código.py

GALÁXIA 6

Python Orientado a Objeto

Galáxia 6

Mundo 1

1.1 Orientação a objeto - definições

Mundo 2

2.1. Pré-configurações

2.2 Criando uma class

2.2.1. Sintaxe da class

2.2.2. Método __init__:

2.2.3. Parâmetro self:

2.3. __name__ == "__main__"

Mundo 3

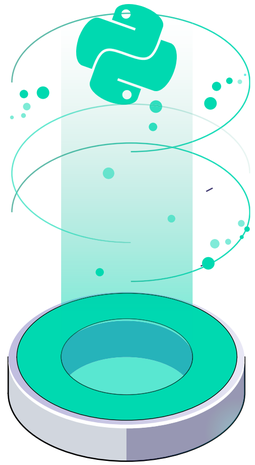
3.1. Criação de um método de instância.

Mundo 4

4.1. Criação de um método de classe.

Mundo 5

5.1. Criação de um métodos estáticos



Mundo 6

6.1. Criação de um getter

6.2. Criação de um setter

Mundo 7

7.1. Variáveis públicas

7.2. Variáveis protegidas

7.3. Variáveis privadas

Mundo 8

8.1. Criando a class banco_de_dados

8.2. Utilizando a class banco_de_dados

Mundo 9

9.1. Agregação

9.2. Aplicação agregação

9.2.1. class Carteira_investimento

9.2.2. class Acoes

9.3. Exemplo

Mundo 10

10.1. Composição

10.2. Aplicação composição

10.2.1. class Empresa

10.2.2. class Endereco

10.3. Exemplo

Mundo 11

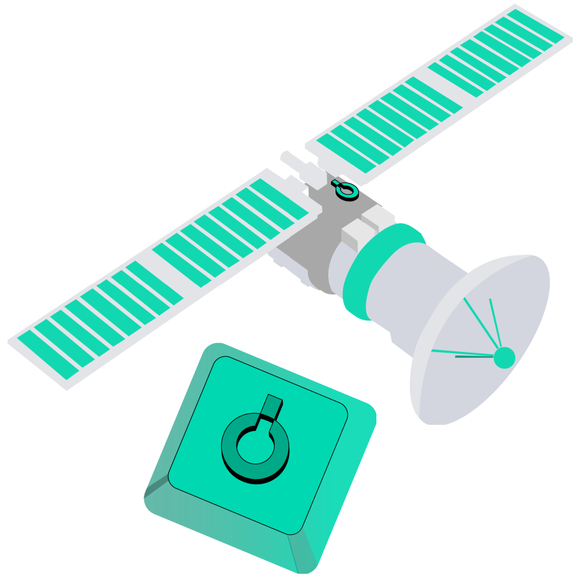
11.1. Herança

11.2. Aplicação composição

11.2.1. class Empresa

11.2.2. class Empresa

11.3. Exemplo



Introdução

Olá, seja bem vindo à Galáxia 6! Neste módulo iremos abordar tudo o que você precisa saber sobre orientação a objeto. O que é, seus principais usos, como você pode utilizar e o que são paradigmas da programação. Diferente dos módulos anteriores, este não é sobre bibliotecas de códigos aberto. Este módulo vai te ensinar como programar igual profissional, te ensinar alguns conceitos que NINGUÉM aborda na internet ou em qualquer outro curso. Então vamos adiante que temos muita coisa pela frente

Mundo 1

1.1. Orientação a objeto - definições

Antes de a gente abordar a orientação a objeto, precisamos entender alguns conceitos e um desses conceitos é o paradigma da programação.

Os paradigmas da programação são diferentes formas de estruturar seus programas. Existem 3 principais paradigmas:

1. Orientação a objeto - Que se preocupa com a criação e manipulação de objetos

2. Programação funcional - Que se preocupa com a aplicação de funções matemáticas para os dados.

3. Programação imperativa - Que se preocupa em como a aplicação deve ser construída.

A orientação a objeto é o paradigma mais popular por ter grande facilidade na hora da manutenção e organização das aplicações.

Na orientação a objeto existem as classes e os objetos. Pense em uma analogia com carros:

Imagine que você decidiu comprar um carro e, neste carro, existem as seguintes características: um motor, cor azul, quatro rodas e câmbio automático.

Além dessas características, o carro também faz ações como: acelerar, desacelerar, tocar música, buzinar e acender farol.

