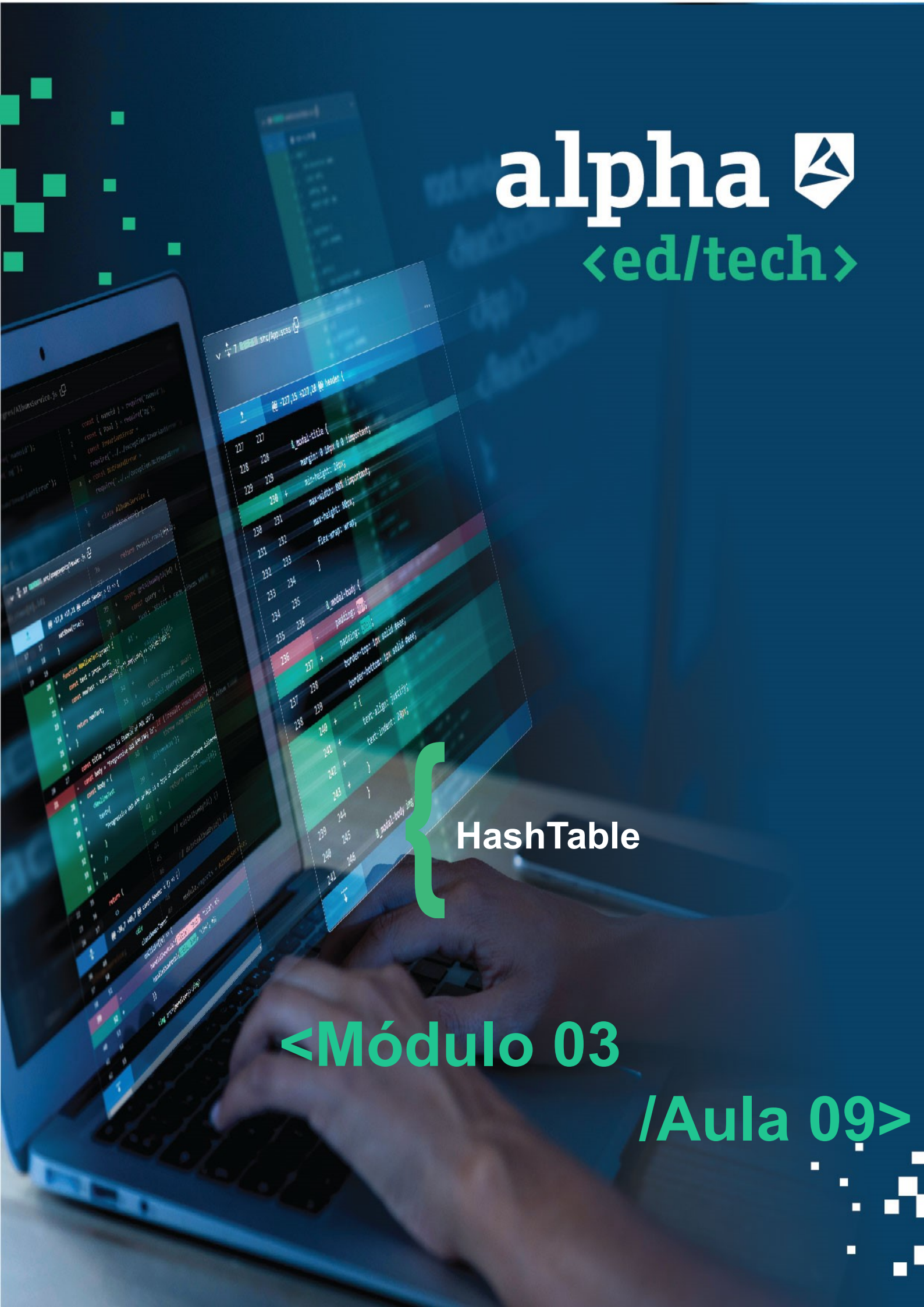

alpha
<ed/tech>
HashTable
<Módulo 03
/Aula 09>

HashTable TDD

HashTable ajuda a resolver muitos problemas da vida real, como indexar tabelas de banco de dados, armazenar valores calculados em cache ou implementar conjuntos.
Frequentemente, ela é abordada em entrevistas de emprego, e o Python utiliza tabelas hash em muitos lugares para tornar as consultas de nomes quase instantâneas.

Python tem sua própria tabela hash chamada "dict" (dicionário), necessário entender como as tabelas hash funcionam nos bastidores. Uma avaliação de codificação pode até desafiá-lo a construir uma.
Estrutura de Dados propiciará alguns desafios que introduzirão conceitos importantes e lhe darão uma ideia de por que as tabelas hash são tão rápidas.

