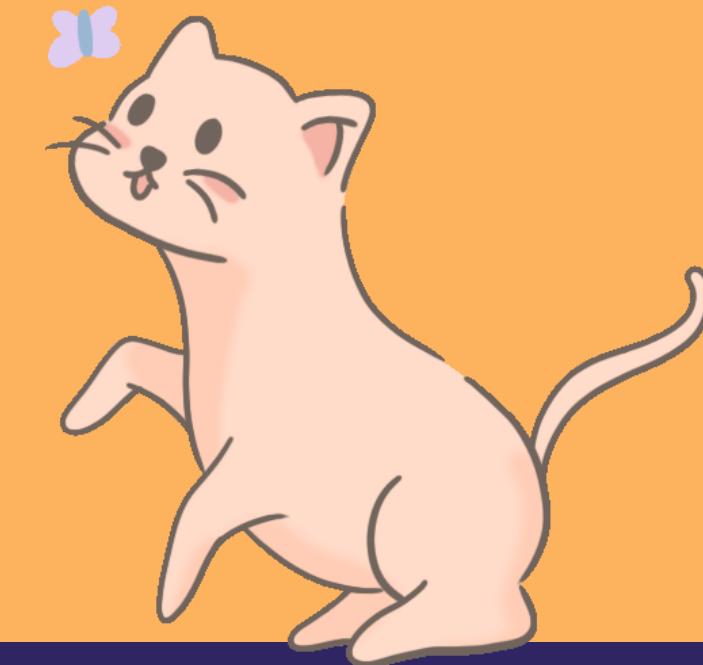


Grupo Katie apresenta:

Minicurso de Python - Poo



Katie's Month



KATIE

Saindo do buraco negro e impulsionando meninas para a computação

O Instituto de Computação (IC) sempre teve poucas mulheres em seu corpo discente, sendo isso um reflexo de como é o mercado de trabalho na área de computação. Assim, se fez necessário a fundação de um projeto que acolhesse e incentivasse mulheres na área.

O nome Katie é uma homenagem à cientista [Katie Bouman](#), que liderou a equipe de cientistas responsável pelo algoritmo que possibilitou a primeira visualização de um buraco negro em nossa galáxia.





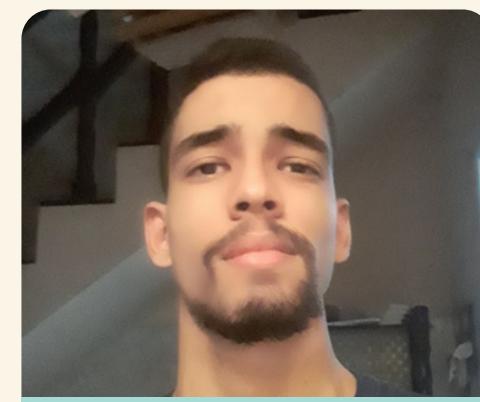
Quem somos nós?



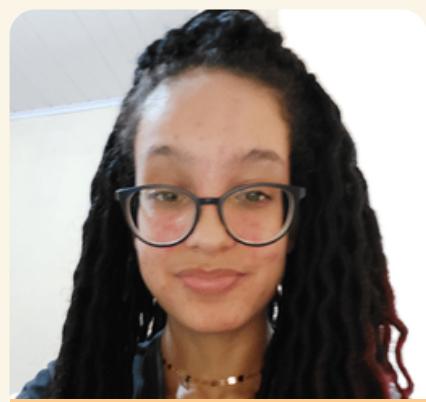
Rebeca Brandão



Evvlyn dos Santos



Mateus Patriota



Ullyanne Patriota



Hiago Cavalcante

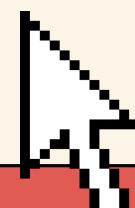
Instrutores

Monitores



Tô com dúvida, e agora?

Não se preocupe! Criamos um grupo no Whatsapp pra você se sentir à vontade, tirar suas dúvidas e compartilharmos conhecimento!





Tô com dúvida, e agora?

* Não se preocupe! Criamos um grupo do **Whatsapp** pra você se sentir à vontade, tirar suas dúvidas e compartilharmos conhecimento!

* Usaremos o **Classroom** para postar links úteis e avisos. Você pode postar sua dúvida lá, mas o grupo do Whatsapp existe justamente para isso.

* Não se sintam incapacitadas(os) por não entender alguma coisa. Ninguém achará isso de você. Estamos aqui para te ajudar :)

Criamos um material complementar bem legal para você utilizar em seus estudos, ele será postado no Classroom.



O que você verá durante o curso

Dia 1 (25/07)

1

Classes, métodos,
atributos e objetos

2

Construtor:
inicializando o
objeto

3

Herança

4

Polimorfismo

5

Composição

6

Encapsulamento



Google Colab

Ferramenta



The logo consists of the word "colab" in a bold, sans-serif font. The letters are primarily orange, with the "c" having a yellow-to-orange gradient. The "o" is solid yellow. The "l" and "a" are solid orange. The "b" has a yellow-to-orange gradient.

{kat:e}



Chat



Trash



Projeto

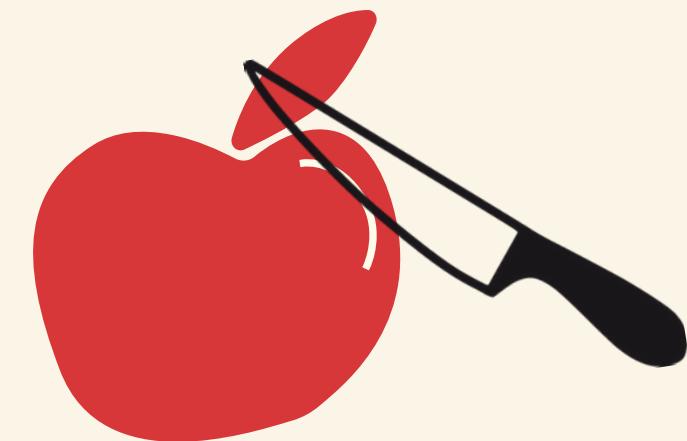
Após este minicurso, você estará totalmente apto para desenvolver...





O que é programação orientada a objetos?

Diferentemente de C ou Fortran, que têm como paradigma a programação procedural, Python toma para si um paradigma de programação criado em 1960: programação orientada a objetos. POO gira completamente em torno da **criação e manipulação de objetos!**





O que é programação orientada a objetos?

E por que objeto?

Imagine que você, cansado de tanto programar, decidiu estudar a natureza! Percebeu que os humanos e as frutas têm muitas características diferentes e a maior delas é que toda fruta é um vegetal e todo humano é um animal. Como bom pesquisador que és, começou a ver **padrões nas coisas** e notou que todo humano respira e que toda fruta amadurece.



O que é programação orientada a objetos?

E por que objeto?

Pensando nessas diferenças e na facilidade de categorizar os seres da natureza, a orientação a objetos foi criada.

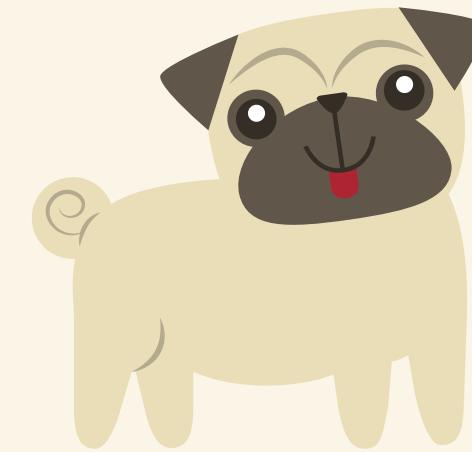
Assim, os programadores viram que a programação procedural não conseguia adaptar-se à vida real, pois o código só poderia ser alterado por meio de funções. A partir disso, o foco passou a ser o objeto.



Classes, métodos, atributos e objetos

Conceito

Classe Cachorro:



Objeto: Ralph

Atributo raça: Pug



Objeto: Paçoca

Atributo raça: Salsicha

Método: sentar



Atributo: Pelagem mista
(múltiplas cores)



Classes, métodos, atributos e objetos

Conceito

Dessa forma, podemos dizer que as classes têm como objetivo definir um determinado grupo de dados de um único tipo ao especificar os elementos de dados que um objeto contém (atributos) e o comportamento pertencente ao objeto (métodos)



Construtor

Inicializando um objeto

```
...  
class Cachorro  
  
class Cachorro:  
    def __init__():  
        pass
```

class: definir uma classe
def __init__: definir construtor



Definindo uma classe

O parâmetro `self` e a criação de atributos

```
...  
class Cachorro  
  
class Cachorro:  
    def __init__(self, raca):  
        self.raca = raca
```

self: representação da instância.

possibilidade de ter atributos distintos e/ou únicos em cada instância.



Definindo uma classe

Métodos

```
...  
class Cachorro  
  
class Cachorro:  
    def __init__(self, raca):  
        self.raca = raca  
    def late(self):  
        print("Au au!")
```

def: criação do método late()



Objeto

```
• • •  
class Cachorro  
  
class Cachorro:  
    def __init__(self, raca):  
        self.raca = raca  
    def late(self):  
        print("Au au!")  
  
Toto = Cachorro("dálmata")
```

**Instanciando nosso primeiro
doguinho, digo, objeto: Totó**



Interação com o objeto

```
class Cachorro

class Cachorro:
    def __init__(self, raca):
        self.raca = raca
    def late(self):
        print("Au au!")

Toto = Cachorro("dálmata")

print(Toto.raca) #dálmata

Toto.late() #Au Au!"
```



Atributos de classe

```
• • •  
class Cachorro  
  
class Cachorro:  
    quantidade = 0  
    cachorros = []  
  
    def __init__(self, raca):  
        self.raca = raca  
    def late(self):  
        print("Au au!")
```

Quantidade = atributo de classe* para
armazenar quantidade de objetos

*Atributo de classe é utilizado para armazenar
informações relevantes à classe como, por exemplo, a
quantidade de objetos de Cachorro que instanciamos.



Mais sobre métodos

```
...  
class Cachorro  
  
class Cachorro:  
    quantidade = 0  
    cachorros = []  
  
    def __init__(self, raca, idade):  
        self.raca = raca  
        self.idade = idade  
  
    def late(self):  
        print("Au au!")
```



```
def mudaIdade(self):  
    self.idade = self.idade + 1 #Incrementação  
  
Toto = Cachorro("dálmata", 2)  
print(Toto.idade) #2  
Toto.mudaIdade()  
print(Toto.idade) #3
```

mudaIdade: método capaz de alterar
atributo "idade".

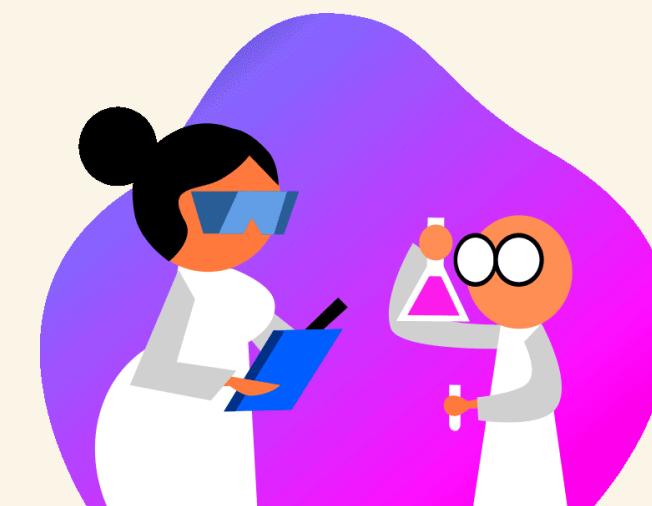


Classes default

O que vimos até agora foi a criação de classes personalizadas, onde definimos as características que uma instância de uma classe deve ter e seus métodos.

Mas você sabia que existem classes definidas pelo próprio Python? São elas `str`, `int`, `float`, `list` etc.

```
str(string) = "Hello World"  
int(integer) = 1  
float = 1.54  
list = [1, 2, 3]
```





Até 27/07!

Nos vemos na próxima aula!

• ° Dúvidas? • °

{kat:e}



Chat



Trash



O que já vimos até agora?

Dia 1 (25/07)

1

Classes, métodos,
atributos e objetos

2

Construtor:
inicializando o
objeto



O que veremos hoje?

Dia 2 (27/07)

3 Herança

4 Polimorfismo



Herança

Todo filho tem algumas características dos pais e chamamos isso de herança genética! Inspirando-se neste conceito de herança, foi criado o conceito de herança na programação e então implementado em Python.





Herança

Sabemos que tanto o gato quanto o humano são seres vivos e portanto nascem e amadurecem. Observe a classe **SerVivo**:

```
class SerVivo:  
    def __init__(self):  
        self.tipo = 'Ser Vivo'  
        self.idade = 0  
  
    def nascer(self, nome):  
        self.nome = nome  
        print(f'Eu sou do tipo {self.tipo}. Meu nome é {self.nome} e acabei de nascer!')  
  
    def amadurecer(self):  
        self.idade += 1  
        print(f'Eu, {self.nome}, amadureci! Minha idade é {self.idade}.')
```



Herança

Então as classes **Humano** e **Gato** podem herdar as propriedades de **SerVivo**!

```
class Humano(SerVivo):
    def __init__(self):
        super().__init__()
        self.tipo = 'Humano'

    def sorrir(self):
        print('(: Eu sou humano e sei rir :)')

class Gato(SerVivo):
    def __init__(self):
        super().__init__()
        self.tipo = 'Gato'

    def miar(self):
        print('Miau! Sou um gatinho e sei miar +^-^+')
```



Herança

Podemos mandar o objeto Humano nascer? Claro!

`humano.nascer('Wilson')`

```
class Humano(SerVivo):
    def __init__(self):
        super().__init__()
        self.tipo = 'Humano'

    def sorrir(self):
        print('(: Eu sou humano e sei rir :)')
```



OBS: o método `nascer()` está em `SerVivo!`



Herança

E se pedirmos ao objeto Gato para sorrir ou ao objeto Humano para miar?

erro de atribuição!

```
class Humano(SerVivo):
    def __init__(self):
        super().__init__()
        self.tipo = 'Humano'

    def sorrir(self):
        print('(: Eu sou humano e sei rir :)')

class Gato(SerVivo):
    def __init__(self):
        super().__init__()
        self.tipo = 'Gato'

    def miar(self):
        print('Miau! Sou um gatinho e sei miar +^-^+')
```





Herança

Vamos agora criar uma nova classe filha de **SerVivo** (ou apenas subclasse): **Cachorro**.

```
...  
class Cachorro  
  
class Cachorro(SerVivo):  
    def __init__(self):  
        self.tipo = 'Cachorro'
```

Você consegue identificar o que está faltando?



Herança

Perceba que diferentemente das outras classes, não chamamos `super().__init__()` em seu construtor. Por conta disso, o interpretador não inicia a nova classe com o construtor da classe mãe. Dessa maneira, nosso objeto Cachorro não herdará nunca o atributo `idade`, somente os métodos.





Herança

Se quisermos, além de usar os atributos da classe mãe, adicionar novos atributos, é somente preciso adicionar ao `__init__` da classe filha.

```
...  
class Criança  
  
class Criança(Humano):  
    def __init__(self):  
        super().__init__()  
        self.dentes = 0  
        self.inocencia = True
```

`Criança` vai herdar atributos e métodos tanto de `Humano` quanto de `SerVivo`.





Herança

E se quisermos um `__init__` na classe filha que receba pelo menos um input do usuário diferente da classe mãe?

```
...  
class CriancaBrincalhona  
  
class CriancaBrincalhona(Crianca):  
    def __init__(self, brinquedoFavorito):  
        super().__init__() #perceba que não recebe brinquedoFavorito  
        self.brinquedoFavorito = brinquedoFavorito
```

Ela recebe `brinquedoFavorito` diferentemente da sua classe mãe



Herança

Em adição, a herança nos possibilita o uso de **métodos sobrescritos**: se um filho tem um método com o mesmo nome de um método do pai, o interpretador executa o do filho.

```
class GatoBranco(Gato):
    def __init__(self):
        super().__init__()

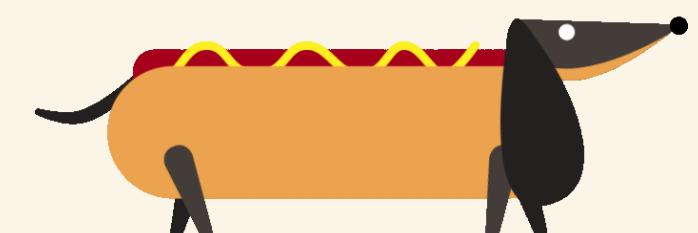
    def miar(self):
        print('Eu sou um gato branco e sou rei do mundo! feed me /'-o-'\\J')
```



Polimorfismo

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas.

Cuidado! Polimorfismo não quer dizer que o objeto fica se transformando. Muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele.

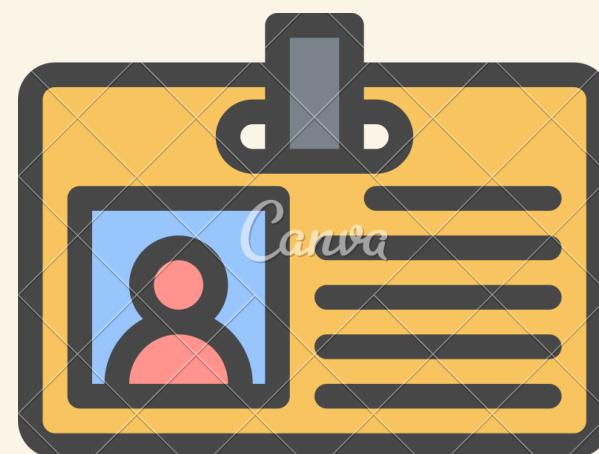




Polimorfismo

Exemplo na vida real

Ao ir em alguns estabelecimentos, há sempre uma repartição restrita a funcionários, permitindo a entrada apenas daqueles que apresentarem seu crachá.





Polimorfismo

Exemplo na vida real

Convertendo isso para código, teríamos uma classe mãe **Funcionário** e classes filhas **Gerente** e **Atendente**, visto que esses últimos, apesar de serem funcionários, possuem funções e responsabilidades diferentes em um banco.





Polimorfismo

Exemplo na vida real



E como permitir que apenas funcionários façam uma determinada ação, mas ao mesmo tempo torná-las individuais, ou seja, que cada tipo de funcionário trabalhe, mas trabalhe de determinada maneira, conforme sua função?



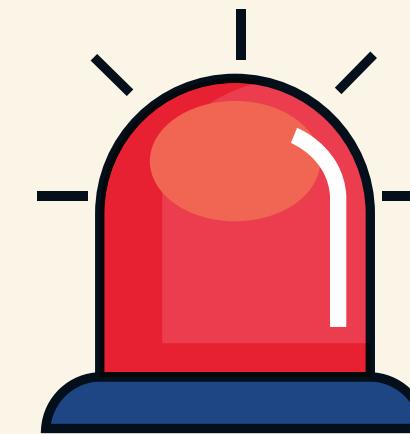
OPÇÃO 1 - vários ifs



Realizando uma ação

```
if funcionario.tipo == "Gerente":  
    print("...Gerenciando")  
elif funcionario.tipo == "Atendente":  
    print("...Atendendo")
```

ALERTA DE CODE SMELL !





OPÇÃO 2 - Polimorfismo!

Ainda bem que temos o **polimorfismo** para salvar o dia!
Não importa como nos referenciamos a um objeto, o método que
será invocado é sempre o que pertence a ele mesmo.





Funcionários

```
class Funcionario():
    def __init__(self, nome):
        self.nome = nome

    def Work():
        pass

class Gerente(Funcionario):
    def __init__(self, nome):
        super().__init__(nome)
        self.tipo = "Gerente"

    def Work():
        print("...Gerenciando")

class Atendente(Funcionario):
    def __init__(self, nome):
        super().__init__(nome)
        self.tipo = "Atendente"

    def Work():
        print("...Atendendo")
```

Assim, chegamos ao nosso objetivo com uma classe mãe **Funcionário** e classes filhas **Gerente** e **Atendente**.





Resultado Final - Polimorfismo

```
...  
Work  
  
Mariana = Gerente("Mariana")  
Rita = Atendente("Rita")  
  
funcionarios = [Mariana, Rita]  
  
for funcionario in funcionarios:  
    print(funcionario.tipo) #Gerente #Atendente  
    funcionario.Work() #...Gerenciando #...Atendendo
```

Legal, né?

Dessa forma, o código está enxuto. A única coisa que precisamos entender, com esse trecho de código, é que queremos que os funcionários trabalhem.





Até 08/10!

Nos vemos na próxima - e última - aula!

• ° dúvidas? • °

{kat:e}



Chat



Trash



O que já vimos até agora?

Dia 1 (04/10)

1

Classes, métodos,
atributos e objetos

2

Construtor:
inicializando o
objeto

3

Herança

4

Polimorfismo



O que veremos hoje?

Dia 3 (08/10)

5 Composição

6 Encapsulamento



Composição



Atenção! Este é um dos conceitos fundamentais de POO!

A **composição** torna possível que uma classe tenha como atributos um ou mais objetos de outras classes e permite uma associação entre objetos.

Composição \neq Herança



Composição

```
class Conta

class Conta:
    def __init__(self, numero, cliente, saldo):
        self.numero = numero
        self.saldo = saldo
        self.limite = limite

    def transfere(self, destino, valor):
        print(f"Transferindo R${valor} para {destino}")
```

Classe Conta criada para um sistema de *internet banking*.



Composição

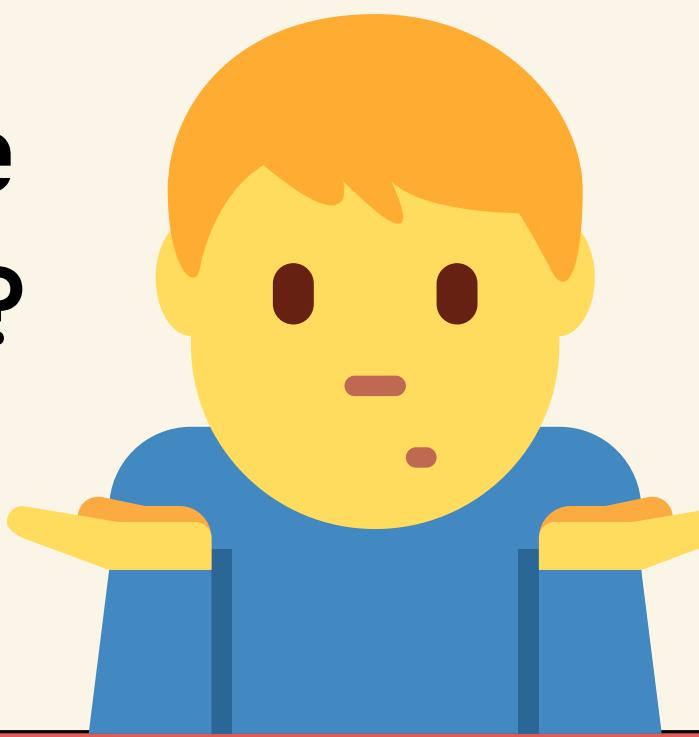
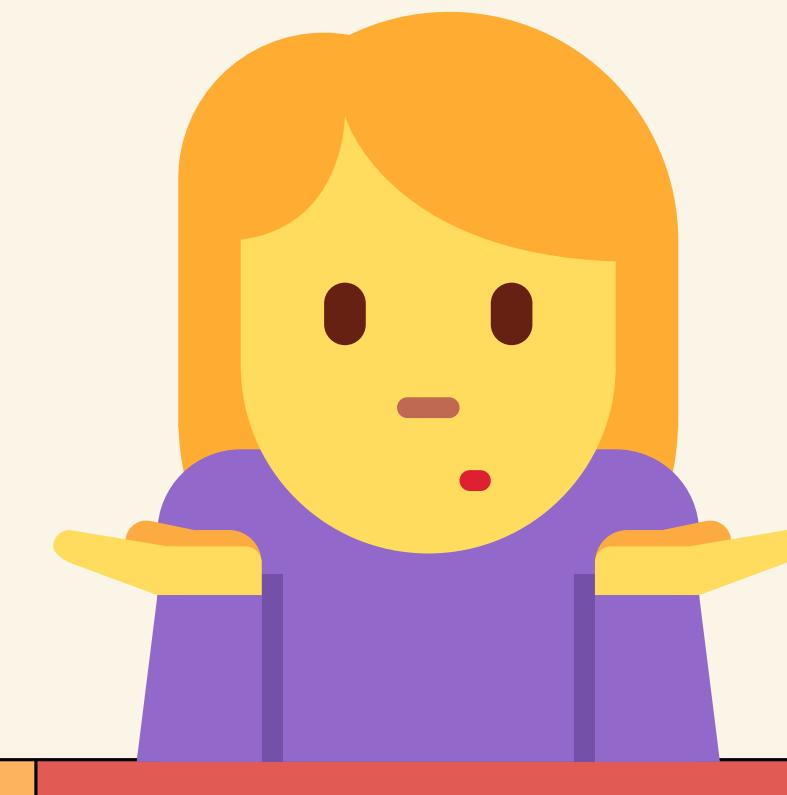
Como toda conta possui histórico de transações, foi criada também a Classe Histórico.

```
class Historico:  
    def __init__(self):  
        self.transacoes = []  
        self.dataAbertura = datetime.datetime.today()  
  
    def imprime(self):  
        print(f"data de abertura: {self.dataAbertura}")  
        print("Transações: ")  
  
        for transacao in self.transacoes:  
            print(transacao)
```



Composição

Será que conseguimos
aproveitar algum
método/atributo da classe
Conta na classe **Histórico**?





Composição

A resposta é **não!**

Um objeto instanciado de **Conta** é completamente diferente de um objeto de **Histórico**. Eles possuem os próprios métodos e atributos, não conseguimos estabelecer uma relação de herança.





Composição

E é nesse momento que usaremos a composição! Ela fará com que cada objeto de **Conta** possua um **histórico**. Assim, criamos um atributo em **Conta** que recebe um objeto de **Histórico**.

```
class Conta:  
    def __init__(self, numero, cliente, saldo):  
        self.numero = numero  
        self.saldo = saldo  
        self.limite = limite  
        self.historico = Historico()
```



Encapsulamento

Separar em partes

Objetivo: esconder de todos os usuários de uma classe algumas informações que não são necessárias ao uso da classe.



Importância: permite que a implementação das funcionalidades de uma classe sejam alteradas sem que o código que a usa precise mudar.





Encapsulamento

```
class Conta:  
    def __init__(self, saldo):  
        self._saldo = saldo  
  
    def get_saldo(self):  
        return self._saldo  
  
    def set_saldo(self, saldo):  
        if(saldo < 0):  
            print("O Saldo não pode ser negativo")  
        else:  
            self._saldo = saldo
```



Problema: Como nossa classe é projetada com atributos públicos, podemos quebrar a interface e simplesmente acessar e alterar os atributos que deveriam ser protegidos.



Encapsulamento

💡 Solução: Manter os atributos protegidos e decorar os métodos com um decorador que se chama *property*.

```
class Conta:
    def __init__(self, saldo=0.0):
        self._saldo = saldo

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

    @saldo.setter
    def saldo(self, saldo):
        if(saldo < 0):
            print("O Saldo não pode ser negativo")
        else:
            self._saldo = saldo
```



Encapsulamento

Mais sobre o *property*:

Possui métodos extras, como um getter e um setter.

Quando aplicado em um objeto, nos retorna uma cópia dele com as suas funcionalidades. Logo, somos capazes de chamar nossos métodos sem os parênteses e como se fossem atributos públicos.

decorator property

```
@property  
def func(self):  
    return self._func
```

é equivalente a:

```
def func(self):  
    return self._func  
  
func = property(func)
```



Encapsulamento

 **IMPORTANTE:** Só devemos usar encapsulamento quando ele for realmente necessário, pois geralmente não são todos os atributos de uma classe que precisam desse recurso.



