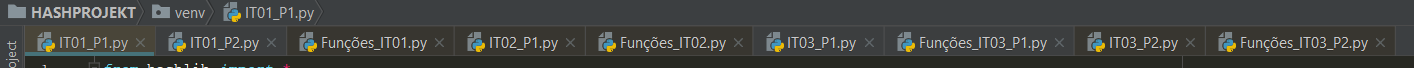
Nome: Gustavo Hammerschmidt.

Relatório HASHPROJEKT

Meu projeto foi dividido em itens, alguns deles divididos em partes. A figura abaixo mostra a ordem que deve ser seguida quando utilizar o programa.



A foto acima pode ser encontrada na mesma pasta como Ordem dos Arquivos.

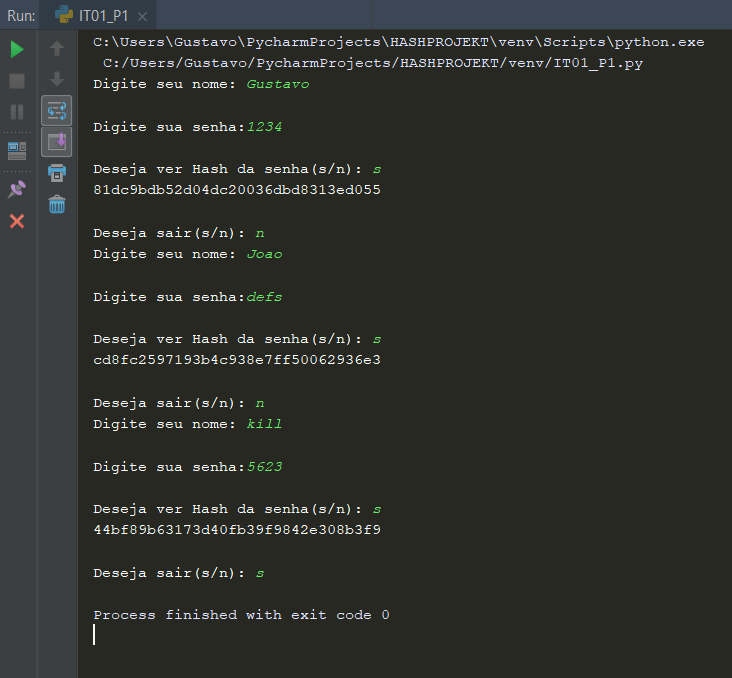
O item 01 P1 executa função: mainCall que executa, do arquivo Funções\_IT01.py, as seguintes funções:

- Cadastrar -> responsável por escrever o arquivo .txt com o nome e senha do usuário, também chama a função hash\_encrypt que, por sua vez, “faz” o HASH da senha cadastrada por meio da biblioteca hashlib utilizando as funções encode, hexdigest e md5. Dentro da função cadastrar, há um loop para fazer cadastros em sequência.

- Banco\_De\_Dados -> é a função responsável por salvar os nomes e as senhas COM HASH de usuários em um único arquivo: banco\_De\_Dados.txt . A Banco\_De\_Dados utiliza outras duas funções: hash\_encrypt : “faz” o HASH da senha fornecida; e pegar\_senha : procura no arquivo Info\_De\_{Nome fornecido} a senha do usuário e a retorna.

- only\_hash -> função que pega a senha COM HASH do usuário do arquivo Info\_De\_{Nome fornecido} e a salva em um arquivo único hash\_Banco\_Dados.txt junto com outras senhas COM HASH.

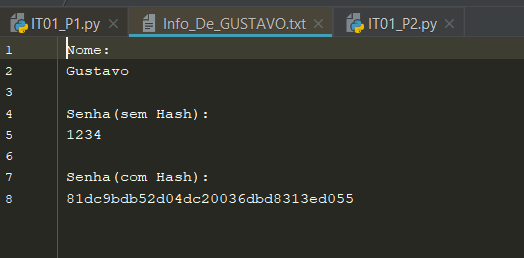
Foto execução de mainCall:



Ao executar o item 02 pela primeira vez, três arquivos serão criados:

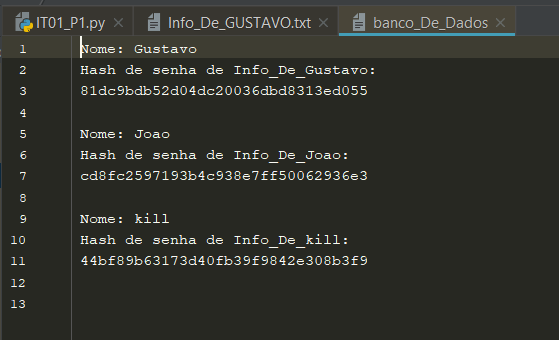
- Info\_De\_{Nome fornecido}.txt -> um arquivo .txt individual com o nome fornecido no campo entre colchetes.

Foto de exemplo:



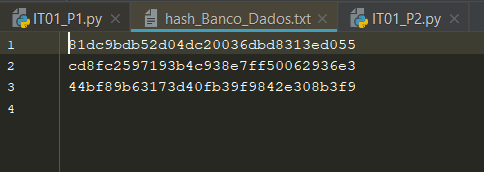
- Banco\_De\_Dados.txt -> um banco de dados único com os nomes e as senhas com HASH dos usuários, nome do arquivo fica: Banco\_De\_Dados.txt .

Foto de Exemplo:



- hash\_Banco\_Dados.txt -> arquivo único que contém somente as senhas COM HASH dos usuários, ele será usado posteriormente para facilitar a quebra das senhas HASH com maior facilidade. O arquivo criado se chamará: hash\_Banco\_Dados.txt .

Foto de Exemplo:



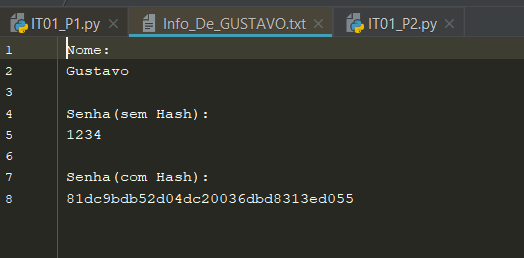
O item 01 P2 faz a autenticação de usuário previamente cadastrado, para tal, utiliza as seguintes funções do arquivo Funções\_IT01.py:

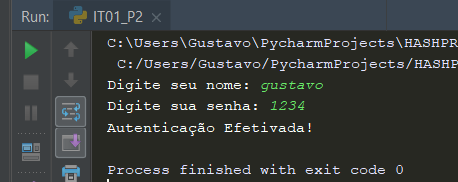
- hash\_encrypt -> faz o HASH do parâmetro senha que será concedido pelo o usuário, ela utiliza as funções da biblioteca hashlib: encode, hexdigest e md5.

- pegar\_hash -> pega a senha COM HASH salva no perfil do usuário.

Depois de executado as funções acima, é feito a comparação entre a senha digitada e a salva. