PAE - Tutorial 1 - Python : Conceitos Básicos

Neste tutorial, são definidos conceitos básicos da linguagem de programação Python. Esta linguagem está sendo largamente utilizada para o desenvolvimento de software científico, como por exemplo, aprendizado de máquinas (*machine learning*) e aprendizado profundo (*deep learning*) que estão potencializando aplicações muito bem sucedidas na área de inteligência artificial.

No Tutorial 2, você aprenderá a base para desenvolver aplicações de grande interesse em várias modalidades de engenharia. Serão ilustradas diversas funções suportadas por bibliotecas disponíveis para utilização em aplicações desenvolvidas em Python. Mas para chegar a esse ponto é fundamental que você entenda bem os conceitos ilustrados neste tutorial.

Python é uma linguagem genérica, que permite o desenvolvimento de uma aplicação completa, não focada somente no desenvolvimento de métodos numéricos para engenharias. Uma outra característica não menos importante, é a facilidade de manipulação de conjuntos de dados em Python, que favorece a criação de um código compacto e versátil.

Para facilitar a leitura quando um parágrafo ou trecho de código não couber no final da página, será transferido para a próxima página, portanto, não estranhe espaços em branco ao final de uma página.

Neste tutorial, você aprenderá a programar em Python de forma incremental. Na próxima seção, você irá aprender a instalar Python e o ambiente de desenvolvimento que você irá utilizar para editar e executar o projeto apresentado neste tutorial, que irá crescendo incrementalmente de forma a ilustrar os conceitos que estão sendo apresentados.

No início, você verá um programa muito simples que irá gradativamente sendo aumentado para ilustrar cada novo conceito que você precisará conhecer para programar em Python. Desta forma, cada novo conceito será explicado e ilustrado na prática por um trecho de código, que você deverá digitar e executar. Esse processo de aprendizagem é muito efetivo, e será a base para que você seja capaz de desenvolver aplicações em Python de grande interesse para engenharias.

1 - Instalação do Python e do IDE PyCharm

Para fixar os conceitos que serão apresentados neste tutorial, é fundamental que cada trecho de código apresentado seja implementado e testado. Desta forma, será necessário instalar:

- a versão estável mais recente do Python;
- e a versão mais recente do IDE (*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado) PyCharm da Jet Brains.

No site https://www.python.org/, selecione a opção Downloads e a opção da versão mais recente do Python para Windows (atualmente: Python 3.9.6), baixe e instale no seu computador o arquivo executável da versão mais recente do Python (por exemplo: python-3.9.6-amd64.exe).

No site https://www.jetbrains.com/pt-br/pycharm/ selecione a opção Baixar e novamente esta opção para a baixar a versão gratuita Community do PyCharm, baixe e instale em seu computador o arquivo executável da versão mais recente do PyCharm Community (por exemplo: pycharm-community-2021.1.3.exe).

Para implementar e testar os trechos de código apresentados nesta apostila, crie um projeto no IDE PyCharm com o nome PAE (sigla de Programação Aplicada à Engenharia):

- selecione File -- New Project
- na janela Create Project
  - mantenha a seleção de New environment using para criar um ambiente específico para o seu projeto
    - em Location
      - navegue para escolher um diretório para armazenar o seu projeto
      - complete o diretório com o nome do projeto : PAE
    - em Basic Interpreter
      - selecione a versão Python 3.9
  - mantenha a seleção de Create a main.py welcome script
  - finalize a criação ativando o botão Create

Crie um diretório padrão para armazenar os diretórios e arquivos fontes do seu projeto:

- selecione a aba Project no canto esquerdo da tela para visualizar a janela lateral Project
- clique na linha com o nome do seu projeto e selecione, com o botão direito do mouse
  - New -- Directory
    - preencha com o nome src (abreviação para source)
      - diretório onde tradicionalmente são armazenados os arquivos fontes do projeto
        - em geral organizados em subdiretórios de src
- selecione o arquivo main.py e mova-o para o diretório src

Ajuste o arquivo main.py:

- na janela de projeto (lateral esquerda)
  - clique no arquivo main.py para visualizar o seu código na janela à direita da janela de projeto
- remova a seleção de ponto de parada para teste passo a passo (debug)
  - clicando no círculo vermelho no lado esquerdo de uma das linhas do código
- remova todos os comentários : strings iniciados com o caracter #
- altere o código do arquivo main.py para ficar da seguinte forma:

```
def imprimir(linguagem_programação):
    print(f'Vamos aprender a programar na linguagem {linguagem_programação}.')

if __name__ == '__main__':
    imprimir('Python')
```

Execute o programa, clicando na seta verde no canto superior direito da sua janela:

Você visualizará o resultado do processo na janela Run (execução)

```
Vamos aprender a programar na linguagem Python.

Process finished with exit code 0
```

O código em Python é organizado em blocos de comandos. Quanto mais indentado (com Tab) for o bloco de comandos, mais interno é o bloco.

Na execução, o Python atribui à variável pré-definida __name__ o string '__main__', e executa o corpo principal do programa, que neste caso tem somente o comando condicional **if**. Variáveis pré_definidas são cercadas por duplo underline, como é o caso da variável __name__.

O comando condicional if testa com sucesso que a variável __name__ é igual ao (contém o mesmo valor que) string '__main__'. Pelo fato do teste do comando if ser bem sucedido, o seu corpo interno de comandos é executado: neste exemplo, somente a chamada da função imprimir, passando como argumento o string 'Python'.

A função imprimir recebe o parâmetro (entre parênteses) linguagem_programação, para o qual foi atribuído como argumento o string 'Python'. Ao ser chamada a função executa o seu corpo interno de comandos, que neste exemplo, é composto somente pela chamada da função pré-definida (*builtin*) print, utilizada para imprimir strings. Funções pré-definidas são disponíveis em Python sem que seja necessário defini-las.

O argumento passado para a função print é um string formatado (f'...') para compor trechos fixos ('Vamos aprender a programar na linguagem ' e '.') com trechos variáveis (entre chaves: { }). O trecho variável recebe o valor passado como argumento ('Python') para o parâmetro linguagem_programação. Como resultado é impresso na tela do seu computador, o string: Vamos aprender a programar na linguagem Python.

O resultado da execução é mostrado na janela de execução (*Run*). Adicionalmente, como mensagem de final de execução é impressa a mensagem: Process finished with exit code 0, cuja tradução é: Processo concluído com código de saída 0. O código 0 indica que a execução foi concluída com sucesso. Caso seja reportado o código 1, a mensagem indica execução mal sucedida. Esta mensagem se repete em todas as execuções e será omitida nas ilustrações da seção 2.

Este exemplo muito simples, com uma breve explicação, é o seu primeiro contato com a execução de um programa em Python. Na próxima seção, vamos conceituar e ilustrar variáveis, tipos, funções e comandos condicionais.

2 - Variáveis, Tipos, Funções e Comandos Condicionais

Nesta seção vamos ilustrar vários conceitos de forma incremental. Para facilitar o entendimento dos conceitos, em cada subseção serão ilustrados somente a função e o trecho de código do programa principal, tratados na respectiva subseção.

O código completo do programa é formado pelo código de todas as funções e pelo corpo do programa definido com base na concatenação de todos os trechos ilustrados nas subseções. Portanto, ao estudar cada subseção acrescente ao código do seu programa, os códigos parciais apresentados na seção que você está estudando. Ao concluir o estudo das quatro subseções, você terá o programa completo da seção 2.

Variáveis são utilizadas para armazenar valores que são utilizados para serem processados por funções. Comando condicionais são são utilizados para executar blocos internos de comandos quando uma determinada condição de execução é verdadeira.

2.1 - Definição de variáveis de vários tipos e impressão por chamada de função

Um módulo é um arquivo, com extensão py, que contém código Python: funções, classes (veremos na seção 5) e variáveis.

Funções utilizadas nesta seção serão definidas em um módulo Python separado. Para criar esse módulo:

- selecione o diretório src dos fontes do seu projeto
- com o botão direito do mouse selecione as operações: New -- Python File
- no campo Name informe o nome do arquivo (módulo):
  - variáveis__tipos__funções__comandos_condicionais

No módulo variáveis__tipos__funções__comandos_condicionais defina uma função com o seguinte código:

```
def imprimir_variável(nome_variável, variável):
    print(f'variável {nome_variável} do tipo {type(variável)} com o valor : {variável}')
```

No módulo main defina o seguinte código:

```
from variáveis__tipos__funções__comandos_condicionais import imprimir_variável

def ilustrar__variáveis__tipos__funções__comandos_condicionais():
    print('\n1 - definição de variáveis de vários tipos e impressão por chamada de função')
    disciplina = 'Programação Aplicada à Engenharia'
    carga_horária = 72
    nota_mínima_aprovação = 6.0
    é_disciplina_externa = True
    imprimir_variável('disciplina', disciplina)
    imprimir_variável('carga_horária', carga_horária)
    imprimir_variável('nota_mínima_aprovação', nota_mínima_aprovação)
    imprimir_variável('é_disciplina_externa', é_disciplina_externa)


if __name__ == '__main__':
    ilustrar__variáveis__tipos__funções__comandos_condicionais()
```

Para utilizar uma função definida em outro módulo você precisa importar essa função do seu módulo de origem. As palavras reservadas **from** e **import** são utilizadas para importar a função imprimir_variável do módulo variáveis__tipos__funções__comandos_condicionais.

O corpo do programa principal (bloco interno ao comando: if __name__ == '__main__':) é composto pela chamada da função ilustrar__variáveis__tipos__funções__comandos_condicionais.

Execute o programa. Você verá o seguinte resultado:

```
2.1 - definição de variáveis de vários tipos e impressão por chamada de função
variável disciplina do tipo <class 'str'> com o valor : Programação Aplicada à Engenharia
variável carga_horária do tipo <class 'int'> com o valor : 72
variável nota_mínima_aprovação do tipo <class 'float'> com o valor : 6.0
variável é_disciplina_externa do tipo <class 'bool'> com o valor : True
```

Em todas as ilustrações que serão apresentadas neste tutorial, a primeira linha será utilizada para imprimir (utilizando a função print) o número da subseção sendo ilustrada (neste caso: 2.1) e o seu conteúdo. Os caracteres \n, utilizados no string a ser impresso, indicam que deve ser pulada uma linha antes de realizar a impressão do string.

Vamos descrever o código da subseção 2.1. Como já vimos no final da seção 1, quando o programa é executado, ocorre a execução do corpo do programa principal, que neste exemplo, é somente a chamada da função ilustrar__variáveis__tipos__funções__comandos_condicionais.

Na função ilustrar__variáveis__tipos__funções__comandos_condicionais são ilustrados:

- comandos de atribuição, no quais quatro variáveis são criadas e inicializadas com valores de diferentes tipos;
- chamadas da função imprimir_variável, para as quatro variáveis criadas anteriormente.

Variáveis são utilizadas para armazenar dados (ex: nota_mínima_aprovação = 6.0) que serão utilizados na execução do seu programa. O nome de uma variável pode ser composto por uma única palavra (ex: disciplina) com letras minúsculas ou por várias palavras interligadas por *underline* (ex: carga_horária).

Uma variável pode ser inicializada com um valor de um determinado tipo. Esse valor poderá ser alterado durante a execução do programa. Por enquanto, vamos considerar que os valores atribuídos a uma variável podem ser dos seguintes tipos:

- str : string --- ex: disciplina = 'Programação Aplicada à Engenharia';
- int : inteiro positivo ou negativo --- ex: carga_horária = 72;
- float : ponto flutuante positivo ou negativo --- ex: nota_mínima_aprovação = 6.0;
- bool : boolean com valores True ou False --- ex: é_disciplina_externa = True.

Função tem um nome, a partir do qual é chamada, e uma lista de parâmetros, para os quais são atribuídos uma lista de argumentos na chamada da função. Apesar do nome de uma função ser formado por palavras separadas por *underline* (ou por uma única palavra), da mesma forma que uma variável, é recomendável que a primeira palavra seja um verbo no infinitivo para designar a ação que será realizada na execução da função (ex: imprimir_variável) ou um substantivo designando uma função sendo realizada (ex: fatorial).

Quando o nome da função por formado por nomes compostos, fica mais legível separar o nome composto (ou os nomes compostos, se for o caso) por dois caracteres *underline*, como por exemplo na função ilustrar__variáveis__tipos__funções__comandos_condicionais. Neste caso, as partes do nome da função (ilustrar, variáveis, tipos, funções, comandos_condicionais) foram separadas por dois caracteres *underlines*, para diferenciar da separação das palavras do nome composto comandos_condicionais, separadas por um único caracter *underline*.

Parâmetros são variáveis cujo escopo é o bloco interno de uma função. O escopo é o intervalo de linhas do programa na qual a variável é válida e acessível. No caso de uma função os parâmetros são válidos somente para utilização nos comandos do corpo interno da função. O parâmetro é nomeado da mesma forma que uma variável (ex: nome_variável) . Na chamada da função é passado um argumento para cada um dos parâmetros da função.

O argumento é um valor, uma variável, ou uma chamada de função. No código atual são passados: um valor e uma variável. Tomando como exemplo a chamada imprimir_variável('carga_horária', carga_horária):

- o primeiro argumento passado é o valor 'carga_horária' do tipo str, que como consequência da chamada de função é atribuído ao parâmetro nome_variável da função imprimir_variável;
- e o segundo argumento passado é a variável carga_horária, cujo valor do tipo int atribuido à variável previamente (carga_horária = 72), é atribuído ao parâmetro variável da função imprimir_variável.

A definição de uma função inicia com a palavra reservada **def** e é separada de seu corpo interno de comandos (indentados com Tab) pelo caracter **:**. Uma palavra reservada faz parte da linguagem e não pode ser utilizada como nome de variável. Neste tutorial, para chamar atenção, as palavras reservas da linguagem Python são ilustradas em negrito.

A sua lista de parâmetros é cercada por parênteses. Esta lista pode ter vários, apenas um ou nenhum parâmetro. Mesmo que a lista seja vazia, os parênteses precisam ser representados na definição e na chamada da função.

O corpo de comandos da função imprimir_variável é composto somente pela chamada da função print, que imprime um string formatado com f'...', concatenando strings fixos e strings gerados a partir da utilização de variáveis (ex: nome_variável) ou de funções (ex: type(variável)), encapsulados por chaves ({ }). A função type é utilizada para retornar o tipo de uma variável.

