EXERCÍCIOS

DESCRIÇÃO

Resolva os exercícios abaixo.

1. Com recursividade e a linguagem de sua preferência, faça um contador para explodir uma bomba, no formato do exemplo abaixo:

entrada:

```
5
```

saída:

```
5
4
3
2
1
BOOM!
```

```python
def contagem_regressiva(n):
    if n > 0:
        print(n)
        contagem_regressiva(n - 1)
    else:
        print("BOOM!")

n = int(input("Digite um número: "))
contagem_regressiva(n)
```

2. Crie um exemplo de código utilizando *labeled loop*.

```java
public class Bomba {
   public static void main(String[] args) {
       int n = 5;
       bomba:
       for (int i = n; i > 0; i--) {
           System.out.println(i);
           if (i == 1) {
               break bomba;
           }
       }
       System.out.println("BOOM!");
   }
}
```

3. Pesquise algum recurso de linguagem de programação não visto em aula e demonstre seu uso prático através de um exemplo:

```python
# Python
'''
Podemos explorar o conceito de herança na orientação a objetos.
A herança é um recurso que permite criar uma nova classe a partir de
uma classe existente. A nova classe, chamada de subclasse, herda os
atributos e métodos da classe existente, chamada de superclasse.
'''

class Foguete:
   def __init__(self, fabricante, motor, combustivel):
       self._fabricante = fabricante
       self._motor = motor
       self._combustivel = combustivel

   @property
   def fabricante(self):
       return self._fabricante
```

```
    @property
    def motor(self):
        return self._motor

    @property
    def combustivel(self):
        return self._combustivel

    def specs(self):
        print(f"Fabricante: {self.fabricante}")
        print(f"Motor: {self.motor}")
        print(f"Combustível: {self.combustivel}")

class FalconHeavy(Foguete):
    def __init__(self, fabricante, motor, combustivel, vezes_reutilizado):
        super().__init__(fabricante, motor, combustivel)
        self._vezes_reutilizado = vezes_reutilizado

    @property
    def vezes_reutilizado(self):
        return self._vezes_reutilizado

    def falcon_heavy_specs(self):
        print(f"Vezes reutilizado: {self.vezes_reutilizado}")

'''
Neste exemplo, criei a classe FalconHeavy que herda atributos da
classe Foguete.


Ainda é possível explorar o conceito de heranças múltiplas, onde uma
classe pode herdar atributos e métodos de mais de uma classe.
'''

class Starship(FalconHeavy):
    def __init__(self, fabricante, motor, combustivel, vezes_reutilizado, prototipo,
usos_da_nave):
```

```
        super().__init__(fabricante, motor, combustivel, vezes_reutilizado)
        self._prototipo = prototipo
        self._usos_da_nave = usos_da_nave

    @property
    def prototipo(self):
        return self._prototipo

    @property
    def usos_da_nave(self):
        return self._usos_da_nave

    def starship_specs(self):
        print(f"Protótipo: {self.prototipo}")
        print(f"Usos da nave: {self.usos_da_nave}")

class Tripulante:
    def __init__(self, nome, idade, funcao):
        self._nome = nome
        self._idade = idade
        self._funcao = funcao

    @property
    def nome(self):
        return self._nome

    @property
    def idade(self):
        return self._idade

    @property
    def funcao(self):
        return self._funcao

    def tripulante_data(self):
        print(f"\nNome: {self.nome}")
        print(f"Idade: {self.idade}")
```

```
        print(f"Função: {self.funcao}")

class Tripulacao:
    def __init__(self):
        self._tripulacao = []

    @property
    def tripulacao(self):
        return self._tripulacao

    def adicionar_tripulante(self, tripulante):
        self._tripulacao.append(tripulante)

    def adicionar_tripulantes(self, tripulantes):
        self._tripulacao.extend(tripulantes)

    def remover_tripulante(self, tripulante):
        self._tripulacao.remove(tripulante)

    def tripulacao_data(self):
        for tripulante in self._tripulacao:
            tripulante.tripulante_data()

    def __str__(self):
        return "\n".join(str(tripulante) for tripulante in self._tripulacao)

class Voo(Starship, Tripulacao):
    def __init__(self, fabricante, motor, combustivel, vezes_reutilizado, prototipo,
usos_da_nave, tripulacao, origem, destino, duracao):
        Starship.__init__(self, fabricante, motor, combustivel, vezes_reutilizado,
prototipo, usos_da_nave)
        Tripulacao.__init__(self)
        self.adicionar_tripulantes(tripulacao)
        self._origem = origem
        self._destino = destino
        self._duracao = duracao
```

```
    @property
    def origem(self):
        return self._origem

    @property
    def destino(self):
        return self._destino

    @property
    def duracao(self):
        return self._duracao

    def voo_data(self):
        print("Dados da nave:\n")
        self.starship_specs()
        print("\nDados do Foguete:\n")
        self.specs()
        self.falcon_heavy_specs()
        print("\nDados da tripulação:")
        self.tripulacao_data()
        print(f"\nOrigem: {self.origem}")
        print(f"Destino: {self.destino}")
        print(f"Duração: {self.duracao}")

'''
Nesta última parte, criei a classe Tripulante, a classe Tripulação e a classe voo.
A classe Tripulante é responsável por armazenar informações sobre um tripulante.
A classe Tripulação é responsável por armazenar uma lista de tripulantes e
manipular esses dados.
A classe voo é responsável por armazenar informações sobre um voo, e nessa classe,
utilizei heranças múltiplas para herdar atributos e métodos das classes Starship e
Tripulação.
'''

dados_tripulantes = [
    ("Elijah Baley", 49, "Piloto"),
    ("R. Daneel Olivaw", 57, "Operador de sistemas"),
```

```
    ("R. Giskard Reventlov", 56, "Operador de sistemas"),
    ("R. Sammy", 1, "Passageiro"),
    ("Gladia Delmarre", 35, "Passageira"),
    ("Fastolfe", 70, "Cientista"),
    ("Dors Venabili", 30, "Passageira"),
    ("Hari Seldon", 60, "Passageiro"),
    ("Jezebel Baley", 30, "Passageira"),
    ("Amadiro", 40, "Diretor do Instituto de Robótica de Aurora"),
    ("Vasilia Fastolfe", 40, "Cientista"),
]

tripulantes01 = [Tripulante(*dados) for dados in dados_tripulantes[:3]]
tripulantes02 = [Tripulante(*dados) for dados in dados_tripulantes[3:8]]
tripulantes03 = [Tripulante(*dados_tripulantes[2]),  # Giskard
                 Tripulante(*dados_tripulantes[8]),  # Jezebel
                 Tripulante(*dados_tripulantes[9]),  # Amadira
                 Tripulante(*dados_tripulantes[10])] # Vasilia

tripulacao01 = Tripulacao()
tripulacao01.adicionar_tripulantes(tripulantes01)

tripulacao02 = Tripulacao()
tripulacao02.adicionar_tripulantes(tripulantes02)

tripulacao03 = Tripulacao()
tripulacao03.adicionar_tripulantes(tripulantes03)



voo01 = Voo("SpaceX", "Raptor", "CH4", 0, "SN8", 1, tripulacao01.tripulacao, "Boca
Chica, EUA, Terra", "Tycho, Lua", "00:00:02:06:00:00")
voo01.voo_data()

'''
Nesta demonstração, é possível ver o emprego de 3 paradigmas de programação:
1. Programação Orientada a Objetos:
   Classes e Objetos:
   - Classes: Foguete, FalconHeavy, Starship, Tripulante, Tripulacao e Voo
```

```
   - Objetos: Intâncias de Tripulante, Tripulacao e Voo
   Encapsulamento:
   - Atributos privados: _fabricante, _motor, _combustivel, _vezes_reutilizado,
     _prototipo, _usos_da_nave, _tripulacao, _origem, _destino e _duracao
   - Métodos getters para acessar atributos privados, como @property
   Herança:
   - Herança Simples: 'Falcon Heavy' herda de 'Foguete'
   - Herança Múltipla: 'Voo' herda de 'Starship' e 'Tripulação'
   Polimorfismo:
   - Métodos sobrescritos e estendidos, como 'specs()' em 'Foguete' e
     'falcon_heavy_specs()' em 'FalconHeavy'
2. Programação Funcional:
   Embora o código esteja fortemente orientado a objetos, é possível notar
   a presença de características do paradigma funcional.
   Funções como Objetos de Primeira Classe:
   - Utilização de funções para a criação de lista de objetos, como na lista
     'tripulantes01', 'tripulantes02' e 'tripulantes03'.
3. Programação Procedural:
   Além dos paradigmas anteriores, é possível notar a presença de características
   do paradigma procedural.
   Como em voo01 = Voos(...); voo01.voo_data(), onde há uma clara sequência de
   instruções para criar uma instância de 'Voo' e invocar o método 'voo_data()'.
'''
```

