Nome da Empresa

PyGame + POO

Start War

Um **sprite** é um elemento gráfico do jogo, e possui 2 propriedades importantes: image e rect. Image é a imagem em si, carregada do disco (os principais formatos são suportados, tais como JPG, GIF, PNG e BMP) e rect representa o retângulo virtual que contém a imagem (imagine um retângulo circunscrito à imagem).

Nossa primeira classe é muito simples, **herdada** da classe **Sprite** nativa do PyGame

```
class Nave(pygame.sprite.Sprite):
    def __init__(self,groups) -> None:
        super().__init__(groups)
        self.__image = pygame.image.load(os.path.join("assets","img","ship.png")).convert_alpha()
        self.__image = pygame.transform.scale(self.__image,(35,30))
        self.image = self.__image
        self.__rect = self.image.get_rect(center=(1200/2, 650/2))
        self.rect = self.__rect
        # Criando um timer para o disparo
        self.__pode_disparar = True
        self.__time_tiro = None
```

Herança

REPETINDO CÓDIGO?

Como toda empresa, nosso banco possui funcionários. Um funcionário tem um nome, um cpf e um salário.
Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente.

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario
```

```
class Gerente:

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciados):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario
        self._senha = senha
        self._qtd_gerenciados = qtd_gerenciados

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```

Herança

A **Herança** possibilita que as classes compartilhem seus atributos, métodos e outros membros da classe entre si. Para a ligação entre as classes, a herança adota um relacionamento esquematizado hierarquicamente.

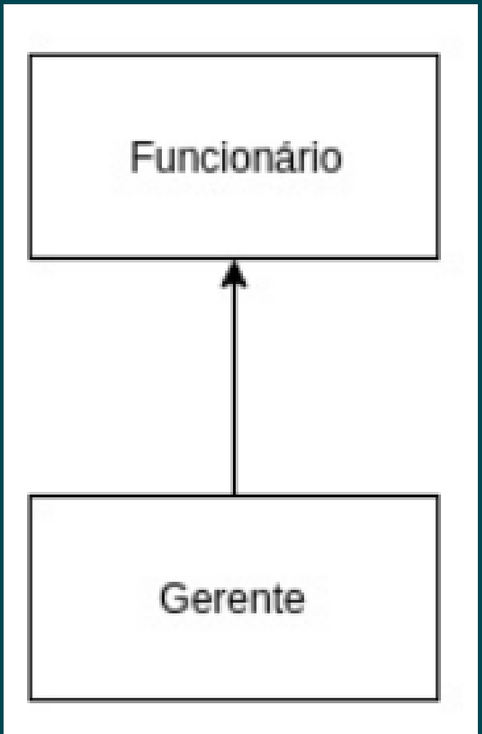
Na Herança temos dois tipos principais de classe:

- **Classe Base:** A classe que concede as características a uma outra classe.
- **Classe Derivada:** A classe que herda as características da classe base.

O fato de as **classes derivadas** herdarem atributos das classes bases assegura que programas orientados a objetos cresçam de forma linear e não geometricamente em complexidade. Cada nova classe derivada não possui interações imprevisíveis em relação ao restante do código do sistema.

Herança

No nosso caso, gostaríamos de fazer com que **Gerente** tivesse tudo que um **Funcionario** tem, gostaríamos que ela fosse uma extensão de Funcionario . Fazemos isso acrescentando a classe mãe entre parenteses junto a classe filha:




```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_funcionarios):
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```

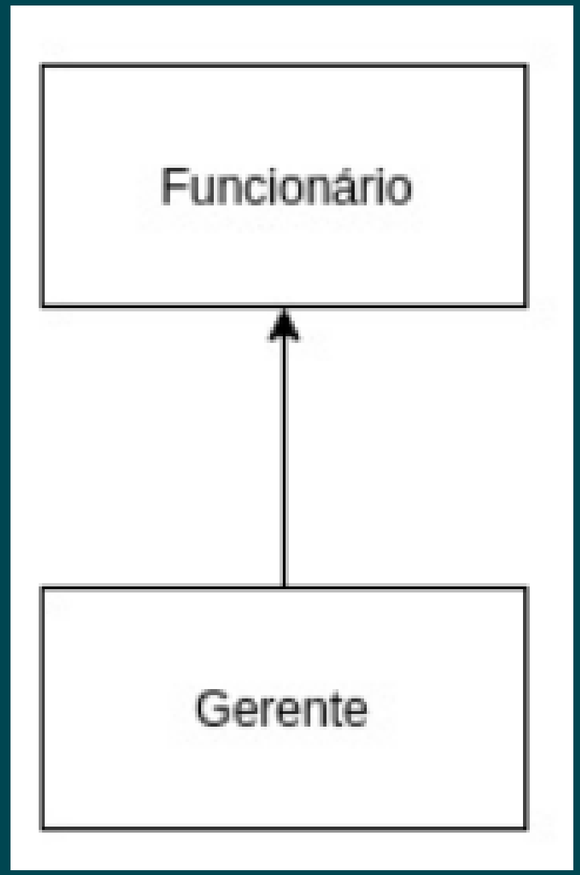
Herança