2.2 - Utilizando comando condicional e operador + para concatenar strings

No módulo variáveis__tipos__funções__comandos_condicionais, acrescente a definição da função calcular_status_aprovação_aluno:

```
def calcular_status_aprovação_aluno(média):
    if média >= 6.0: return 'aprovado'
    elif 4.0 <= média < 6.0: return 'de exame'
    else: return 'reprovado'
```

Para ilustrar a chamada de uma função que utiliza um comando condicional, e a utilização do operador + para concatenar strings, comente o código do trecho de ilustração da seção 2.1 e acrescente o código de ilustração da seção 2.2 na definição da função de ilustração da seção 2:

```
def ilustrar_variáveis_tipos_funções_comandos_condicionais():
    # print('\n2.1 - definição de variáveis de vários tipos e impressão por chamada de função')
    # disciplina = 'Programação Aplicada à Engenharia'
    # carga_horária = 72
    # nota_mínima_aprovação = 6.0
    # é_disciplina_externa = True
    # imprimir_variável('disciplina', disciplina)
    # imprimir_variável('carga_horária', carga_horária)
    # imprimir_variável('nota_mínima_aprovação', nota_mínima_aprovação)
    # imprimir_variável('é_disciplina_externa', é_disciplina_externa)
    print('\n2.2 - função com comando condicional e operador + para concatenar strings')
    nota_aluno = 5.5
    print('status do aluno : ' + calcular_status_aprovação_aluno(nota_aluno))
    print('nota do aluno : ' + str(nota_aluno))
```

Para comentar um trecho de código, no IDE PyCharm, basta selecionar as linhas do trecho de código a ser comentado e utilizar o comando Ctrl /. Quando você quiser remover os comentários, basta selecionar o trecho comentado e utilizar novamento o comando Ctrl /.

Para que a função calcular_status_aprovação_aluno possa ser utilizada no módulo main, será necessário incluir a importação dessa função:

```
from variáveis_tipos_funções_comandos_condicionais import imprimir_variável,\
calcular_status_aprovação_aluno
```

O caracter \ é utilizado para informar que o código da linha continua na linha seguinte. Você só precisará utilizá-lo quando o código de uma linha tiver que ser representado em mais de uma linha.

Lembre-se nas próximas subseções, de acrescentar as importações das funções originárias de outros módulos.

A saída da execução do programa é:

```
2.2 - função com comando condicional e operador + para concatenar strings
status do aluno : de exame
nota do aluno : 5.5
```

As duas chamadas da função print utilizam o operador + para concatenar strings. Na primeira chamada é passado como um dos argumentos a chamada da função obter_status_aprovação_aluno. O corpo desta função é composto por um comando condicional composto de três partes:

- teste da condição do **if** (se) : **if** média >= 6.0:
  - condição testada : se média maior ou igual a 6.0
  - caso este teste seja bem sucedido
    - execução de bloco interno: **return** 'aprovado'
- caso o teste da condição do **if** seja mal sucedido
  - teste da condição do **elif** (senão se) : **elif** 4.0 <= média < 6.0:
    - condição testada : se média maior ou igual a 4.0 e menor que 6.0
    - caso este teste seja bem sucedido
      - execução de bloco interno : **return** 'de exame'
- caso teste da condição do **elif** seja mal sucedido
  - execução de bloco interno ao **else** (senão) : **return** 'reprovado'

Devido ao fato, de que neste código ilustrativo, os blocos internos do **if**, **elif** e **else** tem apenas um único comando, o implementador tem a opção de alinhar esse comando após o caracter **:**, em vez de indentar o comando na próxima linha, desde que o código alinhado caiba na mesma linha.

Assim sendo, a forma indentada (em geral, encontrada nos códigos Python)

```
def calcular_status_aprovação_aluno(média):
    if média >= 6.0:
        return 'aprovado'
    elif 4.0 <= média < 6.0:
        return 'de exame'
    else:
        return 'reprovado'
```

poderá ser substituída pela forma alinhada (mais compacta).

```
def calcular_status_aprovação_aluno(média):
    if média >= 6.0: return 'aprovado'
    elif 4.0 <= média < 6.0: return 'de exame'
    else: return 'reprovado'
```

A palavra reservada **return** é utilizada para retornar um valor, ou uma lista de valores (entre vírgulas). Observe que uma função pode não retornar nenhum valor, tal como a função imprimir_variável ilustrada na subseção 2.1.

Na segunda chamada da função print em um dos argumentos passados é utilizada a chamada da função str, para converter o valor obtido da variável nota_aluno do tipo float para o tipo str. Essa conversão é necessária, porque para utilizar o operador + para concatenar strings é necessário que as partes concatenadas sejam strings. Essa conversão não foi necessária na primeira chamada, porque a função obter_status_aprovação_aluno retorna um valor do tipo str. É importante pontuar que somente o valor lido é convertido, mas que o tipo da variável continua associado ao valor atribuído à variável, ou seja, a variável nota_aluno continua com o valor 5.5 do tipo float.

2.3 - Utilizando comando condicional aninhado (interno)

Um comando condicional pode ter no bloco de comandos de qualquer uma das duas partes, um outro comando condicional. Esse comando condicional interno (contido por outro comando condicional externo) é chamado de comando condicional aninhado, porque sua utilização ocorre em um bloco de comandos de um comando condicional externo.

No módulo variáveis__tipos__funções__comandos_condicionais, acrescente a definição da função calcular_expectativa_aprovação_aluno:

```
def calcular_expectativa_aprovação_aluno(estudo_antecipado_tutoriais,
    teste_implementações_tutoriais, percentual_realização_exercícios_propostos):
    if percentual_realização_exercícios_propostos == 100:
        if estudo_antecipado_tutoriais and teste_implementações_tutoriais == 'completo':
            return 'expectativa muito alta de aprovação',\
                   '100% dos exercícios e estudo completo dos tutoriais'
        elif estudo_antecipado_tutoriais and teste_implementações_tutoriais == 'parcial':
            return 'expectativa alta de aprovação',\
                   ' 100% dos exercícios e estudo parcial dos tutoriais'
        else: return 'expectativa média de aprovação',\
                     ' 100% dos exercícios e nenhum estudo dos tutoriais'
    elif 70 <= percentual_realização_exercícios_propostos < 100:
        if estudo_antecipado_tutoriais and teste_implementações_tutoriais == 'completo':
            return 'expectativa alta de aprovação',\
                   'pelo menos 70% dos exercícios e estudo completo dos tutoriais'
        else: return 'expectativa média de aprovação',\
                     'pelo menos 70% dos exercícios e estudo parcial dos tutoriais'
    elif 50 <= percentual_realização_exercícios_propostos < 70:
        if estudo_antecipado_tutoriais and teste_implementações_tutoriais == 'completo':
            return 'expectativa média de aprovação',\
                   'pelo menos 50% dos exercícios e estudo completo dos tutoriais'
        else: return 'expectativa baixa de aprovação',\
                     'pelo menos 50% dos exercícios e estudo parcial dos tutoriais'
    else:
        if estudo_antecipado_tutoriais and teste_implementações_tutoriais == 'completo':
            return 'expectativa baixa de aprovação',\
                   'menos de 50% dos exercícios e estudo completo dos tutoriais'
        else: return 'expectativa muito baixa de aprovação',\
                     'menos de 50% dos exercícios e estudo parcial dos tutoriais'
```

Na lista de valores retornados, o caracter \ é utilizado para representar quebra de linha, pois a lista de strings retornada não cabe em uma única linha.

Para ilustrar chamadas de uma função que utiliza comandos condicionais aninhados, comente os trechos de ilustração das subseções 2.1 e 2.2 e acrescente o trecho de ilustração da seção 2.3. Por simplicidade, os trechos comentados foram omitidas no código ilustrado a seguir.

```
def ilustrar__variáveis__tipos__funções__comandos_condicionais():
    print('\n2.3 - chamadas de função que utiliza comandos condicionais aninhados')
    expectiva_aprovação, justificativa = calcular_expectativa_aprovação_aluno(
        estudo_antecipado_tutoriais = True, teste_implementações_tutoriais = 'completo',
        percentual_realização_exercícios_propostos = 100)
    print("para : estudo_antecipado_tutoriais = True, teste_implementações_tutoriais = 'completo',
           percentual_realização_exercícios_propostos = 100")
    print('- ' + expectiva_aprovação + '\n- justificativa: ' + justificativa)
    expectiva_aprovação, justificativa = calcular_expectativa_aprovação_aluno(
        estudo_antecipado_tutoriais = True, teste_implementações_tutoriais = 'parcial',
        percentual_realização_exercícios_propostos = 80)
    print("para : estudo_antecipado_tutoriais = True, teste_implementações_tutoriais = 'parcial',
           percentual_realização_exercícios_propostos = 80")
    print('- ' + expectiva_aprovação + '\n- justificativa: ' + justificativa)
```

O caracter **(** ao final da linha, ou o caracter **,** separando uma lista de argumentos, dispensam o uso do caracter \ para representar quebra de linha.

A saída da execução do programa é:

```
2.3 - chamadas de função que utiliza comandos condicionais aninhados
para : estudo_antecipado_tutoriais = True, teste_implementações_tutoriais = 'completo',
       percentual_realização_exercícios_propostos = 100
- expectativa muito alta de aprovação
- justificativa: 100% dos exercícios e estudo completo dos tutoriais
para : estudo_antecipado_tutoriais = True, teste_implementações_tutoriais = 'parcial',
       percentual_realização_exercícios_propostos = 80
- expectativa média de aprovação
- justificativa: pelo menos 70% dos exercícios e estudo parcial dos tutoriais
```

Na lista de argumentos passados para uma função, os caracteres **(** ou **,** são utilizados para representar quebra de linha em substituição ao caracter \.

Comando condicionais aninhados são comandos condicionais que fazem parte do bloco de comandos interno de qualquer uma das três partes de outro comando condicional.

Para que um bloco interno a uma parte de um comando condicional seja executado a condição testada deve ser verdadeira (True). Se a variável não for booleana é necessário compará-la com algum valor. Por exemplo: **if** percentual_realização_exercícios_propostos < 100. No entanto, se a variável da condição for booleana não é necessário compará-la com True ou False, porque o valor de uma variável booleana já é True ou False.

A comparação utilizando uma variável booleana

- **if** estudo_antecipado_tutoriais:

é uma simplificação da comparação

- **if** estudo_antecipado_tutoriais == True:

Pelo mesmo motivo, a comparação:

- **if not** estudo_antecipado_tutoriais:

é uma simplificação da comparação

- **if** estudo_antecipado_tutoriais == False:

Conforme ilustrado na função calcular_expectativa_aprovação_aluno, as partes **if** e **else** podem ser utilizadas uma única vez no comando condicional; no entanto, a parte **elif** pode ser utilizada várias vezes, testando condições distintas.

2.4 - Função recursiva

Nesta seção, serão ilustradas duas formas de calcular o fatorial de um número inteiro: (a) utilizando iteração; ou (b) utilizando recursividade. Uma função que utiliza iteração executa um bloco de comandos repetidas vezes (iteração). Uma função recursiva chama a si própria.

No módulo variáveis__tipos__funções__comandos_condicionais, acrescente a definição das seguintes funções:

```
def fatorial2(n):
    if n == 1: return 1
    else: return n * fatorial2(n - 1)

def fatorial1(n):
    fatorial = n
    while n > 1:
        fatorial *= n - 1
        n -= 1
    return fatorial
```

Para ilustrar a implementação de fatorial por uma função utilizando iteração e por outra utilizando recursão, comente os trechos de ilustração das subseções anteriores e acrescente o trecho de ilustração da seção 2.4.

```
def ilustrar__variáveis__tipos__funções__comandos_condicionais():
    print('\n2.4 - chamada de função com implementação recursiva ou não recursiva com iteração')
    n = 5
    print(f'função não recursiva (utilizando iteração) : fatorial({n}) = {fatorial1(n)}')
    print(f'função recursiva : fatorial2({n}) = {fatorial2(n)}')
```

A saída da execução do programa é:

```
2.4 - chamada de função com implementação recursiva ou não recursiva com iteração
função não recursiva (utilizando iteração) : fatorial(5) = 120
função recursiva : fatorial2(5) = 120
```

É possível representar fatorial(n) = n x (n - 1) ... 1. Na implementação da função fatorial1 é utilizado o comando de iteração while. Este comando testa uma condição de continuidade da iteração, neste exemplo: n > 1. Enquanto a condição for verdadeira, o bloco interno ao comando será executado iterativamente (repetidas vezes). Em Python o operador * representa uma multiplicação.

Também é possível representar fatorial(n) = n x fatorial (n - 1). Na implementação da função fatorial2, a função chama a si própria (recursividade), tornando o código mais compacto.

Uma soma, multiplicação, divisão e subtração podem ser representadas de uma forma mais compacta. Por exemplo, a soma x = x + 8 pode ser representada por x += 8. Esse tipo de simplificação foi utilizado para realizar uma multiplicação e uma subtração na função fatorial1.

Ao concluir a seção 2, você pode remover os comentários dos trechos de código que haviam sido comentados.

3 - Estruturas de Dados e Comandos de Iteração

Um conjunto de dados é ordenado (ou indexado), quando seus elementos podem ser lidos ou alterados a partir de um índice ou de um intervalo de índices (subconjunto de elementos do conjunto). Portanto, um conjunto não ordenado não pode ser indexado.

O índice inicial de um conjunto de elementos é 0 (zero). Para indexar um conjunto de elementos você pode utilizar uma notação que indica o intervalo de valores a ser acessado: [índice_inicial_inclusivo : índice_final_exclusivo]. Índice inicial inclusivo significa que intervalo de valores vai iniciar com esse índice. Índice final exclusivo significa que intervalo de valores vai finalizar com esse índice menos 1. Por exemplo, se conjunto for indexado no intervalo [3:9], serão acessados os elementos do conjunto do índice 3 até o índice 8.

Um conjuntos de dados é imutável quando após a atribuição dos seus elementos, um novo elemento não pode ser inserido, e seus elementos não podem ser alterados ou removidos.

Você não necessita utilizar uma biblioteca externa para criar e utilizar os conjuntos de dados incorporados na linguagem Python de uma forma bastante compacta.

Para ilustrar essa seção crie o módulo Python estruturas_dados__comandos_iteração.

3.1 - String : conjunto ordenado de caracteres

String é utilizado especificamente para manipular um conjunto de caracteres.