4. Pesquise em que linguagem a maior parte do UNIX foi escrita.

```
# A maior parte do UNIX foi escrita em C.
```

5. O que significa dizer que um programa é *confiável*?

```
'''
Dizer sobre a confiabilidade de um programa, é falar sobre a capacidade
desse programa de executar suas funções corretamente, de maneira a
evitar comportamentos inesperados e lidar com entradas inesperadas.
De maneira a oferecer robustez sem comprometer outros aspectos importantes
como a simplicidade e a capacidade do código de ser mantido e modificado.
'''
```

6. Quais são as vantagens de implementar uma linguagem com um interpretador?

```
'''
A utilização de um interpretador em comparação com um compilador,
oferece as vantagens de edição e execução imediatas, sem a necessidade
do processo de compilação, o processo de desenvolvimento do código se torna
mais rápido dado a essa natureza da facilidade de execução.
Também há maior portabilidade do código, pois basta que o interpretador
esteja disponível na máquina para que o código seja executado.
'''
```

7. Marque V para as afirmações verdadeiras e F para as falsas:
    a. (F) Java tem tipagem estática e dinâmica.
    b. (F) Python tem tipagem estática e forte.
    c. (V) Python tem tipagem dinâmica e forte.
    d. (V) JavaScript tem tipagem fraca e dinâmica.

PESO DA AVALIAÇÃO

Conforme plano de ensino.

OBSERVAÇÕES

- Plágio = **ZERO** **(inclui cópia ou simples alteração de trabalho de colegas)**
- Geração de respostas através de ferramentas de Inteligência Artificial = **ZERO**