Curso Intermediário II de Python para mulheres

APOIO:

Relembrando...

FUNÇÕES

Funções são sub-rotinas no código que servem para executar um procedimento muitas vezes, evitando que você tenha que reescrevê-lo mais de uma vez.

Uma funcionalidade importante é o fato que, caso precise realizar alguma alteração ou correção, ela vai ser feita nesta sub-rotina e não em diversas partes do código.

Exemplos

```
def soma(a, b):
    return a + b

print(soma(1, 2))
>>> 3
print(soma('PyLadies', ' São Paulo'))
>>> PyLadies São Paulo
```

```
def multiplica(a,b):
    return a*b

a = int(input("Diga o primeiro número: "))
b = int(input("Agora o segundo: "))

print(multiplica(a,b))
```

Programação Orientada a Objetos

"A orientação a objetos é um modelo de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos." - Wikipedia

Na programação, podemos obter representações de atividades reais realizadas por nós. Por exemplo, quem nunca fez o **algorítimo de trocar uma lâmpada?** Da mesma forma, podemos também representar objetos reais.

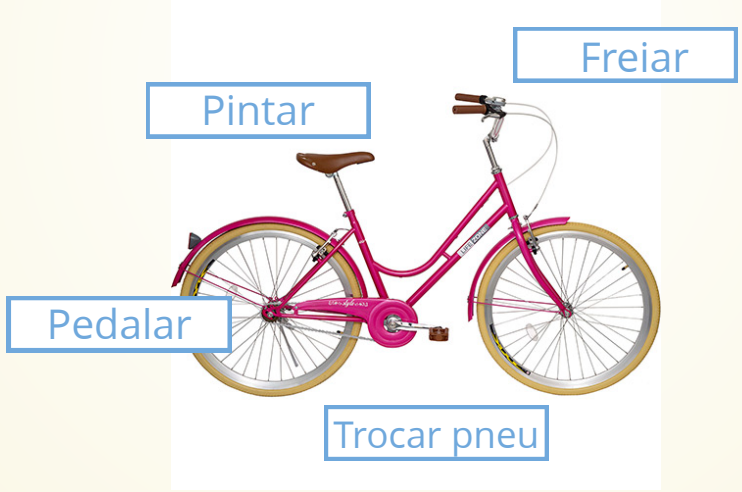
Representação de um objeto

Para representar um objeto da vida real, vamos tomar como exemplo uma **bicicleta**.

A bicicleta possui algumas **características**:

Representação de um objeto

Além disso, ela realiza (ou você realiza nela) algumas **atividades**:



Representação de um objeto

Na programação, as características e as atividades da bicicleta recebem outros nomes.

Atributos são as características do objeto, informações que estão ali como variáveis apenas guardando valores sobre ele.

Lanterna | Banco | Campainha

Métodos são as atividades do objeto, que atuam como funções que apenas aquele objeto realiza.

Pedalar | Pintar | Freiar

Teoria de orientação a objetos

O que acabamos de ver faz parte de todo um conceito de teorias sobre orientação a objetos. Essas teorias definem o que uma linguagem orientada a objetos é capaz de fazer e qual a sua diferença das linguagens que não o são. **Vocês sabem qual a diferença de Python para C?**

```
meu_dicionario = {"nome":"Julia", "idade": 19}
```