```python
class Nave(pygame.sprite.Sprite):
    def __init__(self,groups) -> None:
        super().__init__(groups)
        self.__image = pygame.image.load(os.path.join("assets","img","ship.png")).convert_alpha()
        self.__image = pygame.transform.scale(self.__image,(35,30))
        self.image = self.__image
        self.__rect = self.image.get_rect(center=(1200/2, 650/2))
        self.rect = self.__rect
        # Criando um timer para o disparo
        self.__pode_disparar = True
        self.__time_tiro = None
```

Herança

Método __init__() de umaclasse-filha

A primeira tarefa de Python ao criar uma instância de uma classe-filha é atribuir valores a todos os atributos da classe-pai. Para isso, o método **__init__()** de uma classe-filha precisa da ajuda de sua classe-pai.




```
class Gerente(Funcionario):

    def __init__(self, nome, cpf, salario, senha, qtd_funcionarios):
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios
```

```
class Gerente(Funcionario):

    def __init__(self, nome, cpf, salario, senha, qtd_funcionarios):
        super().__init__(nome, cpf, salario)
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios
```

Polimorfismo - O que é?

Polimorfismo, em Python, é a capacidade que uma subclasse tem de ter métodos com o mesmo nome de sua superclasse, e o programa saber qual método deve ser invocado, especificamente (da super ou sub).Ou seja, o objeto tem a capacidade de assumir diferentes formas (polimorfismo).
Vamos criar a classe Superclasse que tem apenas um método, o hello(). Instanciamos um objeto e chamamos esse método:

```
class Super:
 def hello(self):
  print("Olá, sou a superclasse!")

teste = Super()
teste.hello()
```

Polimorfismo - O que é?

Agora vamos criar outra classe, a Sub, que vai herdar a Superclasse e vamos definir nela um método de mesmo nome hello(), mas com um texto diferente:

```
class Super:
 def hello(self):
  print("Olá, sou a superclasse!")

class Sub (Super):
 def hello(self):
  print("Olá, sou a subclasse!")

teste = Sub()
teste.hello()
```

O resultado vai ser:

- **Olá, sou a subclasse!**

Veja bem, Sub herda a **Superclasse**, ou seja, tudo que nem na superclasse (atributos e métodos), vai ter na subclasse.
Porém, quando chamamos o método hello(), ele vai invocar o método da subclasse e não da superclasse! O **Python** entende: "Opa, ele instanciou um objeto da subclasse. Por isso vou invocar o método da subclasse e não da superclasse"

Polimorfismo - O que é?

Agora vamos criar outra classe, a Sub, que vai herdar a Superclasse e vamos definir nela um método de mesmo nome hello(), mas com um texto diferente:

```
class Super:
 def hello(self):
  print("Olá, sou a superclasse!")

class Sub (Super):
 def hello(self):
  print("Olá, sou a subclasse!")

teste = Sub()
teste.hello()
```

O resultado vai ser:

- **Olá, sou a subclasse!**

Veja bem, Sub herda a **Superclasse**, ou seja, tudo que nem na superclasse (atributos e métodos), vai ter na subclasse.

Porém, quando chamamos o método hello(), ele vai invocar o método da subclasse e não da superclasse! O **Python** entende: "Opa, ele instanciou um objeto da subclasse. Por isso vou invocar o método da subclasse e não da superclasse"

Polimorfismo - O que é?

Subsubclasse, que vai herdar a **Sub**.
Hora, se a **Subsubclasse** herda a Sub, e a Sub herda a Super, então a Subsubclasse também herda tudo da Super.
Porém, quando instanciamos um objeto da Subsub e invocamos o método **hello()**, ele vai rodar o método da **Subsub**

```
class Super:
 def hello(self):
  print("Olá, sou a superclasse!")

class Sub (Super):
 def hello(self):
  print("Olá, sou a subclasse!")

class Subsub (Sub):
 def hello(self):
  print("Olá, sou a subsubclasse!")

teste = Subsub()
teste.hello()
```