Antes de finalizar...

Sugestão para o projeto



Printando um objeto

```
class Gato(SerVivo):
    def __init__(self, nome, nacionalidade):
        super().__init__(nome, nacionalidade)
        self.tipo = 'Gato'

gato1 = Gato("Garfield", "estadunidense")
```

O que acontece se
printarmos um
objeto?



Sugestão para o projeto



Printando um objeto

```
class Gato(ServVivo):
    def __init__(self, nome, nacionalidade):
        super().__init__(nome, nacionalidade)
        self.tipo = 'Gato'

gato1 = Gato("Garfield", "estadunidense")
print(gato1) #<__main__.Gato object at 0x7f8780ab8cd0>
```

O que acontece se
printarmos um
objeto?



Sugestão para o projeto

• • •

Printando um objeto

```
class Gato(Servivo):
    #__init__
    def __str__(self):
        return f"""
        -----
        Sou um {self.tipo} {self.nacionalidade}!
        Mas você pode me chamar de {self.nome}
        -----
        """
        gato1 = Gato("Garfield", "estadunidense")
        print(gato1) #vai chamar o __str__
```

método `__str__`:
útil para printar as
informações do seu
objeto



Projeto

Após a data de entrega do projeto, iremos disponibilizar o código de uma das maneiras de implementá-lo

Lembrando que o critério de emissão do certificado é a entrega do projeto

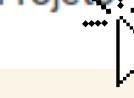




That's all, folks!

Finalizamos o nosso conteúdo por aqui! Esperamos que você tenha aprendido bastante e que tenha sido uma experiência bem legal :))

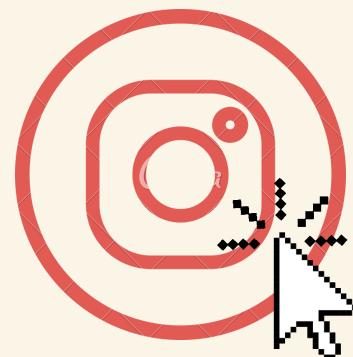
Chegou a hora de fazer o **projeto**! A atividade está no Classroom e pode ser feita até o dia **18/10**. Fique à vontade para consultar os monitores!

 Slides - 04/10	Item postado em Ontem
 Apostila do Curso de Python: Orientação a ...	Item postado em Ontem
 Projeto 	Data de entrega: 18 de out.

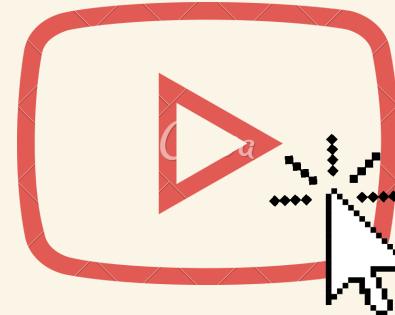


Onde nos encontrar?

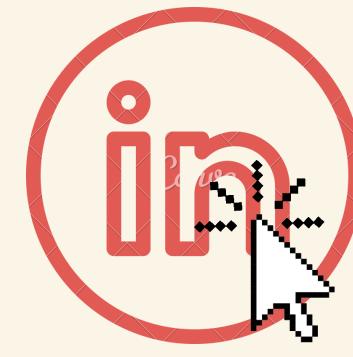
Quer ficar sempre por dentro das novidades? Nos acompanhe em nossas redes sociais!



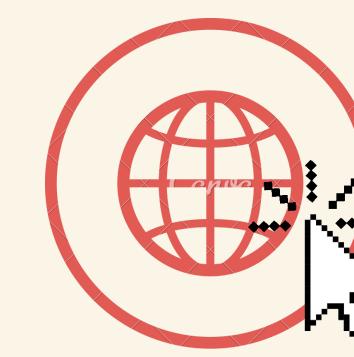
@katie.ufal



KATIE - Projeto de
Extensão IC/UFAL



bit.ly/LinkedinKatie



bit.ly/SiteKatie



File Edit View Label

10:00 AM



Chat



Trash