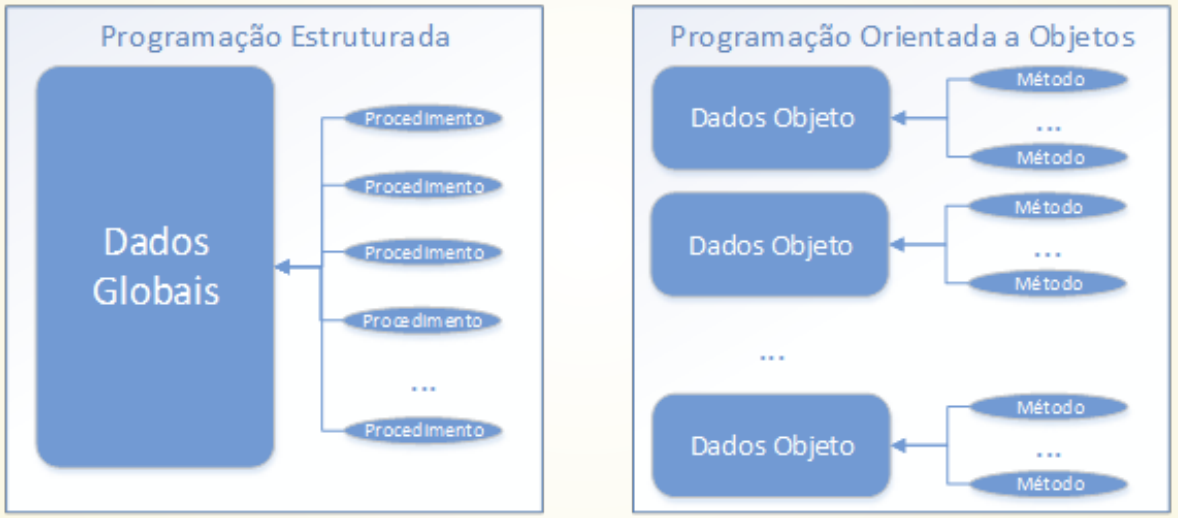
```
struct {
    char nome[50];
    int  idade;
};
```

Programação Estruturada x POO

Na programação **estruturada**, o sistema pode ser dividido em três partes: sequência, seleção e iteração. Isso significa que cada parte do software é um procedimento que pode vir a se repetir, mas que ao todo são apenas vários algorítimos, um atrás do outro.

Já na orientação a objetos, podemos incluir entidades chamadas **objetos** com comportamentos e características próprias que vão trabalhar como parte destes procedimentos, ou seja, são novos **tipos de variáveis personalizados** que podemos inserir nos nossos algorítimos, criando conceitos novos.

Estruturada x POO



Pilares da Orientação a Objetos

ABSTRAÇÃO

A **abstração** de um objeto é o que acabamos de fazer com a bicicleta. Tivemos a capacidade de *abstrair* um objeto real em atributos e métodos. Da mesma forma, devemos aplicar ainda mais essa capacidade quando trabalharmos com objetos não tão reais, como um objeto **Tempo**. Ele não é tangível para nós, mas podemos associar algumas das suas características: *horas, minutos, segundos*; e algumas das suas funções: *adiantar, atrasar, cronometrar, ler o tempo atual, etc.*

Pilares da Orientação a Objetos

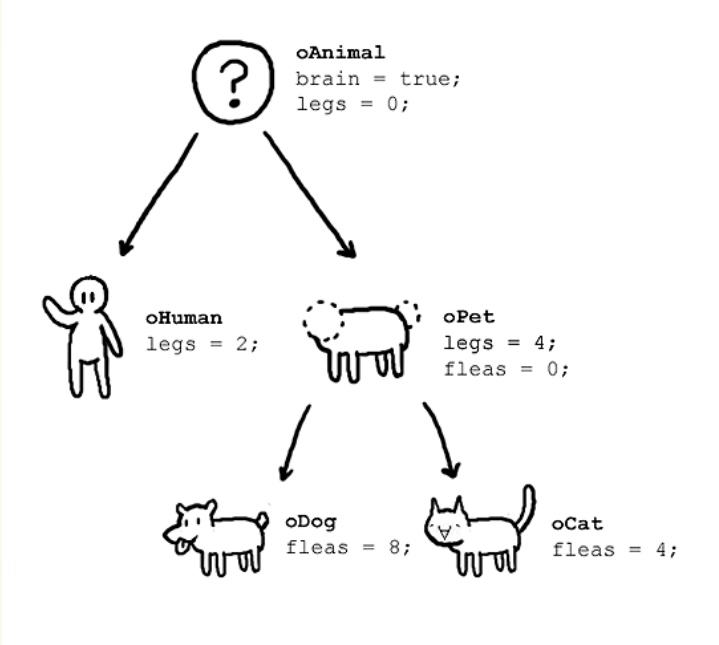
HERANÇA E POLIMORFISMO

Novamente como na vida real, o evolucionismo também foi aplicado na programação. Vamos considerar que objeto é absolutamente todas as **coisas** que existem no mundo. Portanto, um animal é um objeto. Quando um animal nasce de outro, ele tem as mesmas característica que o seu antepassado **mais** algumas características que ele mesmo evoluiu e desenvolveu. **Somos quase Pokémons!**

As características do antepassado ele recebeu como **herança**, já o que ele evoluiu e desenvolveu faz parte do seu **polimorfismo**.

