Universidade Federal de Viçosa
Departamento de Informática
Centro de Ciências Exatas e Tecnológicas

INF 101 – Introdução à Programação II

Classes

Classes e Objetos

- Todos os tipos de dados que já vimos do Python até agora (`int`, `float`, `bool`, `str`, `list`, `tuple`, `dict`, `set`, `file`), desde os mais simples aos mais sofisticados, são todos, na realidade, *classes*
- Os valores desses tipos são *objetos*
- As operações que fazemos com os valores são *métodos*
- A partir desta aula, vamos aprender a definir os nossos próprios tipos usando o constructo denominado `class`


Exemplos

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type('Olá, gente!')
<class 'str'>
>>> type([1, 2, 3, 4, 5])
<class 'list'>
```


Programação Orientada a Objetos

- A programação orientada a objetos é uma técnica de programação que organiza os programas em classes e objetos ao invés de usar apenas funções
- O assunto é muito importante e extenso merecendo muito estudo e prática para ser completamente compreendido
- O objetivo desta aula é somente apresentar o básico da orientação a objetos de modo a introduzir os conceitos e estimular o aprendizado desta técnica


Objetos

- Podemos entender um objeto em Python como a representação de um objeto do mundo real
- Essa representação é modelada pela quantidade de detalhes que podemos ou queremos considerar
- Este modo de agir na construção de modelos do mundo real é chamado, em computação, de abstração


Objetos

- Por exemplo, um aparelho de televisão, o televisor, tem uma marca e um tamanho de tela
- Podemos também pensar no que podemos fazer com o aparelho, por exemplo, mudar o canal de recepção, ligar ou desligar o aparelho
- Vamos ver, a seguir, como podemos modelar isto em Python


Modelagem de um Televisor

```
class Televisor:
    def __init__(self):
        self.ligado = False
        self.canal = 2

tv = Televisor()
print(tv.ligado)
print(tv.canal)
tv_sala = Televisor()
tv_sala.ligado = True
tv_sala.canal = 4
print(tv.canal)
print(tv_sala.canal)
```


Modelagem de um Televisor

- Para criar uma nova classe, usamos a palavra-chave `class` para indicar a declaração de uma nova classe
- Quando declaramos uma classe, estamos criando um novo tipo de dados
- Este novo tipo de dados define seus próprios métodos e atributos
- Os atributos que definimos para a classe `Televisor` na transparência anterior foram: `ligado` e `canal`


Modelagem de um Televisor

- E o método definido foi o construtor que é denotado sempre, em Python, pelo identificador `__init__`
- Por sua vez, o identificador `self` denota um objeto instanciado da classe
- Observe que `self` foi declarado como parâmetro do construtor, mas ele pode ser usado em qualquer outro método ou função
- Os objetos que foram criados (instanciados) a partir desta classe são: `tv` e `tv_sala`


Exercício

- Adicione os atributos `tamanho` e `marca` à classe Televisor
- Crie dois objetos `Televisor` e atribua marcas e tamanhos diferentes a eles
- Depois imprima o valor desses atributos de modo a verificar a independência dos valores de cada instância


Modelagem de um Televisor

- Vamos ver agora como modelar um comportamento à classe `Televisor`, definindo dois novos métodos: `mudaCanalParaCima` e `mudaCanalParaBaixo`


Modelagem de um Televisor

```
class Televisor:
    def __init__(self):
        self.ligado = False
        self.canal = 2

    def mudaCanalParaCima(self):
        self.canal = self.canal + 1

    def mudaCanalParaBaixo(self):
        self.canal = self.canal - 1
```


Uso da Classe Televisor

```
>>> tv = Televisor()
>>> tv.mudaCanalParaCima()
>>> tv.mudaCanalParaCima()
>>> tv.canal
4
>>> tv.mudaCanalParaBaixo()
>>> tv.canal
3
```


Passagem de Parâmetros

- Um problema com nossa classe `Televisor` é que não controlamos os limites dos canais
- Na realidade, podemos obter valores esdrúxulos: canais negativos ou canais de número muito alto, como 32765 ou -2, por exemplo
- É claro que isto está incongruente com um televisor real
- Precisamos melhorar nosso modelo!
- Vamos modificar o construtor de modo a receber o canal mínimo e o máximo suportados pelo televisor


Passagem de Parâmetros

```
class Televisor:
    def __init__(self, min, max):
        self.ligado = False
        self.canal = 2
        self.cmin = min
        self.cmax = max

    def mudaCanalParaCima(self):
        if self.canal + 1 <= self.cmax:
            self.canal += 1

    def mudaCanalParaBaixo(self):
        if self.canal - 1 >= self.cmin:
            self.canal -= 1
```


Uso da Classe Televisor

```
tv = Televisor(1, 99)
for x in range(120):
    tv.mudaCanalParaCima()
print(tv.canal)
for x in range(120):
    tv.mudaCanalParaBaixo()
print(tv.canal)
```


Exercícios

1. Atualmente, a classe `Televisor` inicializa o canal com 2. Modifique a classe `Televisor` de modo a receber o número do canal inicial em seu construtor
2. Modifique a classe `Televisor` de modo que, se pedirmos para mudar o canal para baixo, aquém do mínimo, ele vá para o canal máximo. Se mudarmos para cima, além do canal máximo, que volte ao canal mínimo. Ou seja, as mudanças de canal ficarão rodando dentro do intervalo `cmin` e `cmax`


Banco de Varejo

- Vamos usar como outro exemplo de uso de classes a modelagem de um banco de varejo
- Um banco de varejo mantém contas correntes para pessoas físicas poderem realizar depósitos e saques de dinheiro
- Cada conta corrente pode ter um ou mais clientes
- Cada cliente terá apenas o nome e o número de seu telefone de contato


Banco de Varejo

- A conta corrente apresenta o saldo e a lista de operações de saques e depósitos realizados até o momento
- Quando um cliente fizer saque, o saldo de sua conta será diminuído do valor do saque
- Quando fizer depósito, o saldo de sua conta será aumentado do valor do depósito
- Por ora, o banco não oferece contas especiais que poderiam ter saldos negativos

Banco de Varejo

- Vamos começar criando uma classe para representar cliente
- Como dissemos, cliente é muito simples, tem apenas dois atributos: nome e telefone

```
class Cliente:
    def __init__(self, nome, telefone):
        self.nome = nome
        self.telefone = telefone
```


Banco de Varejo

- Suponha que vamos colocar as definições das classes que modelam bancos de varejo em um arquivo-fonte próprio denominado `bancos.py`
- Assim para usarmos a classe `Cliente` que está definida lá, podemos escrever:

```
from bancos import Cliente
cliente1 = Cliente("Ana Lopes", "3123-4567")
cliente2 = Cliente("Rui Castro", "3234-6789")
```


Banco de Varejo

- Para resolver o problema do banco, precisamos definir outra classe agora para representar uma conta corrente do banco com seus clientes e respectivo saldo
- A classe `Conta` é definida recebendo clientes, número da conta e saldo em seu construtor `__init__`, onde `clientes` é uma lista contendo os correntistas da conta; considere o número da conta como um *string* ou um inteiro
- A classe ainda terá os métodos: `resumo`, `saque` e `deposito` cujo significado é o que se espera


Banco de Varejo

```
class Conta:
    def __init__(self, clientes, numero, saldo=0)
        self.clientes = clientes
        self.numero = numero
        self.saldo = saldo

    def resumo(self):
        print("Cc nº %s Saldo: %10.2f" %
              (self.numero, self.saldo))
```

Banco de Varejo

- Continuação da classe `Conta`:

```
def saque(self, valor):
    if self.saldo >= valor:
        self.saldo -= valor

def deposito(self, valor):
    self.saldo += valor
```


Exercício

- Escreva um programa de teste para a classe `Conta` supondo uma conta conjunta dos clientes João Silva e Maria Silva; invente um número para a conta
- Faça alguns testes como por exemplo:

```
conta.resumo()
conta.saque(1000.00)
conta.resumo()
conta.saque(50.00)
conta.resumo()
conta.deposito(234.56)
conta.resumo()
```


Banco de Varejo

- Embora nossa conta corrente comece a funcionar, porém ainda não temos um registro das operações realizadas na conta
- Esse registro pode ser implementado como uma lista que, na realidade, representará o extrato da conta
- Sendo assim, vamos alterar a classe `Conta` de modo a refletir este acréscimo da lista contendo o extrato da conta


Banco de Varejo

```
class Conta:
    def __init__(self, clientes, numero, saldo=0)
        self.clientes = clientes
        self.numero = numero
        self.saldo = 0.00
        self.operacoes = []
        self.deposito(saldo) # primeira operação

    def resumo(self):
        print("Cc nº %s Saldo: %10.2f" %
              (self.numero, self.saldo))
```

Banco de Varejo

- Continuação da classe Conta melhorada:

```
def saque(self, valor):
    if self.saldo >= valor:
        self.saldo -= valor
        self.operacoes.append(("SAQUE", valor))

def deposito(self, valor):
    self.saldo += valor
    self.operacoes.append(("DEPÓSITO", valor))
```


Banco de Varejo

- Continuação da classe Conta melhorada:

```
def extrato(self):
    print("Extrato Cc nº %s\n" % self.numero)
    for op in self.operacoes:
        print("%10s %10.2f" % (op[0], op[1]))
    print("\n     Saldo: %10.2f\n" % self.saldo)
```


Exercício

- Altere seu programa de teste para imprimir o extrato de cada conta
- Por exemplo, use o seguinte:

```
from bancos import Cliente, Conta
cliente1 = Cliente("João Silva", "3345-7890")
cliente2 = Cliente("Maria Silva", "3345-7890")
conta1 = Conta([cliente1], 1, 1000.00)
conta2 = Conta([cliente1, cliente2], 2, 500.00)
conta1.saque(50.00)
conta2.deposito(300.00)
conta1.saque(190.00)
conta2.deposito(95.26)
conta2.saque(245.00)
conta1.extrato()
conta2.extrato()
```


Banco de Varejo

- Para modelar banco, precisamos de mais outra classe
- Agora para armazenar todas as contas de um banco
- Como atributos, teremos o nome do banco e a lista de suas contas
- Como métodos, teremos a operação de abertura de conta corrente e a listagem de todas as contas do banco


Banco de Varejo

```
class Banco:

    def __init__(self, nome):
        self.nome = nome
        self.contas = []

    def abreConta(self, conta):
        self.contas.append(conta)

    def listaContas(self):
        for c in self.contas:
            c.resumo()
```


Exercício

- Faça o seguinte teste das classes que criamos até agora:

```
from banco import Cliente, Conta, Banco
cliente1 = Cliente("João Silva", "3456-7890")
cliente2 = Cliente("Maria Silva", "3456-7890")
cliente3 = Cliente("José Vargas", "2351-1809")
contaJM = Conta([cliente1, cliente2], 76534,
                100.00)
contaJ= Conta([cliente3], 80297, 10.00)
banco1 = Banco("Tatu")
banco1.abreConta(contaJM)
banco1.abreConta(contaJ)
banco1.listaContas()
```


Exercícios

- Altere o programa de modo que uma mensagem de saldo insuficiente seja exibida caso haja tentativa de sacar mais dinheiro que o saldo disponível
- Modifique o método `resumo` da classe `Conta` para exibir o nome e o telefone do(s) cliente(s)
- Crie uma nova conta tendo João Silva e José Vargas como clientes e saldo inicial igual a 500.00

Herança

- A metodologia de programação orientada a objetos permite modificar nossas classes adicionando ou modificando atributos e métodos de uma classe já existente
- A nova classe é chamada de *classe derivada* ou *subclasse* e a classe original é chamada de *classe base* ou *superclasse*
- E a capacidade de fazer isso é chamada de *herança*


Herança

- Herança é um dos conceitos fundamentais do paradigma de orientação a objetos
- Para ilustrar o conceito, vamos tomar nosso banco de varejo que, para atrair novos clientes, passou a oferecer contas especiais aos clientes
- Uma conta especial permite ao cliente sacar a descoberto até um determinado limite de crédito pré-aprovado
- As operações de depósito, extrato e resumo serão as mesmas de uma conta normal
- Para tanto, vamos criar a classe `ContaEspecial` que herda o comportamento da classe `Conta`


Herança

```
class ContaEspecial(Conta):

    def __init__(self, clientes, numero,
                 saldo=0.00, limite=0.00):
        super().__init__(clientes, numero, saldo)
        self.limite = limite

    def saque(self, valor):
        if (self.saldo + self.limite) >= valor:
            self.saldo -= valor
            self.operações.append(("SAQUE", valor))
```


Herança

- Faça o seguinte teste:

```
from banco import Cliente, Conta, ContaEspecial, Banco
cliente1 = Cliente("João Silva", "3456-7890")
cliente2 = Cliente("Maria Silva", "3456-7890")
cliente3 = Cliente("José Vargas", "2351-1809")
contaJM = ContaEspecial([cliente1, cliente2], 76534, 100.00, 500.00)
contaJ = Conta([cliente3], 80297, 10.00)
banco1 = Banco("Tatu")
banco1.abreConta(contaJM)
banco1.abreConta(contaJ)
banco1.listaContas()
contaJM.saque(50.00)
contaJ.deposito(300.00)
contaJM.saque(190.00)
contaJ.deposito(95.26)
contaJ.saque(245.00)
contaJM.extrato()
contaJ.extrato()
```


Exercício

- Na classe `Banco`, modique a estrutura de dados de armazenamento das contas para dicionário em vez de lista como proposto na transparência 32
- As chaves do dicionário devem ser os números das contas
- Observe que os respectivos comandos nos métodos `__init__`, `abreConta` e `listaContas` têm de ser adequadamente modificados para lidar com dicionário
- Para a listagem das contas em `listaContas`, imprima-as em ordem crescente do número da conta