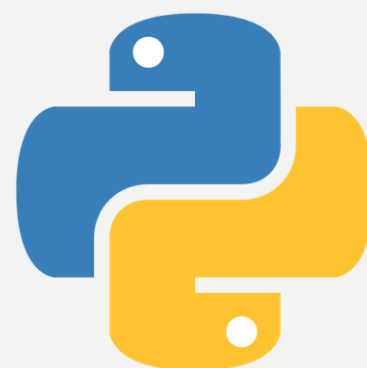


INTRODUÇÃO A LINGUAGEM PYTHON



SUMÁRIO

- Introdução;
- Os 19 Princípios da Linguagem Python;
- Áreas de atuação em Python;
- Construindo o ambiente para escrever seus códigos;
- Escrevendo seu primeiro código em Python;
- Variáveis e tipos da linguagem Python;
- Lógica em Python;
- Estruturas de Repetição em Python;
- As Coleções presentes em Python



SUMÁRIO

- Expressões e Funções em Python;
- Tratamento de erros e debugação;
- Iteradores, Geradores e Decoradores;
- Programação Orientada a Objetos presentes em Python;
- Realizando testes com Python;
- Considerações Finais;
- Referências



INTRODUÇÃO A LINGUAGEM DE PROGRAMAÇÃO PYTHON

- Python é uma linguagem de programação de **alto nível**, **interpretada**, **orientada a objetos**, **dinâmica** e **interativa**. É amplamente utilizada em vários domínios, incluindo **desenvolvimento web**, **análise de dados**, **inteligência artificial**, **automação de tarefas**, entre outros. Este e-book apresenta uma breve introdução ao Python, incluindo uma visão geral da linguagem, o mercado de trabalho para desenvolvedores Python, dicas para aprender [Python](#) e considerações finais.



INTRODUÇÃO A LINGUAGEM DE PROGRAMAÇÃO PYTHON

Python foi desenvolvido por Guido van Rossum no final dos anos 1980 e foi lançado em 1991. É uma linguagem de programação de código aberto, o que significa que é livre para uso, distribuição e modificação. Python é considerada uma linguagem fácil de aprender e usar devido à sua sintaxe simples e elegante. É uma linguagem interpretada, o que significa que o código é executado diretamente pelo interpretador, sem a necessidade de compilação. Python é uma linguagem orientada a objetos, o que significa que os programas são compostos de objetos, que são instâncias de classes.



Saiba mais sobre o criador de Python! Acesse:
<https://www.youtube.com/watch?v=wjebq3d4IGk>



CARACTERÍSTICAS PRINCIPAIS

Algumas das características distintivas do Python são:

- Sintaxe **simples e legível**;
- Forte **tipagem dinâmica**;
- Suporte a **várias** plataformas;
- Grande **biblioteca padrão**;
- **Interação** com outras linguagens de programação;
- Capacidade de **programação orientada a objetos**;
- **Fácil aprendizado** e uso.



19 PRINCÍPIOS DE PYTHON

- 1º Bonito é melhor do que feio.
- 2º Explícito é melhor do que implícito.
- 3º Simples é melhor do que complexo.
- 4º Complexo é melhor do que complicado.
- 5º Horizontal é melhor do que aninhado.
- 6º Esparso é melhor que denso.



19 PRINCÍPIOS DE PYTHON

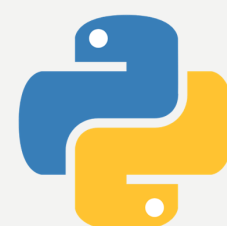
- 7°A legibilidade conta.
- 8°Casos especiais não são especiais o suficiente para quebrar as regras.
- 9°Porém, a praticidade supera a pureza.
- 10°Os erros nunca devem passar silenciosamente.
- 11°A menos que sejam explicitamente silenciados.
- 12°Diante da ambiguidade, recuse a tentação de adivinhar.



19 PRINCÍPIOS DE PYTHON

- 13° Deve haver uma, e de preferência apenas uma, forma óbvia de se fazer algo.
- 14° Embora essa forma possa não ser óbvia no início, a menos que você seja holandês.
- 15° Agora é melhor do que nunca.
- 16° Mas “nunca” é melhor do que “imediatamente agora”.
- 17° Se a implementação é difícil de explicar, é uma má ideia.
- 18° Se a implementação for fácil de explicar, pode ser uma boa ideia.
- 19° Namespaces são uma ótima ideia – vamos fazer mais disso!

Fonte: Disponível em: <https://didatica.tech/a-linguagem-python/>. Acesso em: 03/06/2023.