Pilares da Orientação a Objetos

HERANÇA E POLIMORFISMO



Aplicando OO no Python

Todos os objetos que criamos dentro da programação são chamados de **classes**. Classes são como um tipo padrão do Python feito especificamente para receber atributos e métodos. A partir delas, é como se estivéssemos definindo tipos completamente novos.

Para nossa bicicleta, por exemplo, teríamos uma classe chama **bicicleta** com os atributos e métodos que definimos anteriormente. Quer ver?

Exemplos

CRIAÇÃO DE CLASSES

```
class Bicicleta():
    def __init__(self):
        self.cor = “rosa”
        self.rodas = 2
        self.banco = 1
        self.lanterna = True
        self.campainha = False
        self.velocidade = 0

    def pedalar(self):
        self.velocidade += 5

    def freiar(self):
        self.velocidade = 0

    def pintar(self, cor_nova):
        self.cor = cor_nova
```

__init__ é um *método de inicialização* utilizado pelo *construtor,* ou seja, é ele quem passa para a classe as informações que ela precisa sempre que uma nova *instância* for criada.

self é uma referência à própria classe, ou seja, ela está referenciando ela mesma.

Exemplos

CRIAÇÃO DE CLASSES

```
class Bicicleta():
    def __init__(self):
        self.cor = “rosa”
        self.rodas = 2
        self.banco = 1
        self.lanterna = True
        self.campainha = False
        self.velocidade = 0

    def pedalar(self):
        self.velocidade += 5

    def freiar(self):
        self.velocidade = 0

    def pintar(self, cor_nova):
        self.cor = cor_nova
```

```
minha_caloi10 = Bicicleta()
minha_caloi10.lanterna = False
minha_caloi10.pintar("Preto")

print(minha_caloi10.lanterna)
>>> False
print(minha_caloi10.cor)
>>> 'Preto'
print(minha_caloi10.rodas)
>>> 2
```

Exemplos

HERANÇA DE CLASSES

```
class Caloi10(Bicicleta):
    def __init__(self):
        Bicicleta.__init__(self)
        self.cor = "Preto"
        self.lanterna = False
```

```
minha_caloi10 = Caloi10()
minha_caloi10.pedalar()
print(minha_caloi10.velocidade)
>>> 5
```

Exemplos

POLIMORFISMO

```
class Caloi10(Bicicleta):
    def __init__(self):
        Bicicleta.__init__(self)
        self.cor = "Preto"
        self.lanterna = False
        self.marcha = 1

    def troca_marcha(self, marcha):
        self.marcha = marcha

    def pedalar(self):
        if self.marcha == 1:
            self.velocidade += 2
        elif self.marcha == 2:
            self.velocidade += 3
        elif self.marcha == 3:
            self.velocidade += 4
        else:
            self.velocidade += 5
```

```
minha_caloi10 = Caloi10()
minha_caloi10.pedalar()
minha_caloi10.trocar_marcha(2)
minha_caloi10.pedalar()
minha_caloi10.trocar_marcha(3)
minha_caloi10.pedalar()
print(minha_caloi10.velocidade)
>>> 9
```

Glossário

Relembrando as novas palavras que acabamos de aprender:

- **Atributo:** uma característica de um objeto;
- **Método:** uma função do objeto ou que realizamos com ele;
- **Construtor:** é um componente interno da linguagem que cria o objeto sempre que necessário, como uma "fábrica", a partir das informações passadas pelo **método de inicialização**.
- **Instância:** um objeto específico de um determinado tipo de objetos.

Prática

Escreva uma classe Televisão com atributos como canais, volume e ligada. Crie métodos para ligar, desligar, mudar de canal e aumentar ou diminuir o volume.

Prática

```
class Televisao():
    def __init__(self):
        self.canais = ["Globo", "SBT", "Record"]
        self.canal_atual = 0
        self.ligada = False
        self.volume = 10

    def liga(self):
        self.ligada = True

    def desliga(self):
        self.ligada = False

    def muda_canal(self, canal):
        self.canal_atual = canal
        print("Canal atual: %s" % self.canais[canal])

    def aumenta_volume(self):
        self.volume += 1

    def abaixa_volume(self):
        self.volume -= 1
```

Prática

Escreva uma classe Cliente que tenha como atributos os dados de um usuário (nome, telefone, idade, etc) e métodos alterar_nome, alterar_idade, etc.

Prática

```
class Cliente():
    def __init__(self):
        self.nome = ""
        self.idade = ""
        self.telefone = ""

    def muda_nome(self, nome):
        self.nome = nome
        print("Novo nome: %s" % nome)

    def muda_idade(self, idade):
        self.idade = idade
        print("Nova idade: %d" % idade)

    def muda_telefone(self, telefone):
        self.telefone = telefone
        print("Novo telefone: %s" % telefone)
```

Prática

Crie uma classe para implementar uma conta corrente. A classe deve possuir os seguintes atributos: número da conta, nome do correntista e saldo. Os métodos são os seguintes: alterar nome, depósito e saque;

Prática

```
class ContaCorrente():
    def __init__(self):
        self.numero_conta = "xxxxx-x"
        self.nome = "Fulana da Silva"
        self.saldo = 0

    def alterar_nome(self, nome):
        self.nome = nome
        print("Nome do correntista alterado para: {0}".format(nome))

    def deposito(self, valor):
        self.saldo += valor
        print("Depósito realizado no valor de R${0:.2f}".format(valor))

    def saque(self, valor):
        if self.saldo > valor:
            self.saldo -= valor
            print("Saque realizado no valor de R${0:.2f}".format(valor))
        else:
            print("Não foi possível realizar o saque.")
            print("Saldo insuficiente.")
```

Projeto OO: Criando um Tamagushi

Crie um jogo de Tamagushi (Bichinho Eletrônico):

1. **Atributos:** Nome, Fome, Saúde;
2. **Métodos:** Alterar Nome, Alimentar, Cuidar;
3. **Obs:** Existe mais uma informação que devemos levar em consideração, o Humor do nosso tamagushi, este humor é uma combinação entre os atributos Fome e Saúde, ou seja, um campo calculado, então podemos criar um método para calcular esse humor.

Dicas:

1. Crie a classe do Tamagushi primeiro;

2. Coloque o programa em repetição até que seja feito um comando para sair do jogo.

Projeto OO: Criando um Tamagushi

1º PASSO

```
class Tamagushi():

    def __init__(self):
        self.nome = "Fofinho"
        self.fome = 50
        self.saude = 50
```

```
class Tamagushi():
    ...

    def alterar_nome(self,nome):
        self.nome = nome
        print("Meu nome é {0}!".format(self.nome))
```

Projeto OO: Criando um Tamagushi

1º PASSO

```
class Tamagushi():
    ...

    def alimentar(self):
        if self.fome < 100:
            self.fome += 10
            print("Nham nham!")
        else:
            print("Parece que comi um elefante! D:")
```

Projeto OO: Criando um Tamagushi

1º PASSO

```
class Tamagushi():
    ...
    def cuidar(self):
        interacao = randint(0,2)
        if self.saude < 100:
            self.saude += 10

        if interacao == 0:
            print("*abraço*")
        elif interacao == 1:
            print("*lambeijo*")
        else:
            print("*pula em cima*")
            print("Melhor mamãe do mundo!")
```

Projeto OO: Criando um Tamagushi

1º PASSO

```
class Tamagushi():
    ...

    def calcula_humor(self):
        media = 0
        if self.saude < 50:
            media += 1
        elif self.saude == 100:
            media += 5
        else:
            media += 3

        if self.fome < 50:
            media += 1
        elif self.fome == 100:
            media += 5
        else:
            media += 3

        media = media/2
```

Projeto OO: Criando um Tamagushi

1º PASSO

```
class Tamagushi():
    ...

    def calcula_humor(self):
        ...
        if media == 1:
            print("Que dia péssimo! Estou me sentindo muito mal >:(")
        elif media == 2:
            print("Tô meio mal :/")
        elif media == 3:
            print("Tô melhorando... :)")
        elif media == 5:
            print("O dia está incrível! Estou super feliz :D")
        else:
            print("Tô confuso... não sei o que tô sentindo (?)")
```

