

# Lógica de Programação com Python



## Boas Vindas!

Seja muito bem-vindo(a) ao curso de Lógica de Programação com *Python*. Nele abordaremos os principais conceitos relacionados ao pensamento lógico, a programação de computadores e a sua aplicação para a resolução de problemas cotidianos.

Problemas aliás, com os quais muitas vezes interagimos e se quer nos damos conta da lógica e estruturas de programação necessárias para a suas resoluções. Neste sentido, procuramos abordar de forma clara, conceitos que lhe possibilitarão desenvolver um pensamento lógico computacional, por meio da exposição de exemplos desenvolvidos com a linguagem *Python*.

Persista durante o decorrer de todo o curso, tenha em mente que ele é um importante “pontapé” inicial para o entendimento do pensamento lógico aplicado à uma linguagem de programação, e que essa base lhe servirá com um excelente alicerce para que o que virá à frente.

Enfim, com esse conteúdo, o qual preparamos com carinho, temos certeza de que você aprenderá bastante, portanto lhe desejamos um excelente aprendizado!

## 1. LÓGICA DE PROGRAMAÇÃO (COM PYTHON)

### 1.1 INTRODUÇÃO À PROGRAMAÇÃO E LÓGICA DE PROGRAMAÇÃO

Nesta seção, serão apresentados um breve histórico da “introdução à programação e da lógica de programação”.

#### 1.1.1 HISTÓRIA

A história da lógica de programação possui vários personagens importantes, entre eles destacam-se:

- **Ada Lovelace:** Na década de 1800, a programadora produziu notas sobre o conceito de programação de uma máquina denominada “Máquina Analítica” ou “Engenho Analítico de Babbage” que foi projetada por Charles Babbage engenheiro e inventor britânico. Augusta Ada Lovelace é considerada por muitos a primeira programadora da história e seus conceitos descritos nas referidas notas pavimentaram o início da programação de computadores.
- **Alan Turing:** Em 1930 o matemático e criptógrafo inglês propôs a “Máquina de Turing”, dispositivo teórico que se tornou base para o conceito moderno de algoritmo e computação. Turing contribuiu para a quebra do código da máquina nazista [Enigma<sup>1</sup>](#) durante a Segunda guerra mundial.
- **Konrad Zuse:** Na década de 1940, Zuse construiu sua máquina Z3, considerada o primeiro computador programável eletromecânico.
- **John Tukey:** Na década de 1950 foi cunhado por Tukey o termo “Software”. Nesta época, as primeiras linguagens de programação como Fortran e Cobol foram desenvolvidas.

- **Dennis Ritchie:** Em 1970, Ritchie do Bell Labs <<https://www.bell-labs.com/>> desenvolveu a linguagem de programação denominada C.
- **Década de 1980:** A revolução dos computadores pessoais trouxe linguagens como BASIC, Pascal e C++ para uma ampla gama de programadores.
- **Tim Berners-Lee:** Na década de 1990, a WWW - World Wide Web foi criada dando origem à programação para a mesma área.
- **Década de 2000 . . . :** A partir desta década, a programação se torna mais acessível com o crescimento de linguagens de alto nível como o Python, Ruby e JavaScript.

### FATO CURIOSO

Em forma de ficção científica e com fatos históricos, foram lançados dois filmes que exploram os feitos e conceitos trabalhados por Ada Lovelace e Alan Turing. O primeiro é o “**Conceiving ADA**” ou em português: **Conceber Ada** de 1997 e o “**The Imitation Game**” ou em português: **O Jogo da Imitação**” de 2014.



Figura 1: Capa do filme *The Imitation Game*.

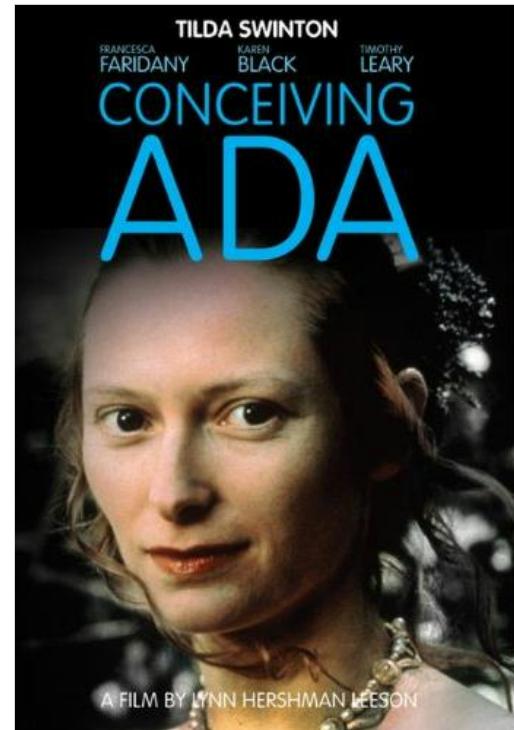


Figura 2: Capa do filme *Conceiving ADA*.

Mais informações sobre os filmes *The Imitation Game* e *Conceiving Ada* podem ser encontradas nos links do (IMDb - Internet Movie Database, 2023) em: <<https://www.imdb.com/title/tt2084970/>> e <<https://www.imdb.com/title/tt0118882/>>

<sup>1</sup> Foi uma máquina eletromecânica de criptografia/descriptografia muito utilizada pelos soldados alemães para transmitirem mensagens secretas. Ela utilizava rotores mecânicos e sistemas elétricos que em conjunto, os quais geravam 1 sextilhão de possibilidades. O mecanismo consistia de um teclado, um conjunto de discos rotativos “rotores” dispostos em fila e de um mecanismo de avanço que fazia movimentar alguns rotores para posições diferentes quando as teclas eram pressionadas.

## 1.2 ALGORITMOS: CONCEITOS E EXEMPLOS

Um **algoritmo** pode ser entendido como o sequenciamento de passos lógicos para a solução de um problema. Um exemplo é a observação das etapas necessárias para trocar uma lâmpada ou trocar o pneu de um carro apresentados através da Tabela 1 (BEZI, 2003).

Tabela 1: Exemplo simplificado de passos lógicos para se atingir objetivos.

TROCAR UMA LÂMPADA (1)	TROCAR UMA LÂMPADA (2)	TROCAR O PNEU DE CARR
Pegue uma escada.	Pegue uma lâmpada nova.	Afrouxar ligeiramente as porcas.
Posicione a escada embaixo da lâmpada.	Pegue uma escada.	Suspender o carro.
Pegue uma lâmpada nova.	Posicione a escada embaixo da lâmpada.	Retirar as porcas e o pneu.
Suba a escada.	Suba a escada.	Colocar o pneu reserva.
Retire a lâmpada velha.	Retire a lâmpada velha.	Apertar as porcas.
Coloque a lâmpada nova.	Coloque a lâmpada nova.	Abaixar o carro.
-	-	Dar o reaperto final nas porcas.

Esta modelagem em passos lógicos para a solução de problemas pode envolver ainda, um **pseudocódigo** e/ou um diagrama visualizado chamado de **fluxograma**. A construção dos códigos segue o ciclo do processamento de dados, o qual envolve **entrada, processamento e saída** (BEZERRA, 2003). Na Figura 3 a seguir, podemos visualizar as três etapas básicas para construção e execução de quaisquer algoritmos e/ou programas computacionais.



**Figura 3:** Ciclo do Processamento de Dados.

### Atenção

Em programação, possuímos várias formas de solucionar um mesmo problema. Na Tabela 1, pudemos visualizar a resolução de um problema de duas maneiras diferentes.

A etapa de **entrada de dados** envolve a captura de todos os recursos (dados) necessários para o cálculo ou processamento das informações que serão realizadas pelo programa. Um exemplo é o cálculo da média semestral, no qual primeiro precisamos inserir as notas (**Nota 1** e **Nota 2**) para que o programa as armazene e posteriormente gere um resultado.

O **processamento** é o tratamento das entradas (os cálculos que serão realizados), como por exemplo, sabemos que o cálculo da média é “**Media = (Nota 1 + Nota 2) / 2**” ou “**Média é igual a nota 1, mais a nota 2, divididas por 2**”. A **saída** é um processo mais simples, se trata da apresentação dos resultados que podem ser exibidos na tela do computador, armazenados em um banco de dados numa impressora ou mesmo na memória do computador.

O **pseudocódigo** é a estruturação da resolução do problema em uma sequência lógica, porém, de forma textual e o **fluxograma** representa o funcionamento do código de forma visual ou gráfica. As três etapas apresentadas podem ser classificadas como estruturais e não representam o código em execução, porém, podem ser utilizadas para melhor entender e solucionar o problema.

Na Tabela 2, pudemos visualizar a estrutura de um **pseudocódigo** aplicado na construção de uma solução à cálculo da média aritmética entre duas notas **N1** e **N2**.

### São elementos estruturais:

- **Algoritmo:** Nome do programa que será desenvolvido.
- **Var:** Declaração de variáveis que utilizaremos na solução. Podemos defini-las como as estruturas onde alocaremos as informações de entrada e saída do programa.
  - Ao final da declaração das variáveis, devemos utilizar o “**:**” e posterior a este símbolo, atribuir um **tipo de dado**.
  - **Possuímos os tipos:**
    - **Numéricos:** Podem ser divididos em **inteiros** e **reais**.
      - Os **inteiros** são os números inteiros, positivos e negativos, sem a existência de casas decimais (ou ponto flutuante) ou seja, a vírgula.

- Exemplo: **2**
- Os **reais** são os números que podem ter casas decimais (ponto flutuante).
- Exemplo: **2,0**
- **Literais:** São **variáveis** que armazenam texto.
- Exemplo: **Professor Eduardo**
- **Lógicos:** São **variáveis** que armazenam apenas duas informações, **verdadeiro** ou **falso**.

Estas definições serão mais detalhadas no item “**Tipos de Dados**”, mais a frente em nosso curso.

