe, ela deve usar o construtor da classe herdada e incorporar o método **super()**. É através do método **super()** que fazemos acesso ao construtor da **Super Classe (Classe Mãe ...)**

Nota: quando uma classe herda de outra classe, a classe herdada é conhecida por:
[Pessoa] - é a classe herdada neste exemplo
- Super Classe

- Classe Mãe
- Classe Pai
- Classe Base
- Classe Genérica

Quando uma classe herda de outra classe, ela é chamada de:
[Cliente, Funcionario] - neste exemplo são as classes que herdam de outra classe
- Sub Classe
- Classe Filha
- Classe Específica

Veja no exemplo a seguir todo processo:

```python
class Pessoa:

    def __init__(self, nome, sobrenome, cpf,):
        self.__nome = nome
        self.__sobrenome = sobrenome
        self.__cpf = cpf

    def nome_completo(self):
        return f"{self.__nome} {self.__sobrenome}"

# Observe como tratamos herança. A classe Cliente é agora do tipo Pessoa, observe.


class Cliente(Pessoa):
    """Dizemos que Cliente Herda de Pessoa"""

    def __init__(self, nome, sobrenome, cpf, renda):
        super().__init__(nome, sobrenome, cpf)  # Acessa o construtor da Super Classe
        self.__renda = renda


# Observe como tratamos herança. A classe Funcionário é agora do tipo Pessoa, observe.


class Funcionario(Pessoa):
    """Dizemos que Funcionario Herda de Pessoa"""

    def __init__(self, nome, sobrenome, cpf, matricula):
        super().__init__(nome, sobrenome, cpf)  # Acessa o construtor da Super Classe
        self.__matricula = matricula
```

```python
# Instanciando os objetos

cliente1 = Cliente('Artur', 'Dutra', '123.456.789-00', 10000)
funcionario1 = Funcionario('Mario', 'Silva', '321.456.987-11', 1111)
```

8.1. Outro conceito importante em POO é o de Sobrescrita de Métodos (Overriding)

Sobrescrita de Métodos, ocorre quando reescrevemos/reimplementamos um método presente na Super Classe em classes filhas, a fim de, conferir ajustes que o tornem mais adequado, ou melhor dizendo ideal para a Sub Classe ou Classe Filha. Para demonstrar esse fato refatoraremos a classe **Funcionario** e sobrescreveremos o **Método** nome_completo, para que no lugar do sobrenome ele escreva o número de matrícula.

Observe o que foi feito na classe **Funcionario**:

```python
class Funcionario(Pessoa):
    """Dizemos que Funcionario Herda de Pessoa"""

    def __init__(self, nome, sobrenome, cpf, matricula):
        super().__init__(nome, sobrenome, cpf)  # Acessa o construtor da Super Classe
        self.__matricula = matricula

    def nome_completo(self):  # Reaproveitando o Método nome completo
        print(f"\nAcesso a Super Classe através do método super()\n{super().nome_completo()}")  # Observe que mesmo
        # sobre escrevendo, ao suar o método super() temos acesso a qualquuer método e atributos da Super Classe.
        print(f"\nAcessando cpf que já é uma herança de Pessoa, veja como fazer:\n{self._Pessoa__cpf}")
        return f"Funcionário: {self.__matricula} Nome: {self._Pessoa__nome}"
```

9. Propriedades – Properties

POO - Propriedades - **Properties**

A melhor forma de ter acesso a atributos privados é criando métodos para manipulá-los, esses métodos em especial possuem um nome específico, sendo conhecidos como **Getters** e **Setters.**

Em linguagens de programação como o Java, ao declararmos atributos privados nas classes, costumamos a criar métodos públicos para a manipulação desses atributos. Esses métodos são conhecidos por **Getters** e **Setters**. Neste processo, os **Getters** retornam o valor do atributo e os **Setters** alteram o valor dele.

Embora funcione muito bem em Python, aqui temos algo mais interessante que podemos aplicar, em Python o recomendado é trabalhar com **Properties**, essa sim é a forma **Pythonica** de acessar atributos privados utilizando Python.

Uma propriedade em Python é criada através de um **decorator** específico **@property.**

Um **Setter** em Python requer o uso de um **decorator** especial **@nome_atributo.setter**

Atenção: em Python é possível criar métodos com **@property**, portanto, fique atento ao seu projeto e busque sempre a melhor forma de implementá-lo. Não basta o sistema estar executando corretamente, é preciso que ele esteja utilizando a melhor e mais otimizada forma de implementação.