Projeto OO: Criando um Tamagushi

2º PASSO

```
bichinho = """  _____000000000_____________________00000000000_____
__0000_______00_____________________00_________0000___
_00__________000000000____00000000_____________00___
_0_________000_____________________000_________00_
_0_______000_____________________000_________0_
_00_____00_____________________________00_____00_
__00___00_____0000_____________0000_____00___000__
___00000_____000000_____________000000_____00000____
_____00_______0000_____________0000_______00_______
_____00_____________00000000_____________00_______
_____00_____________00____00_____________00_______
_____0_____________000000_____________0_________
_____00_____________00_____________00_______
_____000_____________00_____________000_______
_____000_____________00_____________000_____
_____0000_____________00_____________0000_____
_____0000000_0000_0000000"""
```

Projeto OO: Criando um Tamagushi

3º PASSO

```
tg = Tamagushi()
```

```
def estatisticas():
    print("")
    print("{0}".format(bichinho))
    print("Estatísticas de {0}:".format(tg.nome))
    print("Fome: {0}%".format(tg.fome))
    print("Saúde: {0}%".format(tg.saude))
```

```
def menu():
    print("")
    print("Selecione uma opção:")
    print("1 - Alimentar")
    print("2 - Cuidar")
    print("3 - Mudar nome")
    print("4 - Ver humor")
    print("0 - Sair")
```

Projeto OO: Criando um Tamagushi

4º PASSO

```
op = 1
```

```
while (op != 0):
    estatisticas()
    menu()
    op = int(input("\n"))
    if op == 1:
        tg.alimentar()
    elif op == 2:
        tg.cuidar()
    elif op == 3:
        nome = input("Qual será meu novo nome? ")
        tg.alterar_nome(nome)
    elif op == 4:
        tg.calcula_humor()
    time.sleep(2)

print("Até mais! :3")
```

Relembrando...

MÓDULOS E PACOTES

Conforme vamos escrevendo as instruções, o programa vai ficando muito longo e difícil de dar manutenção. Sendo assim, o melhor é dividi-los em vários arquivos, que são os **módulos**.

Os módulos são arquivos com extensão **.py** que contém definições e declarações (funções, variáveis, etc).

As definições em um módulo podem ser importadas em outros módulos, ou seja, reusar um código já pronto, sem fazer o "copia" e "cola", sem duplicar.

Pacotes são um conjunto de módulos organizados hierarquicamente.

Exemplos

MÓDULOS

```
def hello_ladies():
    print(“Hello, Ladies!”)

def hello_lady(lady):
    print(“Hello, {0}!”.format(lady))
```

hello.py

```
import hello

hello.hello_ladies()
hello.hello_lady(“Julia”)

# ou
from hello import hello_ladies, hello_lady
hello.hello_ladies()
hello.hello_lady(“Julia”)

# ou
from hello import *
hello.hello_ladies()
hello.hello_lady(“Julia”)
```

outro.py

Exemplos

PACOTES

```
def func_laranja():
    print("Eu tenho uma laranja!")
```

laranja.py

```
def func_uva():
    print("Eu tenho uma uva!")
```

uva.py

```
def func_pera():
    print("Eu tenho uma pêra!")
```

pera.py

```
from laranja import func_laranja
from uva import func_uva
from pera import func_pera
```

__init__.py

```
Frutas/
├── __init__.py
├── laranja.py
├── pera.py
└── uva.py
```

```
import Frutas

Frutas.func_laranja()
Frutas.func_uva()
Frutas.func_pera()
```

outro.py

Compreensão de pacotes

Para criar pacotes, é mais fácil primeiro compreender o funcionamento de um pacote pronto. Para isso, vamos estudar o Tkinter, uma biblioteca/pacote de interfaces gráficas.

A documentação do Tkinter para referência pode ser encontrada em: https://docs.python.org/2/library/tkinter.html

Compreensão de pacotes

```
import tkinter

top = tkinter.Tk()

def hello_world():
    l = tkinter.Label(top, text="Hello world!")
    l.pack()

w = tkinter.Button(top, text="Clica em mim!", command=hello_world)
w.pack()

top.mainloop()
```

O que estamos utilizando ali? Classes? Funções?

Vamos observar passo a passo!

Compreensão de pacotes

```
import tkinter
```

O primeiro passo é importar a biblioteca que iremos utilizar com um import**.**

```
top = tkinter.Tk()
```

Em seguida, criamos uma instância da **classe** Tk. Como acabamos de ver, essa instância é um **objeto**. E que tipo de objeto é esse que estamos abstraindo? No caso do Tkinter, um objeto Tk é uma janela. Ou seja, estamos criando uma nova janela do programa.

