STEM²D

Go girls!

CODE

like a girl

BREAK

- Essa palavra-chave é utilizada tanto com o **while** quanto com o **for** para que o laço de repetição seja interrompido caso alguma condição seja verdadeira.

```
i = 10
while i > 2
    print(i)
    if i == 6:
        break
    i = i - 1
```

```
10
9
8
7
6
```

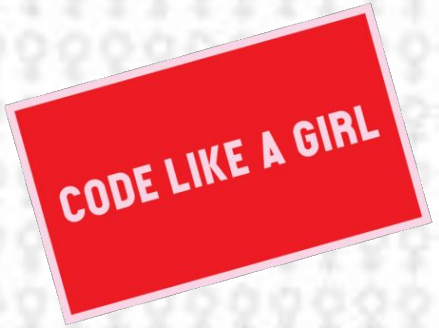
STEM2D

Go girls!

Code like a girl!

EXERCÍCIO

- Faça um programa que leia um nome de usuário e a sua senha e não aceite a senha igual ao nome do usuário, mostrando uma mensagem de erro e voltando a pedir as informações.



EXERCÍCIO

- Os números primos possuem várias aplicações dentro da Computação, por exemplo na Criptografia. Um número primo é aquele que é divisível apenas por um e por ele mesmo. Faça um programa que peça um número inteiro e determine se ele é ou não um número primo.

GIRL CODE

FUNÇÕES

- No universo da programação, funções, também chamadas de **sub-rotinas** (ou sub-programas), são **trechos de um algoritmo** (sequência de instruções) que encerram em si um pedaço da solução de um problema maior.

- Para atingirem determinados objetivos, tais funções podem ou não necessitar de **entradas** (também chamadas de **parâmetros**).

FUNÇÕES

- Os parâmetros especificam qual informação você deve providenciar para que uma função possa ser utilizada.

- A sintaxe de uma função é a seguinte:

```
def nomeDaFuncao (parametro1, parametro2, ...):
    fazer alguma coisa
```

- E para chamá-la no código, basta utilizar a seguinte sintaxe:

```
nomeDaFuncao (parametro1, parametro2, ...)
```

FUNÇÕES

- As funções podem ou não retornar valores como resultados.

```
def funcaoExemplo (x):
    if x>10:
        y = x/10
    else:
        y = x
    return y
```

```
print(funcaoExemplo(10))
```

```
def codeLikeAGirl():
    print("Let's code!")
```

```
codeLikeAGirl()
```

STEM2D

Go girls!

Code like a girl!

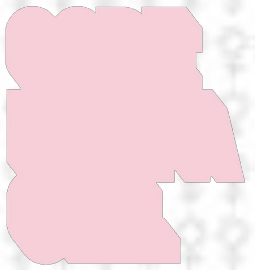
EXERCÍCIO

- Faça um programa, com uma função que necessite de três argumentos, e que forneça a soma desses três argumentos.

Dica: lembre-se da sintaxe de declaração e chamada de funções em *Python*:

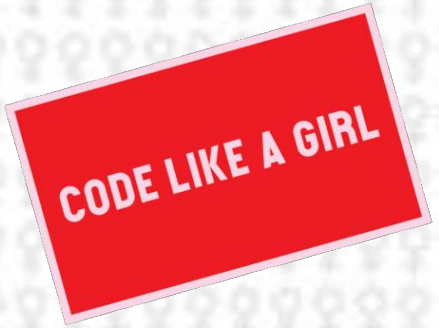
```
def nomeDaFuncao (parametro1, parametro2, ...):
    fazer alguma coisa
```

```
nomeDaFuncao (parametro1, parametro2, ...)
```



EXERCÍCIO

- Escreva uma função que recebe os catetos de um triângulo retângulo e retorna o comprimento da sua hipotenusa.



FUNÇÕES

- Para quê devemos criar funções, se podemos escrever livremente comandos ao longo do código para atingir os mesmos objetivos?

  Suponha que você queira imprimir o resultado da **funcaoExemplo** para os valores de entrada: 8, 9, 10, 20, 21 e 22.

```
def funcaoExemplo (x):
  if x>10:
      y = x/10
  else:
      y = x
  return y
```

FUNÇÕES

- Sem declarar uma função, provavelmente você escreveria:

- Note que este código é longo, repetitivo e difícil de entender.

```
i = 8
while (i <= 10):
    if i>10:
        y = i/10
    else:
        y = i
        print(y)
    i=i+1

i = 20
while (i <= 22):
    if i>10:
        y = i/10
    else:
        y = i
        print(y)
    i=i+1
```

FUNÇÕES

- Mas se utilizarmos uma chamada da função **funcaoExemplo** dentro dos laços de repetição, poderemos fazer o mesmo trabalho com muito menos esforço:

```
i = 8
while (i <= 10):
    print(funcaoExemplo (i))
    i=i+1
i = 20
while (i <= 22):
    print(funcaoExemplo (i))
    i=i+1
```

VETORES

- Um **vetor** é uma variável que nos permite guardar diversos valores de diversos **tipos diferentes**.
- Ele possui posições consecutivas enumeradas (**índices**) e por meio delas podemos acessar o conteúdo no vetor.

5	3,14159265	"STEM"	904	"casa"
v[0]	v[1]	v[2]	v[3]	v[4]

- Para declarar um vetor utilizamos a seguinte sintaxe:

nomeDoVetor = **[**elemento1, elemento2, ...**]**

VETORES

- A construção de um vetor pode ser feita:
  a) fornecendo os seus elementos;

```
vet = [12, 34, 23, 7]
notas = [8.3 ,7.5 , 10]
nome = ["fernanda", "luiza", "marcela"]
lista = [12, 'livro', 7.332, 'casa']
```

  b) utilizando a função **append()**;

```
vet = [12, 34, 23, 7]
vet.append(10)
```

```
vet = [] # define um vetor vazio
for i in range(5):
    vet.append(i)
```

VETORES

- Para **manipularmos vetores** utilizamos uma técnica denominada **indexação**, que consiste em informar o índice da posição que desejamos acessar entre colchetes:

```
lista = [12, 'livro', 7.332, 'casa']
print(lista[0])
print(lista[3])
```

12
'casa'

VETORES

- Para **alterar o valor armazenado em uma posição** do vetor basta acessar o índice que deseja alterar e atribuir outro valor:

```
lista = [12, 'livro', 7.332, 'casa']
lista[1] = 3
print(lista)
```

[12, 3, 7.332, 'casa']

VETORES

- Podemos realizar **operações aritméticas** com os elementos de um vetor:

```
lista = [12, 'livro', 7.332, 'casa']
print(2.5 + lista[2])
```

9.832

VETORES

- Podemos retirar um elemento do vetor por meio da função **del()**, inserir um elemento em uma posição utilizando a função **insert()**, descobrir o tamanho do vetor com a função **len()** e, ainda, limpar o vetor com a função **clear()**.

```
A = ["a", "b", "c" ]
del(A[0])
print(A)
A.insert(0, "y")
print(A)
print(len(A))
A.clear()
print(A)
```

```
["b", "c"]

["y", "b", "c"]

3

[ ]
```

VETORES

- Além de acessar uma única posição do vetor, podemos **acessar um intervalo de posições** dele, descrevendo o intervalo de índices a ser acessado dentro da indexação:

```
lista = [1, 2, 3, 4, 5, 6, 7, 8]
print(lista[1:4])
```

```
[[2, 3, 4]
```

- **Cuidado**: o índice informado após os dois pontos corresponde a um índice após o fim do intervalo (o elemento com esse índice não fará parte do intervalo).

STEM2D

Go girls!

Code like a girl!

EXERCÍCIO

- Faça um algoritmo que solicite ao usuário números e os armazene em um vetor de 20 posições. Crie, então, uma função que recebe o vetor preenchido e substitua todas as ocorrências de valores negativos por zero, as ocorrências de valores menores do que 10 por 1 e as demais ocorrências por 2.

EXERCÍCIO

- Faça uma função que receba duas listas e retorne **True** se são iguais ou **False**, caso contrário. Duas listas são iguais se possuem os mesmos valores e na mesma ordem.