No módulo estruturas_dados__comandos_iteração defina as seguintes funções:

```
def converter_texto(texto, tipo):
    if tipo == 'maiúscula': return texto.upper()
    elif tipo == 'minúscula': return texto.lower()
    elif tipo == 'capital' : return texto.captalize()
    else: return 'tipo de conversão desconhecido'

def contar_caracteres_tipo(texto, tipo):
    total_caracteres_tipo = 0
    for caracter in texto:
        if tipo == 'letra_maiúscula' and caracter.isupper(): total_caracteres_tipo += 1
        elif tipo == 'letra_minúscula' and caracter.islower(): total_caracteres_tipo += 1
        elif tipo == 'número' and caracter.isnumeric(): total_caracteres_tipo += 1
    return total_caracteres_tipo

def mostrar_mensagem(objetivo, texto, tipo, resultado):
    print("texto '%s' - %s %s : %s" % (texto, objetivo, tipo, resultado))
```

No módulo main importe essas funções.

```
from estruturas_dados__comandos_iteração import mostrar_mensagem, contar_caracteres_tipo,
     converter_texto
```

No módulo main defina a função ilustrar__estruturas_dados__comandos_iteração com o trecho de código de ilustração da subseção 3.1.

```
def ilustrar__estruturas_dados__comandos_iteração():
    print('\n3.1 - manipular strings')
    texto = '2021 - Programação Aplicada à Engenharia'
    print("texto '%s' limitado no intervalo [7:18] : %s" % (texto, texto[7:18]))
    print("texto '%s' limitado no intervalo [-10:] : %s" % (texto, texto[-10:]))
    ação = 'contar caracteres com'
    tipo = 'letra_maiúscula'
    mostrar_mensagem(ação, texto, tipo, contar_caracteres_tipo(texto, tipo))
    tipo = 'número'
    mostrar_mensagem(ação, texto, tipo, contar_caracteres_tipo(texto, tipo))
    ação = 'converter para'
    tipo = 'minúscula'
    mostrar_mensagem(ação, texto, tipo, converter_texto(texto, tipo))
    tipo = 'maiúscula'
    mostrar_mensagem(ação, texto, tipo, converter_texto(texto, tipo))
```

No corpo principal do programa, comente a chamada da função de ilustração da seção 2 e acrescente a chamada da função de ilustração da seção 3.

```
if __name__ == '__main__':
    # ilustrar__variáveis__tipos__funções__comandos_condicionais()
    ilustrar__estruturas_dados__comandos_iteração()
```

A saída da execução do programa é:

```
3.1 - manipular strings
texto '2021 - Programação Aplicada à Engenharia' limitado no intervalo [7:18] : Programação
texto '2021 - Programação Aplicada à Engenharia' limitado no intervalo [-10:] : Engenharia
texto '2021 - Programação Aplicada à Engenharia' - contar caracteres com letra_maiúscula : 3
texto '2021 - Programação Aplicada à Engenharia' - contar caracteres com número : 4
texto '2021 - Programação Aplicada à Engenharia' - converter para minúscula :
    2021 - programação aplicada à engenharia
texto '2021 - Programação Aplicada à Engenharia' - converter para maiúscula :
    2021 - PROGRAMAÇÃO APLICADA À ENGENHARIA
```

Na função mostrar_mensagem são utilizados demarcadores %s para compor o string passado como argumento na chamada da função print:

- print("texto '%s' - %s %s : %s" % (texto, objetivo, tipo, resultado))

Os demarcadores %s indicam que posições no string que serão substituídas pelas variáveis entre parênteses após o caracter %. Ou seja, o primeiro %s será substituído pelo valor do parâmetro texto, o segundo %s pelo valor do parâmetro objetivo, o terceiro %s pelo valor do parâmetro tipo e o quarto %s pelo valor do parâmetro resultado.

No variável texto = '2021 - Programação Aplicada à Engenharia':

- o primeiro índice (0) corresponde ao caracter '2' que inicia o substring '2021';
- e o último índice (39) corresponde ao caracter 'a' que finaliza o substring 'Engenharia'.

'Programação' é o substring obtido de texto[7:18], que corresponde à indexação do texto no intervalo 7 até 17, dado que o índice final do intervalo é exclusivo, ou seja, não incluído no intervalo.

'Engenharia' é o substring obtido de texto[-10:]. O índice inicial como -10 significa 10 índices anteriores ao final do string. O índice final exclusivo é 40, dado que o último índice incluído no intervalo é 39. Então para o índice inicial -10 corresponde a 40 - 10 = 30. O índice final omitido significa o índice final exclusivo, que no caso é 40. Portanto texto[-10:] é equivalente a texto[30:40].

Na função contar_caracteres_tipo é utilizado o comando de iteração: **for** caracter **in** texto:. Esse comando copia cada caracter da variável texto, do tipo str, e atribui à variável caracter a cada repetição de execução do bloco de comandos interno ao comando **for**. Então, a cada repetição de execução (loop) do bloco interno do comando **for**, a variável caracter é disponibilizada com o valor do próximo caracter do string atribuído anteriormente à variável texto, para ser utilizada com uma variável nos comandos do bloco interno do comando for. Por exemplo, para texto = 'Programação', na primeira execução do comando a variável caracter recebe o valor 'P' (primeiro caracter do string 'Programação'). Na próxima execução recebe o valor 'r' (segundo caracter), e assim por diante, até assumir o valor 'a' (último caracter do string 'Programação'). Note que as variáveis caracter e texto, poderiam ter qualquer outro nome.

A função contar_caracteres_tipo utiliza três funções aplicáveis a strings:

- isupper : retorna True se todas os caracteres do string (ex: string.isupper()) são letras maísculas;
- islower : idem para letras minúsculas;
- isnumeric : idem para caracteres numéricos.

Note que neste caso, a função está sendo aplicada a um string que contém um único caracter, com o objetivo de totalizar quantos caracteres de um dado tipo (letra maíscula, letra minúscula ou caracter numérico) existem no string do parâmetro texto.

Na função converter_texto, são utilizadas as seguintes funções aplicáveis a strings::

- upper : converte todos as letras de um string para letras maísculas;
- lower : idem para letras minúsculas;
- captalize : converte todas as palavras de um string para palavras iniciando com a primeira letra maiúscula e os demais letras da palavra com letras minúsculas.

O objetivo da função converter_texto é o de converter as letras de um string para o tipo passado como argumento.

3.2 - Lista (list) : conjunto ordenado

Lista suporta o armazenamento de qualquer tipo de elemento, e aceita elementos duplicados. No módulo estruturas_dados__comandos_iteração defina a seguinte função:

```
def converter_valores_negativos_matrizes(matriz):
    matriz_convertida = []
    for índice_linha, linha in enumerate(matriz):
        linha_convertida = []
        for índice_coluna, valor in enumerate(linha):
            if valor >= 0: linha_convertida.append(valor)
            else: linha_convertida.append(-valor)
        matriz_convertida.append(linha_convertida)
    return matriz_convertida
```

Para ilustrar o uso de listas, na função ilustrar__estruturas_dados__comandos_iteração comente o trecho que ilustra a subseção 3.1 (será omitido no código mostrado a seguir) e inclua o trecho que ilustra a seção 3.2.

```
def ilustrar__estruturas_dados__comandos_iteração():
    print('\n3.2 - manipular listas')
    cursos_engenharia_faen = ['alimentos', 'energia', 'produção']
    print('conjunto inicial : ' + str(cursos_engenharia_faen))
    cursos_engenharia_faen.insert(1, 'civil')
    cursos_engenharia_faen.insert(3, 'mecânica')
    print('inserindo elementos na posição 1 e 3 do conjunto : ' + str(cursos_engenharia_faen))
    print('intervalo [2:4] de elementos do conjunto : ' + str(cursos_engenharia_faen[2:4]))
    cursos_engenharia_faen.append('química')
    print('apendando elemento no conjunto : ' + str(cursos_engenharia_faen))
    del cursos_engenharia_faen[-1]
    print('removendo último elemento no conjunto : ' + str(cursos_engenharia_faen))
    del cursos_engenharia_faen[0:3]
    print('removendo os 3 primeiros elementos do conjunto : ' + str(cursos_engenharia_faen))
    matriz = [[2, -7, 2], [8, 5, -4]]f
    print('matriz com números negativos : ' + str(matriz))
    print('matriz com números negativos convertidos para positivos : '
       + str(converter_valores_negativos_matrizes(matriz)))
```

A saída da execução do programa é:

```
3.2 - manipular listas
conjunto inicial : ['alimentos', 'energia', 'produção']
inserindo elementos na posição 1 e 3 do conjunto : ['alimentos', 'civil', 'energia', 'mecânica',
'produção']
intervalo [2:4] de elementos do conjunto : ['energia', 'mecânica']
apendando elemento no conjunto : ['alimentos', 'civil', 'energia', 'mecânica', 'produção',
'química']
removendo último elemento no conjunto : ['alimentos', 'civil', 'energia', 'mecânica', 'produção']
removendo os 3 primeiros elementos do conjunto : ['mecânica', 'produção']
matriz com números negativos : [[2, -7, 2], [8, 5, -4]]
matriz com números negativos convertidos para positivos : [[2, 7, 2], [8, 5, 4]]
```

Inicialmente é atribuída à variável cursos_engenharia_faen uma lista com 3 elementos:

- ['alimentos', 'energia', 'produção']

Como a lista é um conjunto ordenado, você poderá referenciar os índices dos elementos da lista para realizar operações. Vamos ilustrar: (a) inserção de elementos na lista, a partir de um dado índice; (b) cópia de uma sublista baseada na definição de um intervalo de índices; (c) a inserção de um elemento no final da lista (apendar um elemento); e (d) a remoção de sublistas definidas por um único elemento (único índice) ou por vários elementos em sequência (intervalo de índices).

Após imprimir a lista inicial, a função insert é utilizada para incluir o elemento 'civil' na lista inicial, na posição 1, deslocando o elemento que estava na posição 1 para a próxima posição (2) e os elementos seguintes idem. A seguir, é inserido o elemento 'mecânica' na posição 3 da lista, e impressa a lista resultante, que agora já conta com cinco elementos.

A sublista gerada a partir do intervalo [2:4] da lista é impressa. Lembre-se que o último elemento é exclusivo, portanto, serão impressos os elementos indexados pelos índices 2 e 3 da lista.

A função append é utilizada para apendar um elemento, ou seja, inserir-lo como último elemento da lista, com a impressão da lista resultante.

A função del é utilizada para remover o último elemento da lista, indexado por -1 e, posteriormente, uma sublista indexada pelo intervalo [0:3], com a impressão das listas resultantes em ambos os casos.

Uma matriz pode ser representada como uma lista de listas. Foi utilizado um exemplo de utilização de matrizes, a conversão dos valores negativos de uma matriz para valores positivos, com a função converter_valores_negativos_matrizes. Para evitar a alteração dos valores da matriz original é criada um nova matriz (matriz_convertida) e montada um nova linha, a partir de cada linha da matriz original, com os valores originais (se forem positivos ou zero) e com os valores convertidos (se forem negativos). Cada nova linha montada é apendada na nova matriz, inicializada como uma lista vazia.

A função converter_valores_negativos_matrizes ilustra como acessar as linhas da matriz (listas internas), e os valores de cada linha utilizando dois loops de iteração e a função enumerate, que retorna o índice e os elementos de um conjunto. A matriz é uma lista cujos elementos são as suas linhas, representadas como listas internas. Cada linha é uma lista cujos elementos são os seus valores, correspondentes aos valores da matriz. O loop externo itera em cada linha da matriz obtendo : índice_linha e linha. O loop interno itera em cada coluna da linha obtendo : índice_coluna e valor. É possível utilizar os dois índices para indexar qualquer valor da matriz, ou seja, o valor da matriz, com índice de linha i e índice de coluna j é representado como: matriz[i][j].

Nesta seção foram ilustradas operações com uma lista de strings, mais você pode utilizar qualquer tipo como elemento de uma lista, inclusive uma outra lista. Indo mais além, você pode definir uma lista com elementos de vários tipos, como por exemplo:

- ['Dourados', 'UFGD', 2021, True, ['azul', 'branco']].

3.3 - Dicionário (dict) : conjunto de elementos não ordenado acessado a partir de chaves

O dicionário é composto de pares chave-valor. A chave é utilizada para inserção e posterior recuperação, alteração ou remoção do valor associado a ela. Chaves podem ser dos seguintes tipos: str, int, float, bool. Os valores podem ser de qualquer tipo: str, int, float, bool, list (lista), dict (dicionário), e outros tipos que serão ilustrados no restante deste tutorial.

No módulo estruturas_dados__comandos_iteração defina as seguintes funções:

```
def imprimir_lista_valores_indexados_por_mesma_chave(dicionário, chave_interna):
    valores = ''
    for chave in dicionário: valores +=  ' - ' + str(dicionário[chave][chave_interna])
    print("disciplinas indexadas pela chave '%s' :\n%s" % (chave_interna, valores))

def imprimir_chaves_dicionário(nome, dicionário):
    chaves = ''
    for chave in dicionário: chaves +=  ' - ' + str(chave)
    print("chaves do dicionário '%s' :\n%s" % (nome, chaves))
```

Para ilustrar o uso de dicionários, na função ilustrar__estruturas_dados__comandos_iteração comente os trechos que ilustram as subseções 3.1 e 3.2 e inclua o trecho que ilustra a seção 3.3.

```
def ilustrar__estruturas_dados__comandos_iteração():
    print('\n3.3 - manipular dicionários')
    país = {}
    país['nome'] = 'Brasil'
    país['continente'] = 'América do Sul'
    país['regiões'] = ['sul', 'sudeste', 'centro oeste', 'nordeste', 'norte']
    país['Índice de Desenvolvimento Humano'] = 0,765
    país['Índice de Percepção da Corrupção'] = 38
    país['regime democrático'] = True
    print('dicionário com vários tipos de elementos :\n  %s' % (país))
    chave = 'nome'
    print("indexando valor do dicionário '%s' pela chave '%s' : %s" % ('país', chave, país[chave]))
    chave = 'regiões'
    print("indexando valor do dicionário '%s' pela chave '%s' : %s" % ('país', chave, país[chave]))
    imprimir_chaves_dicionário('país', país)
    print("lista de chaves do dicionário '%s' :\n %s" % ('país', list(país)))
    disciplina = {'título':'Vibrações Mecânicas', 'categoria':'Mecânica Aplicada',
       'carga horária':72, 'modalidade':'prática'}
    print("dicionário '%s' com os pares chave-valor iniciais : \n  %s" % ('disciplina', disciplina))
    disciplina['modalidade'] = 'teórica'
    del disciplina['carga horária']
    print("dicionário '%s' após alteração da modalidade e remoção da carga horária :\n  %s"
       % ('disciplina', disciplina))
    disciplinas_engenharia_mecânica = {}
    disciplinas_engenharia_mecânica['Vibrações Mecânicas']\
        = {'título':'Vibrações Mecânicas', 'categoria':'Mecânica Aplicada', 'carga horária':72,
           'modalidade':'teórica'}
    disciplinas_engenharia_mecânica['Ensaios Mecânicos de Materiais']\
        = {'título':'Ensaios Mecânicos de Materiais', 'categoria':'Tecnologia Mecânica',
           'carga horária':36, 'modalidade':'teórica_prática'}
    disciplinas_engenharia_mecânica['Mecânica dos Fluidos Experimental']\
        = {'título':'Mecânica dos Fluidos Experimental', 'categoria':'Fenômenos de Transporte',
           'carga horária':36, 'modalidade':'prática'}
    disciplinas_engenharia_mecânica['Sistemas Térmicos de Potência']\
        = {'título':'Sistemas Térmicos de Potência', 'categoria':'Termodinâmica Aplicada',
          'carga horária':72, 'modalidade':'teórica_prática'}
    disciplinas_engenharia_mecânica['Métodos Numéricos para Engenharia']\
        = {'título':'Métodos Numéricos para Engenharia', 'categoria':'Engenharia Geral',
           'carga horária':72, 'modalidade':'teórica_prática'}
    chave_externa = 'Mecânica dos Fluidos Experimental'
    print("dicionário '%s' indexado com a chave '%s' :\n %s" % ('disciplinas_engenharia_mecânica',
       chave_externa, disciplinas_engenharia_mecânica[chave_externa]))
    chave_interna = 'modalidade'
    print("dicionário '%s' indexado com chave externa '%s' e chave interna '%s' : %s"
       % ('disciplinas_engenharia_mecânica', chave_externa, chave_interna,
       disciplinas_engenharia_mecânica[chave_externa][chave_interna]))
    imprimir_lista_valores_indexados_por_mesma_chave(disciplinas_engenharia_mecânica, 'categoria')
```

