
FIAP

PYTHON – AULA 5
LÓGICA DE PROGRAMAÇÃO APLICADA

AGENDA

Aula 1	Instalando e conhecendo o Python e as ferramentas do curso. Revisitando lógica de programação em Python.
Aula 2	Revisitando lógica de programação em Python – continuação.
Aula 3	Trabalhando com listas, tuplas e dicionários. Leitura e escrita de JSON.
Aula 4	Leitura e escrita de arquivos. Introdução a Numpy e Pandas.

AGENDA

Aula 5	Orientação a objeto no Python.
Aula 6	Trabalhando com os algoritmos. Ordenação e Recursão. – Pesquisa em largura.
Aula 7	Algoritmo de Dijkstra.
Aula 8	Algoritmos Genéticos.

PROGRAMANDO EM PYTHON

ORIENTAÇÃO A OBJETOS

PARADIGMA DA LINGUAGEM ORIENTADA À OBJETOS

A ideia da orientação a objetos é desenvolver programas baseados em objetos do mundo real.

Para onde você olhar irá se deparar com objetos, sejam eles animados ou inanimados: pessoas, animais, carros, motos, etc.

Os objetos possuem:

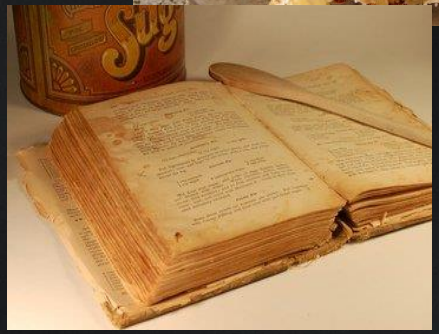
- Características - tamanho, forma, cor, etc
- Comportamentos – andar, falar, comer, etc.

Na linguagem O.O a unidade de programação é a CLASSE a partir do qual os objetos são instanciados (criados)

CLASSES

A classe é o projeto (receita) para a criação de objetos

- Atributos (variáveis) que são as características;
- Métodos (semelhantes a funções) que são os comportamentos do objeto a ser criado;



- A classe é a “receita” de um bolo e o objeto é o bolo.
- Pode-se fazer muitos bolos com uma única receita.
- Você não come a receita e sim o bolo.

CLASSES

- Definir uma classe em Python é simples, basta utilizar a palavra reservada class.
- Por convenção, classes sempre são definidas com maiúsculas. Em caso de nome composto, adota-se o camelCase.

```
class Cliente():
```

DECLARAÇÃO DE ATRIBUTOS

Para declarar os atributos de uma classe, basta especificar o nome do atributo por meio de variáveis em um método construtor.

```
def __init__(self, nome, endereco, cpf):
    self.nome = nome
    self.endereco = endereco
    self.cpf = cpf
```

O construtor é o método de inicialização do objeto.

OBJETOS

Para a instância de objetos, evocamos a classe e passamos os valores para os atributos definidos no construtor.

```
cliente_jose = Cliente("José", "Rua 123 de oliveira 4", "123456789")
print("Cliente: " + cliente_jose.nome)
print("Endereço: " + cliente_jose.endereco)

cliente_maria = Cliente("Maria", "Rua 123 de oliveira 4", "123456799")
print("Cliente: " + cliente_maria.nome)
print("Endereço: " + cliente_maria.endereco)
```

DECLARAÇÃO DE MÉTODOS

Os métodos são declarados por meio de funções.

Vamos imaginar que cada cliente criado tem um atributo “situação” que é definido com o valor booleano False no construtor.

```
class Cliente():

    def __init__(self, nome, endereco, cpf):
        self.nome = nome
        self.endereco = endereco
        self.cpf = cpf
        self.situacao = False
```

DECLARAÇÃO DE MÉTODOS

Vamos precisar de métodos para ativar e desativar o cliente quando necessário.

```
def ativar(self):
    self.situacao = True
    print("Cliente : ", self.nome + " ativado(a) com sucesso!")

def desativar(self):
    self.situacao = False
    print("Cliente : ", self.nome + " desativado(a) com sucesso!")
```

EXECUÇÃO DE MÉTODOS

Para executar, basta usar a variável que guarda a referência ao objeto e chamar / evocar o método escolhido.

```
cliente_jose = Cliente("José", "Rua 123 de oliveira 4", "123456789")
print("Cliente: " + cliente_jose.nome)
print("Endereço: " + cliente_jose.endereco)

cliente_jose.ativar()

cliente_maria = Cliente("Maria", "Rua 123 de oliveira 4", "123456799")
print("Cliente: " + cliente_maria.nome)
print("Endereço: " + cliente_maria.endereco)

cliente_maria.ativar()
```

ENCAPSULAMENTO

Uma das características da orientação a objetos é permitir o encapsulamento (proteção) dos atributos de uma classe.

Deste modo, só é possível manipular os atributos através de regras bem definidas implementadas nos métodos.

Existem três tipos de modificadores de acesso nas linguagens orientadas a objetos, que são:

- Public: atributos e métodos podem ser acessados e modificados em qualquer lugar do projeto;
- Private: atributos e métodos só poderão ser invocados, acessados e modificados por seu próprio objeto.
- Protected: atributos e métodos só poderão ser invocados, acessados e modificados por classes que herdam de outras classes.

ENCAPSULAMENTO

- Para definir um atributo como privado, adicionamos dois underlines (__) antes do nome do atributo ou do método.

```
class Cliente():

    def __init__(self, nome, endereco, cpf):
        self.__nome = nome
        self.__endereco = endereco
        self.__cpf = cpf
        self.__situacao = False
```

ENCAPSULAMENTO

- Basta fazer os ajustes nos métodos para a exibição dos atributos privados

```
def ativar(self):
    self.situacao = True
    print("Cliente : ", self.__nome + " ativado(a) com sucesso!")

def desativar(self):
    self.situacao = False
    print("Cliente : ", self.__nome + " desativado(a) com sucesso!")
```

ENCAPSULAMENTO

- Tentativas de acesso as variáveis privadas (não visíveis) retornam erro.

```
cliente_jose = Cliente("José", "Rua 123 de oliveira 4", "123456789")
print("Cliente: " + cliente_jose.nome)
print("Endereço: " + cliente_jose.endereco)
```

```
Traceback (most recent call last):
  File "C:\Users\Emerson Abraham\PycharmProjects\orientacao_objetos\Cliente.py", line 41, in <module>
    print("Cliente: " + cliente_jose.nome)
AttributeError: 'Cliente' object has no attribute 'nome'
```

ENCAPSULAMENTO

Para resolver este problema vamos criar métodos getter e setter para as variáveis.

```
def get_nome(self):
    return self.__nome

def set_nome(self, nome):
    self.__nome = nome

def get_endereco(self):
    return self.__endereco

def set_endereco(self, endereco):
    self.__endereco = endereco

def get_cpf(self):
    return self.__cpf

def set_cpf(self, cpf):
    self.__cpf = cpf
```

ENCAPSULAMENTO

Fazemos a chamada por meio das variáveis que referenciam os objetos.

```
cliente_jose = Cliente("José", "Rua 123 de oliveira 4", "123456789")
print("Cliente: " + cliente_jose.get_nome())
print("Endereço: " + cliente_jose.get_endereco())
```

Getter - leitura
Setter - escrita

Obs: devemos utilizar estes métodos com parcimônia, ou seja, não criar indiscriminadamente para todas as variáveis. É importante ver se faz sentido, pois os demais métodos da classe também permitem acesso as variáveis privadas.

COMPOSIÇÃO

A composição é um tipo de associação que conecta objetos de mais de duas classes.

Se pensarmos em uma classe Conta (conta de banco), quais atributos podemos definir?

Nome do cliente?

Cpf do cliente?

Endereço do cliente?

Espere, estes atributos são mesmo de Conta ou de um Cliente que deve pertencer a uma Conta?