ÁREAS DE ATUAÇÃO EM PYTHON

Python é uma das linguagens de programação mais populares atualmente. De acordo com a pesquisa de desenvolvedores da Stack Overflow, Python é a **terceira linguagem de programação** mais popular em 2021, atrás apenas de **JavaScript** e **SQL**. Python é amplamente utilizado em várias áreas, incluindo **desenvolvimento web, análise de dados, automação de tarefas, inteligência artificial e aprendizado de máquina**.

O **mercado de trabalho** para desenvolvedores Python é muito promissor. De acordo com o site Glassdoor, o **salário médio** de um desenvolvedor Python nos Estados Unidos é de cerca de **US\$ 90.000 por ano**. Além disso, há **muitas vagas** de emprego disponíveis para desenvolvedores Python em todo o mundo.



CONSTRUINDO O AMBIENTE PARA PROGRAMAR EM PYTHON

Antes de começar a aprender Python, você precisará instalar o interpretador Python em seu computador. Você pode baixar o Python gratuitamente no site oficial (www.python.org).

- **IMPORTANTE:** A Instalação depende do seu sistema operacional e pode sofrer variações, em caso de dúvidas, consulte:
- Instalação no Windows: <https://python.org.br/instalacao-windows/>
- Instalação no Linux: <https://python.org.br/instalacao-linux/>
- Instalação no Mac Os: <https://python.org.br/instalacao-mac/>

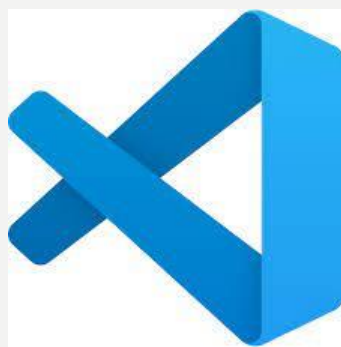
Você vai encontrar o passo a passo para fazer a instalação em cada um desses sistemas, se você não usa nenhum deles, acesse:

<https://www.python.org/download/other/>



ESCOLHENDO UMA IDE PARA ESCREVER SEUS CÓDIGOS

Uma IDE (Integrated Development Environment) é um ambiente de desenvolvimento integrado, que oferece **ferramentas e recursos** para **facilitar a programação** e o **desenvolvimento de software**. Ela combina um **editor de código-fonte**, **recursos de depuração** e **compilador/interpretador** em uma **única interface**, proporcionando um ambiente de trabalho **completo** para os programadores.



ESCOLHENDO UMA IDE PARA ESCREVER SEUS CÓDIGOS

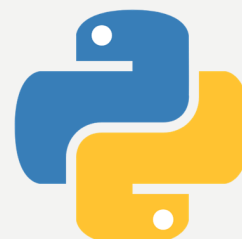
- Existem diversas opções no mercado:
- Pycharm, ideal para quem desenvolve em Python:
<https://www.jetbrains.com/pt-br/pycharm/>
- Visual Studio Code. Nela é possível escrever em diversas linguagens de programação, inclusive Python: <https://code.visualstudio.com/>;
- Existe a versão *online* do Visual Studio Code, que pode ser acessada pelo navegador: <https://vscode.dev/>;



PRIMEIRO CÓDIGO EM PYTHON

- Escrever códigos utilizando a linguagem de programação Python é relativamente simples.
- Veja um exemplo que pode ser utilizado por você para imprimir o comando “Olá Mundo!” na tela:

```
python Copy code  
  
print("Olá, mundo!")
```

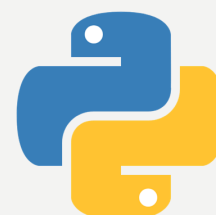


PRIMEIRO CÓDIGO EM PYTHON

- Para executar esse código, você pode copiá-lo para um arquivo com a extensão `.py` (por exemplo, `meu_programa.py`) e, em seguida, executá-lo usando o interpretador [Python](#). Dependendo do seu ambiente de desenvolvimento, você pode abrir o terminal ou **prompt** de comando, navegar até a pasta onde o arquivo está localizado e executar o seguinte comando:

```
python meu_programa.py
```

Copy code



VARIÁVEIS E TIPOS EM PYTHON

Variáveis:

- Uma variável em Python é um nome que você atribui a um valor para poder usá-lo posteriormente no seu programa. Ela é como uma caixa onde você pode armazenar dados. Por exemplo, você pode criar uma variável chamada "idade" e atribuir a ela o valor 25:

```
python Copy code  
  
idade = 25
```

- No exemplo temos que, "idade" é o nome da variável e 25 é o valor atribuído a ela. Agora, você pode usar essa variável em seu programa para realizar operações ou exibições.