A saída da execução do programa é:

```
3.3 - manipular dicionários
dicionário com vários tipos de elementos :
   {'nome': 'Brasil', 'continente': 'América do Sul',
   'regiões': ['sul', 'sudeste', 'centro oeste', 'nordeste', 'norte'],
   'Índice de Desenvolvimento Humano': (0, 765), 'Índice de Percepção da Corrupção': 38,
   'regime democrático': True}
indexando valor do dicionário 'país' pela chave 'nome' : Brasil
indexando valor do dicionário 'país' pela chave 'regiões' :
   ['sul', 'sudeste', 'centro oeste', 'nordeste', 'norte']
chaves do dicionário 'país' :
 - nome - continente - regiões - Índice de Desenvolvimento Humano - Índice de Percepção da Corrupção
 - regime democrático
lista de chaves do dicionário 'país' :
 ['nome', 'continente', 'regiões', 'Índice de Desenvolvimento Humano',
  'Índice de Percepção da Corrupção', 'regime democrático']
dicionário 'disciplina' com os pares chave-valor iniciais :
  {'título': 'Vibrações Mecânicas', 'categoria': 'Mecânica Aplicada', 'carga horária': 72,
   'modalidade': 'prática'}
dicionário 'disciplina' após alteração da modalidade e remoção da carga horária :
  {'título': 'Vibrações Mecânicas', 'categoria': 'Mecânica Aplicada', 'modalidade': 'teórica'}
dicionário 'disciplinas_engenharia_mecânica' indexado com a chave
  'Mecânica dos Fluidos Experimental' :
 {'título': 'Mecânica dos Fluidos Experimental', 'categoria': 'Fenômenos de Transporte',
  'carga horária': 36, 'modalidade': 'prática'}
dicionário 'disciplinas_engenharia_mecânica' indexado com chave externa
  'Mecânica dos Fluidos Experimental' e chave interna 'modalidade' : prática
disciplinas indexadas pela chave 'categoria' :
 - Mecânica Aplicada - Tecnologia Mecânica - Fenômenos de Transporte - Termodinâmica Aplicada
 - Engenharia Geral
```

Esse trecho de código ilustra várias situações:

- dicionário país sendo acrescido de pares chave-valor com valores de vários tipos
- dicionário país sendo indexado pela chave 'nome'
- idem para a chave 'região'
- iteração para imprimir chaves do dicionário país
  - função imprimir_chaves_dicionário itera nas chaves do dicionário
    - **for** chave **in** dicionário:
- impressão de lista de chaves do dicionário país
- impressão do dicionário disciplina
- impressão do dicionário disciplina após
  - após alteração do valor da chave 'modalidade'
  - e remoção do par chave-valor indexado pela chave 'carga horária'
- dicionário disciplinas_engenharia_mecânica sendo acrescido de pares chave-valor com:
  - chave : título de uma disciplina
  - valor : dicionário representando pares chave-valor de uma disciplina
- dicionário disciplinas_engenharia_mecânica indexado pela chave 'Mecânica dos Fluidos Experimental'
- disciplina interna ao dicionário disciplinas_engenharia_mecânica
  - indexado pelas chave externa 'Mecânica dos Fluidos Experimental'
  - e pela chave interna 'modalidade'
- disciplinas internas ao dicionário disciplinas_engenharia_mecânica indexadas pela chave 'categoria'
  - função imprimir_lista_valores_indexados_por_mesma_chave
    - itera nas chaves do dicionário para montar string com os valores nos dicionários internos indexados pela chave passada como parâmetro

3.4 - Tupla (tuple) : conjunto imutável e ordenado de elementos

Tupla pode ser acessada através de índices e aceita elementos de qualquer tipo. Seus elementos não podem ser alterados individualmente, nem inseridos ou removidos.

No módulo estruturas_dados__comandos_iteração defina as seguinte função:

```
def criar__lista_coordenadas__geradas__por__função(função, horizontal_inicial,
       horizontal_final_exclusiva, passo):
    coordenadas_percurso = []
    for coordenada_horizontal in range(horizontal_inicial, horizontal_final_exclusiva, passo):
        coordenada_vertical = função(coordenada_horizontal)
        coordenadas_percurso.append((coordenada_horizontal, coordenada_vertical))
    return coordenadas_percurso
```

Para ilustrar o uso de tuplas, na função ilustrar__estruturas_dados__comandos_iteração comente os trechos que ilustram as subseções 3.1, 3.2 e 3.3, e inclua o trecho que ilustra a seção 3.4.

```
def ilustrar__estruturas_dados__comandos_iteração():
    print('\n3.4 - manipular tuplas')
    áreas_engenharia_mecânica = ('Mecânica Aplicada', 'Tecnologia Mecânica',
       'Fenômenos de Transporte', 'Termodinâmica Aplicada', 'Engenharia Geral')
    print('tupla : ' + str(áreas_engenharia_mecânica))
    reta = lambda x: 5*x + 2
    print('coordenadas da reta : %s'
       % (criar__lista_coordenadas__geradas__por__função(reta, -5, 6, 2)))
    parábola = lambda x: x ** 2
    print('coordenadas da parábola : %s'
       % (criar__lista_coordenadas__geradas__por__função(parábola, -5, 6, 2)))
```

A saída da execução do programa é:

```
3.4 - manipular tuplas
tupla : ('Mecânica Aplicada', 'Tecnologia Mecânica', 'Fenômenos de Transporte', 'Termodinâmica
Aplicada', 'Engenharia Geral')
coordenadas da reta : [(-5, -23), (-3, -13), (-1, -3), (1, 7), (3, 17), (5, 27)]
coordenadas da parábola : [(-5, 25), (-3, 9), (-1, 1), (1, 1), (3, 9), (5, 25)]
```

Inicialmente é criada e impressa a tupla áreas_engenharia_mecânica, com strings correspondentes às áreas da Engenharia Mecânica.

Vamos aproveitar para aprender dois conceitos muito úteis: (a) definição de função anônima; e (b) passagem de função como parâmetro.

A palavra reservada lambda é utilizada para criar uma função anônima (sem nome), como por exemplo lambda x: 5*x + 2, que é equivalente a f(x) = x * 5 + 2. A seguir a função anônima é atribuída à variável reta. Neste caso, a variável reta passa a ser uma função com parâmetro x e corpo 5*x + 2. Posteriormente, no código acima, é defina a função anônima lambda x: x ** 2, equivalente a f(x) = x**2, que é atribuída à variável parábola. Os caracteres x**n indicam: x elevado a n; ou seja, x**2 é x elevado ao quadrado.

A função criar__lista_coordenadas__geradas__por__função recebe uma função como argumento do seu primeiro parâmetro função. Seus outros parâmetros são: horizontal_inicial, horizontal_final_exclusiva e passo. Para iterar nas coordenadas horizontais é utilizado o comando de iteração:

- **for** coordenada_horizontal **in** range(horizontal_inicial, horizontal_final_exclusiva, passo):

A função range é utilizada para gerar um intervalo de valores, partindo de um valor inicial, incrementando esse valor com um passo (valor do incremento), e parando antes que o valor final seja atingido (exclusivo) ou ultrapassado.

A passagem de uma função como parâmetro, é um mecanismo muito poderoso, pois permite que uma única função, neste caso a função criar__lista_coordenadas__geradas__por__função, utilize uma função distinta para realizar seu processamento, cada vez que é chamada com uma outra função como parâmetro.

3.5 - Set : conjunto não ordenado sem elementos duplicadas

Set não aceita elementos de mesmo valor duplicados, e seus elementos não pode ser acessados através de índices.

Para ilustrar o uso de sets (conjuntos não ordenados sem elementos duplicados), na função ilustrar__estruturas_dados__comandos_iteração comente os trechos que ilustram as subseções 3.1, 3.2, 3.3 e 3.4, e inclua o trecho que ilustra a seção 3.5.

```
def ilustrar__estruturas_dados__comandos_iteração():
    print('\n3.5 - manipular conjuntos (sets)')
    faculdades1 = {'FACET', 'FAEN', 'FCS'}
    faculdades2 = {'FACET', 'FAEN', 'FADIR'}
    print('união de %s com %s : %s' %(faculdades1, faculdades2, faculdades1.union(faculdades2)))
    print('intersecção de %s com %s : %s' % (faculdades1, faculdades2,
        faculdades1.intersection(faculdades2)))
```

A saída da execução do programa é:

```
3.5 - manipular conjuntos (sets)
união de {'FAEN', 'FACET', 'FCS'} com {'FAEN', 'FADIR', 'FACET'} : {'FADIR', 'FACET', 'FAEN', 'FCS'}
intersecção de {'FAEN', 'FACET', 'FCS'} com {'FAEN', 'FADIR', 'FACET'} : {'FAEN', 'FACET'}
```

Nesta ilustração são definidos dois conjuntos (sets), ambos com três faculdades da UFGD. A função union é utilizada para gerar um conjunto a partir da união dos conjuntos iniciais. Note que as faculdades FAEN e FACET, que fazem parte dos dois conjuntos iniciais não são íncluídas duas vezes no conjunto gerado, porque esse tipo de conjunto não aceita elementos duplicados.

A função intersection é utilizada para gerar um conjunto a partir da interseção dos conjuntos iniciais. Note que somente as faculdades que fazem parte dos dois conjuntos iniciais, FAEN e FACET, são incluídas no conjunto gerado.

4 - Leitura e Escrita em Arquivos

Nesta seção, vamos ilustrar a leitura e a escrita de dados em dois formatos de arquivos muito utilizados: CSV e JSON.

Para ilustrar essa seção crie o módulo leitura_escrita_arquivos. Para armazenar os arquivos que serão gerados, crie o diretório dados no projeto, no mesmo nível hierárquico do diretório src.

4.1 - Salvar/recuperar matriz (lista de listas) em/de arquivo CSV

O formato CSV (Comma Separated Values), Valores Separados por Vírgulas, é muito utilizado para representar dados de planilhas ou de base de dados.

No módulo leitura_escrita_arquivos defina as seguintes funções :

```
def carregar_arquivo_csv(nome_arquivo):
    arquivo = open('dados/'+ nome_arquivo + '.csv', 'r')
    matriz_str = arquivo.read().strip('\n')
    arquivo.close()
    matriz = matriz_str.split('\n')
    for índice_linha, linha_str in enumerate(matriz):
        linha = list(linha_str.split(', '))
        matriz[índice_linha] = linha
        for índice_coluna, valor in enumerate(linha):
            valor = valor.strip()
            linha[índice_coluna] = valor
    return matriz

def salvar_arquivo_csv(nome_arquivo, matriz):
    arquivo = open('dados/' + nome_arquivo + '.csv', 'w')
    for linha in matriz:
        valores_str = ', '.join(map(str, linha))
        arquivo.write(f"{valores_str}\n")
    arquivo.close()
```

No módulo main, defina a função ilustrar_leitura_escrita_arquivos com o trecho de código de ilustração da subseção 4.1.

```
def ilustrar_leitura_escrita_arquivos():
    print('\n4.1 - salvar/recuperar matriz (lista de listas) em/de arquivo CSV')
    disciplinas_matriz1 = [
        ['título', 'categoria', 'carga horária', 'modalidade'],
        ['Vibrações Mecânicas', 'Mecânica Aplicada', 72, 'teórica'],
        ['Ensaios Mecânicos de Materiais', 'Tecnologia Mecânica', 36, 'teórica_prática'],
        ['Mecânica dos Fluidos Experimental', 'Fenômenos de Transporte', 36, 'prática'],
        ['Sistemas Térmicos de Potência', 'Termodinâmica Aplicada', 72, 'teórica_prática'],
        ['Métodos Numéricos para Engenharia', 'Engenharia Geral', 72, 'teórica_prática']
    ]
    salvar_arquivo_csv('disciplinas1', disciplinas_matriz1)
    disciplinas_matriz2 = carregar_arquivo_csv('disciplinas1')
    for disciplina in disciplinas_matriz2: print(disciplina)
```

No corpo principal do programa, comente as chamadas das funções que ilustram as seções 2 e 3, e acrescente a chamada da função que ilustra a seção 4.

```
if __name__ == '__main__':
    # ilustrar_variáveis_tipos_funções_comandos_condicionais()
    # ilustrar_estruturas_dados_comandos_iteração()
    ilustrar_leitura_escrita_arquivos()
```

A saída da execução do programa é:

```
4.1 - salvar/recuperar matriz (lista de listas) em/de arquivo CSV
['título', 'categoria', 'carga horária', 'modalidade']
['Vibrações Mecânicas', 'Mecânica Aplicada', '72', 'teórica']
['Ensaios Mecânicos de Materiais', 'Tecnologia Mecânica', '36', 'teórica_prática']
['Mecânica dos Fluidos Experimental', 'Fenômenos de Transporte', '36', 'prática']
['Sistemas Térmicos de Potência', 'Termodinâmica Aplicada', '72', 'teórica_prática']
['Métodos Numéricos para Engenharia', 'Engenharia Geral', '72', 'teórica_prática']
```

Cria uma matriz (lista de listas) de disciplinas. A função salvar_arquivo_csv é chamada para salvar a matriz de disciplinas em um arquivo com extensão csv. Utiliza a função carregar_arquivo_csv para carregar o conteúdo do arquivo gerado previamente e itera na linhas do arquivo, obtendo e imprimindo cada linha (lista com o valores de uma disciplina).