COMPOSIÇÃO

- *É ai que entre a composição, ou seja, um tipo de vínculo forte entre o objeto todo (conta) e o objeto parte (cliente).*
- *Neste caso, separamos as responsabilidades, sendo que a classe Cliente ficará com dados relativos a cliente e a classe Conta com dados sobre a conta.*
- *Basta criar na classe Conta um atributo que recebe um objeto do tipo Cliente. Assim sendo, não conseguimos criar uma conta sem antes ter um cliente, faz sentido ?*

```
class Conta():
    def __init__(self, nome, endereco, cpf, numero_conta):
        self.cliente = Cliente(nome, endereco, cpf)
        self.saldo = 0.0
        self.numero_conta = numero_conta
```

Exercícios

1. Crie um novo projeto denominado orientacao_objetos.
2. Crie um arquivo banco.py
3. Defina a classe Cliente com o construtor e os atributos:
    - nome
    - endereço
    - cpf
    - telefone
    - declare os métodos getter / setter para todos os atributos

4. defina a classe Conta com o construtor e os seguintes atributos:
    - numero da conta
    - saldo
    - Cliente
    - declare métodos para depositar, sacar e exibir o saldo do cliente.
    - declare o método de transferência entre contas (transferência é saque de uma conta e depósito em outra)

5. Faça simulações, instanciando dois clientes e duas contas.

Exercícios

```python
class Cliente():

    def __init__(self, nome, endereco, cpf):
        self.__nome = nome
        self.__endereco = endereco
        self.__cpf = cpf
        self.__situacao = False

    def get_nome(self):
        return self.__nome

    def set_nome(self, nome):
        self.__nome = nome

    def get_endereco(self):
        return self.__endereco

    def set_endereco(self, endereco):
        self.__endereco = endereco

    def get_cpf(self):
        return self.__cpf

    def set_cpf(self, cpf):
        self.__cpf = cpf
```

```python
class Conta():
    def __init__(self, nome, endereco, cpf, numero_conta):
        self.cliente = Cliente(nome, endereco, cpf)
        self.saldo = 0.0
        self.numero_conta = numero_conta

    def depositar(self, valor):
        self.saldo += valor

    def sacar(self, valor)->bool:
        if(self.saldo>=valor):
            self.saldo -= valor
            return True
        return False

    def transferir(self, Conta, valor):
        teste = self.sacar(valor)
        if(teste==True):
            Conta.depositar(valor)

    def exibir_saldo(self):
        print("Cliente: " + self.cliente.get_nome())
        print(self.saldo)
```

Exercícios

```
75  conta_maria = Conta("Maria", "Rua 123 de oliveira 4", "123456799", 1)
76  print("Cliente: " + conta_maria.cliente.get_nome())
77
78  conta_jose = ContaCorrente("José", "Rua 123 de oliveira 4", "123456789", 2)
79  print("Cliente: " + conta_jose.cliente.get_nome())
80
81  conta_jose.depositar(1000)
82  conta_maria.depositar(2000)
83
84  conta_jose.sacar(500)
85  conta_maria.sacar(1000)
86
87  conta_maria.exibir_saldo()
88  conta_jose.exibir_saldo()
89
90  conta_jose.transferir(conta_maria, 100)
91
92  conta_maria.exibir_saldo()
93  conta_jose.exibir_saldo()
94
```

Herança e Polimorfismo

Na ciência da computação, a palavra CLASSE deriva de classificação e foi “emprestada” da biologia. O filósofo grego Aristóteles já fazia estudos de classificação de seres vivos durante o século IV A.c.