**Tabela 2:** Estrutura de pseudocódigo com exemplo aplicado para o cálculo de média.

Estrutura	Exemplo
<b>Algoritmo</b> <nome_do_algoritmo>; <declaração_de_variáveis>; <b>Início</b> <Corpo_do_algoritmo>; <b>Fim</b>	<b>Algoritmo</b> Calc_Media; <b>Var</b> N1, N2, Media: <b>real</b> ; <b>Início</b> <b>Leia</b> (N1, N2); <b>Media</b> $\leftarrow$ (N1+N2)/2; <b>Se</b> Media $\geq$ 7 <b>então</b> <b>Escreva</b> (“Aprovado”); <b>Senão</b> <b>Escreva</b> (“Reprovado”); <b>FimSe</b> <b>Fim</b>

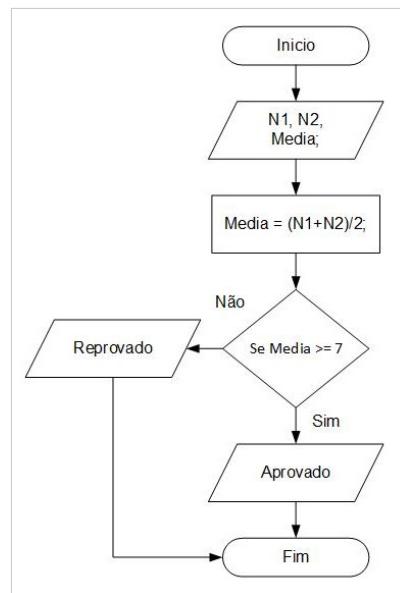
Com relação ao fluxograma ou modelagem visual da solução do problema, podemos utilizar as notações de representação das e descritas em processamento de dados. Na Tabela 3 podemos visualizar alguns símbolos e suas definições (BEZERRA, 2003).

**Tabela 3:** Definição de alguns símbolos utilizados no fluxograma de processos.

Símbolo	Definição
	<b>Início/Fim:</b> Símbolo de entrada e finalização do fluxograma.
	<b>Entrada/Saída:</b> Símbolo de entrada e saída de dados.
	<b>Processamento/Expressão:</b> Símbolo de cálculo e processamento de informações.
	<b>Estrutura condicional/Laço de repetição:</b> Estrutura condicional e de repetição de funções. Teste condicional.

Símbolo	Definição
→	<b>Fluxo de informações:</b> Indica o fluxo de informações/dados.

Na Figura 4 podemos visualizar um exemplo de construção de um fluxograma aplicado a modelagem de um *software* de cálculo de média, com aprovação caso a média calculada seja maior ou igual a sete “**Media  $\geq 7$** ” e reprovação caso contrário.



**Figura 4:** Exemplo de construção de um fluxograma de processos.

### Importante

Para a modelagem de uma solução, eliminação de incertezas e compreensão de um desafio, podemos utilizar várias formas de desenvolvimento, por exemplo, *brainstorming*, estrutura analítica de projetos, diagramas UML - *Unified Modeling Language* e os Métodos Ágeis os quais serão abordados ao longo deste e outros de nossos cursos. É importante ressaltar que podemos utilizar diferentes modelos e boas práticas para a resolução de problemas. Não devemos nos limitar, mas sim procurar utilizar as ferramentas mais apropriadas ao nosso propósito de alcançar os melhores resultados em nosso processo de desenvolvimento.

## 1.2.1 ALGORITMOS NATURAIS

Os **algoritmos naturais** envolvem padrões observáveis na natureza ou inspirados em eventos da natureza. São exemplos de algoritmos naturais:

- **Algoritmos Genéticos**
- **Algoritmo de Colônia de Formigas**
- **Algoritmo de Redes Neurais**

**Algoritmos genéticos** possuem a sua execução baseada na teoria da evolução de *Charles Darwin*. O objetivo é a utilização da seleção, cruzamento ou mesmo a mutação dos dados (População) para otimizarmos o resultado do problema analisado (GOLDBERG 1989). Para exemplificarmos a sua aplicação, podemos citar a escolha de rotas usando o GPS - *Global Positioning System*. Neste caso, quando o executamos, devemos passar por algumas fases:

- **Inicialização:** Criamos uma população inicial de rotas a partir da origem e destino.
- **Avaliação:** Analisamos as ramificações das rotas para chegarmos ao destino. Neste momento, verificamos a distância, desempenho e eficiência de cada caminho.

- **Seleção:** É o momento de selecionar as melhores rotas envolvidas. Com a seleção, produzimos uma nova geração (Opção) de As melhores rotas segundo os parâmetros apontados, como por exemplo, menor distância ou menor tempo possuem maior probabilidade de serem selecionadas.
- **Mutação:** As mutações podem ser caracterizadas por mudanças aleatórias nas rotas geradas. No exemplo trabalhado, podemos evidenciar com as alterações de rotas durante o percurso mediante aumento de tempo ou engarrafamento.
- **Gerações:** O processo de seleção e mutação poderá ser repetido várias vezes durante a execução. A geração chegará ao fim quando uma condição satisfatória seja alcançada, ou seja, a normalização do percurso.
- **Melhor solução:** É a que melhor atende aos critérios de desempenho, otimização e eficiência da aplicação que utiliza o GPS.

No caso dos **algoritmos de colônia de formigas**, funcionam baseados no comportamento das formigas. Neste contexto, em seu comportamento colaborativo a cada teste de caminho ou solução, as próximas intervenções possuirão resultados melhorados com o aprendizado das ações e erros anteriores (DORIGO e STÜTZLE, 2004). O funcionamento do algoritmo pode ser exemplificado através de uma situação problema onde alguns usuários pretendem encontrar o menor caminho (Mais eficiente) para se gerenciar uma central de logística (Entregas). Tal funcionamento envolve as etapas de:

- **Inicialização:** Os funcionários são posicionados em pontos de saída da cidade para melhor analisar o espaço de busca (Ruas).
- **Construção de soluções:** Cada funcionário realizará um teste em sua rota baseada em cálculos de probabilidade para encontrar a melhor rota (Tempo) e o melhor caminho (Curto).
- **Avaliação:** As rotas geradas são avaliadas de acordo com métricas previamente definidas (tempo, distância, dificuldades, riscos, etc.).
- **Atualização de feromônio:** De forma colaborativa, os usuários comunicam as soluções que geram os melhores resultados (Entregas de pacotes).
- **Evaporação de feromônio:** À medida que o tempo passa, as variáveis coletadas mudam, pois rotas rápidas podem ficar congestionadas, serem interditadas e etc. Este evento é positivo, uma vez que os funcionários passam a testar novas rotas e ganham novo conhecimento.
- **Iteração:** O processo de construção de soluções, avaliação e atualização de feromônio é repetido várias vezes.
- **Melhor solução:** A melhor solução encontrada durante as iterações é considerada a solução final.

**Algoritmos de redes neurais** são inspirados no funcionamento do cérebro humano. São utilizados para a aprendizagem de máquinas onde podemos utilizá-las para o reconhecimento de padrões, classificação de dados e previsão de informações. No exemplo abaixo aplicamos os algoritmos para prever eventuais atrasos (AGGARWAL, 2018). O algoritmo envolve as etapas:

- **Coleta de dados:** Nesta etapa analisamos as informações históricas sobre as entregas. Melhores caminhos, melhores horários, menor distância, análise de tempo e o histórico de atrasos em entregas.
- **Preparação de dados:** Os dados são classificados ou organizados, baseado na massa de dados (Quantidade), separamos para usar como referência para entender o comportamento e a outra parte para avaliar ou testar o resultado.
- **Validação e teste:** Com base nas informações levantadas, a solicitação de entrega é avaliada pelo algoritmo através da comparação com os dados levantados anteriormente.
- **Uso operacional:** Executamos o algoritmo e baseado nas informações coletadas no momento para realizar a entrega, verificar qual é a estimativa para realizá-la.
- **Melhorias contínuas:** Algoritmos de redes neurais podem ser ajustados continuamente de acordo com a chegada de novas informações. Estas informações melhorarão a precisão da solução.

## 1.2.2 LÓGICA DE RESOLUÇÃO DE PROBLEMA

Se trata da habilidade de decompor um problema complexo em atividades ou etapas menores. Veja o exemplo: quando definimos apenas “**Software de Controle de Vendas**”, possuímos dificuldade em começar a solucionar o problema através da programação apenas com essa informação. Na decomposição, quebramos o problema em etapas cada vez menores, até termos etapas muito simples.

## 1.3 ESTRUTURAS DE CONTROLE DE FLUXO: SEQUÊNCIA, DECISÃO E REPETIÇÃO

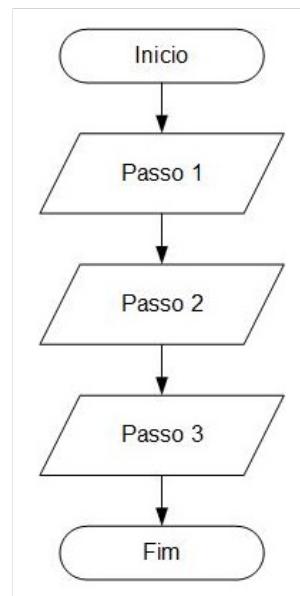
As estruturas de controle de fluxo são essenciais na programação para o direcionamento do fluxo de execução do programa de forma eficiente e lógica. Na linguagem *Python* possuímos ao menos três tipos de estruturas de controle de fluxo:

- **Sequência**
- **Decisão**
- **Repetição**

Na **estrutura de sequência**, possuímos a forma mais básica de execução dos comandos inseridos no código. Nelas as instruções são executadas em ordem, ou seja, linha após linha. Na Figura 5 podemos visualizar as impressões que ocorrerão sequencialmente (RAMALHO, 2015) e na Figura 6, a estruturação do mesmo exemplo através de um fluxograma.

```
print("Passo 1")
print("Passo 2")
print("Passo 3")
```

**Figura 5:** Instruções de impressão linha a linha.



**Figura 6:** Exemplo de instruções sequenciais em fluxograma.

Como podemos visualizar, na Figura 5 as impressões na tela são realizadas a partir do comando `print("Passo N")`, sendo executadas sequencialmente linha a linha.