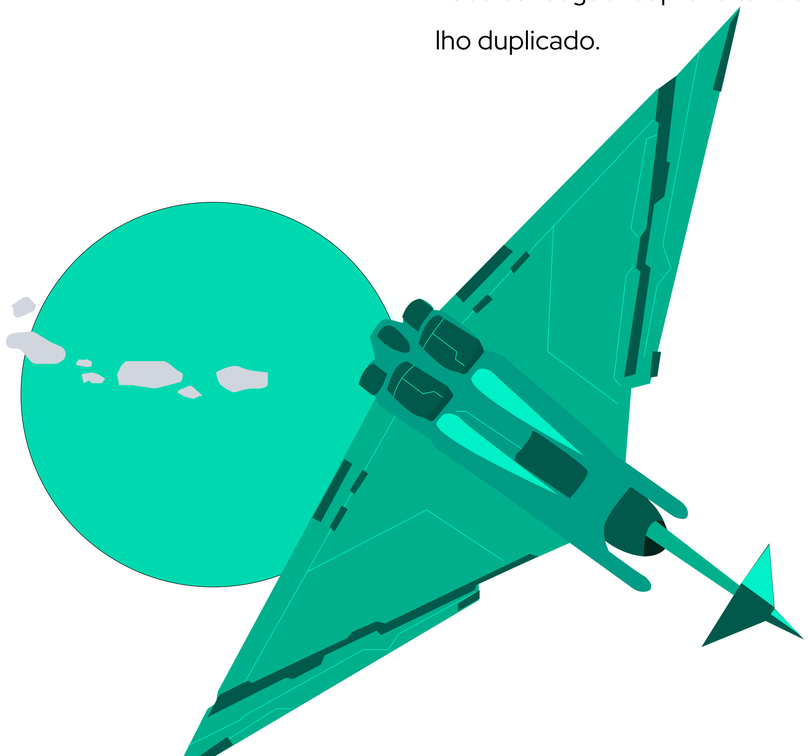
Podemos dizer então que: Seu carro novo é um objeto onde suas características são seus atributos e suas ações são seus métodos.

Porém, existem diversos carros "parecidos" com este na própria concessionária que você foi. Todos possuem volante, banco, vidros, quatro rodas, etc.

Por mais que eles sejam diferentes, todos têm suas próprias ações e características, ou seja, atributos e métodos. Podendo ser iguais ou diferentes.

E por mais que eles pareçam muito iguais, cada carro é único, sendo diferentes instâncias de uma mesma classe.

A parte boa da orientação a objeto é que além de uma organização você consegue reaproveitar boa parte do seu código evitando trabalho duplicado.



Mundo 2

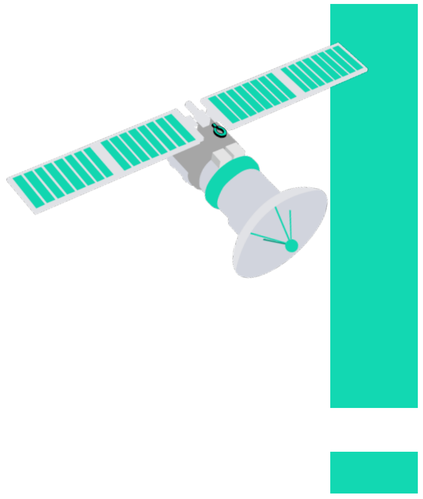
Neste mundo daremos os primeiros passos em direção à programação orientada a objeto.

2.1. Pré-configurações

Neste mundo não utilizaremos o Jupyter Notebook. Isso porque trabalhar com orientação a objeto é muito mais fácil e utilizado no VSCode. Então, caso não tenha baixado, volte à Galáxia 1 e faça o passo a passo para instalação do VSCode.

2.2. Criando uma class

Vamos ver alguns atributos e formas das class.



2.2.1. Sintaxe da class

As class, assim como estruturas de repetição, possuem suas hierarquias através das tabulações. A criação da classe é feita com a seguinte sintaxe:

```
class _nome_escolhido_:
```

2.2.2. Método __init__:

Este é um método construtor. Ele vai construir objetos assim que a classe for iniciada. Vai ser a primeira coisa que a classe fará quando iniciada. Assim como qualquer outra função, sua hierarquia funciona através de tabulações. E este método obedece a seguinte sintaxe:

```
def __init__( ):
```

A classe tem uma funcionalidade, que a princípio pode trazer estranheza. Ao criar qualquer função, você pode fazer com que alguns parâmetros sejam definidos antes, e com o método construtor __init__ não seria diferente.

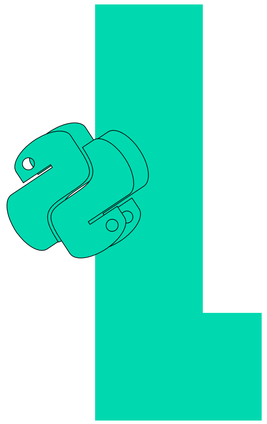
2.2.3. Parâmetro self:

Por meio deste parâmetro poderemos acessar os atributos e métodos de uma classe em Python. Ele refere que algum objeto receberá as propriedades atribuídas da class.