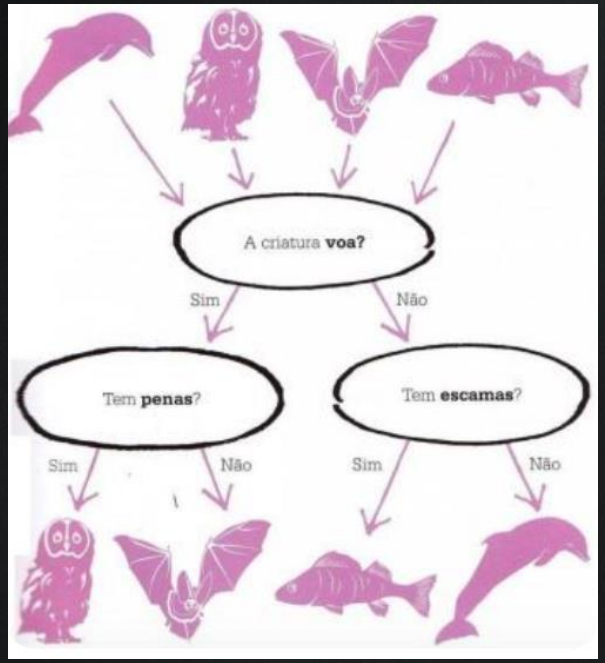


Imagem extraída de O Livro da Filosofia, 2011

A ideia por trás de classificação envolve a busca de características comuns entre os seres vivos. Na biologia o ramo que se ocupa deste trabalho é chamado de Taxonomia.