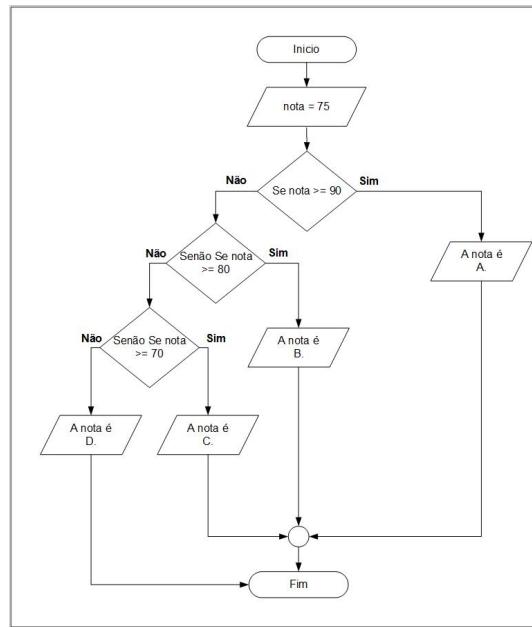
Em **estruturas condicionais**, possuímos instruções necessárias para o controle na tomada de decisões no programa, como por exemplo, o `if`, `elif`, e o `else` (RAMALHO, 2015).

O "`if`" pode ser lido como “**se**”, e neste ponto nos perguntamos, **se** a nota for maior? **Se** a idade for maior? O “`elif`” implica na pergunta, “**se a primeira condição**” não for verdadeira, a segunda será? E enquanto adicionarmos o comando, a pergunta se repete. Somente ao finalizar as condições ou perguntas, adicionamos o “`else`” que pode ser lido como “**senão**” para findar a estrutura condicional (RAMALHO, 2015).

Na Figura 7 podemos visualizar um exemplo de estrutura condicional apresentado em código Python e na Figura 8, a estruturação do mesmo exemplo através de um fluxograma. O exemplo inicia com a declaração de uma variável que recebe o valor “75” e testa se a mesma é maior ou igual a “90” para imprimir o texto “A nota é A.”, senão se nota for maior ou igual a “80” para imprimir que “A nota é B.”, senão se a nota for maior ou igual a “70” o texto será “A nota é C.” senão, se o valor não se encaixar em nenhum teste anterior, imprimirá “A nota é D.”

```
nota = 75
if nota >= 90:
    print("A nota é A.")
elif nota >= 80:
    print("A nota é B.")
elif nota >= 70:
    print("A nota é C.")
else:
    print("A nota é D.")
```

**Figura 7:** Estrutura condicional aplicada em Python.

**Figura 8:** Exemplo de estrutura condicional implementada por fluxograma.

### 1.3.1 TESTE DE MESA

Os testes de mesa servem para testarmos as entradas, o processamento e a saída do algoritmo desenvolvido. Esta execução pode ser comparada a prova real que é realizada em matemática. É importante ressaltar que neste caso, é denominado de “**Teste de Mesa**”.

Na Tabela 4 podemos visualizar um exemplo de teste de mesa referente a um algoritmo de cálculo da média entre dois números.

**Tabela 4:** Exemplo de execução do teste de mesa para avaliarmos a dinâmica de entrada e saída.

Entrada: Número 1	Entrada: Número 2	Expressão	Saída
3	3	Media = (Numero 1 + Numero 2) / 2	3
5	7	Media = (Numero 1 + Numero 2) / 2	6
8	12	Media = (Numero 1 + Numero 2) / 2	10

### 1.3.2 COMANDOS DE ITERAÇÃO E REPETIÇÃO

As estruturas de repetição executam um bloco de comandos repetidamente. Em linguagem *Python* possuímos os laços **for** e **while** para a sua implementação. Com relação ao **for**, o utilizamos para iterar (Iteração quer dizer que a cada looping, leia outro elemento da lista) valores ou itens delimitados a um número específico de repetições (RAMALHO, 2015).

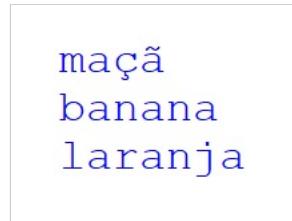
Na Figura 9, apresentamos um exemplo de aplicação do bloco de repetição do *loop for* para a impressão de uma lista de frutas. O conceito de **lista** será melhor detalhado no tópico “**Tipos de Dados**”.

```

frutas = ["maçã", "banana", "laranja"]
for fruta in frutas:
    print(fruta)
    
```

**Figura 9:** Exemplo de estrutura de repetição no *loop for* para a impressão de uma lista de frutas.

Para o exemplo acima, o **for** será executado em *looping* até que os nomes de todas as frutas sejam impressos na tela, ao final, o programa será encerrado. Na Figura 8 podemos visualizar o resultado da sua execução.



**Figura 10:** Saída do exemplo de repetição através do loop for /loop **for**.

O bloco de repetição **while** é executado enquanto uma condição for verdadeira (no exemplo da Figura 11 a condição é "**contado 5**"). A leitura do comando "**while**" pode ser realizada da seguinte forma, "**enquanto**" o estado da variável estiver verdadeira, o bloco executado, quando for falso, a execução será encerrada.

Na linha onde possuímos a variável denominada "**contador**" na figura 11, a mesma funciona como iterador de valor, ou seja, a cada *looping*, será somado um "**+= 1**" e quando ela for igual a "**5**", o programa será encerrado (RAMALHO, 2015).

Com os operadores "**+=**", a execução se torna equivalente a "**contador = contador + 1**".

```
contador = 0
while contador < 5:
    print("Contagem:", contador)
    contador += 1
```

**Figura 11:** Exemplo de estrutura de repetição **while** com o operando de soma.

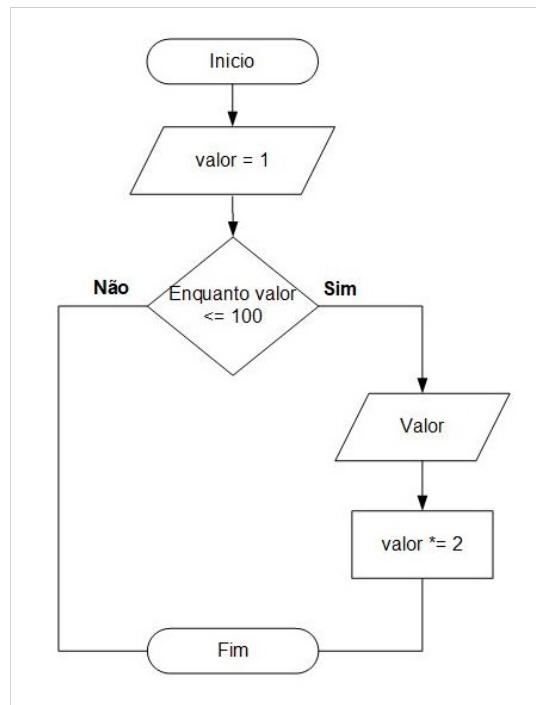
Nas Figuras 12 e 13, o bloco de repetição é executado até a condição ser atendida. Na linha denominada "**valor**", podemos observar a utilização de um iterador com o operador de multiplicação "**\***". Na prática, o código pode ser lido da seguinte forma, **enquanto** o "**valor**" for **menor ou igual** a "**100**", imprimiremos o valor e realizaremos o cálculo "**valor = valor \* 2**" (a expressão **valor \*= 2** é equivalente a **valor=valor\*2**). A Figura 14, ilustra o mesmo exemplo através de um fluxograma.

```
valor = 1
while valor <= 100:
    print("Valor:", valor)
    valor *= 2
```

**Figura 12:** Exemplo do bloco de repetição **while** executada em blocos com operador de multiplicação.

```
Valor: 1
Valor: 2
Valor: 4
Valor: 8
Valor: 16
Valor: 32
Valor: 64
```

**Figura 13:** Saída do código de execução do bloco de repetição **while**.

**Figura 14:** Exemplo de estrutura do bloco condicional **while** implementada por fluxograma.

Podemos combinar o bloco de repetição com o de decisão (**if**, **elif** e **else**), desta forma, podemos criar comportamentos complexos como os representados nas figuras 15 e 16. No exemplo, com a lista "**numeros**", executamos o *loop for* para testarmos cada valor da lista, verificando se é "**par**" ou "**ímpar**". Se o valor dividido por "2" não tiver resto da divisão igual a "0", então ele será "**é par**", senão será "**é ímpar**". A expressão "**if numero % 2 == 0:**" se lê da seguinte maneira: **se o resto da divisão do número por "2" for igual a**

```

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for numero in numeros:
    if numero % 2 == 0:
        print(numero, "é par")
    else:
        print(numero, "é ímpar")
  
```

**Figura 15:** Exemplo de estrutura de repetição e de decisão em um único código.

```

1 é ímpar
2 é par
3 é ímpar
4 é par
5 é ímpar
6 é par
7 é ímpar
8 é par
9 é ímpar
10 é par
  
```

**Figura 16:** Saída do código de execução do bloco de repetição **for** e do bloco de decisão **if** e **else**.

## 1.4 TIPOS DE DADOS

### 1.4.1 VARIÁVEIS, CONSTANTES E OPERADORES

Um dos primeiros conceitos a serem definidos em programação é o conceito de variável. O referido conceito é necessário para trabalharmos com os recursos de programação descritos até agora e que ainda serão contextualizados. No decorrer desta seção, trabalharemos as definições de variáveis, constantes e operadores.

- **Variável:** Se trata de um nome onde atribuímos valores ou recursos. Em Python, não é necessário declararmos explicitamente o tipo da variável, já em outras linguagens, precisamos colocar na declaração ou escrita da variável, o seu tipo, como por exemplo, **intnumero**). No exemplo da Figura 17, a variável "**idade**" e "**media\_de\_notas**" são do tipo inteiro e a variável "**nome**" é do tipo texto.

```
idade = 34
nome = "Eduardo"
media_de_notas = 10
```

**Figura 17:** Exemplo de atribuição de valores às variáveis em Python.

### Importante

Quando escrevemos "**idade = 34**", comunicamos que o valor "**34**" será alocado ou atribuído à variável "**idade**". Neste caso, se solicitarmos via o comando **print()** o valor de idade, a saída deverá ser "**34**". Outro ponto importante é que cada linguagem de programação possui palavras que não podemos utilizar em uma declaração de variável, por exemplo, não podemos criar uma variável chamada "**print**", pois ela é uma palavra reservada da linguagem *Python*. Veremos isso com mais detalhes à frente.