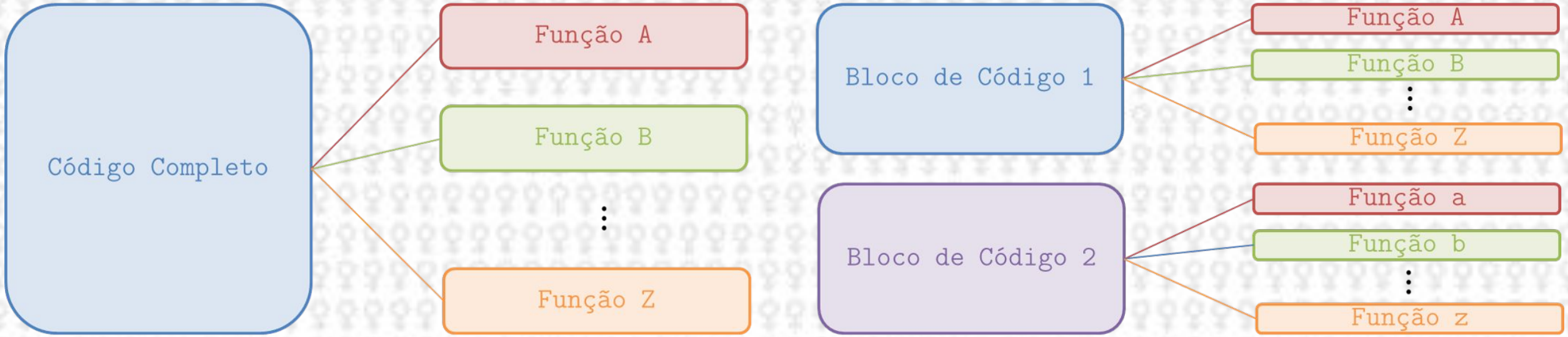
GIRL CODE

INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS (POO)

- **POO** diz respeito a um **padrão de desenvolvimento** que é seguido por muitas linguagens.

- Diferentemente da **programação estruturada**, na qual criamos funções (ou procedimentos) que podem ser aplicadas em todo nosso código, em POO declaramos funções que só podem ser aplicadas a blocos específicos.

INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS (POO)



OBJETOS

- Para programarmos utilizando POO precisamos primeiramente abstrair a realidade para o nosso código dividindo-o em **objetos** que possuem **atributos** e **métodos**. **Atributos** são variáveis exclusivas do objeto. **Métodos** são funções exclusivas do objeto.

```
protagonista:
  #Atributos
    protagonista.imagem = personagem.png
    protagonista.tamanho = (30,120)
    protagonista.velocidade = (15,0)
    protagonista.posicao = (0,500)
  #Métodos
    protagonista.pular()
    protagonista.atirar()
```