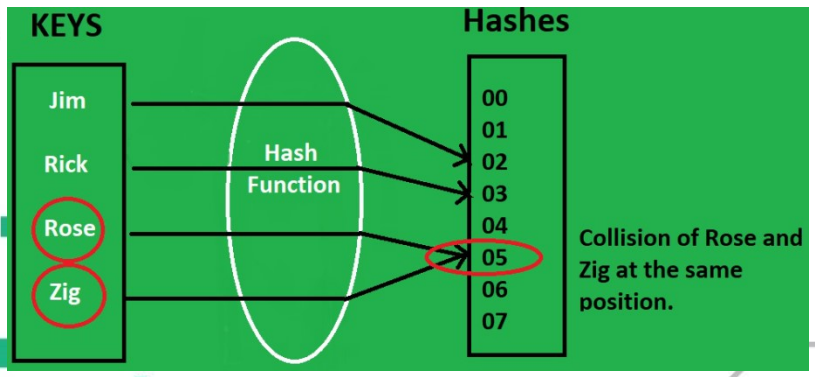
Cuidado para a terminologia não ficar um pouco confusa. Coloquialmente, o termo "tabela hash" ou "hash map" é frequentemente usado de forma intercambiável com a palavra "dicionário". No entanto, há uma diferença sutil entre os dois conceitos, pois o primeiro é mais específico que o último esperado.

Tabela Hash vs. Dicionário: Na ciência da computação, um dicionário é um tipo de dado abstrato composto por chaves e valores organizados em pares.
Além disso, ele define as seguintes operações (métodos da classe) para esses elementos:
•Adicionar um par chave-valor
•Excluir um par chave-valor
•Atualizar um par chave-valor
•Encontrar um valor associado à chave fornecida

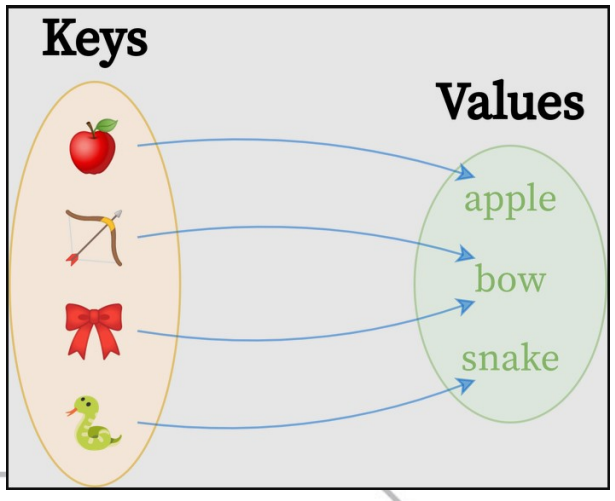
De certa forma, esse tipo de dado abstrato se assemelha a um dicionário bilíngue, onde as **chaves são palavras estrangeiras** e os **valores são suas definições ou traduções para outros idiomas**.

Sempre que você **mapeia** uma coisa para outra ou **associa** um valor a uma chave, você está essencialmente usando uma espécie de dicionário. É por isso que os dicionários também são conhecidos como mapas ou arrays associativos.

Dicionários têm algumas propriedades interessantes. Uma delas é que você pode pensar em um dicionário como uma função matemática que projeta um ou mais argumentos para exatamente um valor.