Em *Python* possuímos diversos tipos de dados em que podemos representar as informações. Os tipos de dados mais comuns são:

- **Números:**

- **int:** Tipo de dado utilizado em números positivos e negativos.
  - Utilizado para a trabalhar dados precisos, como por exemplo, a idade de uma pessoa.
- **float:** Tipo de dado utilizado para ponto flutuante, números fracionários/decimais.
  - Utilizado para a manipulação de informações que necessitam de maior precisão, como a altura e o peso.
- Na Figura 18 é possível visualizar a aplicação dos tipos de dados numéricos, onde o valor inteiro "**10**" é atribuído à variável "**numero\_inteiro**" e valor de ponto flutuante "**10.10**" à variável "**numero\_decimal**".

```
numero_inteiro = 10
numero_decimal = 10.10
```

**Figura 18:** Exemplo de implementação de variáveis numéricas em Python.

- **Textos:**

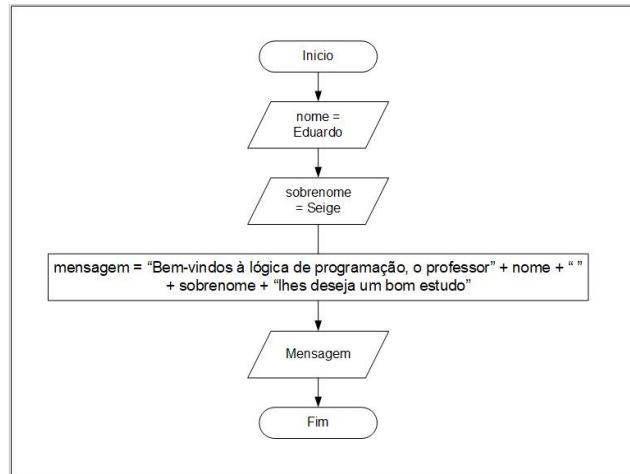
- **str:** Tipo de dado utilizado para armazenar e manipular informações textuais.
  - É importante ressaltar que o tipo de dado do tipo texto, pode ser utilizado nas mais diversas situações, como por exemplo no cadastro do nome das pessoas, campos de senha, passagem de informações para manipulação em páginas Web
- Nas Figuras 19 e 20, podemos visualizar o tipo de dado que trabalha texto em *Python*, na primeira linha atribuímos o texto "**Eduardo**" para a variável "**nome**", na segunda linha, "**Seige**" para a variável "**sobrenome**" e na terceira linha, a concatenação (Soma das variáveis de texto anteriores) e a sua atribuição na variável "**mensagem**". A Figura 21, ilustra o mesmo exemplo através de um fluxograma.

```
nome = "Eduardo"
sobrenome = "Sei"
mensagem = "Bem-vindos à lógica de programação,\n" +\
           "o professor "+\
           nome +\
           "" +\
           "\nLhes deseja um bom estudo."
print(mensagem)
```

**Figura 19:** Exemplo de concatenação de variáveis de texto utilizado na linguagem Python.

Bem-vindos à lógica de programação,  
o professor Eduardo  
lhes deseja um bom estudo.

**Figura 20:** Saída do exemplo de declaração de variáveis textuais e a concatenação de variáveis do tipo texto.



**Figura 21:** Fluxograma de implementação do código de inserção de texto e concatenação.

- **Booleanos:**

- **Booleanos ou Lógicos:** O tipo de dado booleano trabalha com apenas dois valores possíveis, o verdadeiro (**True**) ou falso (**False**).
  - Utilizamos este tipo de variável para validar se o programa pode continuar a em um fluxo de execução ou se algo é apresentado na tela, ou algum outro procedimento será executado.
  - **Exemplo:** Um usuário possui a necessidade de acessar um sistema, porém, se não tiver idade maior que 18 anos, seu acesso será bloqueado. Com as variáveis booleanas podemos testar, se "maior\_idade" for **True** então, "**permitir\_acesso**" será **True**. Na Figura 22 a aplicação em linguagem *Python* do tipo booleano, fica melhor exemplificada.

```

maior_idade = True
permitir_acesso = False
  
```

**Figura 22:** Exemplo de utilização do tipo de dado booleano em *Python*.

### Importante

Cada linguagem de programação possui um número finito de palavras reservadas. A linguagem *Python* possuí ao menos 33 palavras reservadas, as quais não podem ser utilizadas como nomes de funções, variáveis ou métodos.

Na Tabela 5, podemos visualizar as palavras reservadas da linguagem.

**Tabela 5:** Palavras reservadas da linguagem *Python*.

and	as	assert	break
class	continue	def	del

elif	else	except	False
finally	for	from	global
if	import	in	is
lambda	None	nonlocal	not
or	pass	raise	return
True	try	while	with
yield			

- **Constantes:** Em Python, as constantes são representadas como variáveis com letras maiúsculas para indicar que o valor não pode ser alterado. Na Figura 23 visualizamos dois exemplos de variáveis constantes. A "PI" e a "TAXA\_JUROS".

```
PI = 3.14159
TAXA_JUROS = 0.059
```

Figura 23: Exemplo de variáveis constantes em Python.

- **Operadores:** Os operadores são símbolos especiais que realizam operações em valores e variáveis. Os operadores podem ser aritméticos, de comparação, lógicos, de atribuição entre outros. No exemplo abaixo da Figura 24, podemos visualizar algumas operações aritméticas com operadores matemáticos.

```
soma = 7 + 3
subtracao = 10 - 3
multiplicacao = 7 * 3
divisao = 10 / 3
resto_divisao = 7 % 3
potencia = 7 ** 3
```

Figura 24: Exemplo de operadores aritméticos em Python.

- Os operadores de comparação são necessários para avaliarmos condições combinadas e conduzirmos a execução. Na Figura 25, podem ser visualizados os operadores de comparação.

```
variavel_A = 10
variavel_B = 10

igualdade = variavel_A == variavel_B
diferente = variavel_A != variavel_B
maior = variavel_A > variavel_B
menor = variavel_A < variavel_B
maior_igual = variavel_A >= variavel_B
menor_igual = variavel_A <= variavel_B
```

Figura 25: Apresentação dos operadores de comparação.

- Ainda com relação as informações da Figura 25, a seguir é explicado pontualmente como cada operador funciona.

- A "igualdade" ("==") avalia se a "variavel\_a" é igual a "variavel\_b".
- O operador "diferente" ("!=") avalia se a "variavel\_a" é diferente da "variável\_b".
- O operador "maior" (">") avalia se a "variavel\_a" é maior que a "variavel\_b".
- O operador "menor" ("<") avalia se a "variavel\_a" é menor que a "variavel\_b".
- O operador composto "maior\_igual" (">=") avalia se a "variavel\_a" é maior ou igual a "variavel\_b".
- O operador "menor\_igual" ("<=") avalia se a "variavel\_a" é menor ou igual a "variavel\_b".

### PENSE E RESPONDA

Se o nome à esquerda é o nome da variável e à direita é o valor atribuído. Qual seria a importância da utilização de um símbolo de igualdade e a utilização dos dois?

**Resposta esperada:** Um símbolo de igual "=" é uma atribuição e dois "==" é uma igualdade. Na situação apresentada acima, testamos se a "variável\_a" e a "variável\_b" possuem valores iguais e a resultante é atribuída à variável chamada "igualdade"

Ainda em relação às expressões matemáticas, é importante termos clareza de como se organiza a sua ordem de precedência das mesmas. Para facilitar, basta nos guiarmos pela sigla PEMDAS, ou seja: Parênteses, Exponenciação, Multiplicação, Divisão, Adição Subtração. As informações detalhadas podem ser visualizadas na Figura 26.



**Figura 26:** Ordem de execução dos operadores na construção e execução das expressões.

Com o conceito de precedência e organização de execução dos operadores em uma expressão matemática, podemos melhorar a composição da implementação da estrutura condicional definida pelo bloco **if**, **elif** e **else**.

Os operadores lógicos podem melhorar e aumentar a abrangência dos elementos testados em um bloco lógico. A tabela a seguir apresenta exemplos da implementação de portas lógicas ao código.

O "and" ou "e" se trata da estrutura onde duas condições são testadas e ambas têm que ser verdadeiras para gerar uma saída verdadeira.

No caso do "or" ou "ou", pelo menos uma das condições deve ser verdadeira para produzir uma saída verdadeira.

O operador "not" ou "não" (negação), é um inversor de polaridade/estado, ou seja, se a entrada for verdadeira, a saída será falsa; se a entrada for falsa, a saída será verdadeira. Na Tabela 6 podemos visualizar a relação das portas (entradas) apresentadas.

**Tabela 6:** Relação das portas lógicas aplicadas a situação programada.

Operador	Entrada 1	Entrada 2	Saída
E	Verdadeira	Verdadeira	Verdadeira
E	Verdadeira	Falsa	Falsa
OU	Verdadeira	Verdadeira	Verdadeira
OU	Verdadeira	Falsa	Verdadeira
NOT	Verdadeira	-	Falsa
NOT	Falsa	-	Verdadeira

Para o operador lógico “e” ou “**and**” podemos explicitar a seguinte situação, onde um jogador deve pressionar o caractere “W” e movimentar o mouse para direcionar o olhar de um personagem. Neste caso, a movimentação do mouse direciona o personagem e o “W” faz com que o personagem ande para frente.

Na situação do operador lógico “ou” ou “**or**”, imaginemos que o jogador possui duas formas de alterar as armas utilizadas pelo personagem, a tecla numérica “1” e também as teclas acima dos caracteres. Na situação declarada, o jogador pode utilizar apenas das opções para ativar o comportamento, pressionar as duas teclas ao mesmo tempo inviabilizaria a substituição.

O operador lógico de negação ou “**not**”, pode ser aplicado na ativação de uma lanterna, onde se a mesma estiver ativada/ligada pressionarmos um botão invertemos o seu sinal, ou seja, a desligamos, e na situação oposta se a mesma estiver desligada, a ativamos/ligamos.