A função salvar_arquivo_csv :

- utiliza a função open para criar a variável arquivo para escrita de um arquivo no sub diretório dados do projeto
- itera na linhas da matriz
  - utiliza a função map para aplicar a função str a todos os valores da linha da matriz
  - utiliza a função join, aplicável a um string, para gerar um string agrupando os valores separados por ', '
  - utiliza a função write para escrever o string de cada linha no arquivo finalizando com o caracter '\n'
- utiliza a função close para fechar o arquivo

A função carregar_arquivo_csv :

- utiliza a função open para criar a variável arquivo para leitura de arquivo no sub diretório dados do projeto
- utiliza a função read para ler o conteúdo do arquivo
- utiliza a função strip para remover o caracter \n (pula linha) do final do arquivo;
- itera no string da matriz gerada obtendo o índice e string de cada linha da matriz
  - utiliza a função split para remover os caracteres ', ' entre subtrings de cada linha
  - cria uma lista com os strings gerados a partir da linha;
  - substitui o string da linha na matriz, pela lista de strings gerados a partir da linha;
  - itera na lista de strings da linha obtendo o índice do string e o string
    - utiliza a função strip para remover caracteres em branco em torno do string
    - substitui o string na matriz pelo string sem caracteres em branco ao redor
- retorna a matriz lida e processada

4.2 - Salvar/recuperar dicionário de dicionários em/de arquivo JSON

O formato JSON (JavaScript Objetcs Notation), Notação de Objetos Java Script, é muito utilizado para representar objetos (ver seção 5), que são suportados pelas linguagens de programação mais utilizadas atualmente. Esse formato é equivalente ao formato de um dicionário.

No módulo leitura_escrita_arquivos, importe o módulo json e defina as seguintes funções:

```
import json

def carregar_arquivo_json(nome_arquivo):
    arquivo = open('dados/' + nome_arquivo + '.json', 'r')
    dicionário = json.load(arquivo)
    return dicionário

def salvar_arquivo_json(nome_arquivo, dicionário):
    arquivo = open('dados/' + nome_arquivo + '.json', 'w')
    json.dump(dicionário, arquivo)
    arquivo.close()
```

Na função ilustrar_leitura_escrita_arquivos comente o trecho de código de ilustração da subseção 4.1, e acrescente o trecho de ilustração da subseção 4.2.

```
def ilustrar_leitura_escrita_arquivos():
    print('\n4.2 - salvar/recuperar dicionário de dicionários em/de arquivo JSON')
    disciplinas_dicionários1 = {}
    disciplinas_dicionários1['Vibrações Mecânicas']\
       = {'título':'Vibrações Mecânicas',   'categoria':'Mecânica Aplicada', 'carga_horária':72,
       'modalidade':'teórica'}
    disciplinas_dicionários1['Ensaios Mecânicos de Materiais']\
       = {'título':'Ensaios Mecânicos de Materiais', 'categoria':'Tecnologia Mecânica',
       'carga_horária':36, 'modalidade':'teórica_prática'}
    disciplinas_dicionários1['Mecânica dos Fluidos Experimental']\
       = {'título':'Mecânica dos Fluidos Experimental', 'categoria':'Fenômenos de Transporte',
       'carga_horária':36, 'modalidade':'prática'}
    disciplinas_dicionários1['Sistemas Térmicos de Potência']\
       = {'título':'Sistemas Térmicos de Potência', 'categoria':'Termodinâmica Aplicada',
       'carga_horária':72, 'modalidade':'teórica_prática'}
    disciplinas_dicionários1['Métodos Numéricos para Engenharia']\
       = {'título':'Métodos Numéricos para Engenharia', 'categoria':'Engenharia Geral',
       'carga_horária':72, 'modalidade':'teórica_prática'}
    salvar_arquivo_json('disciplinas2', disciplinas_dicionários1)
    disciplinas_dicionários2 = carregar_arquivo_json('disciplinas')
    for chave in disciplinas_dicionários2: print(disciplinas_dicionários2[chave])
```

A saída da execução do programa é:

```
4.2 - salvar/recuperar dicionário de dicionários em/de arquivo JSON
{'título': 'Vibrações Mecânicas', 'categoria': 'Mecânica Aplicada', 'carga_horária': 72,
'modalidade': 'teórica'}
{'título': 'Ensaios Mecânicos de Materiais', 'categoria': 'Tecnologia Mecânica', 'carga_horária':
36, 'modalidade': 'teórica_prática'}
{'título': 'Mecânica dos Fluidos Experimental', 'categoria': 'Fenômenos de Transporte',
'carga_horária': 36, 'modalidade': 'prática'}
{'título': 'Sistemas Térmicos de Potência', 'categoria': 'Termodinâmica Aplicada', 'carga_horária':
72, 'modalidade': 'teórica_prática'}
{'título': 'Métodos Numéricos para Engenharia', 'categoria': 'Engenharia Geral', 'carga_horária':
72, 'modalidade': 'teórica_prática'}
```

Cria um dicionário de dicionários, sendo que cada dicionário interno representa uma disciplina. A função salvar_arquivo_json é chamada para salvar o dicionário de disciplinas em um arquivo com extensão json. Utiliza a função carregar_arquivo_json para carregar o conteúdo do arquivo gerado previamente e itera na linhas do arquivo, obtendo e imprimindo cada linha (dicionário representando uma disciplina).

A função salvar_arquivo_ json :

- utiliza a função open para criar a variável arquivo para escrita de um arquivo no sub diretório dados do projeto
- utiliza a função json.dump para salvar um dicionário em um arquivo com extensão json
- utiliza a função close para fechar o arquivo

A função carregar_arquivo_json :

- utiliza a função open para criar a variável arquivo para leitura de arquivo no sub diretório dados do projeto
- utiliza a função json.load para carregar um dicionário a partir de um arquivo com extensão json
- retorna o dicionário lido

4.3 - Converter dicionário para objeto

No módulo leitura_escrita_arquivos defina a classe de objetos Disciplina.

```
class Disciplina():

     def __init__(self, título, categoria, carga_horária, modalidade):
         self.título = título
         self.categoria = categoria
         self.carga_horária = carga_horária
         self.modalidade = modalidade

     def __str__(self):
         return self.título + ' - ' + self.categoria + ' - ' + str(self.carga_horária) + ' - '
               + self.modalidade
```

Na função ilustrar_leitura_escrita_arquivos comente o trecho de código de ilustração da subseção 4.1, mantenha o trecho da subseção 4.2 sem comentar (cria uma variável que será utilizada no trecho da subseção 4.3), e acrescente o trecho de ilustração da subseção 4.3.

```
def ilustrar_leitura_escrita_arquivos():
    print('\n4.3 - converter dicionário para objeto')
    disciplinas_objetos = {}
    for chave in disciplinas_dicionários2: # iterando nas chaves dos dicionários internos
        disciplina_objeto = Disciplina(**disciplinas_dicionários2[chave])
        disciplinas_objetos[chave] = disciplina_objeto
    for chave in disciplinas_objetos:
        print('%s - %s' % (disciplinas_objetos[chave], type(disciplinas_objetos[chave])))
```

A saída da execução do programa (a saída do trecho da subseção 2 foi omitida) é:

```
4.3 - converter dicionário para objeto
Vibrações Mecânicas - Mecânica Aplicada - 72 - teórica - <class
'leitura_escrita_arquivos.Disciplina'>
Ensaios Mecânicos de Materiais - Tecnologia Mecânica - 36 - teórica_prática - <class
'leitura_escrita_arquivos.Disciplina'>
Mecânica dos Fluidos Experimental - Fenômenos de Transporte - 36 - prática - <class
'leitura_escrita_arquivos.Disciplina'>
Sistemas Térmicos de Potência - Termodinâmica Aplicada - 72 - teórica_prática - <class
'leitura_escrita_arquivos.Disciplina'>
Métodos Numéricos para Engenharia - Engenharia Geral - 72 - teórica_prática - <class
'leitura_escrita_arquivos.Disciplina'>
```

Embora classes de objetos, seja o assunto da seção 5, vamos antecipar a ilustração da classe Disciplina, para que você possar aprender como converter um dicionário para objeto e vice-versa.

O paradigma orientado a objeto tem sido largamente utilizado e suportado pelas linguagens modernas. Esse paradigma se apoia nos seguintes conceitos:

- você pode representar uma entidade do mundo real em um novo tipo para sua linguagem, descrito com uma classe de objetos (em geral, denominada simplesmente de classe);
  - ex: a entidade disciplina do mundo real é representada pela classe Disciplina
- a classe é composta de dados de vários tipos e de métodos (funções aplicáveis a esses dados);
- a partir de novo tipo (classe) você pode criar variáveis denominadas objetos, inicializando os dados da classe com diferentes valores; neste caso, você pode criar várias disciplinas com diferentes valores para os dados;
  - atributos da entidade no mundo real são representados como dados da classe Disciplina
    - título, categoria, carga_horária, modalidade (veja os dados dos objetos na saída).

A orientação a objetos também utiliza o conceito de herança, que veremos na seção 5.

Neste exemplo, a classe disciplina é composta de dois métodos:

- a função com nome padronizado __init__
  - utilizada para definir e inicializar os dados da classe;
- e a função com nome padronizado __str__
  - utilizada para gerar um string a partir da concatenação dos respectivos dados da classe.

Todos os métodos tem self como primeiro argumento, o que não ocorre com as funções externas a uma classe. Os dados e métodos de uma classe são referenciados no código da classe com o prefixo self para indicar que pertencem à classe.

Iterando nas chaves do dicionário disciplinas_dicionários2, carregado de um arquivo json no trecho de ilustração da subseção 4.3, cada dicionário (disciplina), indexado pela chave obtida na iteração, é convertido para o um objeto da classe Disciplina e inserido no dicionário disciplinas_objetos, associado a mesma chave (título da disciplina). Iterando nas chaves do dicionário disciplinas_objetos, imprime cada objeto e o seu tipo (classe Disciplina do módulo leitura_escrita_arquivos).

Esse exemplo utiliza implicitamente os dois métodos definidos na classe Disciplina: __init__ e __str__.

A conversão de um dicionário para um objeto é realizada chamando implicitamente o método __init__ a partir do nome da classe (neste caso: Disciplina). Esse método recebe como parâmetro o dicionário que representa uma disciplina. A notação ** que antecede o dicionário é utilizada para passar uma lista de argumentos de tamanho variável, o que é muito útil, pois cada classe pode ter uma quantidade distinta de dados. Neste exemplo, os argumentos são os pares chave-valor do dicionário que representa um disciplina, que são utilizados para mapear nos dados definidos do método __init__ da classe Disciplina.

Quando o objeto do dicionário disciplinas_objetos é utilizado como argumento (disciplinas_objetos[chave]) da função print, o método __str__ da classe Disciplina é chamado implicitamente para convertar o objeto no string que o representa.

4.4 - Converter objeto para dicionário

Na função ilustrar_leitura_escrita_arquivos comente o trecho de código de ilustração da subseção 4.1, mantenha o trecho das subseções 4.2 e 4.3 sem comentar, e acrescente o trecho de ilustração da subseção 4.4.

```
def ilustrar_leitura_escrita_arquivos():
    print('\n4.4 - converter objeto para dicionário')
    disciplinas_dicionários3 = {}
    for chave in disciplinas_objetos:
        disciplina_objeto = disciplinas_objetos[chave]
        disciplinas_dicionários3[chave] = disciplina_objeto.__dict__
    for chave in disciplinas_dicionários3:
        print('%s - %s' % (disciplinas_dicionários3[chave], type(disciplinas_dicionários3[chave])))
```

A saída da execução do programa é:

```
4.4 - converter objeto para dicionário
{'título': 'Vibrações Mecânicas', 'categoria': 'Mecânica Aplicada', 'carga_horária': 72,
'modalidade': 'teórica'} - <class 'dict'>
{'título': 'Ensaios Mecânicos de Materiais', 'categoria': 'Tecnologia Mecânica', 'carga_horária':
36, 'modalidade': 'teórica_prática'} - <class 'dict'>
{'título': 'Mecânica dos Fluidos Experimental', 'categoria': 'Fenômenos de Transporte',
'carga_horária': 36, 'modalidade': 'prática'} - <class 'dict'>
{'título': 'Sistemas Térmicos de Potência', 'categoria': 'Termodinâmica Aplicada', 'carga_horária':
72, 'modalidade': 'teórica_prática'} - <class 'dict'>
{'título': 'Métodos Numéricos para Engenharia', 'categoria': 'Engenharia Geral', 'carga_horária':
72, 'modalidade': 'teórica_prática'} - <class 'dict'>
```

Itera nas chaves do dicionário disciplinas_objetos, gerado no trecho de ilustração da subseção 4.3. O dicionário que representa cada objeto da classe Disciplina, indexado pela chave obtida na iteração, é obtido através do dado pré-definido __dict__. Esse dicionário é inserido no dicionário disciplinas_dicionários3, associado à mesma chave (título da disciplina). Itera nas chaves do dicionário disciplinas_dicionários3, imprime cada dicionário e o seu tipo (classe dict).

5 - Classes e Objetos

Com vimos antecipadamente na seção 4.3, linguagens orientadas a objetos suportam a representação de entidades do mundo real em novos tipos da linguagem, facilitando muito o desenvolvimento de aplicações que processem essas entidades. Esses novos tipos são classes de objetos, denominadas simplemente de classes. Esse conceito é muito poderoso, dado que em uma linguagem orientada a objetos novos tipos podem ser criados para representar entidades de interesse para uma dada aplicação, tornando o código muito mais legível e mais fácil de ser mantido (corrigido e ampliado, quando necessário).

Na seção 5.1, serão definidas várias classes (novos tipos) e criados objetos, que são variáveis geradas a partir de atribuição de valores aos dados de uma classe. Além de representar atributos mapeados a partir de entidades do mundo real, uma classe pode ter dados que representam referências a objetos de outras classes.

Na seção 5.2, veremos o conceito de herança, que permite definir uma classe mais específica (subclasse) que herda dados e métodos comuns de uma classe mais genérica (superclasse), e pode acrescentar dados ou métodos específicos, aos dados e métodos herdados. Métodos são funções definidas em uma classe, que tem acesso a todos os dados da classe (definidos e inicializados no método __init__).

Para ilustrar essa seção crie o módulo classes_objetos.