VARIÁVEIS E TIPOS EM PYTHON

Tipos de dados:

- Os tipos de dados em Python definem o tipo de valor que uma variável pode armazenar. Alguns dos tipos de dados básicos em Python incluem:
- Inteiro (int): representa números inteiros, como 1, 2, -3, etc.
- Ponto flutuante (float): representa números decimais, como 3.14, 2.5, -0.5, etc.
- String (str): representa sequências de caracteres, como "Olá", "Python", "123", etc.
- Booleano (bool): representa valores lógicos True (verdadeiro) ou False (falso).
- Por exemplo, aqui estão algumas variáveis com diferentes tipos de dados em Python:



VARIÁVEIS E TIPOS EM PYTHON

- Por exemplo, aqui estão algumas variáveis com diferentes tipos de dados em Python:

```
python Copy code  
  
idade = 25 # tipo int  
altura = 1.75 # tipo float  
nome = "João" # tipo str  
estudante = True # tipo bool
```



VARIÁVEIS E TIPOS EM PYTHON

- No exemplo anterior, "idade" é do tipo int (inteiro), "altura" é do tipo float (ponto flutuante), "nome" é do tipo str (string) e "estudante" é do tipo bool (booleano).
- É importante observar que, em Python, você não precisa declarar explicitamente o tipo de uma variável antes de atribuir um valor a ela. O tipo da variável é inferido automaticamente com base no valor atribuído.



LÓGICA EM PYTHON

- A lógica é uma parte fundamental da programação, incluindo na linguagem Python. Através da lógica, podemos criar instruções e estruturas de controle que permitem que nossos programas tomem decisões, repitam tarefas e realizem cálculos complexos.
- A linguagem Python fornece várias estruturas de controle que nos permitem implementar lógica em nossos programas.



LÓGICA EM PYTHON

- Conheça algumas estruturas comumente usadas em Python:
- 1º Condicionais (if-else): Com a estrutura condicional "if", podemos executar um bloco de código somente se uma condição for verdadeira. Se a condição for falsa, podemos usar a estrutura "else" para executar outro bloco de código. Por exemplo:

```
python Copy code  
  
idade = 18  
if idade >= 18:  
    print("Você é maior de idade.")  
else:  
    print("Você é menor de idade.")
```



LÓGICA EM PYTHON

- 2º Loops (for e while): Os loops permitem que uma parte do código seja repetida várias vezes. O loop "for" é usado quando sabemos antecipadamente quantas vezes queremos repetir o código. O loop "while" é usado quando queremos repetir o código enquanto uma condição for verdadeira. Aqui estão exemplos de ambos:


```
python Copy code  
  
# Loop for  
for i in range(1, 5):  
    print(i)  
  
# Loop while  
contador = 0  
while contador < 5:  
    print(contador)  
    contador += 1
```



LÓGICA EM PYTHON

- Operadores lógicos: Python também possui operadores lógicos que nos permitem combinar condições. Os operadores mais comuns são "and" (e), "or" (ou) e "not" (não). Eles nos ajudam a criar expressões lógicas mais complexas. Veja um exemplo:

python

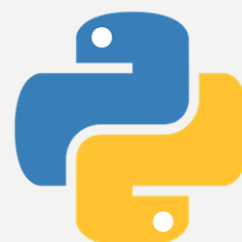
 Copy code

```
idade = 25
if idade >= 18 and idade <= 65:
    print("Você está na idade ativa.")
```



LÓGICA EM PYTHON

- No exemplo anterior temos que a variável **idade** possui o valor de **25** e o código escrito em Python irá verificar através de **if** se a idade atender os requisitos para uma pessoa ser considerada maior de idade, caso ela tenha 18 ou mais anos de idade irá imprimir na tela através do comando **print** a mensagem: “Você está na idade ativa”.



LÓGICA EM PYTHON


- **Comparadores:**
- **==:** Verifica se duas expressões são iguais;
- **!=:** Verifica se duas expressões são diferentes.
- **<:** Verifica se a expressão à esquerda é menor que a expressão à direita.
- **>:** Verifica se a expressão à esquerda é maior que a expressão à direita.
- **<=:** Verifica se a expressão à esquerda é menor ou igual à expressão à direita.
- **>=:** Verifica se a expressão à esquerda é maior ou igual à expressão à direita.



COLEÇÕES UTILIZADAS EM PYTHON: LISTAS

- Uma lista é uma coleção ordenada e mutável de elementos.
- Os elementos em uma lista podem ser de diferentes tipos (por exemplo, números, strings, objetos, etc.).
- Os elementos em uma lista são acessados por meio de índices baseados em zero.
- Exemplo:

python

 Copy code

```
lista = [1, 2, 3, 'a', 'b', 'c']
```



COLEÇÕES UTILIZADAS EM PYTHON: TUPLAS

- Uma tupla é uma coleção ordenada e imutável de elementos.
- Os elementos em uma tupla podem ser de diferentes tipos.
- Os elementos em uma tupla são acessados por meio de índices baseados em zero.
- Exemplo:

```
python
```

[Copy code](#)

```
tupla = (1, 2, 3, 'a', 'b', 'c')
```



COLEÇÕES UTILIZADAS EM PYTHON: CONJUNTOS

- Um conjunto é uma coleção desordenada de elementos únicos e não indexados.
- Os elementos em um conjunto não possuem uma ordem específica.
- Conjuntos são úteis para realizar operações de conjuntos, como união, interseção e diferença.
- Exemplo:

```
python
```

[Copy code](#)

```
conjunto = {1, 2, 3, 'a', 'b', 'c'}
```



COLEÇÕES UTILIZADAS EM PYTHON: DICIONÁRIOS

- Um dicionário é uma coleção desordenada de pares chave-valor.
- Os elementos em um dicionário são acessados por meio de suas chaves, em vez de índices.
- Os valores em um dicionário podem ser de diferentes tipos.
- Exemplos:

```
python
```

[Copy code](#)

```
dicionario = {'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'}
```



TRATAMENTO DE ERROS EM PYTHON

- Em Python, você pode lidar com erros e exceções usando blocos try-except. O bloco try é usado para envolver o código que pode gerar uma exceção, e o bloco except é usado para tratar a exceção e fornecer um comportamento alternativo. Aqui está um exemplo básico:

```
python Copy code  
  
try:  
    # Código que pode gerar uma exceção  
    resultado = dividir(10, 0) # Divisão por zero  
    print(resultado)  
except ZeroDivisionError:  
    # Tratamento da exceção ZeroDivisionError  
    print("Erro: Divisão por zero!")
```



TRATAMENTO DE ERROS EM PYTHON

- No exemplo anterior, a função dividir é chamada com argumentos que causariam uma exceção de divisão por zero. O bloco try envolve essa chamada, e o bloco except especifica o tipo de exceção que será tratada (neste caso, `ZeroDivisionError`). Se ocorrer uma exceção do tipo especificado dentro do bloco try, o fluxo de controle será desviado para o bloco except correspondente.



TRATAMENTO DE ERROS EM PYTHON

- Você também pode adicionar vários blocos except para tratar diferentes tipos de exceção:

```
python Copy code

try:
    # Código que pode gerar uma exceção
    arquivo = open("arquivo.txt", "r")
    conteudo = arquivo.read()
    numero = int(conteudo)
    resultado = 10 / numero
    print(resultado)
except FileNotFoundError:
    # Tratamento da exceção FileNotFoundError
    print("Erro: Arquivo não encontrado!")
except ValueError:
    # Tratamento da exceção ValueError
    print("Erro: Valor inválido!")
except ZeroDivisionError:
    # Tratamento da exceção ZeroDivisionError
    print("Erro: Divisão por zero!")
except Exception as e:
    # Tratamento de exceções genéricas
    print("Erro:", str(e))
```



TRATAMENTO DE ERROS EM PYTHON

- No exemplo anterior, diferentes tipos de exceção são tratados usando blocos except separados. O bloco except Exception as e captura exceções genéricas e fornece informações adicionais sobre o erro.



TRATAMENTO EM PYTHON

- Além do bloco try-except, você também pode usar os blocos else e finally. O bloco else é opcional e é executado se nenhuma exceção ocorrer dentro do bloco try. O bloco finally também é opcional e é executado independentemente de ter ocorrido uma exceção ou não.
- Veja o exemplo:

```
python Copy code  
  
try:  
    # Código que pode gerar uma exceção  
    resultado = dividir(10, 2)  
except ZeroDivisionError:  
    # Tratamento da exceção ZeroDivisionError  
    print("Erro: Divisão por zero!")  
else:  
    # Executado se nenhuma exceção ocorrer  
    print("Resultado:", resultado)  
finally:  
    # Sempre executado, independentemente de exceções  
    print("Finalizando o programa")
```



TRATAMENTO DE ERROS EM PYTHON

- No último exemplo, se nenhuma exceção ocorrer, o bloco `else` será executado para exibir o resultado. Em seguida, o bloco `finally` será executado, independentemente de ter ocorrido uma exceção ou não.



DEBUGAÇÃO EM PYTHON

- A depuração (debugging) é um processo de identificação e correção de erros em um programa. Python oferece várias ferramentas e técnicas para auxiliar na depuração do código.



DEBUGAÇÃO EM PYTHON:ALGUMAS DAS PRINCIPAIS MANEIRAS DE FAZER A DEPURAÇÃO EM PYTHON:

- 1º Impressão de mensagens: Uma das formas mais simples de depuração é inserir instruções de impressão no seu código para exibir o valor de variáveis ou mensagens de status em pontos-chave do programa. Você pode usar a função `print()` para exibir informações relevantes durante a execução do programa.
- 2º Declaração `assert`: A declaração `assert` permite verificar se uma determinada condição é verdadeira durante a execução do programa. Se a condição for falsa, um `AssertionError` é levantado. Isso pode ser útil para verificar suposições e condições em seu código e detectar erros.



DEBUGAÇÃO EM PYTHON: ALGUMAS DAS PRINCIPAIS MANEIRAS DE FAZER A DEPURAÇÃO EM PYTHON:

- Módulo pdb: O módulo pdb é uma ferramenta de depuração interativa incorporada ao Python. Ele permite pausar a execução do programa em pontos específicos e explorar o estado das variáveis, executar o código passo a passo e muito mais. Você pode inserir comandos de depuração diretamente no seu código ou iniciar o depurador interativo usando `import pdb; pdb.set_trace()`.
- IDEs e editores de texto avançados: Muitas IDEs (Ambientes Integrados de Desenvolvimento) e editores de texto avançados têm recursos integrados de depuração que facilitam a identificação e correção de erros. Essas ferramentas geralmente fornecem recursos como pontos de interrupção, visualização do estado das variáveis, execução passo a passo e inspeção de pilha de chamadas.



DEBUGAÇÃO EM PYTHON: ALGUMAS DAS PRINCIPAIS MANEIRAS DE FAZER A DEPURAÇÃO EM PYTHON:

- **Análise de tracebacks:** Quando ocorre uma exceção, o Python exibe um traceback (rastreamento) que mostra a sequência de chamadas de função que levou ao erro. O traceback pode ser usado para identificar a origem do erro e entender como as funções foram chamadas em seu programa. Analisar o traceback pode ajudar a identificar o local exato do erro e corrigi-lo.
- **Ferramentas de terceiros:** Existem várias ferramentas de depuração de terceiros disponíveis para Python, como o pdb++, ipdb, PyCharm, pdbpp, entre outras. Essas ferramentas oferecem recursos adicionais e uma interface mais amigável para depuração.



DEBUGAÇÃO EM PYTHON: ALGUMAS DAS PRINCIPAIS MANEIRAS DE FAZER A DEPURAÇÃO EM PYTHON:

- Lembre-se de remover ou desativar as instruções de depuração antes de finalizar o código ou enviá-lo para produção, já que elas podem afetar o desempenho do programa.
- A depuração é uma habilidade importante para todo desenvolvedor, e a escolha da técnica ou ferramenta depende das preferências pessoais e das necessidades específicas do projeto. Experimente diferentes abordagens e ferramentas para encontrar o método de depuração que melhor se adapta ao seu estilo de trabalho.



ITERADORES EM PYTHON

- Em Python, os iteradores são objetos que permitem percorrer sequencialmente um conjunto de elementos. Eles fornecem uma maneira eficiente de acessar os elementos de uma coleção, um por vez, sem a necessidade de armazenar todos os elementos na memória. Os iteradores são amplamente utilizados em construções como loops for e na iteração sobre objetos.



ITERADORES EM PYTHON

- Para entender melhor os iteradores em Python, aqui estão algumas informações importantes:
- O protocolo do iterador: Um objeto iterador em Python deve seguir o protocolo do iterador, que consiste em implementar dois métodos: `__iter__()` e `__next__()`. O método `__iter__()` retorna o próprio objeto iterador e é usado para iniciar a iteração. O método `__next__()` retorna o próximo elemento da sequência e é chamado repetidamente até que não haja mais elementos, momento em que ele deve levantar uma exceção `StopIteration`.



