Introdução à Inteligência Artificial

Módulo 01: Introdução à Inteligência Artificial (IA)

Professor: Dr. João Paulo R. R. Leite

Sumário

Introdução ao Python

Objetivo principal

- O **objetivo** desta aula é introduzir a sintaxe de **funções** e **classes** em Python. Através dessas ferramentas, o programador será capaz de tornar o código mais **organizado** e modularizado. Além disso, funções e classes contribuem para a **reutilização** do código.

Funções em Python

Uma função funciona exatamente como um programa. Ela é **executada sequencialmente**, pode possuir **entradas** (parâmetros) e **saída** (retorno) e, dentro delas, é possível utilizar localmente todas as estruturas que vimos até agora (variáveis, listas, loops, condicionais, chamadas para outras funções, etc.). Para criar nossas próprias funções, utilizamos a palavra reservada def.

Não esqueça dos **dois pontos** e da indentação!!

```
def print_menu():
    print("1 - Cadastrar Produto")
    print("2 - Alterar Produto")
    print("3 - Excluir Produto")
    print("4 - Sair do Programa")

# Programa Principal
print("Programa de Gerenciamento:")
print_menu()
```

A função ao lado **não recebe nenhum parâmetro** de entrada e nem retorna nada na saída. É apenas utilizada para impressão de mensagens na tela.

```
Programa de Gerenciamento:
1 - Cadastrar Produto
2 - Alterar Produto
3 - Excluir Produto
4 - Sair do Programa
```

```python
# calcula a tensao, dados os valores de resist e corrente
def voltage(resist, current):
    volt = resist*current
    return volt

# Program Principal
i = float(input("Entre com a corrente: "))
r = float(input("Entre com a resistencia: "))

print("Tensao = ", voltage(i,r))
```

Aqui, a função recebe **dois parâmetros** e **retorna** um valor para o programa principal.

```
Entre com a corrente: 1.5
Entre com a resistencia: 10
Tensao =  15.0
```

Também é possível retornar mais de um valor através da função. De duas formas:

```python
# return list of values
def fibs(num):
    result = [0,1]
    for i in range(2, num):
        a = result[i-1] + result[i-2]
        result.append(a)
    return result

n = int(input("How many elements in sequence? "))
print(fibs(n))
```

```
How many elements in sequence? 10
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```python
import math # wait... what is this?

# returning more than one value
def circle_info(rad):
    area = math.pi*(rad**2)
    perim = 2*math.pi*rad
    return area, perim

r = float(input("Enter circle radius: "))
a, p = circle_info(r)
print("Area = %f, Perimeter = %f" %(a,p))
```

```
Enter circle radius: 10
Area = 314.159265, Perimeter = 62.831853
```

```python
# calcula a tensao, dados os valores de resist e corrente
def voltage(resist=10.0, current=5.0):
    volt = resist*current
    return volt

print("Default values (V) =", voltage())
print("Default value - 1st (V) =", voltage(20))
print("By name (V) =", voltage(current=10.0, resist=2.0))
```

```
Default values (V) = 50.0
Default value - 1st (V) = 100.0
By name (V) = 20.0
```

Quando fizer sentido para sua função, é possível ainda determinar **valores-padrão** (*default*) para os parâmetros de entrada, como na figura ao lado. Caso o usuário não especifique outro valor, o valor "default" será utilizado.

Repare ainda que é possível passar parâmetros fora de ordem, especificando por nome (última linha).

```python
def fun(x,y): #does it modify its values?
    x, y = 10, 20
    print("Inside funcion: ", x, y)

xx, yy = 1, 2
fun(xx, yy)
print("Outside funcion: ", xx, yy)
```

```
Inside funcion:  10 20
Outside funcion:  1 2
```

Importante notar que os valores originais das variáveis passadas como parâmetro para a função **não podem ser modificados** dentro dela. É como a passagem "por valor" de C/C++.

x e y contém apenas os valores de xx e yy, mas não representam as mesmas células de memória.

Programação **Orientada a Objetos**

Até agora, aprendemos as estruturas básicas do Python para fluxo do programa, definimos funções para organização do código e utilizamos os tipos de dados integrados da linguagem.

Agora, iremos aprender a **definir nossos próprios tipos.**

A **programação orientada a objetos** facilita a escrita e manutenção de nossos programas através da criação de classes e objetos. Classes são a definição de um **novo tipo de dados**, que associa **dados e operações** em uma única estrutura. Um objeto pode ser entendido como uma variável cujo tipo é definido por uma classe, ou seja, um objeto é uma instância da classe.

```python
class Television:
    def __init__(self):
        self.is_on = True
        self.channel = 3

tv = Television()
print("Is it on?", tv.is_on)
print("On channel", tv.channel)
```

```
Is it on? True
On channel 3
```

A classe ao lado define um tipo de dados "**Television**", que modela de maneira simplificada um aparelho de TV do mundo real. Uma classe é como se fosse um molde, a partir do qual é possível construir muitas TVs com seus próprios atributos. Algumas estarão ligadas, outras podem estar desligadas. Algumas no canal 3, outras no 5 ou no 13. Chamamos cada uma dessas tvs de "instância".