5.1 - Dicionários, impressão e filtragem de objetos

No módulo classes_objetos defina as seguintes funções e classes :

```
def selecionar_propostas_projetos(propostas_projetos, tipo_projeto, titulação_orientador,
       ano_mínimo_ingresso_aluno):
    filtros = 'Filtros:: '
    if tipo_projeto != None: filtros += 'tipo do projeto: ' + tipo_projeto
    if titulação_orientador != None:
        filtros += ' - titulação do orientador: ' + titulação_orientador
    if ano_mínimo_ingresso_aluno != -1:
        filtros += ' - ano mínimo de ingresso do aluno: ' + str(ano_mínimo_ingresso_aluno)
    propostas_selecionadas_projetos = {}
    for título_projeto in propostas_projetos:
        proposta_projeto = propostas_projetos[título_projeto]
        orientador = proposta_projeto.orientador
        aluno = proposta_projeto.aluno
        if tipo_projeto != None and proposta_projeto.tipo != tipo_projeto: continue
        if titulação_orientador != None and orientador.titulação != titulação_orientador: continue
        if ano_mínimo_ingresso_aluno != -1 and aluno.ano_ingresso < ano_mínimo_ingresso_aluno:
            continue
        propostas_selecionadas_projetos[título_projeto] = proposta_projeto
    return filtros, propostas_selecionadas_projetos


def imprimir_dicionário_objetos(cabeçalho, dicionário):
    print('\n' + cabeçalho)
    n = 1
    for chave in dicionário:
        print(str(n) + ' - ' + str(dicionário[chave]))
        n += 1

class Aluno:

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

    def __str__(self):
        return self.nome + ' - email: ' + self.email + ' - ano de ingresso: '\
            + str(self.ano_ingresso)

class Professor:

    def __init__(self, nome, titulação, email):
        self.nome = nome
        self.titulação = titulação if titulação in ('mestrado', 'doutorado') else 'inválida'
        self.email = email

    def __str__(self):
        return self.nome + ' - email: ' + self.email + ' - titulação: ' + self.titulação

class PropostaProjeto:
    def __init__(self, título, orientador, aluno, tipo):
        self.título = título
        self.orientador = orientador
        self.aluno = aluno
        self.tipo = tipo if tipo in ('Trabalho de Conclusão de Curso', 'Iniciação Científica')
            else 'inválido'

def __str__(self):
    return 'Proposta de ' + self.tipo + ': ' + self.título\
       + '\n    - orientador: ' + str(self.orientador) + '\n    - aluno: ' + str(self.aluno)
```

No módulo main, defina a função ilustrar_classes_objetos, com o trecho de código que ilustra a subseção 5.1.

```
def ilustrar_classes_objetos():
    print('\n5.1 - criar dicionários de objetos - imprimir objetos - filtrar hierarquia de objetos')
    professores = {}
    professores['Marina Andrade'] = Professor('Marina Andrade', 'doutorado',
        'marina.andrade@ufgd.edu.br')
    professores['Diogo Tavares'] = Professor('Diogo Tavares', 'mestrado',
        'diogo.tavares@ufgd.edu.br')
    professores['Talia Menucci'] = Professor('Talia Menucci', 'doutorado',
        'talia.menucci@ufgd.edu.br')
    alunos = {}
    alunos['Ariane Villas'] = Aluno('Ariane Villas', 'ariane.villas@gmail.com', 2017)
    alunos['Angelo Silveira'] = Aluno('Angelo Silveira', 'angelo.silveira@gmail.com', 2016)
    alunos['Fabia Baroni'] = Aluno('Fabia Baroni', 'fabia.baroni@gmail.com', 2018)
    alunos['Alessandro Sales'] = Aluno('Alessandro Sales', 'alessandro.sales@gmail.com', 2017)
    alunos['Alexia Fantoni'] = Aluno('Alexia Fantoni', 'alexia_fantoni@gmail.com', 2018)
    alunos['Tales Oliveira'] = Aluno('Tales Oliveira', 'tales_oliveira@gmail.com', 2018)
    propostas_projetos = {}
    propostas_projetos['Produção de Baterias com Grafeno']\
        = PropostaProjeto('Produção de Baterias com Grafeno', professores['Marina Andrade'],
        alunos['Ariane Villas'], 'Iniciação Científica')
    propostas_projetos['Propriedades Físicas do Grafeno']\
        = PropostaProjeto('Propriedades Físicas do Grafeno', professores['Marina Andrade'],
        alunos['Angelo Silveira'], 'Trabalho de Conclusão de Curso')
    propostas_projetos['Avalições de Tensões em Engates Ferroviários']\
        = PropostaProjeto('Avalições de Tensões em Engates Ferroviários',
        professores['Diogo Tavares'], alunos['Fabia Baroni'], 'Iniciação Científica')
    propostas_projetos['Análise Numérica de Transferência de Calor em Motores']\
        = PropostaProjeto('Análise Numérica de Transferência de Calor em Motores',
        professores['Diogo Tavares'], alunos['Alessandro Sales'], 'Trabalho de Conclusão de Curso')
    propostas_projetos['Vibração em Hélices de Energia Eólica']\
        = PropostaProjeto('Vibração em Hélices de Energia Eólica', professores['Talia Menucci'],
        alunos['Alexia Fantoni'], 'Iniciação Científica')
    propostas_projetos['Vibração em Hélices de Drones']\
        = PropostaProjeto('Vibração em Hélices de Energia Eólica', professores['Talia Menucci'],
        alunos['Tales Oliveira'], 'Trabalho de Conclusão de Curso')
    imprimir_dicionário_objetos('Propostas de Projetos:', propostas_projetos)
    filtros, propostas_selecionadas_projetos = selecionar_propostas_projetos(
        propostas_projetos, None, None, -1)
    imprimir_dicionário_objetos(filtros, propostas_selecionadas_projetos)
    filtros, propostas_selecionadas_projetos = selecionar_propostas_projetos(
        propostas_projetos, 'Iniciação Científica', None, -1)
    imprimir_dicionário_objetos(filtros, propostas_selecionadas_projetos)
    filtros, propostas_selecionadas_projetos = selecionar_propostas_projetos(
        propostas_projetos, 'Iniciação Científica', 'doutorado', -1)
    imprimir_dicionário_objetos(filtros, propostas_selecionadas_projetos)
    filtros, propostas_selecionadas_projetos = selecionar_propostas_projetos(
        propostas_projetos, 'Iniciação Científica', 'doutorado', 2018)
    imprimir_dicionário_objetos(filtros, propostas_selecionadas_projetos)
```

No corpo principal do programa, comente as chamadas das funções que ilustram as seções 1, 2, 3 e 4, e acrescente a chamada da função que ilustra a seção 5.

```
if __name__ == '__main__':
    ilustrar_classes_objetos()
```

A saída da execução do programa é:

```
5.1 - criar dicionários de objetos - imprimir objetos - filtrar hierarquia de objetos

Propostas de Projetos:
1 - Proposta de Iniciação Científica: Produção de Baterias com Grafeno
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
2 - Proposta de Trabalho de Conclusão de Curso: Propriedades Físicas do Grafeno
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
3 - Proposta de Iniciação Científica: Avalições de Tensões em Engates Ferroviários
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
4 - Proposta de Trabalho de Conclusão de Curso: Análise Numérica de Transferência de
    Calor em Motores
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Alessandro Sales - email: alessandro.sales@gmail.com - ano de ingresso: 2017
5 - Proposta de Iniciação Científica: Vibração em Hélices de Energia Eólica
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
6 - Proposta de Trabalho de Conclusão de Curso: Vibração em Hélices de Drones
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Tales Oliveira - email: tales_oliveira@gmail.com - ano de ingresso: 2018

Filtros::
1 - Proposta de Iniciação Científica: Produção de Baterias com Grafeno
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
2 - Proposta de Trabalho de Conclusão de Curso: Propriedades Físicas do Grafeno
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
3 - Proposta de Iniciação Científica: Avalições de Tensões em Engates Ferroviários
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
4 - Proposta de Trabalho de Conclusão de Curso: Análise Numérica de Transferência de
    Calor em Motores
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Alessandro Sales - email: alessandro.sales@gmail.com - ano de ingresso: 2017
5 - Proposta de Iniciação Científica: Vibração em Hélices de Energia Eólica
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
6 - Proposta de Trabalho de Conclusão de Curso: Vibração em Hélices de Drones
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Tales Oliveira - email: tales_oliveira@gmail.com - ano de ingresso: 2018

Filtros:: tipo do projeto: Iniciação Científica
1 - Proposta de Iniciação Científica: Produção de Baterias com Grafeno
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
2 - Proposta de Iniciação Científica: Avalições de Tensões em Engates Ferroviários
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
3 - Proposta de Iniciação Científica: Vibração em Hélices de Energia Eólica
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018

Filtros:: tipo do projeto: Iniciação Científica - titulação do orientador: doutorado
1 - Proposta de Iniciação Científica: Produção de Baterias com Grafeno
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
2 - Proposta de Iniciação Científica: Vibração em Hélices de Energia Eólica
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018

Filtros:: tipo do projeto: Iniciação Científica - titulação do orientador: doutorado
          - ano mínimo de ingresso do aluno: 2018
1 - Proposta de Iniciação Científica: Vibração em Hélices de Energia Eólica
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
```

Inicialmente são definidos os dicionários professores, alunos e propostas_projetos. O dicionário professores é preenchido com três objetos, utilizando o nome do professor como chave e passando argumentos para os parâmetros definidos no método __init__ da classe Professor. Com vimos anteriormente (na seção 4.3, com a criação de objetos da classe Disciplina), o método __init__ é chamado implicitamente a partir da chamada do nome da classe (Disciplina) com os argumentos correspondentes aos parâmetros do método __init__.

Da mesma forma, o dicionário alunos é preenchido com seis objetos da classe Aluno, utilizando o nome do aluno como chave; e o dicionário propostas_projetos é preenchido com seis objetos da classe PropostasProjeto, utilizando o título da proposta de projeto como chave. Uma chave deve identificar um único objeto e, portanto, em geral nomes e títulos não são utilizados como chaves, porque podem existir mais de uma pessoa com o mesmo nome ou de coisas com o mesmo título. Em geral o CPF é utilizado como chave de pessoas e um identificador alfanumérico (composto de letras e números) é utilizado com chave de coisas em geral. Por simplicidade, e dado que nomes e títulos são mais amigáveis para identificar pessoas e coisas, estamos assumindo que num sistema de pequeno porte os nomes e títulos são distintos.

A seguir a função imprimir_dicionário_objetos é utilizada para imprimir os objetos cadastrados no dicionário propostas_projetos. Essa função imprime o string gerado a partir de um número sequencial para cada proposta de projeto, concatenado com o string gerado a partir de cada objeto da classe PropostaProjeto armazenado no dicionário propostas_projetos. Na conversão de um objeto para string, utilizando a função str, o método __str__ da classe do objeto é chamado implicitamente para concatenar os dados e referências do objeto e retornar o string que representa o objeto.

A seguir a função imprimir_dicionário_objetos é chamada quatro vezes para imprimir propostas de projeto selecionadas, passando como argumento a função selecionar_propostas_projetos, tendo com argumentos combinações de valores de três filtros.

A função selecionar_propostas_projetos retorna um string contendo somente os filtros cujos valores devem ser considerados na filtragem. Os valores None para filtros do tipo str ou objetos, -1 para filtros do tipo int e -1.0 para filtros do tipo float são utilizados para sinalizar que esses filtros são devem ser considerados na filtragem.

É comum utilizar filtros inteiros (int) ou ponto flutuantes (float) para indicar que o dado do objeto selecionado deve ser maior que o valor de um filtro definido com valor mínimo, ou menor que um filtro definido como valor máximo.

A função selecionar_propostas_projetos:

- monta um string com nomes e valores dos filtros válidos;
- cria variáveis para comparar com os dados para verificar se os dados de um dado objeto atendem os filtros de seleção;
  - observe que para obter o valor de um dado de um objeto é necessário referenciar o dado a partir do objeto:
    - ex: aluno = proposta_projeto.aluno
      - objeto : proposta_projeto
      - dado do objeto : aluno
- itera nas chaves do dicionário para verificar se cada objeto atende todos os filtros válidos
  - para cada filtro testa se o filtro é válido e se a condição testada falha
    - se o filtro não é válido
      - esse filtro não será considerado na filtragem e, portanto, não há necessidade de testar a condição
    - se o filtro é válido e a condição imposta por esse filtro não é atendida
      - utiliza o comando continue para interromper a execução dos comando do bloco interno à iteração e passar a testar o próximo objeto da iteração
    - se o filtro é válido e a condição imposta por esse filtro é atendida
      - continua no bloco de comandos interno da iteração para testar os demais filtros
  - se todos os filtros forem atendidos
    - insere o objeto no dicionário propostas_selecionadas_projetos
- retorna o string com os filtros válidos e o dicionário com as propostas de projetos selecionadas

As combinações de valores de filtros passadas com argumentos nas quatro chamadas da função selecionar_propostas_projetos obedecem aos seguintes critérios:

- inicialmente todos os filtros são passados com valores inválidos, de tal forma que os objetos retornados sejam os mesmos objetos cadastrados;
- a seguir, é atribuído um valor válido para o primeiro filtro, e esse valor é mantido nas próximas chamadas na função selecionar_propostas_projetos;
- e assim sucessivamente com cada um dos próximos filtros, de forma que a cada nova seleção o conjunto de objetos retornados potencialmente diminui em função da restrição imposta pelo próximo filtro com valor válido.