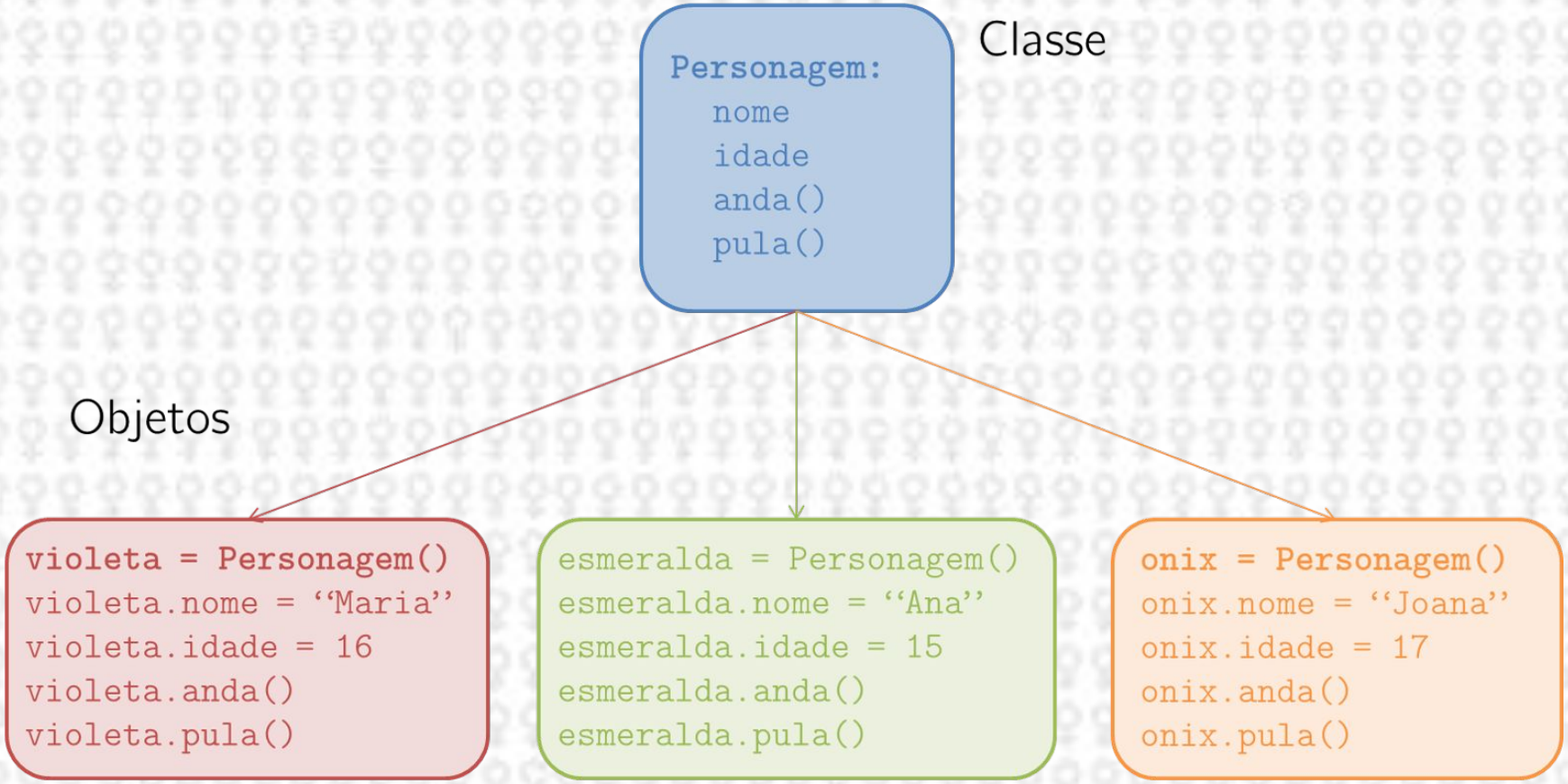
OBJETOS

- Em *Python* usamos a notação **objeto.atributo** para acessar seus atributos e **objeto.metodo(parâmetros)** para acessar seus métodos.

```
print(protagonista.tamanho) #imprime (30, 120)
protagonista.pular() # realiza as acoes descritas em pular
```

CLASSES

- **Classes** funcionam como moldes a partir do qual é possível criar **objetos** com o mesmo formato.



CLASSES

- Em *Python* implementamos **classes** utilizando a seguinte sintaxe:

```
class NomeDaClasse():
    def __init__(self, parametrosDeInicializacao):
        #atributos
        self.atributo1 = 1
        self.atributo2 = 2
        [...]
    #metodos
    def primeiroMetodo(parametro):
        [...]
    def segundoMetodo(parametro):
        [...]
```

CLASSES

```
class Jogador():
    def __init__(self, game):
        self.x = X_CHAO
        self.y = Y_CHAO
        self.carregarImagemPersonagem(game)
        self.largura = self.image.get_width()
        self.altura = self.image.get_height()

    # carrega a imagem do personagem de acordo com a escolha do usuario
    def carregarImagemPersonagem(self, game):
        self.imagem = pygame.image.load(os.path.join('Imagens', 'personagem_principal_FEC_1.png'))

    # esse metodo faz carregar o tiro do jogador na tela
    def atirar(self, game):
        game.tiros.append(Tiro(self.x + self.largura, 0.93*self.pos.y, 'A', -10 - game.dvel))
```

CLASSES

- A palavra **self** presente é usada para se referir a atributos e métodos da própria classe.