No código, foi criada uma nova instância chamada "tv" a partir da classe "Television". A tv está ligada, no canal 3.

Algumas considerações:

1. Utilizamos a palavra reservada class para indicar a declaração de uma nova classe e : (dois pontos) para iniciar seu bloco de instruções. A seguir, são definidos os seus métodos e atributos. Métodos são **funções membro** da classe e atributos são seus **membros de dados**.

2. Inicialmente, definimos um método chamado __init__, que funciona como **construtor** e é chamado toda vez que um objeto da classe Televisao for criado. É o construtor que inicializa nosso objeto (*INITialize*) com seus valores padrão. Ele recebe um parâmetro self, que é o objeto em si (**this**).

3. Dentro da classe, sempre que quisermos nos referir a um membro de dados da classe (ligada, canal), temos que associá-lo a self, como em "**self**.ligada". Se escrevermos apenas "ligada", o interpretador irá reconhecer este identificador como uma variável local, apenas do escopo do método __init__. Ao escrever duas variáveis ligadas ao self, criamos dois membros de dados: **ligada**, do tipo bool e **canal**, do tipo int. No momento da criação da variável "tv", **o método** __init__ **é chamado** e o objeto é inicializado com seus valores padrão (True e 3).

Podemos ainda incrementar a classe, passando parâmetros para o construtor e criando novos métodos:

```
class Television:
    def __init__(self, on=True, ch=3):
        self.is_on = on
        self.channel = ch
    def channel_up(self):
        self.channel += 1
    def channel_down(self):
        self.channel -= 1

tv = Television(False, 5) # initial values
print("Is it on?", tv.is_on)
print("On channel", tv.channel)

tv.channel_up() # method call
tv.channel_up()
tv.channel_down()
print("Now on channel", tv.channel)
```

```
Is it on? False
On channel 5
Now on channel 6
```

Nessa nova versão, o **construtor da classe pode receber valores** que serão colocados inicialmente nos atributos "*is*_on" e "*channel*". Além disso, há dois métodos para mudar o canal "para cima" e "para baixo"

Ao criar a tv, a inicializamos com "False", indicando que está com o visor desligado, mas indicamos que ela está no canal 5. A partir do próprio objeto criado, é possível chamar (*tv.channel_up()*) os métodos para mudança de canal, que refletem no valor do atributo "*channel*" daquele objeto.

Qualquer método pode também receber parâmetros. Vejamos **outro exemplo.**

Vamos escrever um programa para **controle bancário simplificado**, que tenha duas classes, Cliente e Conta. Veja o código:

```python
class Client:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
    def print_profile(self):
        print("%s (%s)" % (self.name, self.phone))

class Account:
    def __init__(self, client, number, balance = 0):
        self.balance = balance
        self.client = client
        self.number = number
    def summary(self):
        print("Owner:", self.client.name)
        print("Number = %s, balance = $%.2f" % (self.number, self.balance))
    def debit(self, val):
        if (self.balance >= val):
            self.balance -= val
    def credit(self, val):
        self.balance += val
```

```python
# creating clients
cl1 = Client("John", "555-1234")
cl2 = Client("Mary", "555-6789")

print("List of clients:")
cl1.print_profile()
cl2.print_profile()

#creating accounts
acc1 = Account(cl1, 1, 1000)
acc2 = Account(cl2, 2, 0)

print("\nAccounts:")
acc1.summary()
acc2.summary()

#managing account 1
acc1.debit(150)
acc1.credit(235)

print("\nAccount updated:")
acc1.summary()
```

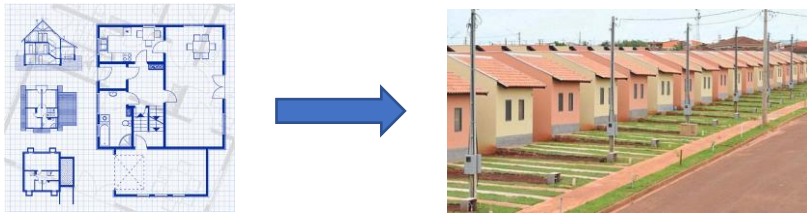
```
List of clients:
John (555-1234)
Mary (555-6789)

Accounts:
Owner: John
Number = 1, balance = $1000.00
Owner: Mary
Number = 2, balance = $0.00

Account updated:
Owner: John
Number = 1, balance = $1085.00
```

Repare como o código orientado a objetos é limpo e legível.

