Módulo 4: Programação Orientada à Objetos

Objetivos de aprendizagem:

1. Compreender o uso do método **init** () e o parâmetro self. Declarar corretamente uma classe/objeto.
2. Reconhecer a diferença entre funções e métodos e o escopo dos métodos, e fazer chamadas de métodos.
3. Entender como o nível de herança afeta as chamadas e variáveis do método.

Vamos considerar o sistema de biblioteca como um exemplo de **Programação Orientada à Objetos**.

Uma biblioteca tem uma série de livros.

- Quais devem ser os dados associados a cada livro?
- Há alguma operação que um livro deve realizar?

Então, construiremos a **classe** LibraryBook para armazenar dados sobre cada livro e os métodos/operações de suporte no sistema da biblioteca.

Classes são plantas, desenhos ou modelos para instâncias. A relação entre uma classe e uma instância é semelhante àquela entre um cortador de biscoitos e um biscoito.

- Um único cortador de biscoitos pode fazer quantos biscoitos forem necessários. O cortador define a forma do biscoito.
- Os biscoitos são comestíveis, mas o cortador de biscoitos não é.

Referências das palestras do professor Daniel Bauer ENGI1006 da Universidade de Columbia.

In [ ]:
```
# LibraryBook é o nome da classe
class LibraryBook:
  """
  A library book
  """

  # pass indica que o corpo/fato da definição da classe está vazio.
  pass
```

In [ ]:
```
# Isto irá criar uma instância da classe.
my_book = LibraryBook()
my_book
```

Out[ ]:
```
<__main__.LibraryBook at 0x7fca565e5d30>
```

In [ ]:
```
type(my_book)
```

Out[ ]:
```
__main__.LibraryBook
```

In [ ]:
```
# Outra maneira de verificar o tipo de algum objeto
isinstance(my_book, LibraryBook)
```

Out[ ]:
```
True
```

Por que usar classes e quando usá-las?

Os objetos simplificam os problemas, fornecendo uma abstração sobre certos tipos de dados e sua funcionalidade.

> Em vez de pensar no problema em termos de cadeias de caracteres individuais, inteiros, etc., podemos agora pensar em termos de LibraryBooks (ou outros objetos).

Encapsulação

- Os dados e a funcionalidade são agrupados em objetos.
- Os métodos fornecem uma interface para o objeto. O ideal é que os dados individuais sejam apenas escritos e lidos através de métodos.
- Isto significa que os detalhes sobre como a funcionalidade é implementada são escondidos do programador. Por exemplo, não sabemos como o método de anexar em listas é implementado.
- Esta ideia permite que as classes sejam compartilhadas (em bibliotecas) e usadas por outros (ou reutilizadas por você) sem a necessidade de ler o código fonte da classe.

4.1: init, parâmetro Self

Campos de dados - Cada instância possui seus próprios dados (a classe pode definir quais nomes os campos de dados têm).

O método **init(self, ...)** é automaticamente executado por Python quando uma nova instância é criada. Este método é chamado de **construtor de classe**; ele inicializa os valores dos dados na classe.

In [ ]:
```
"""
    Um livro da biblioteca.
    """
    class LibraryBook (object):

      """
      O parâmetro self é OBRIGATÓRIO dentro da classe,
      porque ele diz ao programa para buscar/atuar sobre o objeto de instância
      que a executou.
      """
      def __init__(self, title, author, pub_year, call_no):
          self.title = title
          self.author = author
          self.year = pub_year
          self.call_number = call_no
          self.checked_out = False
```

In [ ]:
```
"""
Como já criamos meu_livro como um objeto do LibraryBook,
agora podemos adicionar manualmente o título, autor,... informações associadas ao liv
ro.
"""


my_book.title = "Harry Potter and the Philosopher's Stone"
my_book.author = ('Rowling', 'J.K.')
my_book.year = 1998
my_book.call_number = "PZ7.R79835"
```

In [ ]:
```
# Busque um campo de dados específico de uma instância executando o nome da instância
e o nome do campo
my_book.author
```

Out[ ]:
```
('Rowling', 'J.K.')
```

In [ ]:
```
"""
Ou podemos passar todas as informações para o __init__ para configurar os campos
ao criar a nova instância.
"""

new_book = LibraryBook("Harry Potter and the Sorcerer's Stone",
                       ("Rowling","J.K."), 1998, "PZ7.R79835")

new_book.author
```

Out[ ]:
```
('Rowling', 'J.K.')
```

4.2: Métodos

Os **métodos** contêm a funcionalidade do objeto.