No exemplo abaixo, estamos criando uma class “Empresa” onde terá dois atributos, nome e ticker, que poderão ser acessados depois. Chamaremos essa class, definindo o nome e o ticker.

Exemplo:

```
class Empresa:

    def __init__(self, nome, ticker):

        self.nome = nome
        self.ticker = ticker

objeto1 = Empresa(nome="WEGE", ticker="WEGE3")

print(objeto1.nome)
```



Respostas:

```
>> WEGE
```

Você pode até estar estranhando o self, mas ele se refere ao objeto que a class vai criar. No caso acima, a class é usada para criar o “objeto1” e deste objeto foi retirado o nome e o ticker. Eu sei, eu sei. Pode estar parecendo confuso agora, mas com a prática, as coisas ficarão mais claras.

2.3. __name__ == “__main__”

O python possui algumas funções que são nativas, criadas para facilitar o uso do usuário e essa é uma delas. Quando você roda um arquivo.py o python informa para váriavel “__name__” que é o arquivo principal que está sendo utilizado, ou seja o “__main__”. Quando você importa uma class o python informa que não é o “__main__” e sim uma importação. Claro que isso tudo acontece debaixo dos panos.

Claro que você não precisa decorar isso tudo agora, essas coisas acontecem sem ao menos você perceber. O que você precisa saber é:

Quando você importar um programa, tudo que estiver escrito nele será importado, menos a parte que está dentro da tabulação **if __name__ == “main”**:

if __name__ == “main”:

{ Parte que só será lida pelo arquivo principal }

Então por exemplo:

Se rodarmos este script ele mostrará o ano de fundação da empresa de motor e a cor do carro do nelson. Pois ele é o principal, o “__main__”:

Exemplo:

```python
class Empresa:

    def __init__(self, nome, ticker, ano_fundacao, cnpj): #construtor

        self.nome = nome #atributos da instancia
        self.ticker = ticker

        self.ano_fundacao = ano_fundacao
        self.cnpj = cnpj


class Carro:

    def __init__(self, cor, direcao):

        self.cor_do_carro = cor
        self.tipo_direcao = direcao


if __name__ == "__main__": #isso aqui serve para fazer testes dentro do próprio código. Só vai rodar quando rodar esse código como main

    empresa_de_motor = Empresa(nome = "WEG", ticker = "WEGE3", ano_fundacao = 1960, cnpj = "3981381291")
    carro_do_nelson = Carro(cor = "Preta", direcao = "Elétrica")

    print(empresa_de_motor.ano_fundacao)
    print(carro_do_nelson.cor_do_carro)
```

Respostas:

```
>> 1960
>> Preta
```

Se importarmos esse script, ele não mostrará nada para nossa tela, ao menos que a gente peça. Isso porque aquela parte que está dentro da tabulação do **if __name__ == "main"**: não aparece. Pois como esse não é arquivo "__main__" faz com que tudo que esteja dentro da tabulação "fique invisível" na hora da importação.

Repare que nem o ano de fundação da empresa nem a cor do carro apareceram.

Exemplo:

```python
from galaxia_6_mundo_2 import Empresa

petro = Empresa(nome = "Petrobras", ticker = "PETR4")

print(petro.nome)
```

Respostas:

```
>> Petrobras
```

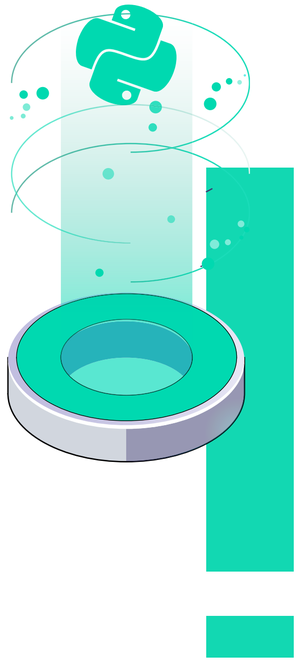
Mundo 3

Neste mundo aprenderemos a criar, modificar e acessar os métodos de instância dentro da programação orientada a objeto.

3.1. Criação de um método de instância.

No caso abaixo estamos criando um método que vai retirar caracteres especiais do cnpj, deixando apenas números. Métodos da instância são apenas funções que podem ser aplicadas a TODOS os casos, desde que respeite o contexto.

Por exemplo, independente do CNPJ colocado ele retornaria apenas o CNPJ com números.



Exemplo:

```python
class Empresa:

    def __init__(self, nome, ticker, ano_fundacao, cnpj): #construtor

        self.nome = nome #atributos da instancia
        self.ticker = ticker
        self.ano_fundacao = ano_fundacao
        self.cnpj = cnpj

    def cnpj_numerico(self):

        self.cnpj_inteiro = int(self.cnpj.replace("-", "").replace(".", "").replace("/", ""))
        print(f"O CNPJ só com números é {self.cnpj_inteiro}.")



if __name__ == "__main__":


    weg = Empresa(nome = "WEG", ticker = "WEGE3", cnpj="84.429.695/0001-11")

    weg.cnpj_numerico()
```

Respostas:

```
>> O CNPJ só com números é 84429695000111.
```

Mundo 4

Neste mundo aprenderemos a criar, modificar e acessar os métodos de classes dentro da programação orientada a objeto.

4.1. Criação de um método de classe.

Sua sintaxe respeita a seguinte ordem:

```
@classmethod
def nome_funcao( self, parâmetro1, parâmetro2 ... ):

{ FUNÇÃO_FEITA_POR_VOCÊ }

return cls(parâmetro1, parâmetro2, parâmetroNovo ,...)
```

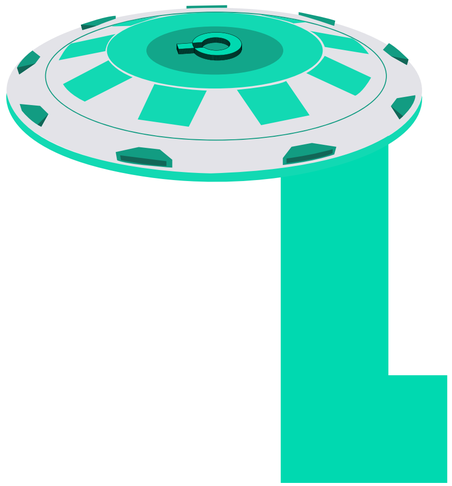
No caso abaixo, estamos criando um método que vai calcular o ano de fundação de acordo com os anos de existência da empresa. Pense no método de classes como se você tivesse botando uma função dentro de uma função já existente para que o resultado saia da forma que você deseja. Como se você tivesse formatando, o valor final, do seu jeito.

Exemplo:

```python
class Empresa:

    ano_atual = 2022

    def __init__(self, nome, ticker, ano_fundacao, cnpj): #construtor

        self.nome = nome #atributos da instancia
        self.ticker = ticker
        self.ano_fundacao = ano_fundacao
        self.cnpj = cnpj

    @classmethod
    def extraindo_empresa_por_ano_existencia(cls, anos_existencia, nome, ticker, cnpj):

        ano_fundacao = cls.ano_atual - anos_existencia
        return cls(nome, ticker, ano_fundacao, cnpj)

if __name__ == "__main__":

    weg = Empresa.extraindo_empresa_por_ano_existencia(nome = "WEG", ticker = "WEGE3", anos_existencia=62, cnpj="84.429.695/0001-11")

    print(weg.ano_fundacao)
```

Respostas:

```
>> 1960.
```



Mundo 5

Neste mundo aprenderemos a criar, modificar e acessar os métodos de estáticos dentro da programação orientada a objeto.

5.1. Criação de um métodos estáticos.

Sua sintaxe respeita a seguinte ordem:

```
@staticmethod
def nome_funcao( ):

{ FUNÇÃO_FEITA_POR_VOCÊ }

return instancia
```

No caso abaixo estamos criando, de forma simples, um método que gera um id aleatório para empresa. O "**@staticmethod**" não é necessário, porém é utilizado para melhorar a organização do código. Como pode ver, ela não está associada a nenhum objeto nem a nada.

Exemplo:

```python
import random

class Empresa:

    ano_atual = 2022 #atributo de classe. Não importa a instância

    def __init__(self, nome, ticker, ano_fundacao, cnpj): #construtor

        self.nome = nome #atributos da instancia
        self.ticker = ticker
        self.ano_fundacao = ano_fundacao
        self.cnpj = cnpj

    @staticmethod #só deixa aqui por organização, uma função normal
    def gera_id():

        id_aleatorio = random.randint(0, 100)
        return id_aleatorio


if __name__ == "__main__":

    id_empresa = Empresa.gera_id()

    print(id_empresa)
```

Mundo 6

Neste mundo aprenderemos a criar getters e setters e a importância deles.

6.1. Criação de um getter

Sua sintaxe respeita a seguinte ordem:

```python
@property
def nome_funcao( self ):
return self._nome_instancia
```

O getter serve para pegar a informação que será tratada.

6.2. Criação de um setter

Sua sintaxe respeita a seguinte ordem:

```
@nome_funcao.setter
def nome_variavel( self, parâmetro1 )

self._nome_instancia = operação
```

O setter serve para você formatar a instância do jeito que deseja.