Os operadores lógicos são usados na construção das estruturas condicionais e laços de repetição, a fim de inserir maiores testes de comportamento do sistema. Nas Figuras 27, 28, 29, 30, 31 e 32 podemos visualizar alguns exemplos do funcionamento das portas lógicas descritas até aqui.

```
a = True
b = False

resultado = a and b
print(resultado)
```

False

**Figura 27:** Porta lógica “**and**” em Python.

**Figura 28:** Saída do exemplo de execução da porta lógica “**and**”.

```
a = True
b = False

resultado = a or b
print(resultado)
```

True

**Figura 29:** Porta lógica “**or**” em Python.

**Figura 30:** Saída do exemplo de execução da porta lógica “**or**”.

```
a = True

resultado = not a
print(resultado)
```

False

**Figura 31:** Porta lógica “**not**” em Python.

**Figura 32:** Saída do exemplo de execução da porta lógica “**not**”.

Na construção de expressões complexas, podemos utilizar várias portas lógicas em sua construção, como por exemplo, a situação onde precisamos pressionar a tecla “W”, e movimentar o mouse para controlar a visualização do personagem, porém, ao mesmo tempo podemos também gerar o efeito desejado se alguma propriedade for ativada isoladamente, o exemplo ilustrado é a inversão da variável “z”. Nas Figuras 33 e 34 podemos visualizar o exemplo citado.

```

x = True
y = False
z = True

resultado = (x and y) or (not z)
print(resultado)

```

False

**Figura 34:** Saída do código de expressão complexa com a utilização de portas lógicas.

**Figura 33:** Exemplo de expressão complexa com portas lógicas.

A saída do exemplo acima é **False** uma vez que, da esquerda para a direita, avaliamos que "x =True" e "y =False" o que gera uma saída **False**. Como invertemos o "z = True", possuiremos então, uma saída **False**.

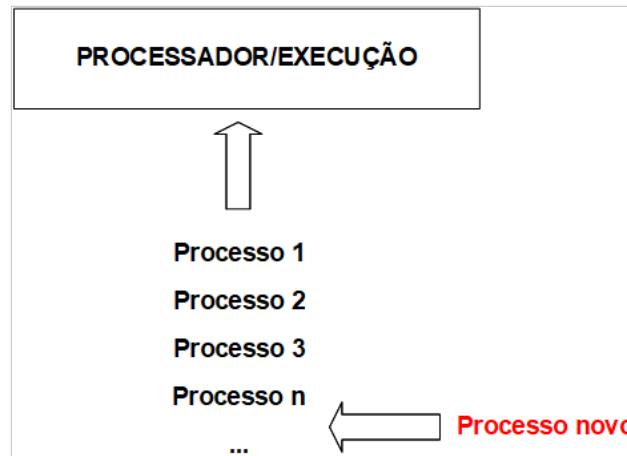
## 1.5 ESTRUTURAS DE DADOS COMPLEXAS

Neste tópico definimos estruturas de dados complexas e apresentamos alguns exemplos em linguagem *Python*. São estruturas consideradas complexas as **filas**, as **pilhas** e as **listas**.

As **filas** podem ser estruturas que seguem a relação FIFO - *First-in and First-out*, ou seja, o primeiro processo a ser declarado e executado é o primeiro a ser consumido e finalizado.

Um exemplo claro, é a fila para a compra ou retirada de qualquer coisa em nosso dia-a-dia. Com a necessidade de irmos retirar a dinheiro no banco ou mesmo retirar algum produto nos Correios, nos quais o controle é baseado pela ordem de chegada, portanto somos o primeiro a solicitar o produto ou estamos em primeiro lugar na fila, seremos o primeiro a consumir o serviço e atingir o objetivo.

A Figura 35 ilustra como é o funcionamento da estrutura de execução em fila.



**Figura 35:** Execução em estrutura de fila.

As **pilhas** são estruturas do tipo LIFO - *Last-in and First-out*, neste contexto, os últimos que forem incluídos na pilha serão consumidos primeiro. Um exemplo comum do nosso dia-a-dia é quando empilhamos um bloco de folhas sulfites e ao final precisamos utilizá-las, neste caso, é melhor consumirmos da última para a primeira, ou seja do topo para baixo.

Na Figura 36, podemos visualizar a estrutura de execução em **pilha** e comparar com a anterior.

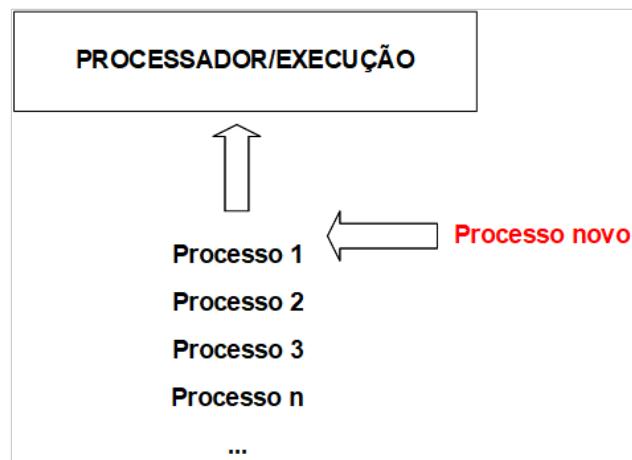


Figura 36: Execução em estrutura de pilha.

Por fim, temos as **listas**, que são estruturas utilizadas também na construção e armazenamento de informações mutáveis e ordenáveis ou na utilização de tuplas, possuímos a inserção de dados imutáveis.

É importante ressaltar que, as filas possibilitam que executemos as funções ou atribuições de recursos sem a necessidade de ocorrer uma vez que, possuímos a associação de cada conteúdo a um índice ou “index” (assim como no sumário de um trabalho escolar, os valores adicionados seguem uma ordem por posição).

Na construção de uma lista uma fila usando a estrutura de lista, adicionamos os dados ao final da sequência de informações armazenadas na lista e em estrutura de pilha, sempre ao início da lista. Em sua implementação, utilizamos o comando **push()** para inserir o elemento no topo da lista e o **enqueue()** ao final.

Na Figura 37, podemos visualizar um exemplo onde a execução/atribuição das informações não está limitada a uma ordem.

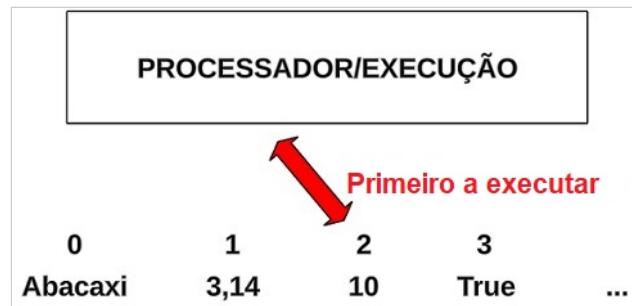


Figura 37: Exemplo de execução em lista.

- **Sequências:**

- **Lista:** Tipo de dado que é classificável como mutável (Podemos modificar o valor da lista) e ordenável (É organizada).
  - Quando utilizamos uma lista de informações em nosso algoritmo ou programa, podemos compará-la com a lista que levamos ao mercado, onde escrevemos os itens organizados num pedaço de papel. No exemplo apresentado na Figura 38, as variáveis "numeros" e "nomes" são listas.
  - As listas são delimitadas por colchetes "[ ]".
  - Elas alocam informações heterogêneas, ou seja, de diferentes tipos, como por exemplo, textos e números conforme o exemplo da variável "lista\_mista".

```

numeros = [1, 2, 3, 4, 5]
nomes = ["Eduardo", "Luis", "Carol"]
pi = 3.14
lista_mista = [1, "Eduardo", pi]
  
```

Figura 38: Exemplo aplicado do tipo de dado sequencial de listas em Python.

- **Tupla:** As **tuplas** são equiparáveis às listas, porém, são imutáveis (Não podem ser mudadas e, portanto, são constantes).
  - Imagine produzir uma tupla com informações sobre as taxas fixas de variação cambial (Cotação do dólar ou euro) para uma empresa do ramo financeiro ou fixar as taxas de juros ao pagar via cartão e/ou em dinheiro vivo (RAMALHO, 2015). No exemplo apresentado na Figura 39, a variável "**dados\_cadastro**" é apresentada.
  - As **tuplas** podem ser delimitadas por parênteses "( )".
  - Elas são heterogêneas, ou seja, assim como a lista, podem alojar valores de diferentes tipos. A principal diferença é que os valores das tuplas não podem ser alterados.

```
coordenadas = (1, 5)
dadosCadastro = ("Eduardo", 34,
                  "eduardo_sei@example.com")
```

**Figura 39:** Exemplo de aplicação do tipo de dado de sequência denominado “tupla” em Python.

- **Dicionário:** Dicionários são estruturas que podem ser alteradas durante a execução do código. Estas estruturas envolvem chaves-valor (RAMALHO, 2015).
  - Um dicionário funciona como uma lista telefônica, possuímos os campos (Chaves) e os valores (Valor) que são as informações associadas à chave.
  - Nas Figuras 40 e 41, podemos visualizar a construção e a exibição de um dicionário. Na variável "**cadastro\_usuario**" possuímos os campos "**nome**", "**idade**" e "**altura**". São valores, "Eduardo", "34" e "1.70". No exemplo da variável "**endereco**", possuímos as chaves, "**rua**", "**cidade**" e "**sigla**". Os valores são, "Rua A", "São Paulo" e "SP".
  - A delimitação dos dicionários é realizada através das chaves "{}".
  - O dicionário é homogêneo, ou seja, seu valor pode ser alterado. Com relação às estruturas anteriores, a diferença é a existência da relação chave-valor separadas por ":".

```
cadastro_usuario = {"nome": "Eduardo", \
                     "idade": 34, \
                     "altura": 1.70}

endereco = {"rua": "Rua A", "cidade": \
                     "São Paulo", \
                     "sigla": "SP"}

print(cadastro_usuario)
print(endereco)
```

**Figura 40:** Exemplo de estruturação de dicionário em Python.

```
{'nome': 'Eduardo',
 'idade': 34,
 'altura': 1.7}

{'rua': 'Rua A',
 'cidade': 'São Paulo',
 'sigla': 'SP'}
```

**Figura 41:** Saída do exemplo de execução de dicionário em Python.

Outras operações envolvem os operadores de "**in**" e "**not in**" para a análise ou teste de se um item está presente na lista citada

Na Figura 42 podemos visualizar a criação de uma lista numérica e a avaliação de se um item está presente ou não na mesma. Neste caso, é retornado à variável o valor de "True" ou "False".

```
presente_na_lista = 10 in \
[2, 4, 6, 8, 10]

nao_presente_na_lista = 11 not in \
[2, 4, 6, 8, 10]
```

**Figura 42:** Operações de análise de item presente ou não em um grupo.

### Dica

Você pode testar os códigos apresentados em nosso curso, sem ter que instalar a linguagem *Python* em seu computador. Para tanto basta acessar o IDE *online*, **OnlineGDB** <[https://www.onlinegdb.com/online\\_python\\_compiler](https://www.onlinegdb.com/online_python_compiler)>, sendo que o mesmo permite ainda que sejam executados e depurados programas também nas linguagens Java, C, C++, C#, HTML entre outras.

