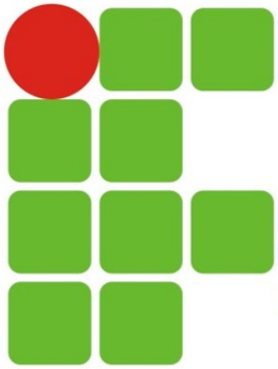

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
RIO GRANDE DO NORTE

Programação Orientada a Objetos

APRESENTAÇÃO DA DISCIPLINA

PROFESSOR:
Demetrios Coutinho
Demetrios.coutinho@ifrn.edu.br

AGENDA

- Herança
- Polimorfismo

OS QUATRO PILARES

ABASTRAÇÃO

ENCAPSULAMENTO

HERANÇA

POLIMORFISMO

Herança

- Tecnicamente, todas as classes que usamos já usam herança, uma vez que elas herdam da classe *object*;
- Os comportamentos fornecidos por essa classe são aqueles com underscore duplo;
- A classe *object* permite que o python trate todos os objetos da mesma maneira, isto é, como um objeto;
- Até agora ao criar nossas classes não explicitamos que herdamos de *object*, mas podemos, conforme segue:

```
1 ▾ class MySubClass(object):
2       pass
```

Herança

```
1 ▾ class MySubClass(object):
2       pass
```

- Isso é **herança**;
- Chamamos de superclasse ou classe pai a classe que estendemos e de subclasse ou classe filha a classe que herda.
  - No caso acima, object é a superclasse e MySubClasse a subclasse;
- Para realizar a herança em python basta colocar o nome da superclasse a qual se quer herdar dentro dos parênteses da definidação da subclasse;

Herança

- O uso mais simples da herança é adicionar funcionalidade a uma classe existente.
- Para ilustrar, veja um simples exemplo de uma classe base que gerencia contatos:

```
1  class Contact:
2      all_contacts = []
3
4      def __init__(self, name, email):
5          self.name = name
6          self.email = email
7          Contact.all_contacts.append(self)
```

- Neste exemplo usamos o que chamamos de variável de classe;
  - A lista *all_contacts* é uma variável de classe porque é compartilhada por todos objetos da classe *Contact*.
- Agora imagine que existem contatos de fornecedores que fazemos pedidos;
  - Poderíamos pensar em adicionar um método **order** na classe Contact, mas isso faria que contatos familiares também tivessem esse método;
  - Vamos criar uma classe Fornecedor (*Supplier* em inglês) que além de herdar de Contact, também possui o método *order*:

Herança

```
1 ▾ class Supplier(Contact):
2 ▾     def order(self, pedido):
3 ▾         print("Pedido para '{}': "
4               "'{}'".format(self.name, pedido))
```

```
1  c1 = Contact('Filipe', 'filipe@gmail.com')
2  f1 = Supplier('Amazon', 'amazon@gmail.com')
```

- Note que o objeto *f1* possui **nome** e **email** pois herda da classe **Contact**:

```
1  print(c1.name, c1.email)
2  print(f1.name, f1.email)
```

```
Filipe filipe@gmail.com
Amazon amazon@gmail.com
```

Herança

- Perceba ainda que apenas o objeto *f1* é capaz de efetuar pedidos:

```
1  f1.order('Preciso de 2 camisas do palmeiras')
```

```
Pedido para 'Amazon': 'Preciso de 2 camisas do palmeiras'
```

```
1  c1.order('Preciso de 2 camisas do palmeiras')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
 last)
<ipython-input-7-c139743b71d5> in <module>
----> 1 c1.order('Preciso de 2 camisas do palmeiras')

AttributeError: 'Contact' object has no attribute 'order'
```

Herança

- Agora perceba como o atributo *all_contacts* é compartilhado por ambos objetos:

In [8]:

```
1  c1.all_contacts
```

Out[8]:

```
[<__main__.Contact at 0x5018e48>, <__main__.Supplier at 0x5018e10>]
```

In [9]:

```
1  f1.all_contacts
```

Out[9]:

```
[<__main__.Contact at 0x5018e48>, <__main__.Supplier at 0x5018e10>]
```

- Como ele é um atributo de classe, você pode acessá-lo diretamente pela classe:

In [10]:

```
1  Contact.all_contacts
```

Out[10]:

```
[<__main__.Contact at 0x5018e48>, <__main__.Supplier at 0x5018e10>]
```

Herdando de built-in

- Um uso interessante desse tipo de herança é adicionar funcionalidade aos recursos built-in do python.
- Para exemplo, podemos criar uma classe *ContactList* que adiciona um método para fazer busca de contatos em uma lista padrão do python.

```
1  class ContactList(list):
2      def search(self, name):
3          matching_contacts = []
4          for contact in self:
5              if name in contact.name:
6                  matching_contacts.append(contact)
7          return matching_contacts
8
9  class Contact():
10     all_contacts = ContactList()
11
12     def __init__(self, name, email):
13         self.name = name
14         self.email = email
15         Contact.all_contacts.append(self)
```

Herdando de built-in

- Testando:

```
1  c1 = Contact('Fulano da Silva', 'fulano1@dominio.com')
2  c2 = Contact('Fulano Sousa', 'fulano2@dominio.com')
3  c3 = Contact('Ciclano Pereira', 'ciclano3@dominio.com')
```

```
1  Contact.all_contacts.search('Fulano')
```

```
Out[13]:

[<__main__.Contact at 0x50b2be0>, <__main__.Contact at 0x50b2b38>]
```

O que significa essa saída? Por que não aparece o nome e o e-mail?

Sobrescrevendo métodos

- Já falamos da sobrecarga de métodos como **__add__** e ***__str__***, basicamente estamos sobrescrevendo a herança dada pela classe *Object*.
- O próprio construtor __init__ é uma sobrecarga de método.
- Por exemplo, imagine que queremos adicionar o atributo *phone_number* nos contatos dos nossos amigos mais próximos.

```
1 ▾ class Friend(Contact):
2 ▾     def __init__(self, name, email, phone):
3           self.name = name
4           self.email = email
5           self.phone = phone
```

- Observe 2 problemas com o código acima:

1. Repetimos código o que pode tornar o processo de manutenção do sistema ruim;
2. Esquecemos de adicionar o contado do amigo na variável de classe *all_contacts*;

Sobrescrevendo métodos

- O que nós realmente precisamos é executar o **__init__** da classe *Contact* e apenas adicionar o telefone do contato.
- Para tanto, podemos usar a palavra chave **super**, que nos permite invocar métodos da **superclasse** diretamente:

```
1 ▾ class Friend(Contact):
2 ▾     def __init__(self, name, email, phone):
3           super().__init__(name, email)
4           self.phone = phone
```

```
1  friend1 = Friend('Beltrano Ferreira', 'beltrano@dominio.com', '2121-3131')
```

```
1  friend1.phone
```

Qualquer método, atributo **não-privado** é herdado pela superclasse.

Sobrescrevendo métodos

- Mais exemplos:

```
class Funcionario:

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

    # outros métodos e properties

    def get_bonificacao(self):
        return self._salario * 0.10
```

```
class Gerente(Funcionario):

def __init__(self, nome, cpf, salario, senha, qtd_gerenciaveis):
    super().__init__(nome, cpf, salario)
    self._senha = senha
    self._qtd_gerenciaveis = qtd_gerenciaveis

    def get_bonificacao():
        return super().get_bonificacao() + 1000

    # métodos e properties
```

Sobrescrevendo métodos

- Mais exemplos:

```
funcionario = Funcionario('João', '111111111-11', 2000.0)
print(vars(funcionario))

gerente = Gerente('José', '222222222-22', 5000.0, '1234', 0)
print(vars(gerente))
```

Saída:

```
{'_salario': 2000.0, '_nome': 'João', '_cpf': '111111111-11'}
{'_cpf': '222222222-22', '_salario': 5000.0, '_nome': 'José', '_qtd_funcionarios': 0, '_senha': '1234'}
```

Polimorfismo

- **Polimorfismo** é a capacidade do objeto de uma subclasse ser referenciado como uma superclasse para que, dependendo da subclasse, possa haver comportamentos diferentes.
- Em outras palavras, diferentes comportamentos podem ocorrer depedendo de qual subclasse está sendo usada, onde não existe a necessidade de sabermos explicitamente que subclasse é essa.

Polimorfismo

- Vamos ver um exemplo.

DÚVIDAS?