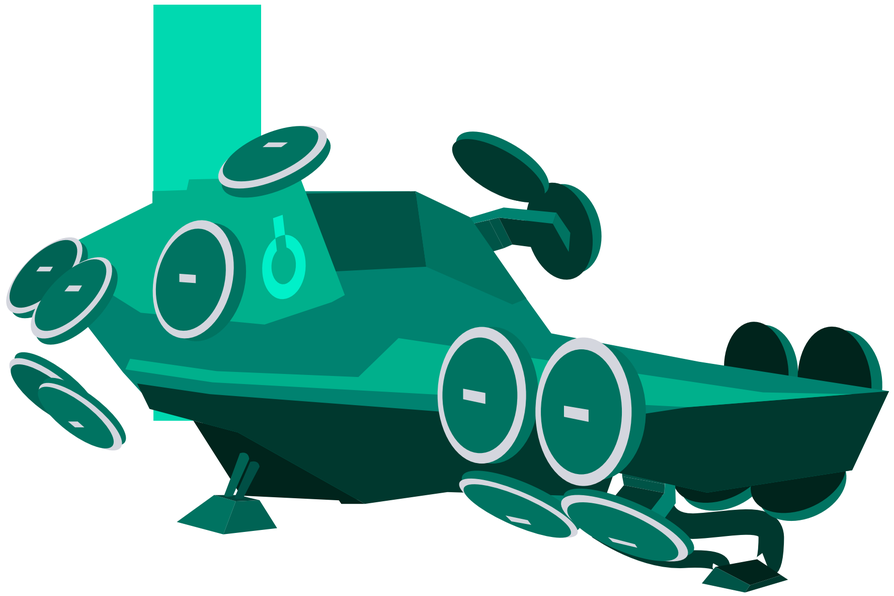
No caso abaixo, estamos criando um “getter” para pegar as informações e definindo um “setter” para formatar a informação para um int.

Exemplo:

```python
class Empresa:

    ano_atual = 2022 #atributo de classe. Não importa a instância

    def __init__(self, nome, ticker, ano_fundacao, cnpj): #construtor

        self.__nome = nome #atributos da instancia
        self.ticker = ticker
        self.ano_fundacao = ano_fundacao
        self.cnpj = cnpj

    #getter
    @property
    def ano_fundacao(self): #pode ser qualquer nome
        return self._ano_fundacao #aqui pode ser qualquer nome

    @ano_fundacao.setter #o mesmo nome do getter
    def ano_fundacao(self, ano): #o importante é a definição aqui, com o atributo sendo o nome da função.

        self._ano_fundacao = int(ano)

if __name__ == "__main__":

    weg = Empresa(nome = "wEg", ticker = "WEGE3", ano_fundacao="1960", cnpj="84.429.695/0001-11")

    print(weg.ano_fundacao, type(weg.ano_fundacao))
```

Respostas:

```
>> 1960 <class 'int'>
```



Mundo 7

Neste mundo aprenderemos sobre variáveis públicas e privadas e sua importância para o tratamento de erros.

7.1. Variáveis públicas

São variáveis que conseguem ser substituídas por qualquer valor e não possuem nenhuma sinalização de que aquela variável não pode ser mudada.

7.2. Variáveis protegidas

São variáveis que conseguem ser substituídas por qualquer valor, porém possuem sinalização de que aquela variável não pode ser mudada, sua sinalização é feita por meio de “_” antes do nome da palavra, por exemplo:

_nomeVariavel

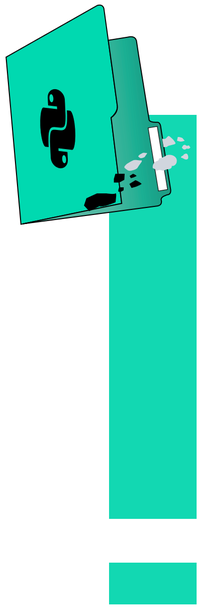
7.3. Variáveis privadas

São variáveis que não podem ser substituídas e possuem sinalização de que aquela variável não pode ser mudada, sua sinalização é feita por meio de "_ _" antes do nome da palavra, por exemplo:

__nomeVariavel

Por mais que você tente, você não conseguirá mudar o valor dessa variável. Essa utilidade serve tanto para atributo quanto para método.

No exemplo abaixo definimos a variável "site" como sendo privada. Repare que mesmo que a gente tente mudar o valor dela, isso não acontece. Uma vez que a variável privada foi definida, ela se torna imutável.



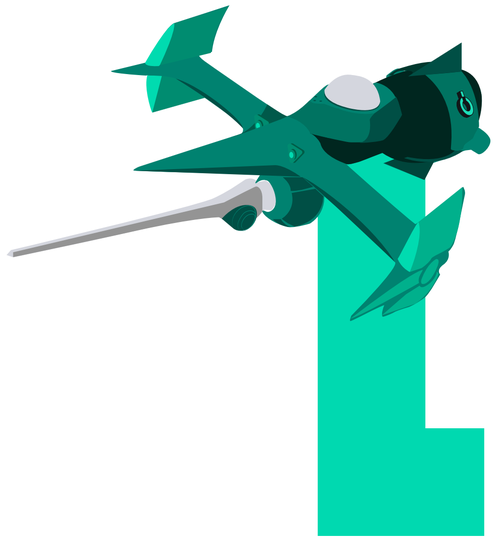
Exemplo:

```python
import wget

class Cvm:

    def __init__(self):

        self.__site = "http://dados.cvm.gov.br/dados/CIA_ABERTA/DOC/ITR/DADOS/"


    def pegando_itr(self, ano_arquivo):

        url = self.__site + f"itr_cia_aberta_{ano_arquivo}.zip"

        wget.download(url)



if __name__ == "__main__":
    baixando_dados = Cvm()


    baixando_dados.__site = "et bilu"

    print(baixando_dados.__site)

    baixando_dados.pegando_itr(2022)
```

Respostas:

Abrirá uma contagem de um download e ao final terá sido baixado um arquivo “itr_cia_202X.zip”

```
>> 100% [..............................................
.......................] 30324054 / 30324054
```



Mundo 8

Neste mundo aprenderemos a criar uma class de conexão de banco de dados que servirá para múltiplas aplicações.

8.1. Criando a class banco_de_dados

No exemplo abaixo, estamos criando uma class fictícia de uma conexão de banco de dados.

Lembrando que: Ela não funcionará. Para que haja a conexão precisamos importar módulos específicos para isso, os veremos mais à frente no nosso módulo de banco de dados.

Porém essa estrutura e a lógica por trás se repetirá mais a frente, então é bom que você entenda e saiba como funciona.

No exemplo abaixo, essa class estabelece uma conexão com banco de dados. Isso faz com que a gente possa retirar e adicionar informações. Estabelecer uma conexão é o primeiro passo antes das trocas de informações.

Exemplo:

```
class Banco_de_dados:

    def __init__(self, senha, user):

        self.senha = senha
        self.user = user

    def iniciar_conexao(self):

        self.conexao = self.user + self.senha

        print('Conexão iniciada com sucesso')
```

8.2. Utilizando a class banco_de_dados

Após criado a class que fará a conexão com o banco de dados. A gente importará ela dentro do nosso método construtor "__init__". Pois queremos que a primeira coisa que a aplicação faça é estabelecer a conexão, e é exatamente isso que este método faz.

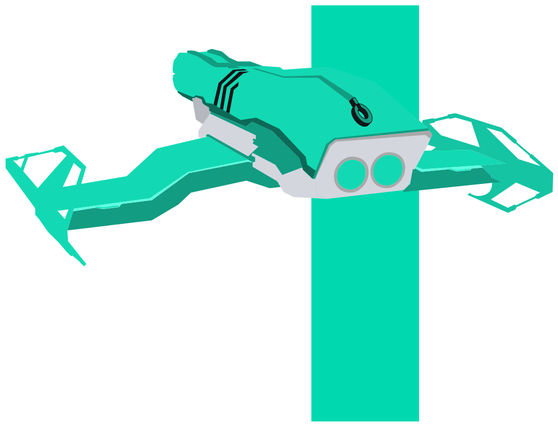
Após estabelecida a conexão inicial, precisaremos de métodos específicos que retirem ou adicionem informações dentro do banco de dados. Aprenderemos esses métodos mais para frente no nosso módulo específico para isso.

Exemplo:

```
class Cotacoes_empresas:

    def __init__(self):

        self.cotacoes = [20, 20.03, 21, 22.30]
        self.bd = Banco_de_dados(user = "edufinance", senha = "codigo.py")

    def soma_cotacoes(self):

        return print(sum(self.cotacoes))

    def colocar_cotacoes_na_base_de_dados(self):

        self.bd.iniciar_conexao()

        colocar_dados = (self.bd.conexao, self.cotacoes)

        print("Dados na base!")

if __name__ == "__main__":

    teste_cotacoes = Cotacoes_empresas()

    teste_cotacoes.colocar_cotacoes_na_base_de_dados()
```

Respostas:

```
>> Conexão iniciada com sucesso
>> Dados na base!
```



Mundo 9

Neste mundo aprenderemos sobre agregação e como utilizar

9.1. Agregação

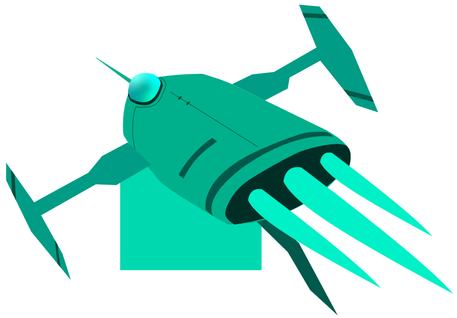
É quando uma class precisa da outra para existir.

9.2. Aplicação agregação

9.2.1. class Carteira_investimento

No exemplo abaixo temos uma carteira de investimentos. Por mais que toda sua estrutura esteja pronta, não há nenhuma informação. Essa carteira de investimentos sem nenhuma informação, é inútil e sem utilidade, logo ela precisa de uma class que complemente ela e que faça com que as informações sejam passadas para si mesma.

E essa class vai ser de "Acoes" que podem compor a carteira de investimentos, ou seja, a "Carteira_investimentos" não serve de muita coisa sem a class "Acoes".



Exemplo:

```
class Carteira_investimento:

    def __init__(self, nome_pessoa):

        self.proprietario = nome_pessoa
        self.carteira = []

    def inserir_acao(self, acao):

        self.carteira.append(acao)

    def listar_acoes(self):

        for acao in self.carteira:

            print(acao.ticker, acao.nome_empresa)
```

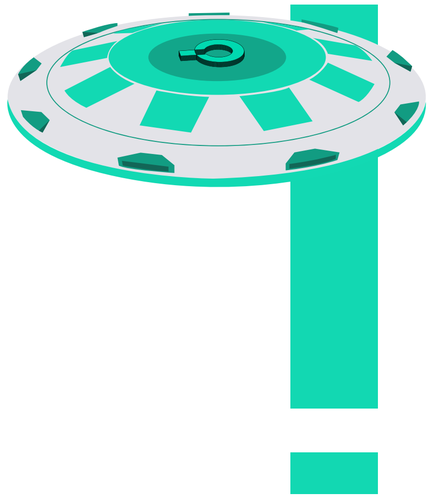
9.2.2. class Acoes

O exemplo abaixo se trata de uma class de ações que define o nome e o ticker das ações, essa class sozinha não serve de muita coisa, já que é apenas o nome e o ticker da ação.

Porém, essa class em conjunto com a "Carteira_investimentos" compõe uma ferramenta do mercado financeiro que pode ser utilizada para análise.

Exemplo:

```
class Acoes:

    def __init__(self, ticker, nome_empresa):

        self.ticker = ticker
        self.nome_empresa = nome_empresa
```



9.3. Exemplo:

Reparem que no exemplo abaixo, ao invés de estar passando uma string para o método "inserir_acao", é um objeto que está sendo passado e este objeto contém as informações necessárias sobre a ação.

```
class Carteira_investimento:

    def __init__(self, nome_pessoa):

        self.proprietario = nome_pessoa
        self.carteira = []

    def inserir_acao(self, acao):

        self.carteira.append(acao)

    def listar_acoes(self):

        for acao in self.carteira:

            print(acao.ticker, acao.nome_empresa)

class Acoes:

    def __init__(self, ticker, nome_empresa):

        self.ticker = ticker
        self.nome_empresa = nome_empresa
```

```python
if __name__ == '__main__':

    carteira_brenno = Carteira_investimento("Brenno")


    weg = Acoes("WEGE3", "Weg")
    petro = Acoes("PETR4", "Petrobras")


    carteira_brenno.inserir_acao(weg)
    carteira_brenno.inserir_acao(petro)


    carteira_brenno.listar_acoes()
```

Respostas:

```
>> WEGE3 Weg
>> PETR4 Petrobras
```

Mundo 10

Neste mundo aprenderemos sobre composição e como utilizá-la.

10.1. Composição

É quando uma class é dona de outra. Elas não têm funcionalidades separadas, já que uma precisa da outra para existir. Se a class principal deixar de existir, a subordinada também deixará.

10.2. class Empresa

No exemplo abaixo temos uma class que contém as informações de uma empresa. Essa class recebe outra class, em forma de objeto, que contém as informações do endereço respectivo da empresa. Podemos dizer que a class "Empresa" é mãe da "Endereco", pois esta última é utilizada por ela. Além disso, um endereço precisa estar associado à alguma coisa, neste caso é a empresa.

Ao final da execução do programa, as informações serão deletadas,isso porque foi utilizado o método "__del__".