**Nota:** O *OnlineGDB* apesar de bastante prático, pode apresentar pequenos erros de interpretação em algumas situações particulares devido as diferentes versões existentes não só em *Python*, como em qualquer outra linguagem de programação. Como alternativa indicamos o **IDLE - Integrated Development and Learning Environment** ou Ambiente Integrado de Desenvolvimento e Aprendizagem o qual apesar de não fazer parte da linguagem *Python*, é disponibilizado conjuntamente em seus pacotes a cada liberação da mesma, desde a sua versão 2.3.

## 1.5.1 Arrays, Matrizes, Listas e Dicionários

Em *Python*, possuímos estruturas denominadas como arrays, matrizes e dicionários.

- **Arrays:** São estruturas homogêneas (mesmo tipo de dado), as quais são implementadas em *Python* através do módulo "array"
  - Algumas funções exigem bibliotecas (módulos que possuem funções, variáveis e recursos preestabelecidos), as quais auxiliam na criação e reutilização de códigos em outros programas.
  - **Arrays** podem ser definidas como estruturas unidimensionais, ou seja, possuem apenas uma linha de informações. Se possuirmos duas ou mais linhas (dimensões), possuiremos então, uma estrutura de **matriz** (linha e coluna).
  - Nas Figuras 43 e 44, podemos visualizar um exemplo de implementação de um **array**. A palavra "**import**" é utilizada para importarmos bibliotecas que possibilitarão por exemplo, usarmos esta estrutura (**Array**).

```
import array
primeiro_array = array.array('i', \
[1, 2, 3, 4, 5])

print(primeiro_array)
```

**Figura 43:** Exemplo de definição de array.

```
array('i', [1, 2, 3, 4, 5])
```

**Figura 44:** Saída do exemplo de execução de um array.

- **Matrizes:** Uma matriz pode ser definida como um vetor bidimensional, ou seja, possui informações para coordenadas. Neste caso vamos nomeá-las como "**linhas**" e "**colunas**".
  - Numa situação problema onde possuímos "2" linhas e "2" colunas, podemos identificá-las como uma matriz "2X2".
  - Sendo que em *Python*, podemos defini-la também como uma lista aninhada, por exemplo, definimos a estrutura padrão: "**= [[Coluna 1], [Coluna 2], [Coluna 3]]**". Nas Figuras 45 e 46, a matriz "**1X3**" implementada em *Python* pode ser observada.

```

matriz = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]

print(matriz)

```

**Figura 45:** Exemplo de definição de matrizes.

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

**Figura 46:** Saída do exemplo de execução das matrizes.

- Para melhor entender o contexto de estruturas complexas, podemos inserir matrizes em listas. Para esta situação, podem visualizar o exemplo a seguir, o qual utiliza uma lista, a processa via um laço "for" e a imprime item por item. Nas Figuras 48, podemos visualizar um exemplo aplicado para tal situação

```

lista = [["Arroz", "Feijao", "Salada"],
          ["Carne", "Frango", "Peixe"],
          ["Agua", "Suco", "Refrigerante"]]

for linha in lista:
    print(linha)

```

**Figura 47:** Estrutura de dados com uma matriz em uma lista.

```
'Arroz', 'Feijao', 'Salada']
['Carne', 'Frango', 'Peixe']
['Agua', 'Suco', 'Refrigerante']
```

**Figura 48:** Saída do exemplo de estrutura de dados com matriz de itens dentro de uma lista.

- Como já citado, listas são estruturas de dados complexos que permitem trabalharmos com dados heterogêneos, ou seja, de diferentes tipos, de forma mutável e ordenada.
  - Nas figuras a seguir podemos visualizar um exemplo de implementação de lista heterogênea, com inserção e remoção de elementos da lista.
  - O comando **append()**, realiza a inserção do valor "5". A inserção é realizada ao final da lista.
  - O comando **remove()** apaga o valor apontado e presente na lista.
  - A saída e o efeito da execução da lista (código implementado) desenvolvida pode ser visualizada nas Figuras 49 e 50.

```

primeira_lista = [1, 2, 3, "abc", True]
primeira_lista.append(5)
primeira_lista.remove(1)

print(primeira_lista)

```

**Figura 49:** Estrutura de dados heterogêneos.

```
[2, 3, 'abc', True, 5]
```

**Figura 50:** Saída do programa de estrutura de dados heterogêneos.

- **Dicionários:** São estruturas complexas com relação chave-valor e a propriedade de mutabilidade. Nas Figuras 51 e 52, podemos visualizar um exemplo de aplicação da estrutura de dicionários para a construção de um sistema de cadastro de usuário.

```
primeiro_dicionario = {"nome": "Eduardo", \
                       "idade": 34, \
                       "cidade": "São Paulo"}
print("Nome:", primeiro_dicionario["nome"])
print("Idade:", primeiro_dicionario["idade"])
print("Cidade:", primeiro_dicionario["cidade"])

primeiro_dicionario["Profissao"] = "Professor"
print(primeiro_dicionario)
```

**Figura 51:** Implementação de um dicionário a partir de uma lista e a inserção de um novo elemento.

```
Nome: Eduardo
Idade: 34
Cidade: São Paulo
{'nome': 'Eduardo',
 'idade': 34, 'cidade': 'São Paulo',
 'Profissao': 'Professor'}
```

**Figura 52:** Saída do programa de execução de um dicionário e lista de inserção de dados.

Ainda em dados complexos, agora vamos testar o funcionamento de **pilhas e filas**. É importante lembrarmos que as pilhas funcionam LIFO e as filas em FIFO. Na prática, ao estruturarmos uma aplicação onde possuímos várias funções, algumas de inserção e outras de eliminação de itens de nossa lista, dependendo da estrutura aplicada teremos um comportamento diferente.

Na Figura 53, estruturaremos uma pilha onde inseriremos elementos e no processo de eliminação, ao invés de eliminar a primeira entrada, o processo será executado da última cadastrada para a primeira.

```
# Criando uma pilha vazia usando uma lista
pilha = []

# Empilhando elementos na pilha
pilha.append(1) #Inserção de informações.
pilha.append(2) #Inserção de informações.
pilha.append(3) #Inserção de informações.

# Desempilhando elementos da pilha
item_desempilhado = pilha.pop()
print("Item desempilhado:", item_desempilhado) # Saída: 3

item_desempilhado = pilha.pop()
print("Item desempilhado:", item_desempilhado) # Saída: 2

# Verificando o topo da pilha
if len(pilha) > 0:
    topo_da_pilha = pilha[-1]
    print("Topo da pilha:", topo_da_pilha) # Saída: 1
else:
    print("A pilha está vazia.")
```

**Figura 53:** Execução de funções dentro de uma classe em estrutura de pilha.

No exemplo desenvolvido, visualizamos a aplicação do conceito de pilha que é utilizado na chamada de funções, as quais são necessárias para conseguirmos chamar as estruturas implementadas antes. No código podemos visualizar que inicialmente criamos variáveis, atribuímos os valores e apenas ao final, realizamos a movimentação dos números ou dados armazenados.

O código e suas funções podem ser definidos da seguinte forma:

- Na linha "`pilha = []`", criamos uma lista vazia.
- A função "`pilha.append()`" insere um valor na lista.
- O comando `pop()` remove o último item adicionado.
- O comando `len()` é a função que apresenta ou atribui o comprimento da variável apontada.
- O resultado pode ser visualizado na Figura 54.

```
Item desempilhado: 3
Item desempilhado: 2
Topo da pilha: 1
```

**Figura 54:** Saída da execução da função de empilhar e desempilhar.

## 1.6 Orientação a Objetos (OO): Conceitos e exemplos básicos

A orientação a objetos (OO), insere os conceitos de classe, atributo, método e objetos permitindo uma série de otimizações (melhorando por exemplo, a reutilização de estruturas de programação, a capacidade otimizada de manutenção e a organização do código)

Com a OO é possível por exemplo declararmos uma classe "`cadastro_de_login`", a qual é podemos reutilizar em programas distintos por exemplo, num programa de "*Gerenciamento de uma padaria*" ou num programa para o "*Gerenciamento de um supermercado*". Isto sem a necessidade de termos que reprogramar a classe "`cadastro_de_login`", ou seja, a mesma é reutilizada, o que sem a otimização disponibilizada pela OO não seria alcançada.

