Nesta aula, vamos aprender a utilizar Orientação a Objetos em Python.

1 Classes, objetos e construtores

Em Python, a classe de um objeto e o tipo de um objeto são sinônimos. Cada objeto do Python tem uma classe (tipo) que é derivada diretamente ou indiretamente da classe **object**. A classe (tipo) de um objeto determina o que ele é e como pode ele ser manipulado. Uma classe encapsula dados, operações e semântica, ou seja, é um agrupamento de valores e suas operações. O usuário de uma classe manipula objetos instanciados dessa classe somente através dos métodos fornecidos por essa classe.

Ao criarmos um número inteiro, por exemplo, estamos criando um objeto da classe **int**. Podemos saber a classe de cada objeto através do comando **type**:

```
1 x = 50
2 l = [1, 2, 3, 4, 5, 6]
3
4 print(type(x))
5 print(type(l))
```

O resultado do código acima será:

```
<class 'int'>
<class 'list'>
```

A classe **list**, por exemplo, possui métodos como **append** (adiciona um elemento ao final da lista), **remove** (remove um elemento da lista), **reverse** (inverte a ordem dos elementos da lista) e **sort** (coloca os elementos da lista em ordem crescente):

```
1 l = [5,2,3,7,1,4,8,9,10,6]
2 l.remove(5)
3 l.append(11)
4 l.sort()
5 l.reverse()
6 print(l)
```

O resultado do código acima será:

```
[11, 10, 9, 8, 7, 6, 4, 3, 2, 1]
```

Para criar nossas próprias classes, utilizamos o comando **class**:

```
1 class Pessoa:
2     def __init__(self):
3         self.nome = input("Digite o nome da pessoa: ")
```

```
4         self.cpf = input("Digite o cpf da pessoa: ")
5         self.idade = int(input("Digite a idade da pessoa: "))
```

O __init__ é um método responsável pela inicialização dos objetos da classe (semelhante ao **construtor** do Java). Ele sempre deve ter este mesmo nome, e seu primeiro parâmetro (por padrão, chamamos de **self**) se refere ao próprio objeto que está sendo acessado no momento (semelhante ao `this` do Java). Ao contrário do Python, o `this` **não** precisa ser incluído na lista de parâmetros dos métodos de uma classe em Java.

Para criar um objeto da classe, o parâmetro **self** não precisa ser passado:

```
1 p1 = Pessoa()
```

Caso a classe possua um atributo cujo valor inicial é fixo, ele pode ser explicitado fora do __init__:

```
1 class Conta:
2     saldo = 0
3
4     def __init__(self, p):
5         self.numero = input("Digite o numero da conta: ")
6         self.titular = p
```

Neste caso, o valor atributo **saldo** sempre inicia zerado. Note que agora o __init__ recebe um parâmetro adicional, que será um objeto da classe **Pessoa** com o titular da conta. Novamente, o **self** não precisa ser passado como parâmetro ao criar os objetos:

```
1 c1 = Conta(p1)
```

2 Métodos

Classes possuem métodos, que definem o comportamento de seus objetos. Em Python, o método é definido de forma semelhante a uma função, mas deve ser identado dentro da classe e seu primeiro parâmetro sempre será o próprio objeto (**self**):

```
1  class Conta:
2      saldo = 0
3
4      def __init__(self, p):
5          self.numero = input("Digite o numero da conta: ")
6          self.titular = p
7
8      def disponivel(self):
9          return self.saldo
10
11     def depositar(self, valor):
```

```
12          self.saldo += valor
13          print("Deposito realizado com sucesso.")
14
15      def sacar(self, valor):
16          if valor <= self.disponivel():
17              self.saldo -= valor
18              print("Saque realizado com sucesso.")
19              return True
20          else:
21              print("Valor indisponivel para saque.")
22              return False
23
24  p1 = Pessoa()
25
26  c1 = Conta(p1)
27
28  c1.depositar(300)
29  c1.sacar(150)
30  c1.sacar(180)
31
32  print("Valor disponivel: R$", c1.disponivel())
```

O resultado do código acima será:

```
Deposito realizado com sucesso.
Saque realizado com sucesso.
Valor indisponivel para saque.
Valor disponivel na conta: R$ 150
```

Lembre-se sempre que:

- O **self** sempre deve ser o primeiro parâmetro na definição de um método.
- Dentro do método, os atributos da classe são sempre precedidos do **self**. Por exemplo, **self.saldo**.
- O **self não** deve ser passado como parâmetro quando chamarmos o método.

3 Herança

Python também permite a criação de classes derivadas, estendendo as classes já existentes. A classe nova é chamada de classe derivada e a classe existente de quem é derivada

é chamada de classe base. Uma classe derivada herda todos os atributos de sua classe base. Isto é, a classe derivada contém todos os atributos da classe contidos na classe base e a classe derivada suporta todas as operações fornecidas pela classe base.

```
1  class ContaCorrente(Conta):
2      limite = 200
3
4      def disponivel(self):
5          return self.saldo + self.limite
6
7  cc1 = ContaCorrente(p1)
8
9  cc1.depositar(300)
10 cc1.sacar(150)
11 cc1.sacar(180)
12
13 print("Valor disponivel: R$", c1.disponivel())
```

Assim a conta corrente **cc1** é também uma Conta, tem os atributos saldo, titular e numero, e além deles possui também o atributo limite. Além disso, o método **disponivel** foi reescrito para incluir o valor do limite da conta corrente.

O resultado do código acima será:

```
Deposito realizado com sucesso.
Saque realizado com sucesso.
Saque realizado com sucesso.
Valor disponivel na conta: R$ 170
```

4 Encapsulamento e modificadores de acesso

O Python não utiliza o termo private, que é um modificador de acesso e também chamado de modificador de visibilidade, para impedir o acesso aos atributos de uma classe. Ao invés disso, inserimos dois underscores (__) na frente do atributo para adicionar esta característica. Por convenção, os programadores utilizam um único underscore na frente do atributo para indicar que ele é protegido (ou seja, só pode ser acessado na própria classe ou em suas classes derivadas):

```
1  class Conta:
2      _saldo = 0
3
4      def __init__(self, p):
5          self._numero = input("Digite o numero da conta: ")
6          self._titular = p
```

```
7
8       def disponivel(self):
9           return self._saldo
10
11      def depositar(self, valor):
12          self._saldo += valor
13          print("Deposito realizado com sucesso.")
14
15      def sacar(self, valor):
16          if valor <= self.disponivel():
17              self._saldo -= valor
18              print("Saque realizado com sucesso.")
19              return True
20          else:
21              print("Valor indisponivel para saque.")
22              return False
23
24  class ContaCorrente(Conta):
25      __limite = 200
26
27      def disponivel(self):
28          return self._saldo + self.__limite
```

Dessa forma, não é possível acessar o limite das contas a partir de um código fora da classe (nem usando `cc1.limite` e nem usando `cc1.__limite`).

5 Exercício

1. Implemente os códigos mostrados neste tutorial, contendo as classes Pessoa, Conta e ContaCorrente.

2. Adicione um novo método para transferir dinheiro entre duas contas.

3. Crie uma segunda ContaCorrente, e tente transferir dinheiro entre elas.