Estes são definidos na classe.

4.2.1: Escrevendo um Método

In [ ]:
```
class LibraryBook(object):
    """
    Um livro da biblioteca.
    """

    def __init__(self, title, author, pub_year, call_no):
        self.title = title
        self.author = author
        self.year = pub_year
        self.call_number = call_no

    """
    Métodos para o LibraryBook
    """

    # Retorna o título e as informações do autor do livro como uma string
    def title_and_author(self):
        return "{} {}: {}".format(self.author[1], self.author[0], self.title)

    # Imprime todas as informações associadas a um livro neste formato
    def __str__(self): # certifique-se de que __str__ retorna uma string!
        return "{} {} ({}): {}".format(self.author[1], self.author[0], self.year, sel
f.title)

    # Retorna uma representação de string do livro com o título e call_number
    def __repr__(self):
        return "<Book: {} ({})>".format(self.title, self.call_number)
```

In [ ]:
```
# A simples chamada da própria instância está desencadeando __repr__()
new_book
```

Out[ ]:
```
<__main__.LibraryBook at 0x7fca565db278>
```

In [ ]:
```
# print está desencadeando a __string__()
print(new_book)
```

```
<__main__.LibraryBook object at 0x7fca565db278>
```

In [ ]:
```
new_book = LibraryBook("Harry Potter and the Sorcerer's Stone",
                       ("Rowling","J.K."), 1998, "PZ7.R79835")

new_book.title_and_author()
```

Out[ ]:
```
"J.K. Rowling: Harry Potter and the Sorcerer's Stone"
```

4.2.2: Chamadas de Método Interno/Externo

A **única** diferença é:

- Externamente/fora da classe, você simplesmente executaria *instanceName.method()*, como `new_book.title_and_author()`
- Internamente/na classe, você utilizaria `self` para indicar para aquela instância específica da classe, como `self.title_and_author()`

4.3: Herança

Exemplo de **instância de** relação.

> nemo é uma instância de ClownFish (Peixe Palhaço).

In [ ]:
```
class ClownFish(object):
    pass

nemo = ClownFish()
```

In [ ]:
```
type(nemo)
```

Out[ ]:
```
__main__.ClownFish
```

In [ ]:
```
isinstance(nemo, ClownFish)
```

Out[ ]:
```
True
```

Mas ClownFish (Peixe Palhaço) é também um peixe, um vertebrado e um animal, e cada um poderia ser uma classe separada.

Neste caso, precisamos ter relações entre as classes.

- A classe ClownFish poderia ter a classe pai Fish,
  - que poderia ter a classe pai Vertebrate,
    - que poderia ter a classe pai Animal...

Esta relação é chamada de relação **is-a** (é-um). Esta relação se estabelece entre uma classe de filhos e sua classe de pais. Cada classe em Python tem pelo menos uma classe de pais.

(Note que a relação é uma relação transitória, portanto, todo ClownFish também é um Animal).

Há uma super classe em Python chamada objeto. Até agora, quando definimos classes, sempre fizemos do objeto o pai direto da classe.

In [ ]:
```
class Animal(object):
    pass

class Vertebrate(Animal):
    pass

class Fish(Vertebrate):
    pass

class ClownFish(Fish):
    pass

class TangFish(Fish):
    pass
```

In [ ]:
```
nemo = ClownFish()
```

In [ ]:
```
isinstance(nemo, ClownFish)
```

Out[ ]: True

In [ ]:
```
isinstance(nemo, TangFish)
```

Out[ ]: False

In [ ]:
```
# A relação is-a (é-um) é transitiva
isinstance(nemo, Animal)
```

Out[ ]: True

In [ ]:
```
# Todas as classes têm uma classe pai de Objeto
isinstance(nemo, object)
```

Out[ ]: True

4.3.1: Métodos Herdados

Por que usar a herança?

Toda classe também tem acesso aos atributos de classe da classe pai. Em particular, os métodos definidos na classe pai podem ser chamados em instâncias de seus "decendentes".

In [ ]:
```
class Fish(Animal):
    def speak(self):
        return "Blub"

class ClownFish(Fish):
    pass

class TangFish(Fish):
    pass
```

In [ ]:
```
dory = TangFish()

"""
A classe TangFish é uma classe filha de Fish, por isso pode acessar speak() da classe Fish.
Esta classe procura primeiro o método de chamada dentro de sua classe e, se não for encontrado, então repete
a busca por cada nível hierárquico superior.
"""
dory.speak()
```

Out[ ]:
```
'Blub'
```

In [ ]:
```
nemo = ClownFish()

# ClownFish é uma classe filha de Fish, por isso pode acessar speak() da classe Fish

nemo.speak()
```

Out[ ]:
```
'Blub'
```

E se quisermos uma funcionalidade diferente para uma classe filha? Podemos **substituir** o método (escrevendo um novo com o mesmo nome).

In [ ]:
```
class TangFish(Fish):
    def speak(self):
        return "Hello, I'm a TangFish instance."
```

In [ ]:
```
dory = TangFish()

# este speak() é da classe TangFish
dory.speak()
```

Out[ ]:
```
"Hello, I'm a TangFish instance."
```

In [ ]:
```
"""
Por outro lado, como a classe ClownFish ainda NÃO
define o speak(), as instâncias de ClownFish ainda estão usando o
speak() da classe pai de Fish.
"""

nemo = ClownFish()
nemo.speak()
```

Out[ ]:
```
'Blub'
```

In [ ]:
```
# O que acontece quando queremos imprimir a instância nemo?
print(nemo)
```

```
<__main__.ClownFish object at 0x7fa894114b50>
```

Quando você escreve seu próprio método especial (como **str**). Você está substituindo o método com o mesmo nome definido no objeto.

In [ ]:
```
#  A declaração print não é fácil de entender, por isso vamos ignorá-la.

class ClownFish(Fish):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return "A ClownFish named "+self.name
```

In [ ]:
```
nemo = ClownFish('Nemo')

print(nemo)
```

```
A ClownFish named Nemo
```

4.3.2: Acessando Variáveis em uma Relação

Em uma relação is-a (é-um), a classe filha pode acessar os atributos da classe pai se não estiver definida na classe filha, ou substituir o valor do atributo do mesmo atributo existente na classe filha.

Entretanto, se uma instância for definida em um dos níveis da classe pai, então ela NÃO poderá acessar os atributos que estão definidos em qualquer um dos níveis inferiores da classe filha.

In [ ]:
```
class Fish(Vertebrate):

    # self.name não é definido na classe Fish, mas é definido na classe ClownFish.
    def __str__(self):
        return "Hello, my name is {}".format(self.name)

class ClownFish(Fish):
    def __init__(self, name):
        self.name = name
```

In [ ]:
```
nemo = ClownFish("nemo")

# O atributo self.name para __str__() é da classe ClownFish
# mas __str__() é da classe Fish
print(nemo)
```

```
Hello, my name is nemo
```

In [ ]:
```
"""
ERROR, porque se nemo é um exemplo de classe Fish,
então NÃO tem o atributo do nome.
"""
nemo = Fish()
print(nemo)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-33-3dff370cc698> in <module>()
      4 """
      5 nemo = Fish()
----> 6 print(nemo)

<ipython-input-31-b5476c627494> in __str__(self)
      3     # self.name is not defined in Fish class, but is defined in the ClownFish
class.
      4     def __str__(self):
----> 5         return "Hello, my name is {}".format(self.name)
      6
      7 class ClownFish(Fish):

AttributeError: 'Fish' object has no attribute 'name'
```

In [ ]:
```
class Fish(Vertebrate):
    def __init__(self, name):
        self.name = name

    # self.name não é definido na classe Fish, mas é definido na classe ClownFish.
    def __str__(self):
        return "Hello, my name is {}".format(self.name)

class ClownFish(Fish):
    def __init__(self, name):
        self.name = name
```

In [ ]:
```
nemo = ClownFish("Nemo")

# __str__() está acessando o self.name a partir do nível filha
print(nemo)
```

```
Hello, my name is Nemo
```

In [ ]:
```
nemo = Fish("clown_fish")

#  __str__ está acessando o atributo self.name da classe Fish
print(nemo)
```

```
Hello, my name is clown_fish
```

In [ ]:
```

```