https://media.geeksforgeeks.org/wp-content/uploads/20210108180437/Chaining2.jpg



https://files.realpython.com/media/dict_map.6ea6c6e33b4b.png

HashTable TDD

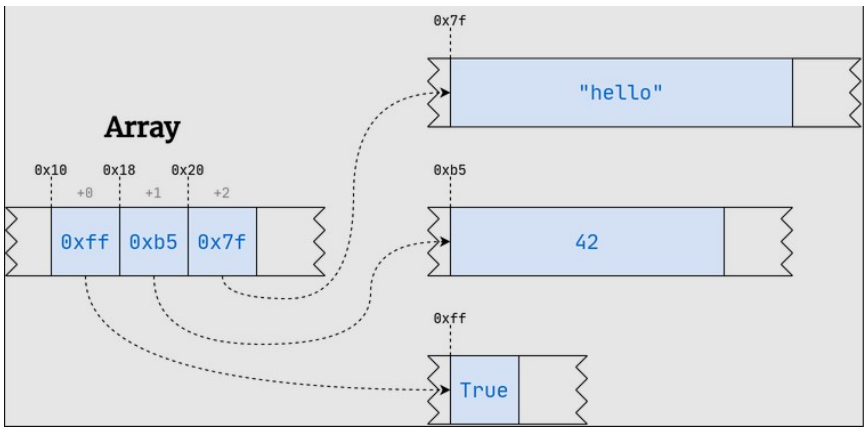
As consequências diretas desse fato são as seguintes:
•**Apenas Pares Chave-Valor**: Você não pode ter uma chave sem o valor ou vice-versa em um dicionário. Eles sempre vão juntos.
•**Chaves e Valores Arbitrários**: Chaves e valores podem pertencer a dois conjuntos disjuntos do mesmo tipo ou tipos diferentes. Tanto chaves quanto valores podem ser quase qualquer coisa, como números, palavras ou até mesmo imagens.
•**Pares Chave-Valor Não Ordenados**: Devido ao último ponto, os dicionários geralmente não definem nenhuma ordem para seus pares chave-valor. No entanto, isso pode depender da implementação específica.
•**Chaves Únicas**: Um dicionário não pode conter chaves duplicadas, pois isso violaria a definição de uma função.
•**Valores Não Únicos**: O mesmo valor pode ser associado a muitas chaves.

Existem conceitos relacionados que **estendem a ideia de um dicionário**. Por exemplo, um **multimapa** permite que você tenha **mais de um valor por chave**, enquanto um mapa bidirecional não apenas mapeia chaves para valores, mas também fornece mapeamento na direção oposta.

O **dict do Python** permite que você realize todas as operações de dicionário, com a sintaxe de colchetes (`[]`) usada em Listas do Python:
•Pode adicionar um novo par chave-valor a um dicionário.
•Pode atualizar o valor ou excluir um par existente identificado por uma chave.
•Pode procurar o valor associado à chave fornecida.
•Pode fazer uma pergunta diferente.

Como o dicionário built-in no Python realmente funciona?
Como ele mapeia chaves de tipos de dados arbitrários e como o faz tão rapidamente?
Encontrar uma implementação eficiente desse tipo de dado abstrato é conhecido como o **problema do dicionário**.
A solução do **dict do Python** aproveita a estrutura de dados da tabela hash.

No entanto, observe que essa não é a única maneira de implementar um dicionário em geral. Outra implementação popular é baseada em uma árvore rubro-negra.



https://realpython.com/python-hash-table/

Vetor de endereços de memória (cada endereço é do tamanho de endereçamento de memória, por exemplo caso se o seu Windows for 32 bits ou 64 bits) onde estão armazenados cada objeto (pode ser uma ficha de empregado, muitos bytes).

HashTable TDD

HashTable: Um Array Com uma Função de Hash

Acessar elementos de sequências em Python é tão rápido, independentemente do índice solicitado.

```
import string
texto = string.ascii_uppercase * 100_000_000

print(len(texto))

print(texto[30_000_000:30_000_050])
```

Há 2,6 bilhões de caracteres provenientes de letras ASCII repetidas na variável texto acima, que você pode contar com a função len() do Python. No entanto, obter o primeiro, do meio, último ou qualquer outro caractere desta string é igualmente rápido.

O mesmo é válido para **todos os tipos de sequências em Python**, como **listas** e **tuplas**.

O segredo para essa velocidade impressionante é que sequências em Python são respaldadas por **um array (estrutura de dados de acesso aleatório)**. Arrays segue dois princípios:

1. O array ocupa um bloco contínuo de memória.
2. Cada elemento no array tem um tamanho fixo conhecido antecipadamente (32 bits ou 64 bits que é o endereçamento do seu Windows/Linux/Processador).

Obs.: **Acesso Aleatório é um péssimo nome**, pois foi uma expressão criada a partir da fita de acesso sequencial contra o disco rígido, que **na verdade é um disco de acesso direto**. Como também, RAM Memória de Acesso Randômico, deveria ser memória de acesso direto.

Quando você **conhece o endereço de memória do início de um array, chamado de offset**, então pode chegar a qualquer elemento desejado no array instantaneamente, simplesmente sabendo o tamanho de cada elemento do array (como int ou float ou complex).

Observação: Ao contrário dos arrays, as listas do Python podem conter elementos heterogêneos de tamanhos variados. Para mitigar isso, o Python adiciona outro nível de indireção introduzindo um array de ponteiros para locais de memória, em vez de armazenar valores diretamente no array.