### 1.6.1 Conceitos Básicos de Orientação a Objetos

- **Classe:** As classes podem ser entendidas como a estrutura e/ou o comportamento de um objeto instanciado na memória. Uma classe pode ser comparada a um mapa ou planta-baixa de uma estrutura que será gerada.
  - **Exemplo:** Pessoa
- **Atributo:** São as características ou propriedades de um objeto. Pode ser um valor ou outra instância de uma classe.
  - **Exemplo:** Nome, Idade, Endereço, Telefone e etc.
- **Método:** São as ações ou os comportamentos observados da classe
  - **Exemplo:** Parado, Andando, Correndo e etc.
- **Objeto:** O objeto pode ser entendido como a classe instanciada na memória
  - **Exemplo:** O objeto possui as características atribuídas na execução do programa, como por exemplo, possuímos o objeto "Pessoa: Eduardo" e o objeto "Pessoa: Luis"

As Figuras 55 e 56 apresentam um exemplo de implementação dos conceitos de **classe, atributo, método e objeto**.

```
class usuario:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def mensagem(self):
        return f"Bem-vindo, meu nome é \
{self.nome} tenho \
{self.idade} anos."

pessoal = usuario("Eduardo", 34)
pessoal2 = usuario("Luis", 99)

print(pessoal.nome)
print(pessoal.idade)
print(pessoal.mensagem())
```

**Figura 55:** Exemplo de criação de objetos e utilização de classes.

No código acima possuímos alguns comandos bastante úteis, os quais detalhamos a seguir:

- O **class** define a classe criada.
- O **\_\_init\_\_** indica a inicialização da classe e seu construtor que passa os seguintes argumentos (**self**, **nome** e **idade**).
  - Métodos: **\_\_init\_\_** e **mensagem**.
  - Atributos: **nome** e **idade**.
- No caso do **self**, se trata da chamada de um objeto da própria classe.
- No **self.nome**, indicamos que o objeto da classe recebeu o atributo "**nome**".
- No **return**, passamos um texto que será impresso, porém, ao iniciar em **f**, podemos inserir uma formatação especial onde pode adicionar a chamada do "objeto-atributo" (**self.nome**) que estão entre os símbolos de chaves "**{ }{ }**".
  - **Atenção:** O "**f**" deve ficar conectado às aspas "**"**", uma vez que estamos aplicando o comando de formatação do texto em linguagem **Python** que será impresso na tela.

Na Figura 56 podemos visualizar a saída da execução do exemplo da classe **usuario**.

**Figura 56:** Saída do exemplo da classe **usuario**.

## 1.6.2 Conceitos de Herança, Polimorfismo e Encapsulamento

São conceitos relacionados a orientação a objetos, o encapsulamento de informações, a herança, o polimorfismo entre outros.

- **Encapsulamento:**

- É o conceito de proteger informações, controlando a visibilidade das mesmas, ou seja, funciona de modo a restringir o acesso a informações sensíveis e/ou compartilhar o que pode ser utilizado em outros locais (telas ou arquivos do programa). Abaixo definimos os principais tipos de encapsulamento.
  - **Públicos:** Visualização total por outras instâncias do programa. Assim como definido anteriormente com as "variáveis globais ou públicas / privadas e locais", atributos públicos definidos por exemplo numa tela de *login*, poderão ser reutilizados em outras partes do programa, como por exemplo, num arquivo de cadastro de clientes.
  - **Protegidos:** Podemos utilizá-las externamente, porém, não são abertos a todos como o público. A sua delimitação se dá através de pacotes. Os pacotes podem ser definidos como pastas onde colocamos os arquivos do programa. Quando declaramos uma variável como protegida, apenas os arquivos da mesma pasta podem chamá-la. É importante ressaltar que os conceitos de encapsulamento protegido, privado e abstrato serão melhor detalhados em outros cursos, como exemplo o de Programação **Python**.
  - **Privados:** A função ou classe que instancia o recurso ou atributo, consome a ação internamente e sem a possibilidade de um recurso externo o utilizar. Neste caso, apenas as funções ou chamadas que criaram o atributo podem executá-la.
  - **Abstratos:** Se trata da restrição aplicada em uma classe ou atributo na geração de um objeto. Quando declaramos com este encapsulamento, o impedimos de ser chamado na execução do programa.

Nas Figuras 57 e 58, podemos visualizar um exemplo de implementação do encapsulamento em **Python** onde:

- O **\_\_saldo** define um atributo reservado à chamada da classe.
- Os métodos **\_\_init\_\_**, **depositar**, **sacar** e **get\_saldo** são utilizados para movimentar os valores bancários, conforme o exemplo trabalhado.
- Através da declaração da classe **conta\_bancaria**, passamos o valor inicial de "**1000**".
- Posteriormente, podemos executar os métodos de **depositar** e **sacar**.
- As chamadas de **get\_saldo()**, são para mostrar que a cada execução, alteramos o valor do saldo inicial.

```
#Encapsulamento: Conta Bancária

class conta_bancaria:
    def __init__(self, saldo):
        self.__saldo = saldo

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

    def sacar(self, valor):
        if self.__saldo >= valor:
            self.__saldo -= valor
        else:
            print("Saldo insuficiente.")

    def get_saldo(self):
        return self.__saldo

conta = conta_bancaria(1000)
print("Saldo inicial:", conta.get_saldo())
conta.depositar(500)
print("Novo saldo:", conta.get_saldo())
conta.sacar(300)
print("Saldo após saque:", conta.get_saldo())
```

**Figura 57:** Exemplo de encapsulamento público para conta bancária.

Na Figura 58, podemos visualizar a saída do código implementado.

```
Saldo inicial: 1000
Novo saldo: 1500
Saldo após saque: 1200
```

**Figura 58:** Saída do código com implementação de encapsulamento público.

- **Herança:**

- O conceito de herança possibilita que pratiquemos o reuso de atributos da classe pai ou superclasse. **Exemplo:** Uma classe **pessoa** possui os atributos **nome** e **idade**. Neste contexto, a subclasse **funcionario** poderá herdar os atributos que classificam qualquer pessoa, como já definido anteriormente.
  - É importante ressaltar que na prática, a herança é o recurso de OO que nos possibilita reutilizar e simplificar o código de melhorar a sua organização.

Na Figura 59 possuímos um exemplo de relação de herança entre superclasses e subclasses, onde:

- O construtor **super().\_\_init\_\_()** da subclasse nos permite chamar alguns recursos da superclasse através de sua subclasse. No exemplo abaixo, dentro da subclasse **carro**, chamamos os atributos **marca** e **modelo**.
- É importante visualizar que, os atributos estão na superclasse e são acessados pela subclasse, ou seja, são reaproveitados sem a necessidade de redeclarar os atributos **marca** e **modelo** na subclasse.

```
#Herança: Veículo e Carro

class veiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

class carro(veiculo):
    def __init__(self, marca, modelo, cor):
        super().__init__(marca, modelo)
        self.cor = cor

    def info(self):
        print(f"Carro: {self.marca} {self.modelo}, Cor: {self.cor}")

# Uso das classes
carrol = carro("Toyota", "Corolla", "Prata")
carrol.info()
```

**Figura 59:** Exemplo de código de herança entre veículos gerados.

Na Figura 60, podemos visualizar o resultado da saída da execução do exemplo acima. Nele podemos visualizar a criação do objeto carro e suas informações.

```
Carro: Toyota Corolla, Cor: Prata
```

**Figura 60:** Criação do objeto carro, com as características de marca, modelo e cor.

- **Polimorfismo:**

- Conceito que possibilita possuirmos várias formas de acessar ou executar uma mesma função. **Exemplo:** Na construção do método denominado "falar" em uma classe animal, podemos possuir vários comportamentos diferentes do objeto original da mesma classe. **Exemplo:** Um cachorro "fala: Au au!" e um gato "fala: Miau!".

Nas Figuras 61 e 62, podemos visualizar exemplos de polimorfismo aplicado em Python.

```
#Polimorfismo: Animais

class animal:
    def falar(self):
        pass

class cachorro(animal):
    def falar(self):
        return "Au au!"

class gato(animal):
    def falar(self):
        return "Miau!"

# Uso das classes
animais = [cachorro(), gato()]

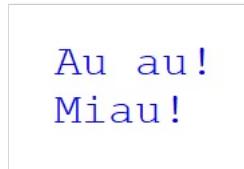
for animal in animais:
    print(animal.falar())
```

**Figura 61:** Exemplo de polimorfismo aplicado ao comportamento dos animais.

No código acima, podemos observar que foi definida a classe **animal** e o seu método **falar**. Note que utilizamos o comando **pass** que o programa não fique parado aguardando qualquer interação do usuário e continue a ler as outras classes do programa.

Posteriormente, definimos as classes **cachorro** e **gato**, para cada uma delas criamos um método **falar**, porém, quando o program instancia uma das classes, teremos um comportamento diferente, uma vez que o **return** de **cachorro** é “*Au au!*” e o de **gato** é “*Miau!*”

A figura a seguir apresenta o comportamento das classes chamadas, para a saída via comando **print**. Como chamamos **cachorro** e **gato**, o resultado será:



**Figura 62:** Comportamento de saída das classes implementadas.

A partir dos conceitos apresentados até aqui, podemos visualizar um outro exemplo com o qual são criadas novas classes, métod ocorrem o instanciamento (declaração) de alguns objetos na memória. O código é então apresentado na Figura 63, e as suas respe saídas na Figura 64.

```
class pessoa:
    def __init__(self, nome, idade):
        self._nome = nome
        self._idade = idade

    def apresentar(self):
        return f"Olá, meu nome é {self._nome} e tenho {self._idade} anos."

class estudante(pessoa):
    def __init__(self, nome, idade, curso):
        super().__init__(nome, idade)
        self._curso = curso

    def apresentar(self):
        return f"Oi, sou {self._nome}, um estudante de {self._curso}."

# Uso das classes
pessoa = pessoa("Eduardo", 34)
print(pessoa.apresentar())

estudante = estudante("Luis", 20, "Computação")
print(estudante.apresentar())
```

**Figura 63:** Exemplo de implementação dos elementos aprendidos até o presente momento.

Na Figura 64, podemos visualizar as informações de saída da execução do exercício de classes **pessoa** e **estudante**.

```
Olá, meu nome é Eduardo e tenho 34 anos.
Oi, sou Luis, um estudante de Computação.
```

**Figura 64:** Saída da execução da classe **pessoa**.

### Importante

Apresentadas as estruturas de base para a programação orientada a objetos, possuímos diferentes tipos de padrão de projeto (*Design Patterns*). Estes padrões evidenciam mais uma vez que em programação possuímos várias formas de resolução de um mesmo problema. Podemos utilizar diferentes recursos com diferentes efeitos, porém, que atingem a solução do problema trabalhado.