5.2 - Herança de dados e métodos

No módulo classes_objetos acrescente as seguintes funções e classes :

```
from datetime import date

def converte_str_para_data(data_str):
    dia, mês, ano = data_str.split('/')
    return Data(int(dia), int(mês), int(ano))

class Data:

    def __init__(self, dia, mês, ano):
        self.dia = dia
        self.mês = mês
        self.ano = ano
```

```
    def __str__(self):
        if self.dia < 10: data_str = '0' + str(self.dia)
        else: data_str = str(self.dia)
        if self.mês < 10: data_str += "/0" + str(self.mês) + "/"
        else: data_str += "/" + str(self.mês) + "/";
        data_str += str(self.ano)
        return data_str

    def __eq__(self, data):
        if str(self) == str(data): return True
        return False

    def __ne__(self, data):
        return not self == data

    def __gt__(self, data):
        if self.ano > data.ano: return True
        elif self.ano < data.ano: return False
        if self.mês > data.mês: return True
        elif self.mês < data.mês: return False
        if self.dia > data.dia: return True
        elif self.ano < data.ano: return False
        return False

    def __lt__(self, data):
        if self.ano < data.ano: return True
        elif self.ano > data.ano: return False
        if self.mês < data.mês: return True
        elif self.mês > data.mês: return False
        if self.dia < data.dia: return True
        elif self.ano > data.ano: return False
        return False

    def __ge__(self, data):
        if self < data: return False
        else: return True

    def __le__(self, data):
        if self > data: return False
        else: return True

    def calcular_idade(self):
        dia_atual_str, mês_atual_str, ano_atual_str = date.today().strftime("%d/%m/%Y").split('/')
        dia_atual, mês_atual, ano_atual = int(dia_atual_str), int(mês_atual_str), int(ano_atual_str)
        idade = ano_atual - self.ano
        if mês_atual < self.mês or (mês_atual == self.mês and dia_atual < self.dia): idade -= 1
        return idade


def selecionar_projetos(projetos, data_máxima_início, titulação_orientador,
    ano_mínimo_ingresso_aluno, modalidade_tcc, ic_com_bolsa):
    filtros = 'Filtros:: '
    if data_máxima_início != None: filtros += 'data máxima de início: ' + str(data_máxima_início)
    if titulação_orientador != None:
       filtros += ' - titulação do orientador: ' + titulação_orientador
    if ano_mínimo_ingresso_aluno != -1:
       filtros += ' - ano mínimo de ingresso do aluno: ' + str(ano_mínimo_ingresso_aluno)
    if modalidade_tcc != None: filtros += '\n          - modalidade do TCC: ' + modalidade_tcc
    if ic_com_bolsa == True: filtros += ' - IC com bolsa'
    elif ic_com_bolsa == False: filtros += ' - IC sem bolsa'
    projetos_selecionados = {}
    for título_projeto in projetos:
        projeto = projetos[título_projeto]
        orientador = projeto.orientador
        aluno = projeto.aluno
        if data_máxima_início != None and projeto.data_início > data_máxima_início: continue
        if titulação_orientador != None and orientador.titulação != titulação_orientador: continue
        if ano_mínimo_ingresso_aluno != -1 and aluno.ano_ingresso < ano_mínimo_ingresso_aluno:
            continue
        if isinstance(projeto, TrabalhoConclusãoCurso):
            if modalidade_tcc != None and modalidade_tcc != projeto.modalidade: continue
        elif isinstance(projeto, IniciaçãoCientífica):
            if ic_com_bolsa in (True, False) and ic_com_bolsa != projeto.com_bolsa: continue
        projetos_selecionados[título_projeto] = projetos[título_projeto]
    return filtros, projetos_selecionados
```

```
class Projeto:

    def __init__(self, título, orientador, aluno, data_início):
        self.título = título
        self.orientador = orientador
        self.aluno = aluno
        self.data_início = data_início

    def __str__(self):
        return self.título + ' - data de início: ' + str(self.data_início)\
           + '\n- orientador: ' + str(self.orientador) + '\n- aluno: ' + str(self.aluno)

class TrabalhoConclusãoCurso(Projeto):

    def __init__(self, título, orientador, aluno, início, modalidade):
        super().__init__(título, orientador, aluno, início)
        self.modalidade = modalidade if modalidade in ('monografia', 'artigo técnico científico',
            'protótipo') else 'inválida'

    def __str__(self):
        return 'Trabalho de Conclusão de Curso: ' + super().__str__()]
            + ' - modalidade: ' +  self.modalidade

class IniciaçãoCientífica(Projeto):

    def __init__(self, título, orientador, aluno, início, com_bolsa):
        super().__init__(título, orientador, aluno, início)
        self.com_bolsa = com_bolsa

    def __str__(self):
        vínculo = 'bolsista' if self.com_bolsa else 'voluntário'
        return  'Iniciação Científica: ' + super().__str__() + '- vínculo: ' +  vínculo
```

Na função ilustrar_classes_objetos, mantenha sem comentar o trecho de código que ilustra a subseção 5.1, e acrescente o trecho de código que ilustra a subseção 5.2.

```
def ilustrar_classes_objetos():
    print('\n5.2 - utilizar herança para especializar classes de objetos')
    projetos = {}
    projetos['Produção de Baterias com Grafeno']\
        = IniciaçãoCientífica('Produção de Baterias com Grafeno', professores['Marina Andrade'],
        alunos['Ariane Villas'], Data(1,8,2019), True)
    projetos['Propriedades Físicas do Grafeno']\
        = TrabalhoConclusãoCurso('Propriedades Físicas do Grafeno', professores['Marina Andrade'],
        alunos['Angelo Silveira'], Data(1,3,2019), 'protótipo')
    projetos['Avalições de Tensões em Engates Ferroviários']\
        = IniciaçãoCientífica('Avalições de Tensões em Engates Ferroviários',
        professores['Diogo Tavares'], alunos['Fabia Baroni'], Data(1,8,2020), False)
    projetos['Análise Numérica de Transferência de Calor em Motores']\
        = TrabalhoConclusãoCurso('Análise Numérica de Transferência de Calor em Motores',
        professores['Diogo Tavares'], alunos['Alessandro Sales'], Data(1,3,2020), 'monografia')
    projetos['Vibração em Hélices de Energia Eólica']\
        = IniciaçãoCientífica('Vibração em Hélices de Energia Eólica',
        professores['Talia Menucci'], alunos['Alexia Fantoni'], Data(1,8,2020), True)
    projetos['Vibração em Hélices de Drones']\
        = TrabalhoConclusãoCurso('Vibração em Hélices de Energia Eólica',
        professores['Talia Menucci'], alunos['Tales Oliveira'], Data(1,3,2021),
        'artigo técnico científico')
    imprimir_dicionário_objetos('Propostas de Projetos:', projetos)
    filtros, projetos_selecionados\
        = selecionar_projetos(projetos, None, None, -1, None, None)
    imprimir_dicionário_objetos(filtros, projetos_selecionados)
    filtros, projetos_selecionados\
        = selecionar_projetos(projetos, Data(1,1,2021), None, -1, None, None)
    imprimir_dicionário_objetos(filtros, projetos_selecionados)
    filtros, projetos_selecionados\
        = selecionar_projetos(projetos, Data(1,1,2021), 'doutorado', -1, None, None)
    imprimir_dicionário_objetos(filtros, projetos_selecionados)
    filtros, projetos_selecionados\
        = selecionar_projetos(projetos, Data(1,1,2021), 'doutorado', 2017, None, None)
    imprimir_dicionário_objetos(filtros, projetos_selecionados)
    filtros, projetos_selecionados\
        = selecionar_projetos(projetos, Data(1,1,2021), 'doutorado', 2017, 'protótipo', False)
    imprimir_dicionário_objetos(filtros, projetos_selecionados)
    filtros, projetos_selecionados\
        = selecionar_projetos(projetos, Data(1,1,2021), 'doutorado', 2017, 'protótipo', False)
    imprimir_dicionário_objetos(filtros, projetos_selecionados)
```

A saída da execução do programa é:

```
5.2 - utilizar herança para especializar classes de objetos

Projetos:
1 - Iniciação Científica: Produção de Baterias com Grafeno - data de início: 01/08/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
             - vínculo: bolsista
2 - Trabalho de Conclusão de Curso: Propriedades Físicas do Grafeno - data de início: 01/03/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
             - modalidade: protótipo
3 - Iniciação Científica: Avalições de Tensões em Engates Ferroviários - data de início: 01/08/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
             - vínculo: voluntário
4 - Trabalho de Conclusão de Curso: Análise Numérica de Transferência de Calor em Motores
    - data de início: 01/03/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Alessandro Sales - email: alessandro.sales@gmail.com - ano de ingresso: 2017
             - modalidade: monografia
5 - Iniciação Científica: Vibração em Hélices de Energia Eólica - data de início: 01/08/2020
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
             - vínculo: bolsista
6 - Trabalho de Conclusão de Curso: Vibração em Hélices de Drones - data de início: 01/03/2021
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Tales Oliveira - email: tales_oliveira@gmail.com - ano de ingresso: 2018
             - modalidade: artigo técnico científico

Filtros::
1 - Iniciação Científica: Produção de Baterias com Grafeno - data de início: 01/08/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
             - vínculo: bolsista
2 - Trabalho de Conclusão de Curso: Propriedades Físicas do Grafeno - data de início: 01/03/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
             - modalidade: protótipo
3 - Iniciação Científica: Avalições de Tensões em Engates Ferroviários - data de início: 01/08/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
             - vínculo: voluntário
4 - Trabalho de Conclusão de Curso: Análise Numérica de Transferência de Calor em Motores
    - data de início: 01/03/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Alessandro Sales - email: alessandro.sales@gmail.com - ano de ingresso: 2017
             - modalidade: monografia
5 - Iniciação Científica: Vibração em Hélices de Energia Eólica - data de início: 01/08/2020
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
             - vínculo: bolsista
6 - Trabalho de Conclusão de Curso: Vibração em Hélices de Drones - data de início: 01/03/2021
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Tales Oliveira - email: tales_oliveira@gmail.com - ano de ingresso: 2018
             - modalidade: artigo técnico científico
```

```
Filtros:: data máxima de início: 01/01/2021
1 - Iniciação Científica: Produção de Baterias com Grafeno - data de início: 01/08/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
             - vínculo: bolsista
2 - Trabalho de Conclusão de Curso: Propriedades Físicas do Grafeno - data de início: 01/03/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
             - modalidade: protótipo
3 - Iniciação Científica: Avalições de Tensões em Engates Ferroviários - data de início: 01/08/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
             - vínculo: voluntário
4 - Trabalho de Conclusão de Curso: Análise Numérica de Transferência de Calor em Motores
    - data de início: 01/03/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Alessandro Sales - email: alessandro.sales@gmail.com - ano de ingresso: 2017
             - modalidade: monografia
5 - Iniciação Científica: Vibração em Hélices de Energia Eólica - data de início: 01/08/2020
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
             - vínculo: bolsista

Filtros:: data máxima de início: 01/01/2021 - titulação do orientador: doutorado
1 - Iniciação Científica: Produção de Baterias com Grafeno - data de início: 01/08/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
             - vínculo: bolsista
2 - Trabalho de Conclusão de Curso: Propriedades Físicas do Grafeno - data de início: 01/03/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
             - modalidade: protótipo
3 - Iniciação Científica: Vibração em Hélices de Energia Eólica - data de início: 01/08/2020
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
             - vínculo: bolsista

Filtros:: data máxima de início: 01/01/2021 - titulação do orientador: doutorado
          - ano mínimo de ingresso do aluno: 2017
1 - Iniciação Científica: Produção de Baterias com Grafeno - data de início: 01/08/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
             - vínculo: bolsista
2 - Iniciação Científica: Vibração em Hélices de Energia Eólica - data de início: 01/08/2020
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
             - vínculo: bolsista

Filtros:: data máxima de início: 01/01/2021 - titulação do orientador: doutorado
          - ano mínimo de ingresso do aluno: 2017 - modalidade do TCC: protótipo
1 - Iniciação Científica: Produção de Baterias com Grafeno - data de início: 01/08/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
             - vínculo: bolsista
2 - Iniciação Científica: Vibração em Hélices de Energia Eólica - data de início: 01/08/2020
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
             - vínculo: bolsista

Filtros:: data máxima de início: 01/01/2021 - titulação do orientador: doutorado
          - ano mínimo de ingresso do aluno: 2017 - modalidade do TCC: protótipo - IC sem bolsa
```

Na classe Data o método __init__ recebe como parâmetros: dia, mês e ano da data. O método __str__, gera um string no formato: dd/mm/aaaa. Quando o objeto da classe Data é inicializado com a data de nascimento de uma pessoa, o método calcular_idade, obtém a data atual e calcula a idade em função da data atual e da data de nascimento.

```
data_nascimento = Data(18,8,1998)
idade = data_nascimento.calcular_idade()
```

Varíaveis com valores inteiros (do tipo int) ou ponto flutuante (do tipo float) podem ser comparadas pela utilização de comparadores relacionais: ==, !=, >, >=, <, <=. Objetos também podem ser variáveis comparáveis, com a utilização dos operadores relacionais, desde que a classe de objetos implemente métodos padronizados para representar cada um desses operadores. Para ilustrar o uso de objetos comparáveis, a classe Data foi definida com seis métodos que representam os seis comparadores relacionais.

A implementação do método __eq__ (*equal*) na classe Data possibilita que o operador relacional == seja utilizado para comparar se dois objetos da classe Data tem datas iguais:

```
data1 = Data(13,8,2021)
data2 = Data(13,8,2021)
if data1 == data2: print('datas iguais')
else: print('datas diferentes')
```

A implementação mais óbvia para o método __eq__, seria retornar um valor verdadeiro (True) para a comparação, se dia, mês e ano das duas datas comparadas forem iguais. Os parâmetros do método __eq__ são: self e data. O parâmetro self recebe o próprio objeto da classe Data. O parâmetro data recebe outro objeto da classe Data cuja data será comparada com a data do objeto self.

```
def __eq__(self, data):
    if self.dia == data.dia and self.mês == data.mês and self.ano == data.ano: return True
    return False
```

Este método funciona muito bem quando são atribuídos dois objetos da classe Data às variáveis a serem comparadas. No entanto para qualquer objeto pode ser atribuído o valor None. Se uma das variáveis, ou as duas, receber o valor None (sigficando que nenhum objeto foi criado), a implementação método __eq__ vai falhar, simplesmente porque não é possível obter dia, mês e ano de um objeto com valor None.

Uma alternativa para solucionar esse problema seria verificar se os objetos self e data são None:

```
def __eq__(self, data):
    if self == None or data == None: return False
    if self.dia == data.dia and self.mês == data.mês and self.ano == data.ano: return True
    return False
```

Essa função é recursiva, pois utiliza o próprio operador ==, cujo método __eq__ se propõe a representar. Neste caso, o método __eq__ utiliza o operador ==, que resulta na chamada do próprio método __eq__, se envolvendo em uma sucessão sem fim de chamadas do método __eq__.

Uma alternativa para solucionar essa problema é converter os objetos self e data em strings, utilizando a função str e, então, utilizar o comparador == para comparar dois strings, o evitará a chamada recursiva do método __eq__ da classe Data. Essa solução funciona para as todas as atribuições possíveis das variáveis sendo comparadas, estando essas variáveis com valores correspondentes a objetos da classe Data ou a None.

```
def __eq__(self, data):
    if str(self) == str(data): return True
    return False
```

A implementação do método __ne__ (*not equal*) na classe Data possibilita que o operador relacional **!=** seja utilizado para comparar se dois objetos da classe Data são diferentes. Essa implementação se aproveita da existência do método __eq__ para negar (com o operador **not**) a comparação que utiliza o operador ==.

```
def __ne__(self, data):
    return not self == data
```

A implementação do método __gt__ (*greater than*) na classe Data possibilita que o operador relacional > seja utilizado para comparar se o objeto self tem uma data posterior (maior que) à data do objeto data. De forma semelhante, a implementação do método __lt__ (*less than*) na classe Data possibilita que o operador relacional < seja utilizado para comparar se o objeto self tem uma data anterior (menor que) à dada do objeto data. A implementação desses métodos pode simplesmente comparar se as partes da data do objeto self (dia, mês e ano) são maiores ou menores (conforme o caso) que as partes da data do objeto data. Neste casos, não há necessidade de checar objetos da classe Data com valor None, pois não faz sentido verificar se uma data é posterior ou anterior a None.