Acompanhe os exemplos a seguir para fixar os conceitos:

```
class Conta:
    contador = 0

    def __init__(self, titular, saldo, limite):
        self.__numero = Conta.contador + 1
        self.__titular = titular
        self.__saldo = saldo
        self.__limite = limite
        Conta.contador += 1

    # Trabalhando com Properties - Getters e Setters Pythonicos

    @property
    def numeros(self):
        return self.__numero

    @property
    def titular(self):
        return self.__titular

    @property
    def saldo(self):
        return self.__saldo

    @property
    def limite(self):
        return self.__limite

    @limite.setter
    def limite(self, novo_limite):
        self.__limite = novo_limite
```

```
    # Métodos de Instâncias

    def extrato(self):
        return f"\nSaldo de {self.__saldo} do cliente {self.__titular}"

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

    def sacar(self, valor):
        self.__saldo -= valor

    def transferir(self, valor, destino):
        self.__saldo -= valor
        destino.saldo += valor

    # Criando um método utilizando @property

    @property
    def valor_total(self):  # observe que não exsiste o atributo valor total, mas podemos criar um método para essa
        # finalidade e ainda utilizar o @Property
        return self.__saldo + self.limite

# Intanciando e testando

conta1 = Conta('Artur', 3000, 5000)
conta2 = Conta('Mario', 1000, 2500)

print(f"Extrato da conta1:{conta1.extrato()}")
print(f"Extrato da conta2:{conta2.extrato()}")
print(f"Características da conta1:{conta1.__dict__}")
print(f"Características da conta2:{conta2.__dict__}")

# Usando properties - observe que o decorator nos permite acesso aos métodos sem o uso dos parênteses
soma = conta1.saldo + conta2.saldo
print(f"\nA soma dos saldos das contas correntes é: {soma}")

# Usando setters Pythonico

print(f"Características da conta1 antes do setter:{conta1.__dict__}")
conta1.limite = 80000
print(f"Características da conta1 após o setter:{conta1.__dict__}")

# Observe que o uso do @property permite acesso ao método sem o uso do parênteses (isso recebe o nome de propriedade)
print(f"Valor total da conta1:{conta1.valor_total}")
print(f"Valor total da conta2:{conta2.valor_total}")
```

10. Herança múltipla

Herança Múltipla nada mais é que a possibilidade de herdar de múltiplas classes, fazendo com que a classe filha herde todos os atributos e métodos de todas as classes herdadas.

Nota: a herança múltipla pode ser feita de duas maneiras, são elas:
- Por Multiderivação Direta
- Por Multiderivação Indireta

Obs.: para Python não há limites para Multiderivação seja ela Direta ou Indireta. Não importa se a derivação é direta ou indireta. A classe que realizar a herança herdará todos os atributos e métodos das Super Classes.

MRO *Method Resolution Order* é a ordem de execução dos métodos (quem será executado primeiro – observe o exemplo da classe Pinguim). São três as formas que Python oferece para que você possa conferir a ordem de execução dos métodos (MRO):

- Via propriedade de classe __mro__
- Via método mro()
- Via help

Observe a seguir um exemplo simples:

```
class Animal:

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

    @property
    def nome(self):
        return f"{self.__nome}"

    def cumprimentar(self):
        return f"Eu sou {self.__nome}"


class Aquatico(Animal):

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

    def nadar(self):
        return f"{self._Animal__nome} está nadando."

    def cumprimentar(self):
        return f"Eu sou {self._Animal__nome} do mar!"


class Terrestre(Animal):

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

    def andar(self):
        return f"\n{self._Animal__nome} está andando."

    def cumprimentar(self):
        return f"Eu sou {self._Animal__nome} da terra!"
```

```python
class Pinguim(Terrestre, Aquatico):  # Fique atento em heranças Multiplas ao MRO Method Resolution Order → a ordem
    # que você instancia o objeto influencia na sua resposta aos métodos que foram sobrescritos, neste caso,
    # o método cumprimentar da classe Terrestre prevalecerá ao método cumprimentar da classe Aquatico.

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

```python
# Instanciando e testando ...

baleia = Aquatico('Willy')
tatu = Terrestre('Xim')
tux = Pinguim('Tux')

print(f"O objeto baleia possui as seguintes características {baleia.__dict__}")
print(f"O animal {baleia.nome}")
print(f"O/A {baleia.nadar()}")
print(f"O cumprimentodo/da {baleia.nome} é: {baleia.cumprimentar()}")

print(tatu.andar())
print(tatu.cumprimentar())

print(tux.andar())
print(tux.nadar())
print(tux.cumprimentar())

# Entendendo o MRO ...
# Podemos descobrir se um objeto é de uma classe ou outra, observe ...

print(f"Tux é uma instância de Pinguim? {isinstance(tux, Pinguim)}")  # True
print(f"Tux é uma instância de Aquatico? {isinstance(tux, Aquatico)}")  # True
print(f"Tux é uma instância de Terrestre? {isinstance(tux, Terrestre)}")  # True
print(f"Tux é uma instância de Animal? {isinstance(tux, Animal)}")  # True
print(f"Tux é uma instância de Object? {isinstance(tux, object)}")  # True
```