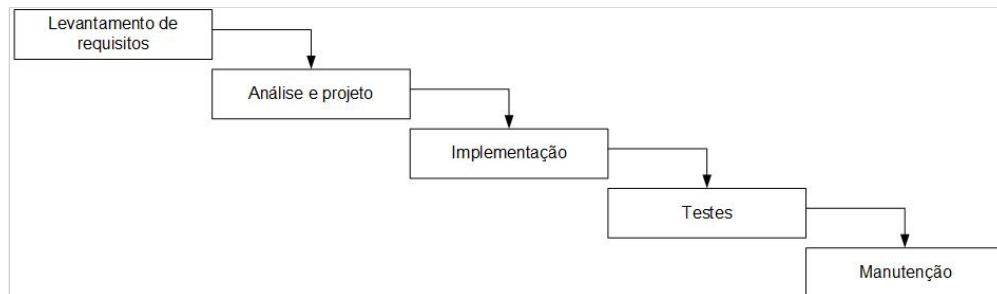
## 1.7 Metodologia Ágil: Papéis e Dinâmica

Antigamente utilizávamos uma abordagem de projeto para a solução de problemas de desenvolvimento de *software*. Como base tal, aplicávamos as boas práticas de gerenciamento de projetos ou mesmo os elementos contidos na UML, por exemplo. Neste cont com a evolução do dinamismo e velocidade de disseminação das informações, a aplicação destes conceitos gerava morosidade e dificuldade em se adequar às práticas mais diligentes e flexíveis exigidas ao longo dos últimos anos.

Como uma forma de solução, já há algum tempo dispomos de métodos ágeis onde diferentemente dos modelos de gestão mais i e de certa forma lineares, possuímos velocidade, iterações pequenas, entregas constantes, *feedbacks* próximos a realidade e um gr volume de mudanças.

Baseado nesta visão, possuímos entre outros, um *framework* bastante utilizado chamado **Scrum** - *Sprint, Cycle, Review, Update Meeting*. Popular entre equipes de empresas de todos os segmentos do mercado, sua aplicação em ambientes de desenvolvimento como objetivos centrais propiciar a otimização de recursos, a colaboração, a flexibilidade, a eficiência e a produtividade/rapidez.

À medida que observamos os modelos de desenvolvimento mais tradicionais, visualizamos a necessidade da implementação em e em outras a dificuldade de mudanças. Na Figura 65, possuímos o ciclo de vida em cascata, no qual os programas passam por um processo de levantamento de requisitos, análise e projeto, implementação, testes e manutenção (SABBAGH, 2016).



**Figura 65:** Ciclo de Vida em Cascata de desenvolvimento de software.

#### Saiba Mais

Para conhecer mais sobre Projetos Tradicionais e Métodos Ágeis, sugerimos que assista ao breve vídeo intitulado “Projetos Ágeis e Tradicionais: qual é a diferença?”, disponível no YouTube em:

< [https://youtu.be/5oFzY1\\_-pXI?si=uleV0cQscMsnSSHD](https://youtu.be/5oFzY1_-pXI?si=uleV0cQscMsnSSHD) >

### 1.7.1 Papéis no Scrum

- **Product Owner (Dono do Produto)**

- Representa os interesses do cliente e do usuário final.
- Define as funcionalidades e prioridades do produto.
- Mantém o *backlog*<sup>1</sup> do produto - uma lista de funcionalidades a serem desenvolvidas.
- *Scrum Master (Mestre Scrum)*:
  - Facilita o processo *Scrum*, garantindo que as práticas e princípios sejam seguidos.
  - Remove obstáculos que atrapalham o desenvolvimento da equipe.
  - Ajuda a equipe a melhorar continuamente.

- **Equipe de Desenvolvimento**

- Grupo de profissionais responsáveis por transformar os itens do *backlog* do produto em incrementos de *software* funcionais.
- Auto-organizadas, multidisciplinares e autogerenciadas.
- Tomam decisões técnicas e trabalham em colaboração.

<sup>1</sup> Uma lista de tarefas que contribui para o entendimento do escopo de um projeto, as suas prioridades e o andamento das suas *sprints* (conjuntos de atividades que duram períodos de tempo limitados).

## 1.7.2 Dinâmica do Scrum

- **Sprint:**

- Um ciclo de desenvolvimento de curto prazo (possui geralmente de 2 a 4 semanas) onde a equipe trabalha em um conjunto de funcionalidades.
- As funcionalidades são retiradas do *backlog* do produto e compõem o *backlog* da *sprint*.

- **Reunião de Planejamento da Sprint:**

- A equipe, o *Product Owner* e o *Scrum Master* definem quais itens do *backlog* do produto serão desenvolvidos na *sprint*.
- A equipe define como esses itens serão implementados.

- **Reuniões Diárias (Daily Standup):**

- Breve reunião diária onde a equipe compartilha o que fez no dia anterior, o que fará no dia atual e quais obstáculos estão enfrentando.

- **Revisão da Sprint:**

- Ao final da *sprint*, a equipe demonstra as funcionalidades desenvolvidas ao *Product Owner* e outros *stakeholders*<sup>1</sup>.
- Recebe *feedback* para ajustes.

- **Retrospectiva da Sprint:**

- A equipe analisa o que funcionou bem e o que pode ser melhorado na *sprint*.

- Define ações para aprimorar o processo.

- **Aplicação em Projetos: (Passos para solucionar o problema)**

- **Definir o Product Owner:**

- Identifique alguém que entenda as necessidades do usuário final e possa priorizar as funcionalidades.

- Escolha o *Scrum Master*:

- Alguém que facilite o processo, remova obstáculos e promova a cultura ágil.

- **Monte a Equipe de Desenvolvimento:**

- Reúna desenvolvedores *Python*, testadores, *designers*, etc., para formar uma equipe multifuncional.

- **Planeje Sprints:**

- Decida a duração das *sprints* e escolha os itens do *backlog* do produto para desenvolver.

- **Realize Reuniões Diárias:**

- Realize as reuniões diárias para acompanhar o progresso e identificar quaisquer bloqueios.

- **Realize Revisões e Retrospectivas:**

- Mostre as funcionalidades desenvolvidas ao *Product Owner* nas revisões.

<sup>1</sup> Pessoas, grupos ou organizações que impactam ou são impactadas por um projeto, ou seja, são os atores cuja participação é indispensável para a realização do mesmo.

### Dica

Para conhecer mais sobre Scrum e/ou buscar formações e certificações oficiais, você pode consultar duas de suas organizações oficiais: O [Scrum.org](https://scrum.org/) <<https://scrum.org/>> e o [Scrum-Institute](https://scrum-institute.org/) <<https://scrum-institute.org/>>, este último inclusive por vezes oferece cursos e livros gratuitos.

Outra referência muito útil sobre o framework é o livro “**Scrum: A Arte de Fazer o Dobro do Trabalho na Metade do Tempo**” do criador do Scrum, Jeff Sutherland.

## BREVE DESPEDIDA...

Chegamos ao final da jornada de nosso curso “**Lógica de Programação com Python**”, durante o mesmo pudemos visualizar conceitos gerais e específicos relacionados à lógica de programação.

Ao longo de nossa trilha, visualizamos conceitos desde o planejamento até a implementação. A partir de técnicas de eliminação de incertezas, levantamos elementos para a resolução do problema lógico (Decomposição) e posteriormente, aprendemos os principais conceitos de linguagens de programação, como por exemplo, variáveis, constantes e as principais características de orientação a objetos.

Ficou claro que a lógica de programação é uma etapa fundamental no processo de aprendizado e implementação de soluções computacionais, sendo a mesma uma habilidade essencial para o desenvolvimento de aplicações, independentemente da linguagem de programação a ser utilizada.

Trouxemos ainda de forma resumida, as características da gestão por boas práticas de projetos tradicionais mais antigos até a aplicação de métodos ágeis para o desenvolvimento de produtos, em especial os de software. E, por fim, discorremos sobre as características do framework *Scrum* e das entregas comuns obtidas através dos métodos ágeis atuais.

Esperamos que ao concluir este curso, o seu entendimento sobre os conceitos sobre o pensamento lógico, a programação e a importância da gestão de projetos, tenham se solidificado e que os mesmos sejam apenas um primeiro passo, rumo a um futuro de muito sucesso e realizações para você na área computacional.

Parabéns por chegar até aqui, e pelo o que ainda está por vir!

## Tags do conteúdo

Algoritmo, Lógica, Fluxograma, Pseudocódigo, UML, Teste de Mesa.

Tipos de Dados, Variáveis, Constantes, Operadores, Instâncias, Atribuições.

Fila, Lista, Pilha, Matriz, FIFO, LIFO.

Blocos de Repetição, Blocos de Sequência, Blocos de Decisão.

Classe, Herança, Objeto, Encapsulamento, Polimorfismo.

Tipos de Dados, Instâncias, Atribuições.

Funções, Parâmetros, Escopo.

Metodologia Ágil, Scrum, Ciclo de Vida, Cascata.

Product Owner, Sprint.

## Referências

AGGARWAL, Charu C. **Neural Networks and Deep Learning: A textbook**. Cham: Springer, 2018.

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. Rio de Janeiro: Elsevier. ISBN-13: 978-85-352-2626-3. 2008.

DORIGO, Marco; STÜTZLE, Thomas. **Ant Colony Optimization**. Cambridge, Massachusetts: The MIT Press, 2004.

GOLDBERG, David E. **Genetic Algorithms in Search, Optimization, and Machine Learning**. Addison-Wesley, 1989.

IMDB. 2023a. Disponível em: <<https://www.imdb.com/title/tt2084970/>>. Acessado em: 13/09/2023 às 02:49.

IMDB. 2023b. Disponível em: <<https://www.imdb.com/title/tt0118882/>>. Acessado em: 13/09/2023 às 02:57.

INCE, Darrel. **The Computer:A Very Short Introduction**. Oxford University Press, 2011.

RAMALHO, Luciano. **Python Fluente**. Novatec, 2015.

SABBAGH, Rafael. **Scrum: Gestão ágil para projetos de sucesso**. São Paulo: Casa do Código, 2016.

### Nota:

Design by Fábrica de Conteúdos Educação © 2024