Compreensão de pacotes

```
def hello_world():
    l = tkinter.Label(top, text="Hello world!")
    l.pack()
```

Após isso, criamos uma função hello_world. Dentro dela, utilizamos um objeto Label, que é um texto. Estamos abstraíndo uma linha de texto!

Após criar uma linha, utilizamos o **método** pack() para "empacotar" a linha. É um atalho do Tkinter para salvar essa ação que fizemos.

Compreensão de pacotes

```
w = tkinter.Button(top, text="Clica em mim!", command=hello_world)
w.pack()
```

Em seguida, criamos uma instância de Button, um botão que vai chamar a função hello_world e empacotamos ele.

```
top.mainloop()
```

Por fim, chamamos a função mainloop da biblioteca, que é uma função que faz a janela ficar aberta e não parar a execução até que o usuário mande.

Compreensão de pacotes

Se observarmos o código (que é aberto) da biblioteca, poderemos ver, por exemplo, como a classe Button foi construída:

```
class Button(Widget):
        def __init__(self, master=None, cnf={}, **kw):
        """Construct a button widget with the parent MASTER."""
                Widget.__init__(self, master, 'button', cnf, kw)
        def tkButtonEnter(self, *dummy):
                self.tk.call('tkButtonEnter', self._w)
        def tkButtonLeave(self, *dummy):
                self.tk.call('tkButtonLeave', self._w)
        def tkButtonDown(self, *dummy):
                self.tk.call('tkButtonDown', self._w)
        def tkButtonUp(self, *dummy):
                self.tk.call('tkButtonUp', self._w)
        def tkButtonInvoke(self, *dummy):
                self.tk.call('tkButtonInvoke', self._w)
        def flash(self):
                self.tk.call(self._w, 'flash')
        def invoke(self):
                return self.tk.call(self._w, 'invoke')
```

Compreensão de pacotes

Não precisamos entender tudo o que está escrito, mas podemos perceber algumas coisas: nosso botão é uma classe que herda de uma classe Widget, possui alguns métodos internos do Tkinter e interage com outros componentes do código.

Nada muito além do que vimos até agora, não é?

Criação de pacotes

Crie um módulo contendo as seguintes funções de interação com o usuário:

- cumprimentar (hello world) com o nome do usuário;
- cálculo da idade do usuário em dias (recebe a data de nascimento e retorna em dias até a data de hoje);

```
def hello_user(user):
    print("Hello, {0}".format(user))

def dias_idade(dia_nasc,mes_nasc,ano_nasc):
    from datetime import date
    dif = date.today() - date(ano_nasc, mes_nasc, dia_nasc)
    return dif.days
```

Criação de pacotes

Crie um módulo contendo:

- uma função de criptografia (você recebe um texto e muda ele para outro usando a lógica que quiser);
- uma função de comparação de criptografia (você recebe um texto e outro já criptografado, criptografa o primeiro e compara os dois);
- uma classe de Usuário com dois métodos:
  - cadastro (recebe nome, usuário e senha, que passará pela criptografia);
  - login (recebe senha e compara com a senha armazenada usando a função criada);

Criação de pacotes

```
def criptografa(senha):
    nova_senha = ""
    for c in senha:
        nova_senha += chr(ord(c)+ 30000)
    return nova_senha

def compara(senha, cript):
    nova_senha = criptografa(senha)
    if nova_senha == cript:
        return True
    else:
        return False
```

Criação de pacotes

```
class Usuario():
    def __init__(self):
        self.nome = ""
        self.usuario = ""
        self.senha = ""

    def cadastra(self, nome, usuario, senha):
        self.nome = nome
        self.usuario = usuario
        self.senha = criptografa(senha)

    def login(self, senha):
        if compara(senha, self.senha):
            return True
```

Criação de pacotes

Crie um pacote contendo os módulos anteriores e faça uso de todos os objetos e funções criados em um novo arquivo.

Curso Intermediário II de Python para mulheres

Julia Rizza

contato@juliarizza.com

APOIO:

Prêmio Mulheres TechemSampa

Google for Entrepreneurs

Rede MULHER EMPREENDEDORA

TechSampa