A implementação do método __ge__ (*greater than or equal*) na classe Data possibilita que o operador relacional >= seja utilizado para comparar se o objeto self tem uma data posterior (maior que) ou igual à dada do objeto data. De forma semelhante, a implementação do método __lt__ (*less than or equal*) na classe Data possibilita que o operador relacional <= seja utilizado para comparar se o objeto self tem uma data anterior (menor que) ou igual à dada do objeto data. As implementações desses métodos utilizam os comparadores > e <, se beneficiando das implementações dos métodos __gt__ e __lt__.

De forma análoga à ilustração da classe Data, uma outra classe pode suportar a comparação de seus objetos com a utilização de operadores relacionais, desde que esta outra classe implemente os seis métodos que representam esses operadores.

O dicionário projetos é preenchido com seis objetos da classe Projeto, utilizando o título do projeto como chave. Também são utilizados os dicionários professores e alunos, as classes Professor e Aluno, e a função imprimir_dicionário_objetos.

No lugar da classe PropostaProjeto é utilizada a classe Projeto como superclasse das subclasses TrabalhoConclusãoCurso e IniciaçãoCientífica.

Na superclasse Projeto são definidos os dados e métodos comuns, que serão herdados pelas subclasses. A superclasse é defina como qualquer outra classe. As suas subclasses, no entanto tem as seguintes diferenças:

- na declaração da subclasse é necessário informar a sua superclasse;
- no método __init__ é necessário repassar, para a superclasse, os parâmetros utilizados para inicializar os dados herdados;
- sobreposição de métodos herdados.

Na declaração da subclasse é necessário informar a sua superclasse, entre parênteses. Observe a exemplo da subclasse TrabalhoConclusãoCurso, declarada como subclasse da superclasse Projeto:

- class TrabalhoConclusãoCurso(Projeto):

Os parâmetros que serão utilizados para inicializar os valores dos dados herdados da superclasse são repassados, no primeiro comando do bloco interno da função. O método __init__ da superclasse é chamado a partir da função super(), que retorna o objeto da superclasse, permitindo chamar um método definido na superclasse com o mesmo nome do método definido na subclasse. Os parâmetros que vão inicializar os dados específicos da subclasse são tratados da mesma forma que na superclasse. Observe o método __init__ da suclasse TrabalhoConclusãoCurso:

```
def __init__(self, título, orientador, aluno, início, modalidade):
    super().__init__(título, orientador, aluno, início)
    self.modalidade = modalidade if modalidade\
        in ('monografia', 'artigo técnico científico', 'protótipo') else 'inválida'
```

Qualquer método herdado da superclasse pode ser sobreposto, ou seja, ter o seu bloco interno de comandos alterado mantendo a mesma assinatura (nome do método e lista de parâmetros). O método __str__ sempre é sobreposto, dado que deverá gerar um string concatenando adicionalmente os dados específicos da subclasse. Veja o exemplo com o método __str__ da subclasse TrabalhoConclusãoCurso, no qual o método __str__ da superclasse também é utilizado para montar o string:

```
def __str__(self):
    return 'Trabalho de Conclusão de Curso: ' + super().__str__() + ' - modalidade: '\
        +  self.modalidade
```

A função imprimir_dicionário_objetos é chamada quatro vezes para imprimir projetos selecionados, passando como argumento a função selecionar_projetos, tendo como argumentos combinações de valores de cinco filtros. Nesta subseção também é utilizado, adicionalmente, um filtro baseado em um dado booleano. Como um dado booleano só pode assumir os valores True e False e ambos são válidos como filtros, o string 'X' é utilizado para informar que qualquer valor de dado atende esse filtro, ou seja, tanto faz se o dado do objeto for True ou False.

A função selecionar_projetos difere da função selecionar_propostas_projetos, no seguintes pontos:

- na montagem de um string com nomes e valores dos filtros válidos
  - para o filtro ic_com_bolsa baseado no dado booleano com_bolsa da classe IniciaçãoCientífica, caso o valor passado seja:
    - 'X' (para indicar filtro inválido): será omitido no string com os filtros válidos
    - True: será montado o string como IC com bolsa
    - False: será montado o string como IC sem bolsa
- para os filtros originários das subclasses TrabalhoConclusãoCurso e IniciaçãoCientífica
  - é utilizada a função isinstance para testar se o objeto projeto é desta subclasse, para poder testar se o filtro é atendido
    - **if** isinstance(projeto, TrabalhoConclusãoCurso):
      - **if** modalidade_tcc != None **and** modalidade_tcc != projeto.modalidade: continue
- idem para o filtro originário da subclasse IniciaçãoCientífica, cujos valores válidos deverão ser True ou False
  - elif isinstance(projeto, IniciaçãoCientífica):
    - if ic_com_bolsa in (True, False) and ic_com_bolsa != projeto.com_bolsa: continue

As combinações de valores de filtros passadas com argumentos nas quatro chamadas da função selecionar_projetos obedecem os mesmos critérios definidos, na subseção 5.1, para a função selecionar_propostas_projetos. No entanto, o resultado neste caso tem três diferenças.

Os filtros gerados a partir de dados das subclasses, só devem ser aplicados a objetos das subclasses, depois que foi testado que o objeto pertence à subclasse (utilizando a função isinstance).

Com a utilização do filtro modalidade do TCC = protótipo, não houve redução da lista de projetos selecionados. Esse filtro é específico para TCC (objeto da subclasse), e já não havia nenhum TCC na lista filtrada até então.

Com a utilização do filtro IC sem bolsa, não sobrou nenhum projeto na lista. Embora esse filtro seja especifico para IC (objeto da subclasse IniciaçãoCientífica) e a lista filtrada até então contasse com dois projetos de IC, os dois projetos da lista eram IC com bolsa e, portanto, não atendiam o filtro.

Como havia sido comentado anteriormente, na subseção 5.1, para demonstrar de forma mais didática a influência de cada filtro, é esperado que a lista de objetos selecionados sofra redução a cada acréscimo de filtro válido e ao final a lista não esteja vazia. Para conseguir este efeito, é recomendado aumentar as lista de objetos cadastrados e a utilização de várias combinações de dados desses objetos.

5.3 - Salvamento/recuperação de objetos em/de arquivos json

No módulo classes_objetos acrescente as seguintes funções :

```
def converter_dicionários_para_objetos_alunos(alunos_dicionários):
    alunos = {}
    for nome_aluno in alunos_dicionários:
        aluno = Aluno(**alunos_dicionários[nome_aluno])
        alunos[nome_aluno] = aluno
    return alunos

def converter_objetos_alunos_para_dicionários(alunos):
    alunos_dicionários = {}
    for nome_aluno in alunos:
        aluno = alunos[nome_aluno]
        alunos_dicionários[nome_aluno] = aluno.__dict__
    return alunos_dicionários

def converter_dicionários_para_objetos_professores(professores_dicionários):
    professores = {}
    for nome_professor in professores_dicionários:
        professor = Professor(**professores_dicionários[nome_professor])
        professores[nome_professor] = professor
    return professores

def converter_objetos_professores_para_dicionários(professores):
    professores_dicionários = {}
    for nome_professor in professores:
        professor = professores[nome_professor]
        professores_dicionários[nome_professor] = professor.__dict__
    return professores_dicionários
```

```
def converter_dicionários_para_objetos_projetos(projetos_dicionários, professores, alunos):
    projetos = {}
    for título_projeto in projetos_dicionários:
        projeto = Projeto(**projetos_dicionários[título_projeto])
        projeto.orientador = professores[projeto.orientador]
        projeto.aluno = alunos[projeto.aluno]
        projeto.data_início = converte_str_para_data(str(projeto.data_início))
        projetos[título_projeto] = projeto
    return projetos

def converter_objetos_projetos_para_dicionários(projetos):
    projetos_dicionários = {}
    for título_projeto in projetos:
        projeto = projetos[título_projeto]
        nome_orientador = projeto.orientador.nome
        nome_aluno = projeto.aluno.nome
        data_str = str(projeto.data_início)
        projeto_referências_por_chaves\
            = Projeto(título_projeto, nome_orientador, nome_aluno, data_str)
        projetos_dicionários[título_projeto] = projeto_referências_por_chaves.__dict__
    return projetos_dicionários
```

Na função ilustrar_classes_objetos, mantenha sem comentar os trechos de código que ilustram as subseções 5.1 e 5.2, e acrescente o trecho de código que ilustra a subseção 5.3.

```
def ilustrar_classes_objetos():
    print('\n5.3 - salvar/recuperar objetos em/de arquivos json')
    salvar_arquivo_json('professores', converter_objetos_professores_para_dicionários(professores))
    salvar_arquivo_json('alunos', converter_objetos_alunos_para_dicionários(alunos))
    salvar_arquivo_json('projetos', converter_objetos_projetos_para_dicionários(projetos))
    professores_recuperados\
        = converter_dicionários_para_objetos_professores(carregar_arquivo_json('professores'))
    print('Professores recuperados do arquivo json')
    for nome_professor in professores_recuperados:
       print('- ' + str(professores_recuperados[nome_professor]))
    alunos_recuperados = converter_dicionários_para_objetos_alunos(carregar_arquivo_json('alunos'))
    print('Alunos recuperados do arquivo json')
    for nome_aluno in alunos_recuperados: print('- ' + str(alunos_recuperados[nome_aluno]))
    projetos_recuperados = converter_dicionários_para_objetos_projetos(
        carregar_arquivo_json('projetos'), professores_recuperados, alunos_recuperados)
    print('Projetos recuperados do arquivo json')
    for título_projeto in projetos_recuperados:
        print('- ' + str(projetos_recuperados[título_projeto]))
```

A saída da execução do programa é:

```
5.3 - salvar/recuperar objetos em/de arquivos json
Professores recuperados de arquivo json
- Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
- Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
- Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
Alunos recuperados de arquivo json
- Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
- Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
- Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
- Alessandro Sales - email: alessandro.sales@gmail.com - ano de ingresso: 2017
- Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
- Tales Oliveira - email: tales_oliveira@gmail.com - ano de ingresso: 2018
Projetos recuperados de arquivo json
- Produção de Baterias com Grafeno - data de início: 01/08/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Ariane Villas - email: ariane.villas@gmail.com - ano de ingresso: 2017
- Propriedades Físicas do Grafeno - data de início: 01/03/2019
    - orientador: Marina Andrade - email: marina.andrade@ufgd.edu.br - titulação: doutorado
    - aluno: Angelo Silveira - email: angelo.silveira@gmail.com - ano de ingresso: 2016
- Avalições de Tensões em Engates Ferroviários - data de início: 01/08/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Fabia Baroni - email: fabia.baroni@gmail.com - ano de ingresso: 2018
- Análise Numérica de Transferência de Calor em Motores - data de início: 01/03/2020
    - orientador: Diogo Tavares - email: diogo.tavares@ufgd.edu.br - titulação: mestrado
    - aluno: Alessandro Sales - email: alessandro.sales@gmail.com - ano de ingresso: 2017
- Vibração em Hélices de Energia Eólica - data de início: 01/08/2020
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Alexia Fantoni - email: alexia_fantoni@gmail.com - ano de ingresso: 2018
- Vibração em Hélices de Drones - data de início: 01/03/2021
    - orientador: Talia Menucci - email: talia.menucci@ufgd.edu.br - titulação: doutorado
    - aluno: Tales Oliveira - email: tales_oliveira@gmail.com - ano de ingresso: 2018
```

Para salvar e recuperar o dicionário de objetos da classe Professor no arquivo professores.json é necessário:

- converter o dicionário de objetos da classe Professor para um dicionário de dicionários, utilizando a função converter_objetos_professores_para_dicionários;
- salvar o dicionário resultante no arquivo professores.json, utilizando a função salvar_arquivo_json (descrita na subseção 4.2);
- recuperar o dicionário de dicionários do arquivo professores.json, utilizando a função carregar_arquivo_json (descrita na subseção 4.2);
- converter o dicionário de dicionários em um dicionário de objetos da classe Professor, utilizando a função converter_dicionários_para_objetos_professores.

A função converter_objetos_professores_para_dicionários:

- itera na chaves (nome_professor) no dicionário professores (de objetos da classe Professor)
    - obtém o objeto professor associado ao nome_professor
    - converte objeto para dicionário (conforme visto na seção 4.4)
    - armazena o dicionário obtido em um dicionário de dicionários
- retorna o dicionário de dicionários

A função converter_dicionários_para_objetos_professores:

- itera na chaves (nome_professor) no dicionário de dicionários professores_dicionários
    - obtém o dicionário associado ao nome_professor
    - converte dicionário para objeto da classe Professor (conforme visto na seção 4.3)
    - armazena o objeto obtido em um dicionário de objetos da classe Professor
- retorna o dicionário de objetos da classe Professor

O código para salvar e recuperar o dicionário de objetos da classe Aluno no arquivo alunos.json é equivalente ao utilizado para a classe Professor.

Para salvar e recuperar o dicionário de objetos da classe Projeto no arquivo projetos.json é utilizado um procedimento semelhante para as classes Professor e Aluno. A diferença ocorre no código das funções de conversão, pois um objeto da classe Projeto referencia três outros objetos, o que não ocorre com objetos das classes Professor e Aluno. Uma referência a um objeto não pode ser armazenada em um arquivo json; é necessário convertê-la para chave do objeto antes de armazenar e recriar o objeto a partir de sua chave após recuperar o dicionário que o representa.

A função converter_objetos_projetos_para_dicionários:

- itera na chaves (título_projeto) no dicionário projetos (de objetos da classe Projeto)
  - obtém o objeto projeto associado ao título_projeto
  - a partir do objeto obtém
    - a chave do objeto professor : projeto.orientador.nome
    - a chave do objeto aluno : projeto.aluno.nome
    - a data de início do projeto e converte para string : str(projeto.data_início)
  - cria um novo objeto da classe Projeto com esses três valores
  - converte para o novo objeto para dicionário (conforme visto na seção 4.4)
  - armazena o dicionário obtido em um dicionário de dicionários
- retorna o dicionário de dicionários

A função converter_dicionários_para_objetos_projetos:

- itera na chaves (título_projeto) no dicionário de dicionários projetos_dicionários
  - obtém o dicionário associado ao seu título_projeto
  - converte dicionário para objeto da classe Projeto (conforme visto na seção 4.3)
  - altera dados do objeto
    - substituindo chaves por objetos obtidos a partir de suas chaves
      - projeto.orientador = professores[projeto.orientador]
      - projeto.aluno = alunos[projeto.aluno]
    - substituindo a data como string pelo objeto da classe Data
      - utilizando a função converte_str_para_data para converter string data para objeto
        - projeto.data_início = converte_str_para_data(str(projeto.data_início))
  - armazena o objeto obtido em um dicionário de objetos da classe Projeto
- retorna o dicionário de objetos da classe Projetos