ITERADORES EM PYTHON

- Função `iter()`: A função `iter()` é usada para criar um objeto iterador a partir de um objeto iterável. Um objeto iterável é qualquer objeto que possa ser percorrido, como listas, tuplas, strings, dicionários, conjuntos e até mesmo arquivos.
- Loops `for`: Os loops `for` em Python são projetados para trabalhar com iteradores. Eles executam automaticamente a chamada aos métodos `__iter__()` e `__next__()` do iterador, percorrendo todos os elementos até que a exceção `StopIteration` seja levantada.



ITERADORES EM PYTHON

- Aqui está um exemplo simples para ilustrar o uso de iteradores em Python:

```
python Copy code  
  
frutas = ['maçã', 'banana', 'laranja']  
  
# Criando um iterador a partir da lista de frutas  
iterador = iter(frutas)  
  
# Percorrendo os elementos usando um loop for  
for fruta in iterador:  
    print(fruta)
```



ITERADORES EM PYTHON

- No exemplo anterior, a função `iter(frutas)` cria um iterador a partir da lista de frutas. Em seguida, o loop `for` percorre cada elemento usando o iterador. A cada iteração, o método `__next__()` é chamado implicitamente para obter o próximo elemento e atribuí-lo à variável `fruta`.
- Os iteradores fornecem uma maneira flexível e eficiente de lidar com grandes quantidades de dados ou sequências que são geradas dinamicamente. Eles permitem um processamento por demanda, economizando recursos de memória e tempo de execução. Além disso, você também pode criar seus próprios objetos iteradores personalizados implementando os métodos `__iter__()` e `__next__()` em suas classes.



ITERADORES EM PYTHON

- É importante mencionar que, a partir do Python 3, existe a construção `yield`, que permite criar geradores, uma forma mais simples e conveniente de criar iteradores em Python. Os geradores são funções especiais que usam a instrução `yield` para produzir uma sequência de valores em vez de retornar um único valor. Eles são uma poderosa ferramenta para a criação de iteradores de forma concisa e elegante.



GERADORES EM PYTHON

- Em Python, os geradores são uma forma especial de função que permite a criação de iteradores de maneira simples e eficiente. Eles são implementados usando a instrução `yield` em vez de `return`. Os geradores são úteis quando você deseja criar sequências de valores que podem ser iterados de forma eficiente, sem a necessidade de armazenar todos os valores na memória.



GERADORES EM PYTHON

- Aqui estão alguns pontos importantes sobre os geradores em Python:
- Declaração yield: O yield é uma instrução que interrompe a execução da função e retorna um valor. No entanto, diferentemente do return, o estado da função é salvo e pode ser retomado posteriormente. Isso permite que a função produza uma sequência de valores ao longo do tempo, em vez de retornar todos de uma vez.



GERADORES EM PYTHON

- **Função geradora:** Uma função geradora é definida usando a palavra-chave `def`, assim como uma função normal, mas em vez de usar `return`, ela usa `yield` para retornar valores. Cada vez que a instrução `yield` é executada, o valor é retornado e a função é suspensa temporariamente até a próxima iteração.
- **Iteração sobre geradores:** Os geradores podem ser iterados usando um loop `for`, assim como outros objetos iteráveis. A cada iteração, o gerador produz o próximo valor usando a instrução `yield`. A iteração continua até que não haja mais valores a serem produzidos pelo gerador.



GERADORES EM PYTHON

- Vantagens dos geradores: Os geradores são eficientes em termos de memória, pois produzem valores sob demanda, evitando a necessidade de armazenar todos os valores em memória. Eles são adequados para lidar com grandes volumes de dados ou sequências infinitas. Além disso, os geradores oferecem uma maneira elegante de escrever código iterativo e simplificam o processo de criação de iteradores personalizados.



GERADORES EM PYTHON

- Aqui está um exemplo simples para ilustrar a criação e uso de um gerador em Python:

```
python Copy code  
  
def contador(max):  
    i = 0  
    while i < max:  
        yield i  
        i += 1  
  
# Criando um gerador a partir da função contador  
meu_gerador = contador(5)  
  
# Iterando sobre o gerador usando um loop for  
for valor in meu_gerador:  
    print(valor)
```



GERADORES EM PYTHON

- No último exemplo, a função contador é um gerador que produz uma sequência de valores de 0 até o valor máximo especificado. A cada iteração, a instrução `yield` é usada para retornar o valor atual. No loop `for`, o gerador é iterado, e a cada iteração, o próximo valor é obtido usando a instrução `yield`.
- Os geradores são muito úteis em situações em que você precisa lidar com grandes quantidades de dados ou sequências que são geradas dinamicamente. Eles oferecem uma maneira eficiente e elegante de criar iteradores em Python, economizando recursos de memória e tempo de execução.