Exemplo:

```
class Empresa:

    def __init__(self, nome_empresa, ticker):

        self.nome = nome_empresa
        self.ticker = ticker
        self.enderecos = []

    def adicionando_enderecos(self, estado, cidade, pais):

        self.enderecos.append(Endereco(estado, cidade, pais))

    def lista_enderecos(self):

        for endereco in self.enderecos:

            print(endereco.estado, endereco.cidade, endereco.pais)

    def __del__(self):

        print(f"{self.nome} foi apagado")
```

10.2.2. class Endereco:

No exemplo abaixo temos uma class que cria um endereço e ao final da execução, ela exclui as informações.

Exemplo:

```
class Endereco:

    def __init__(self, estado, cidade, pais):

        self.estado = estado
        self.cidade = cidade
        self.pais = pais

    def __del__(self):

        print(f"{self.cidade} foi apagado")
```

10.3. Exemplo:

Reparem que no exemplo abaixo, o objeto é criado diretamente no método "adicionando_enderecos". Depois de criado, ele exclui tanto as empresas quanto os endereços.

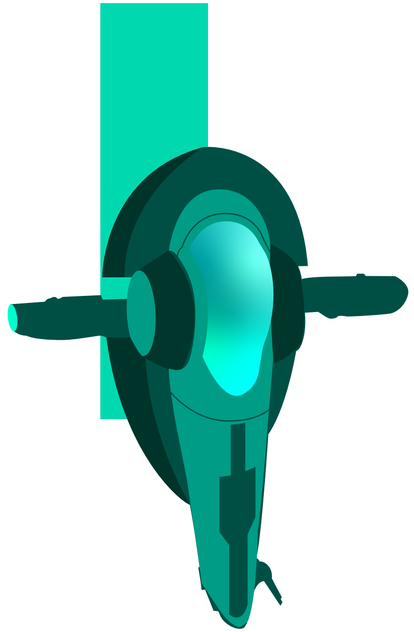
Lembrando que quando a class principal for apagada, a outra também será. Ou seja, quando a empresa for apagada, o endereço será também.

```python
class Empresa:

    def __init__(self, nome_empresa, ticker):

        self.nome = nome_empresa
        self.ticker = ticker
        self.enderecos = [] #Empresa pode ter varios endereços

    def adicionando_enderecos(self, estado, cidade, pais):

        self.enderecos.append(Endereco(estado, cidade, pais))

    def lista_enderecos(self):

        for endereco in self.enderecos:

            print(endereco.estado, endereco.cidade, endereco.pais)

    def __del__(self):

        print(f"{self.nome} foi apagado")
```

```python
class Endereco:

    def __init__(self, estado, cidade, pais):

        self.estado = estado
        self.cidade = cidade
        self.pais = pais

    def __del__(self):

        print(f"{self.cidade} foi apagado")


if __name__ == "__main__":
    weg = Empresa("Weg", "WEGE3")

    weg.adicionando_enderecos(estado = 'SC', cidade = "Jaraguá do Sul", pais = "Brasil")
    weg.adicionando_enderecos(estado = 'Missouri', cidade= 'Washington', pais = "EUA")

    weg.lista_enderecos()

    print("-" * 20)
```

Respostas:

```
>> SC Jaraguá do Sul Brasil
>> Missouri Washington EUA
>> --------------------
>> Weg foi apagado
>> Washington foi apagado
>> Jaraguá do Sul foi apagado
```



Mundo 11

Neste mundo aprenderemos sobre herança e como utilizá-la.

11.1. Herança

Talvez seja a principal característica quando se trata de orientação a objeto. É quando uma class faz parte de outra e herda todas as características dela.

11.2. Aplicação herança

11.2.1. class Pessoa

No exemplo abaixo temos uma class que contém características de uma pessoa. Essa classe cria um objeto com essas características.

Exemplo:

```python
class Pessoa:

    def __init__(self, nome, cidade):

        self.nome = nome
        self.cidade = cidade

    def falar_sobre_futebol(self):

        print(f"{self.nome} está falando sobre futebol. (Vamos flamengo)")
```

11.2.2. class Investidor

No exemplo abaixo temos uma class que atribui as características do objeto criado "Pessoa" por meio da herança. Afinal, todo investidor é uma pessoa.

Exemplo:

```python
class Investidor(Pessoa):

    def comprar_acoes(self):

        print(f"{self.nome} está comprando ações!")
```

11.3. Exemplo:

Reparem como no exemplo abaixo, o objeto "Investidor" está atribuindo características do objeto "Pessoa", por meio da herança.

```python
class Pessoa:

    def __init__(self, nome, cidade):

        self.nome = nome
        self.cidade = cidade


class Investidor(Pessoa):

    def comprar_acoes(self):

        print(f"{self.nome} está comprando ações!")

if __name__ == "__main__":

    brenno = Investidor(nome = "Brenno", cidade = 'Niterói')
    brenno.comprar_acoes()
```

Respostas:

```
>> Brenno está comprando ações
```