A partir de uma única classe, foram criados dois objetos independentes, cada um responsável por gerenciar seu próprio estado interno.



A POO traz grande poder para o programador, que passa a ter a capacidade de gerar "entidades" (classes) que contém **características** (atributos) e **comportamentos** (métodos) capazes de modelar objetos do mundo real de maneira a abstrair a complexidade inerente da realidade, focando apenas nos pontos interessantes para a sua aplicação.

Exercício para Casa:

Acrescente à classe "*Account*" a capacidade de gerar um extrato. Para isso, declare uma variável do tipo lista capaz de armazenar cada uma das operações de débito e crédito (em formato de str). Crie também uma função membro para imprimir todo o extrato. Dica: Utilize a função append() de listas e a utilize.

Herança

A orientação a objetos permite que **criemos novas classes a partir de classes já existentes**, modificando ou adicionando atributos e métodos. Isso é feito através do conceito de Herança.

A criação de uma **hierarquia de classes** (taxonomia) com relacionamentos de herança leva a um programa mais organizado, possibilita a reutilização de código e a utilização de algumas das técnicas mais avançadas de orientação a objetos, como o **polimorfismo**, que veremos a seguir.

Exemplo: Imagine que para nosso sistema bancário, queiramos criar um tipo de conta diferente, **Conta Especial**. Precisamos começar do zero? Para a conta especial, todas as funções de uma conta básica são mantidas, mas queremos que haja um limite de cheque especial (*overdraft*), que permita que saquemos mais dinheiro do que há em conta no momento.

```python
class SpecialAccount(Account): #inherits from Account
    def __init__(self, client, num, bal = 0, overd = 0):
        Account.__init__(self, client, num, bal)
        self.overdraft = overd
    def debit(self, val):
        if (self.balance + self.overdraft >= val):
            self.balance -= val

# creating clients and accounts
cl1 = Client("John", "555-1234")
cl2 = Client("Mary", "555-6789")

acc1 = Account(cl1, 1, 1000)
acc2 = SpecialAccount(cl2, 2, 1000, 200)

print("\nAccounts:")
acc1.summary()
acc2.summary()

#managing account 1
acc1.debit(1100)
acc2.debit(1100)

print("\nAccounts updated (after attempt to debit $1100):")
acc1.summary()
acc2.summary()
```

```
Accounts:
Owner: John
Number = 1, balance = $1000.00
Owner: Mary
Number = 2, balance = $1000.00

Accounts updated (after attempt to debit $1100):
Owner: John
Number = 1, balance = $1000.00
Owner: Mary
Number = 2, balance = $-100.00
```

Ao criar a nova classe, **ela não começa "do zero"**. Ela já parte do ponto de partida em que possui todos os atributos e métodos da classe "Account", colocada entre parênteses.

Portanto, "SpecialAccount" já tem os atributos referentes ao cliente, saldo e número da conta, além dos métodos de saque e crédito.

A nova classe, portanto, precisa apenas **acrescentar o atributo** de "cheque especial", que chamamos de "overdraft" e **substituir o método** de saque (debit), para que ele leve isso em consideração.

Terminologia:
Account: classe **base** ou **superclasse**
SpecialAccount: classe **derivada** ou **subclasse**

SpecialAccount **é uma** *Account*, e pode ser tratada como tal, mas tem algo a mais.

Repare que a **conta especial** da Mary autorizou o saque de 1100 dólares, enquanto a conta básica do John não.

E se, ao invés de substituir um método, eu precisasse apenas **complementá-lo**? Ou seja, ele precisa fazer tudo o que já fazia "mais alguma coisa". Basta chamar inicialmente o método da classe base (incluindo **self** na chamada). Veja um exemplo, utilizando a função para imprimir o resumo da conta:

```
class SpecialAccount(Account): #inherits from Account
    def __init__(self, client, num, bal = 0, overd = 0):
        Account.__init__(self, client, num, bal)
        self.overdraft = overd
    def debit(self, val):
        if (self.balance + self.overdraft >= val):
            self.balance -= val
    def summary(self):
        Account.summary(self)
        print("Overdraft limit = $%.2f" % (self.overdraft))
```

```
Owner: Mary
Number = 2, balance = $-100.00
Overdraft limit = $200.00
```

Exercício para Casa:
Escreva uma classe que modele um **Cliente VIP**. O Cliente VIP herda todas as características de um cliente normal, mas possui acesso a uma linha de crédito, e tem um membro de dados que contém o valor máximo que ele pode pedir emprestado ao banco. Defina nesse cliente, uma função membro chamada empréstimo, que recebe uma conta bancária daquele mesmo cliente como parâmetro e credita nela o valor do empréstimo pedido pelo Cliente.

Polimorfismo

E o polimorfismo? Há suporte?