DECORADORES EM PYTHON

- Em Python, decoradores são uma maneira de modificar ou estender o comportamento de uma função ou classe sem a necessidade de modificar diretamente seu código. Eles permitem adicionar funcionalidades extras a uma função ou classe, encapsulando-as em outra função que recebe a função original como argumento.
- Os decoradores são escritos como funções que envolvem a função ou classe que será decorada. Eles podem executar ações antes e/ou depois da função original ser chamada, ou até mesmo substituir completamente o comportamento da função original.



DECORADORES EM PYTHON

- Exemplo:

```
python Copy code  
  
def decorador(funcao):  
    def funcao_decorada():  
        print("Executando código antes da função original")  
        funcao() # Chamada da função original  
        print("Executando código depois da função original")  
  
    return funcao_decorada  
  
@decorador  
def funcao_original():  
    print("Função original sendo executada")  
  
funcao_original()
```



DECORADORES EM PYTHON

- Nesse exemplo, temos a função decorador, que recebe a função original como argumento. Ela define uma função interna chamada `funcao_decorada`, que envolve a chamada da função original. Em seguida, o decorador retorna a função `funcao_decorada`.
- Ao decorar a função original usando `@decorador`, estamos efetivamente substituindo a função original pela função decorada. Quando chamamos `funcao_original()`, o decorador é aplicado automaticamente, e a função decorada é executada. Isso permite que o decorador adicione o código adicional antes e depois da função original ser chamada.




DECORADORES EM PYTHON

- Os decoradores também podem receber argumentos adicionais, permitindo uma maior flexibilidade na sua utilização. Para isso, é necessário adicionar uma camada extra de função ao decorador.



DECORADORES EM PYTHON: EXEMPLO

python

 Copy code

```
def decorador_com_argumentos(arg1, arg2):  
    def decorador(funcao):  
        def funcao_decorada():  
            print("Argumentos do decorador:", arg1, arg2)  
            funcao() # Chamada da função original  
        return funcao_decorada  
    return decorador  
  
@decorador_com_argumentos("arg1", "arg2")  
def funcao_original():  
    print("Função original sendo executada")  
  
funcao_original()
```



DECORADORES EM PYTHON: EXPLICAÇÃO DO EXEMPLO

- O decorador `decorador_com_argumentos` recebe dois argumentos (`arg1` e `arg2`) e retorna o decorador real. Esse decorador envolve a função original `funcao_original`, adicionando o código extra. Quando chamamos `funcao_original()`, os argumentos do decorador são passados e exibidos antes da execução da função original.
- Os decoradores são uma poderosa ferramenta em Python e são amplamente utilizados em frameworks e bibliotecas para adicionar funcionalidades extras às funções ou classes existentes sem modificar seu código original. Eles permitem uma maior modularidade e reutilização de código, tornando o desenvolvimento mais flexível e eficiente.



PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON

- A programação orientada a objetos (POO) é um paradigma de programação que se baseia no conceito de "objetos" para modelar o mundo real em software. Em Python, a POO é amplamente suportada e oferece recursos poderosos para criar e gerenciar objetos.



PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON: CONCEITOS

- Classes: Uma classe é uma estrutura que define as características e comportamentos de um objeto. Ela serve como um modelo para criar objetos específicos. Em Python, as classes são definidas usando a palavra-chave `class`. Por exemplo:

```
python Copy code  
  
class Pessoa:  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade  
  
    def saudacao(self):  
        print(f"Olá, meu nome é {self.nome} e tenho {self.idade} anos.")
```



PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON: CONCEITOS

- **Objetos:** Um objeto é uma instância de uma classe. Ele contém dados (atributos) e comportamentos (métodos) definidos pela classe. Para criar um objeto em Python, é necessário chamar a classe como se fosse uma função. Por exemplo:

```
python Copy code  
  
pessoa1 = Pessoa("João", 25)
```



PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON: CONCEITOS

- Atributos: Os atributos são as características ou propriedades de um objeto. Eles são representados por variáveis definidas dentro da classe. Em Python, os atributos podem ser acessados usando a notação de ponto (objeto.atributo). Por exemplo:

```
python Copy code  
  
print(pessoa1.nome) # Saída: João  
print(pessoa1.idade) # Saída: 25
```



PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON: CONCEITOS

- Métodos: Os métodos são as funções definidas dentro de uma classe que definem o comportamento dos objetos. Eles são usados para realizar ações específicas nos objetos ou manipular seus atributos. Os métodos são acessados usando a notação de ponto (objeto.metodo()). Por exemplo:

```
python Copy code  
  
pessoa1.saudacao() # Saída: Olá, meu nome é João e tenho 25 anos.
```



PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON: CONCEITOS

- Herança: A herança é um conceito fundamental na POO que permite criar novas classes com base em classes existentes. A classe derivada herda todos os atributos e métodos da classe base. Isso promove a reutilização de código e a organização hierárquica das classes. Em Python, a herança é definida colocando o nome da classe base entre parênteses após o nome da classe derivada. Por exemplo:

```
python Copy code  
  
class Estudante(Pessoa):  
    def __init__(self, nome, idade, curso):  
        super().__init__(nome, idade)  
        self.curso = curso  
  
    def info(self):  
        print(f"Eu sou estudante do curso de {self.curso}.")
```



PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON: CONCEITOS

- Encapsulamento: O encapsulamento é um princípio da POO que consiste em ocultar detalhes internos de uma classe e fornecer uma interface para interagir com o objeto. Em [Python](#), isso é alcançado usando convenções de nomenclatura de atributos e métodos. Por exemplo, um atributo prefixado com `__` é considerado privado e não deve ser acessado diretamente fora da classe.



TESTES EM PYTHON

- Em [Python](#), existem várias bibliotecas e abordagens para realizar testes automatizados. Aqui estão algumas das principais opções disponíveis:



TESTES EM PYTHON

- unittest: O módulo unittest é uma biblioteca de testes unitários incorporada ao Python. Ele fornece uma estrutura para escrever e executar testes de forma organizada. Os testes são definidos como métodos de uma classe de teste que herda da classe unittest.TestCase. O unittest oferece várias asserções úteis para verificar se os resultados esperados são obtidos. Exemplo:

```
python Copy code

import unittest

def soma(a, b):
    return a + b

class TesteSoma(unittest.TestCase):
    def test_soma(self):
        self.assertEqual(soma(2, 3), 5)
        self.assertEqual(soma(0, 0), 0)
        self.assertEqual(soma(-1, 1), 0)

if __name__ == '__main__':
    unittest.main()
```



TESTES EM PYTHON

- **pytest:** O pytest é uma biblioteca de testes mais simples e flexível em comparação com o unittest. Ele permite escrever testes de forma mais concisa e oferece recursos avançados, como descoberta automática de testes, suporte a fixtures e plugins. O pytest funciona bem com a maioria dos frameworks de teste existentes e pode ser executado diretamente na linha de comando. Exemplo:

```
python Copy code  
  
def soma(a, b):  
    return a + b  
  
def test_soma():  
    assert soma(2, 3) == 5  
    assert soma(0, 0) == 0  
    assert soma(-1, 1) == 0
```



TESTES EM PYTHON

- doctest: O módulo doctest permite escrever testes no formato de exemplos de código incorporados em docstrings. Esses exemplos são executados e os resultados são comparados com os resultados esperados especificados nos docstrings. O doctest é útil para testes simples e documentação ao mesmo tempo. Exemplo:

```
python Copy code  
  
def soma(a, b):  
    """  
    Função para somar dois números.  
  
    >>> soma(2, 3)  
    5  
    >>> soma(0, 0)  
    0  
    >>> soma(-1, 1)  
    0  
    """  
    return a + b  
  
import doctest  
doctest.testmod()
```



TESTES EM PYTHON

- Além dessas opções, existem outras **bibliotecas populares** de teste em [Python](#), como nose e unittest.mock, que oferecem recursos adicionais para testes avançados, como mocking e cobertura de código.
- Independentemente da biblioteca escolhida, a prática de teste é fundamental para garantir a qualidade e a confiabilidade do seu código. Os testes automatizados permitem verificar se o código funciona conforme o esperado, detectar regressões em alterações futuras e facilitar a manutenção do código ao longo do tempo.



CONSIDERAÇÕES FINAIS

- Neste e-book você aprendeu teve uma breve introdução sobre alguns conceitos sobre a linguagem Python;
- Espero que você tenha aprendido bastante e continue seus estudos na linguagem de programação tão importante para o mercado de trabalho;
- Compartilhe este conhecimento com seus amigos que também possa se interessar;
- Continue seu aprendizado acessando este curso de [Python](#)



REFERÊNCIAS

- <https://www.python.org/>;
- <https://tecnoblog.net/responde/o-que-e-python-guia-para-iniciantes/>;
- <https://dio.me/curso-intensivo-python/AF09NCKTFOJE>;