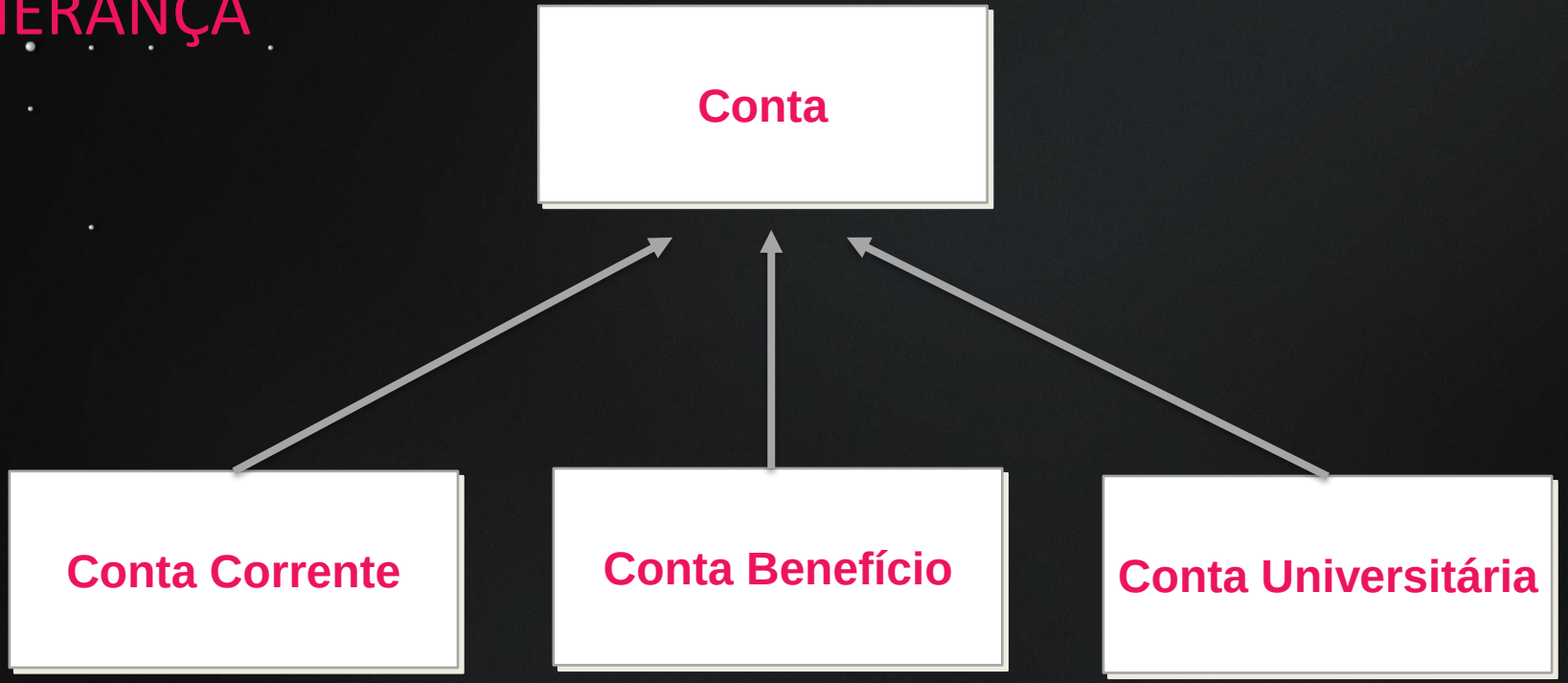
HERANÇA

- Partindo de características genéricas para mais específicas, podemos desenvolver projetos que irão demandar reutilização de software de qualidade (testado e depurado), economizando tempo e facilitando a manutenção.

“A Herança é uma forma de reutilização de software na qual uma nova classe é criada, absorvendo membros de uma classe existente e aprimoradas com capacidades novas ou modificadas” (Deitel, pg 301)

Vejamos um exemplo aplicado ao nosso projeto de banco

HERANÇA



A classe mais genérica denominada Conta é chamada de superclass (mãe) e as classes mais específicas de subclass (filhas).

HERANÇA

- Classe mãe (CONTA), com atributos e métodos característicos (genéricos) para todas as contas.

- As demais contas, herdam características e métodos, que por sua vez podem oferecer recursos para outras classes que venham a herdar destas (herança múltipla)

- Para a utilização da herança, é importante a elaboração de um projeto que abranja a complexidade do sistema e suas possíveis extensões futuras.

- Toda classe mãe deve ser mais genérica possível, afim de abranger as características de suas classes filhas, evitando-se inconsistências.

- Pode “engessar” o código.

HERANÇA

- Para implementar a herança, basta criar a *superclass* e definir que a *subclass* está herdando por meio do construtor

```
class ContaCorrente(Conta):
    def __init__(self, nome, endereco, cpf, numero_conta):
        super().__init__(nome, endereco, cpf, numero_conta)
        self.saldo_investimento=0.0
```

- Como Python possui tipagem dinâmica, precisamos nos certificar, informando os parâmetros da *superclass* e evocando o construtor de Conta no construtor de ContaCorrente

Exercícios

No projeto banco, vamos implementar um classe ContaCorrente que herda de Conta.

1. Crie a classe ContaCorrente e faça a herança, ajustando os construtores.
2. Implemente um novo atributo em conta corrente denominado saldo de investimento
3. Implemente o método de investir que recebe um tipo (string) e um valor para o investimento.
4. Se o investimento for do tipo 1, o valor deve ser sacado da conta do cliente e aplicado no saldo de investimento acrescido de 1%; se o investimento for do tipo 2 o valor aplicado será acrescido de 5%, senão serão acrescidos 10%.
5. Realize as simulações.

Exercícios

```python
class ContaCorrente(Conta):
    def __init__(self, nome, endereco, cpf, numero_conta):
        super().__init__(nome, endereco, cpf, numero_conta)
        self.saldo_investimento=0.0

    def investir(self, produto, valor):
        teste = self.sacar(valor)
        if(teste==True):
            if(produto==1):
                self.saldo_investimento += (valor*1.01)
            elif(produto==2):
                self.saldo_investimento += (valor * 1.05)
            else:
                self.saldo_investimento += (valor * 1.1)
```

Exercícios

banco.py

```
66            self.saldo_investimento += valor * 1.1
67
68    conta_maria = Conta("Maria", "Rua 123 de oliveira 4", "123456799", 1)
69    print("Cliente: " + conta_maria.cliente.get_nome())
70    print(conta_maria.saldo)
71
72    conta_jose = ContaCorrente("José", "Rua 123 de oliveira 4", "123456789", 2)
73    print("Cliente: " + conta_jose.cliente.get_nome())
74    print(conta_jose.saldo)
75
76    conta_jose.depositar(1000)
77    conta_maria.depositar(2000)
78
79    conta_jose.sacar(500)
80    conta_maria.sacar(1000)
81
82    conta_maria.exibir_saldo()
83    conta_jose.exibir_saldo()
84
85    conta_jose.investir(10, 100)
86
87    conta_jose.exibir_saldo()
88    print(conta_jose.saldo_investimento)
```