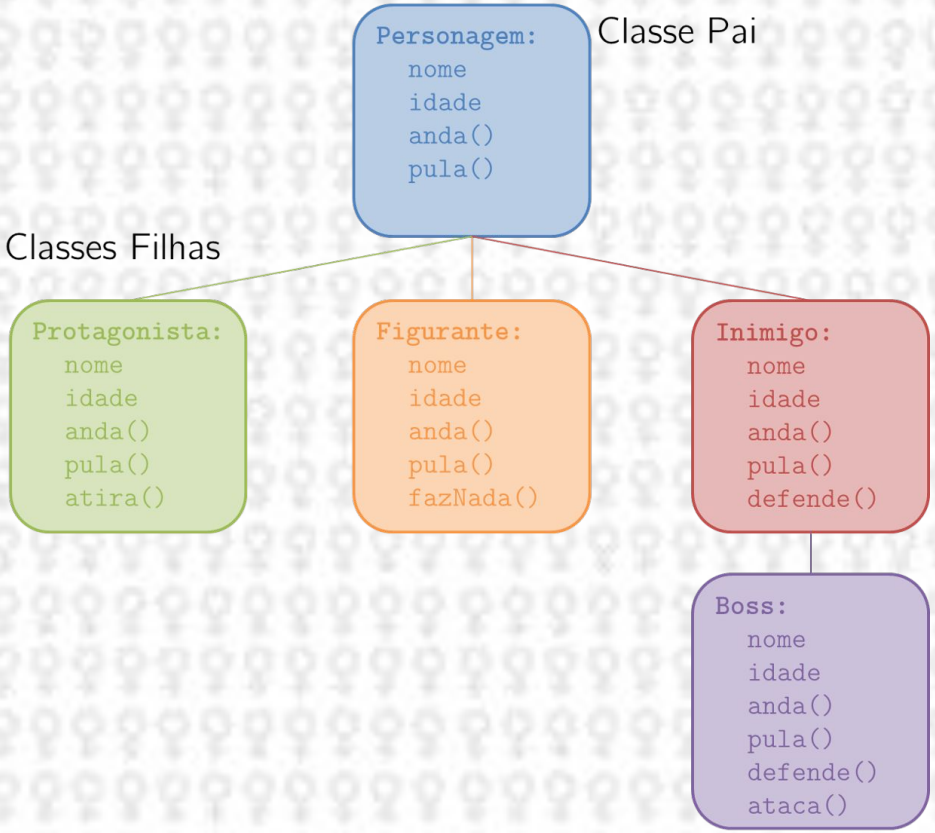
- Para declarar o objeto de uma classe devemos utilizar a seguinte sintaxe:

```
objeto = NomeDaClasse(parametro1, parametro2, ...)
```

```
protagonista = Jogador(imagem)
```

HERANÇA

- Herança é uma característica existente em POO na qual a partir de uma classe podemos criar **classes derivadas**.
- A ideia da herança é que classes-filhas possuirão os mesmos atributos e métodos de suas classes-pai, além de poderem possuir outros atributos e métodos, além daqueles que herdaram.



HERANÇA

- Em *Python* implementamos **herança** utilizando a seguinte sintaxe:

```
from NomeDaClassePai import *
class NomeDaClasseFilha(NomeDaClassePai):
    def __init__(self, parametrosDeInicializacao):
        # constroi a classe pai
        super().__init__() # opcional
        [...]
```

HERANÇA

```
from Tela import *

class TelaDeInicio(Tela):
    def __init__(self, game):
        super().__init__()
        self.name = 'Tela de Inicio'
        self.botaoPlay = pygame.image.load(os.path.join('Imagens', 'play1.png'))
```

STEM2D

Go girls!

Code like a girl!

EXERCÍCIO

- Crie uma classe para implementar uma conta corrente. A classe deve possuir os seguintes atributos: **numeroDaConta**, **nomeDoCorrentista** e **saldo**. Os métodos são os seguintes: **alterarNome**, **alterarNumero**, **deposito**, **saque**, **consultarSaldo**. No construtor, **saldo** possui valor padrão zero e os demais atributos estão vazios.


STEM2D
Go girls!
Muito obrigada!