```python
# Verificando a ordem de execução MRO → ordem de execução dos métodos da classe em questão

print(f"\nPodemos observar a ordem de execução da classe assim: {Pinguim.__mro__}")
print(f"\nPodemos observar a ordem de execução da classe assim: {Pinguim.mro()}")
print(f"\nPodemos observar a ordem de execução da classe assim: {help(Pinguim)}")

"""
  |  Method resolution order:
  |      Pinguim
  |      Terrestre
  |      Aquatico
  |      Animal
  |      builtins.object
"""
```

11. Polimorfismo

Polimorfismo vem do Latim e significa: Poli – muitas; Morfis – formas.

Nota: quando nós reimplementamos um método presente na classe pai em classes filhas estamos realizando uma sobrescrita de métodos (***Overriding***). O ***Overriding*** é a melhor representação do **Polimorfismo**.

Observe no exemplo a seguir, que o ato de sobrescrever o método falar é como sabemos um ***Overriding*** que também é Polimorfismo, logo podemos dizer que Polimorfismo é um nome para ***Overriding***.

```
class Animal(object):

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

    def falar(self):  # Observe o uso do método raise que permite que se lance uma exceção
        raise NotImplementedError('A classe filha deve implementar esse método')

    def comer(self):
        print(f"{self.__nome} está comendo!!!")


class Cachorro(Animal):

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

    def falar(self):
        print(f"{self._Animal__nome} fala uau uau!")


class Gato(Animal):

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

    def falar(self):
        print(f"{self._Animal__nome} fala miauu!")


class Rato(Animal):

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

    def falar(self):
        print(f"{self._Animal__nome} fala roc roc!")
```

```
# Instanciando e testando ...

felix = Gato('Felix')
felix.comer()
felix.falar()

pluto = Cachorro('Pluto')
pluto.comer()
pluto.falar()

cerebro = Rato('Cerebro')
cerebro.comer()
cerebro.falar()
```

12. Sobrecarga

Em linhas gerais nada mais é que do que a capacidade de estarmos executando métodos ou funções de forma diferente a depender dos parâmetros. Em outras palavras através do controle dos parâmetros na OO podemos sobrecarregar em Python. Pode ser que em algum momento o projeto exija algum tipo de sobrecarga, a seguir dois exemplos de sobrecarga um de métodos e outro de operadores em Python.

```
class Classe:

    def metodo(self, parametro=''):
        if type(parametro) == str:
            print(f'o parâmetro é uma string: {parametro}')
        else:
            print(f'o parâmetro não é uma string: {parametro}')


c = Classe()
c.metodo('texto')
c.metodo(5)
```

```python
class Sobrecarga:

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

    # Observe que estamos criando métodos que serão sobrecarregados. Fique atento à Sintaxe

    def __add__(self, outro_numero):
        return self.numero + outro_numero.numero

    def __sub__(self, outro_numero):
        return self.numero - outro_numero.numero

    def __truediv__(self, outro_numero):
        return self.numero / outro_numero.numero

    def __mul__(self, outro_numero):
        return self.numero * outro_numero.numero

    def __mod__(self, outro_numero):
        return self.numero % outro_numero.numero

# Instanciando e testando ...

s1 = Sobrecarga(4)
s2 = Sobrecarga(2)

print(f"\nO tipo de s1 = {type(s1)}\nO tipo s2 = {type(s2)}")

# A seguir note que estamos utilizando operadores matemáticos nativos para operações com objetos

add = s1 + s2
sub = s1 - s2
div = s1 / s2
mul = s1 * s2
mod = s1 % s2

print(f"\n4 + 2 = {add}\n4 - 2 = {sub}\n4 / 2 = {div}\n4 * 2 = {mul}\n4 % 2 = {mod}")

# Utilizando os operadores na sua forma nativa

print(f"\n4 + 2 = {4 + 2}\n4 - 2 = {4 - 2}\n4 / 2 = {4 / 2}\n4 * 2 = {4 * 2}\n4 % 2 = {4 % 2}")
```

13. Bibliografia

[1] https://docs.python.org/3/
[2] Curso Intensivo de Python: Uma introdução prática e baseada em projetos à programação, Editora: Novatec Editora; 1ª edição (21 agosto 2017)
[3] Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos, Editora: Bookman; 1ª edição (1 janeiro 2000)

Niterói, 21 de outubro de 2021

Em conformidade com as propostas de trabalho do FriendsLab.

Prof. Dr. Vitor Hugo Ferreira
Coordenador de Projetos: FriendsLab

(21) 99969-8728
friendslab.sites.uff.br
vhferreira@id.uff.br

Rua Passos da Pátria, 156
Sala 335A, Bloco D
São Domingos, Niterói - RJ