Ponteiros são apenas números inteiros, que sempre ocupam a mesma quantidade de espaço. É costume **denotar endereços de memória usando a notação hexadecimal. O Python e algumas outras linguagens prefixam tais números com 0x.**

Encontrar um elemento em um array é rápido, não importa onde esse elemento esteja fisicamente localizado.
Usar a mesma estratégia em um dicionário. Um array de endereços de memória, para cada elemento do dicionário.

Array – Acesso Direto

Acessar elementos de um array em Python é tão rápido, independentemente do índice solicitado. Mas vamos utilizar um NumPy Array.

```
import numpy as np

# Criando um array de inteiros usando NumPy
array_inteiros = np.array([1, 2, 3, 4, 5])

# Imprimindo o array
print("Array de inteiros:", array_inteiros)

# Realizando operações no array
maximo = np.max(array_inteiros)
minimo = np.min(array_inteiros)
```

As **tabelas hash** recebem seu nome de um truque chamado **hashing**, que permite traduzir uma chave arbitrária em um número inteiro que pode funcionar como um índice em um array regular.

Portanto, em vez de pesquisar um valor por um índice numérico, você o buscará por uma chave arbitrária sem uma perda de desempenho perceptível.

Isso é interessante!
Na prática, o **hashing** não funcionará com todas as chaves, mas a maioria dos tipos incorporados no Python pode ser "**hashada**". Se você seguir algumas regras, poderá **criar seus próprios tipos hasháveis** também.
Entender a Função de Hash que realiza o hashing transformando qualquer objeto em uma **sequência de bytes de tamanho fixo** chamada valor de hash ou código de hash. É um número que pode funcionar como **uma impressão digital ou um resumo**, geralmente muito menor que os dados originais, o que permite verificar sua integridade.

Além de verificar a integridade dos dados e resolver o problema do dicionário, as funções de hash ajudam em outras áreas, incluindo **segurança e criptografia**. Por exemplo, você geralmente armazena senhas hashadas em bancos de dados para mitigar o risco de vazamentos de dados. Assinaturas digitais envolvem o uso de hashing para criar um resumo da mensagem antes da criptografia. Transações de blockchain são outro exemplo primário do uso de uma função de hash para fins criptográficos.

Embora existam muitos algoritmos de hash, todos compartilham algumas propriedades comuns. Implementar corretamente uma boa função de hash é uma tarefa difícil que pode exigir a compreensão de matemática avançada envolvendo números primos.

Uma função de hash criptográfica é um tipo especial de função de hash que deve atender a alguns requisitos adicionais.

hash() built-in

O Python vem com um **módulo hashlib embutido**, que fornece uma variedade de funções de hash criptográficas conhecidas, bem como algoritmos de checksum menos seguros.
A linguagem também possui uma **função global hash()**, usada principalmente para busca rápida de elementos em **dicionários e conjuntos**.

Estudar como ela funciona primeiro para aprender sobre as propriedades mais importantes das funções de hash.

Antes de tentar implementar uma função de hash do zero, espere um momento e analise o hash() do Python para destilar suas propriedades. Isso ajudará você a entender quais problemas estão envolvidos ao projetar sua própria função de hash.

```
import hashlib

# Exemplo usando hashlib para hashing criptográfico (SHA-256)
dados = b"Exemplo de dados para hash"
hash_obj = hashlib.sha256(dados)
hash_resultado = hash_obj.hexdigest()

print(f"Dados: {dados}")
print(f"Hash usando hashlib: {hash_resultado}")
```

Este exemplo usa o algoritmo SHA-256 para criar um hash criptográfico dos dados fornecidos. Certifique-se de tratar os dados adequadamente, especialmente se você estiver lidando com senhas ou informações sensíveis.

Antes de implementar uma função de hash do zero, vamos analisar o hash() do Python para destilar suas propriedades.

A escolha de uma função de hash pode impactar drasticamente o desempenho da sua tabela hash. Portanto, você dependerá da função hash() built-in ao construir uma tabela hash personalizada.

Implementar uma função de hash nesta seção serve apenas como um exercício. Para começar, experimente chamar hash() em algumas literais de tipos de dados embutidos no Python, como números e strings, para ver como a função se comporta.

Chamando hash() com o mesmo argumento dentro da mesma sessão, você continuará obtendo o mesmo resultado.

Valores de hash são imutáveis e não mudam durante a vida útil de um objeto. No entanto, assim que você sair do Python e iniciá-lo novamente, é quase certo que verá valores de hash diferentes em invocações diferentes do Python. Você pode testar isso tentando a opção -c para executar um script de uma linha no terminal