Polimorfismo

- Métodos que funcionam com tipos de dados diferentes são chamadas de polimórficos.
- Voltamos ao exemplo do método de investimento.

- Sempre que o banco definir um novo produto precisamos implementar mais um *if*.
- Nosso código vai crescer indefinidamente.

- E se ao invés de informarmos um número (mágico) para o tipo de produto, pudéssemos informar um produto (objeto) específico?

```
def investir(self, produto, valor):
    teste = self.sacar(valor)
    if(teste==True):
        if(produto==1):
            self.saldo_investimento += (valor*1.01)
        elif(produto==2):
            self.saldo_investimento += (valor * 1.05)
        else:
            self.saldo_investimento += (valor * 1.1)
```

Polimorfismo

- Basta criarmos uma interface IProduto apenas com a declaração do método; sem a implementação.

```
class IProduto(ABC):
    @abstractmethod
    def investir(self, valor):
        pass
```

- Fazemos herança de ABC - realize o importe: *from abc import ABC, abstractmethod*
- E marcamos o método com o decorador *@abstractmethod.* Esse decorador força a reescrita do método nas classes que irão implementar a interface

Polimorfismo

- Posteriormente, criamos uma classe para cada novo produto, implementamos a interface IProduto e reescrevemos o método investir, cada qual com sua própria regra de negócio.

```python
class CDB(IProduto):
    def investir(self, valor) -> float:
        return valor * 1.01

class LCA(IProduto):
    def investir(self, valor) -> float:
        return valor * 1.05

class Fundos(IProduto):
    def investir(self, valor) -> float:
        return valor * 1.1
```

Polimorfismo

- Por último, ajustamos o método de investir da classe ContaCorrente. Vamos renomear para calcular investimento.

```
def calcular_investimento(self, produto: Type[IProduto], valor):
    teste = self.sacar(valor)
    if(teste==True):
      self.saldo_investimento += produto.investir(valor)
```

- O método agora recebe um objeto do tipo produto que pode ser um CDB, LCA ou Fundos. Como os produtos definidos implementam IProduto, são um tipo de produto, portanto, temos um parâmetro versátil ou polimórfico.

- Para usar Type faça o import: from typing import Type

Polimorfismo

- Para simular, basta instanciar os produtos uma única vez, evocar o método de conta corrente, informando o valor e tipo de investimento.

```
cdb = CDB()
lca = LCA()
fundos = Fundos()

conta_jose.calcular_investimento(cdb, 200)
```

Polimorfismo

- O cálculo será efetuado em função da regra de negócio do objeto que está sendo enviado como argumento para o método.

- Agora quando surgir um novo produto bancário, basta a instituição criar uma classe, fazer a herança e reescrever o método. Muito mais limpo.

- Essa prática propicia alta coesão e baixo acoplamento e é conhecida por ser um *Design Pattern* de nome *Strategy*