"Através dele, objetos de classes diferentes de uma mesma hierarquia de classes podem ser tratados de maneira igual em alguns cenários em que isso seja conveniente, mas respondem a uma **mesma mensagem** (chamada de método) de **maneiras diferentes**, segundo sua própria especificação."

O que aconteceria, portanto, caso declarássemos uma **lista de contas** e colocássemos nela tanto **contas comuns** quanto **contas especiais** sem qualquer ordem pré-concebida? E se percorrêssemos essa lista pedindo para que cada item **imprimisse o resumo** de seus próprios atributos? A resposta seria igual para a chamada de método ou diferente? A conta especial imprimiria seu limite?

```python
# creating clients and accounts
cl1 = Client("John", "555-1234")
cl2 = Client("Mary", "555-6789")
cli3 = Client("Peter", "777-8956")
cli4 = Client("Susan", "756-2364")

acc1 = Account(cl1, 1, 1000)
acc2 = SpecialAccount(cl2, 2, 1000, 200)
acc3 = Account(cli3, 3, 1000)
acc4 = SpecialAccount(cli4, 4, 10000, 200)

#create list of accounts
acc_list = [acc1, acc2, acc3, acc4]

print("Account summaries:")
for acc in acc_list:
    print("\nAccount:")
    acc.summary()
```

```
Account summaries:

Account:
Owner: John
Number = 1, balance = $1000.00

Account:
Owner: Mary
Number = 2, balance = $1000.00
Overdraft limit = $200.00

Account:
Owner: Peter
Number = 3, balance = $1000.00

Account:
Owner: Susan
Number = 4, balance = $10000.00
Overdraft limit = $200.00
```

Construímos uma lista com variáveis de classes diferentes (*Account* e *SpecialAccount*), relacionadas entre si por herança (todas “são” *Account*, no fim das contas). A seguir, a **mesma mensagem** foi enviada para cada um dos itens (“imprima seu resumo”), e **cada um respondeu e acordo com sua própria especificação**: polimorfismo.

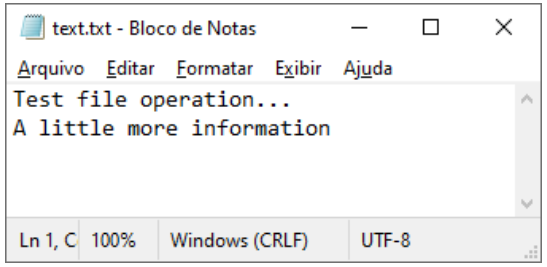
Entrada e Saída em arquivos:

Arquivos são utilizados para **armazenar dados permanentemente** em memória não volátil (por exemplo, um HD ou SSD). Em Python, operações envolvendo arquivos precisam seguir a seguinte ordem:

1) **Abrir** o arquivo;
2) **Ler** ou **escrever** nele;
3) **Fechar** o arquivo.

```
f = open("text.txt", 'w') # file mode: write
f.write("Test file operation...")
f.close()

f = open("text.txt", 'a') # file mode: append
f.write("\nA little more information")
f.close()
```



A função ***open*** recebe o nome do arquivo e o modo de abertura (w = escrita, r = leitura, a = *append*). Caso o arquivo seja aberto em modo de escrita (w ou a) e ele ainda não exista, um novo arquivo é criado. Se for no modo de leitura, o código gera um erro:

```
FileNotFoundError: [Errno 2] No such file or directory
```

```
f = open("text.txt", 'r')
content = f.read()
f.close()
print("File content:", content)
```

```
File content: Test file operation...
A little more information
```

```python
f = open("text.txt", 'w') # file mode: write
f.write("This is one line...")
f.close()

f = open("text.txt", 'a') # file mode: append
f.write("\nSecond line")
f.write("\nthird line")
f.write("\nfourth line")
f.close()

f = open("text.txt", 'r')
content = f.readline()
print("First line:", content)
content = f.readlines()
print(content)
f.close()
```

```
First line: This is one line...

['Second line\n', 'third line\n', 'fourth line']
```

No exemplo ao lado, vemos outra maneira de realizar a leitura do arquivo, utilizando as funções **readline()**, que retorna o conteúdo linha a linha, e **readlines()**, que retorna uma lista de strings em que cada um de seus elementos é uma linha de texto do arquivo lido.

Apoio

Este projeto é apoiado pelo Ministério da Ciência, Tecnologia e Inovações, com recursos da Lei nº 8.248, de 23 de outubro de 1991, no âmbito do [PPI-Softex| PNM-Design], coordenado pela Softex.

João Paulo Reus Rodrigues Leite
Universidade Federal de Itajubá
e-mail: joaopaulo@unifei.edu.br

UNIFEI

Softex

MCTI FUTURO
FUTURO DO TRABALHO, TRABALHO DO FUTURO

GOVERNO FEDERAL
BRASIL
UNIÃO E RECONSTRUÇÃO