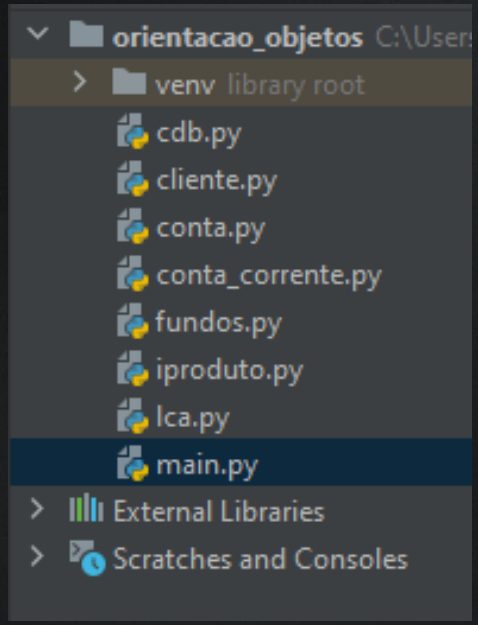
Exercícios

Faça uso do polimorfismo e do *design pattern Strategy.*

1. Elabore a interface IProduto com um método abstrato denominado investir
2. Defina cinco diferentes produtos bancários; crie as classes, implemente a interface IProduto e faça a reescrita do método investir
3. Elabore diferentes regras de negócio para os produtos.
4. Faça o ajuste no método da conta corrente para receber um produto.
5. Reescreva (customize) o método de exibirSaldo em ContaCorrente. O método deve exibir o saldo, saldo de investimento e saldo total da conta.
6. Faça diferentes simulações

Ajustes Finais

Vamos ajustar nosso código em uma estrutura de projeto. Separe as classes, interface e executável em arquivos diferentes. A estrutura final deve ficar conforme a imagem abaixo.



Faça os *imports,* conforme exemplo: *from conta_corrente import ContaCorrente*.
Execute novamente as simulações a partir do arquivo: main.py

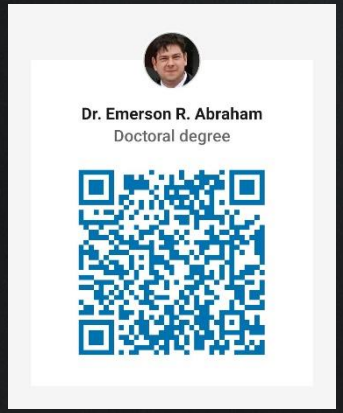
BIBLIOGRAFIA BÁSICA

- BEAZLEY, David. Python Essential Reference, 2009.
- BHARGAVA, ADITYA Y. Entendendo Algoritmos. Um guia ilustrado para programadores e outros curiosos. São Paulo: Ed. Novatec, 2017
- CORMEN, THOMAS H. et al. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2002
- COSTA, Sérgio Souza. Recursividade. Professor Adjunto da Universidade Federal do Maranhão.
- DOWNEY, ALLEN B. Pense em Python. Pense como um cientista da computação. São Paulo: Ed. Novatec, 2016
- GRANATYR, Jones; PACHOLOK, Edson. IA Expert Academy. Disponível em: https://iaexpert.academy/
- KOPEC, DAVID. Problemas clássicos de ciência da computação com Python. São Paulo: Ed. Novatec, 2019
- LINDEN, Ricardo. Algoritmos Genéticos. 3 edição. Rio de Janeiro: Editora Moderna, 2012
- MCKINNEY, WILLIAM WESLEY. Python para análise de dados. Tratamento de dados com Pandas, Numpy e Ipython. São Paulo: Ed. Novatec, 2018

BIBLIOGRAFIA BÁSICA

- TENEMBAUM, Aaron M. Estrutura de Dados Usando C. Sao Paulo: Makron Books do Brasil, 1995.
- VELLOSO, Paulo. Estruturas de Dados. Rio de Janeiro: Ed. Campus, 1991.
- VILLAS, Marcos V & Outros. Estruturas de Dados: Conceitos e Tecnicas de implementacao. RJ: Ed. Campus, 1993.
- PREISS, Bruno P. Estrutura de dados e algoritmos: Padrões de projetos orientados a objetos com Java. Rio de Janeiro: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. Lógica de programação e estruturas de dados. 2016.
- SILVA, Osmar Quirino. Estrutura de Dados e Algoritmos Usando C. Fundamentos e Aplicações. Rio de Janeiro: Editora Ciência Moderna, 2007.
- ZIVIANI, N. Projeto de algoritmos com implementações em pascal e C. São Paulo: Editora Thomsom, 2002.

OBRIGADO



FIAP


FIAP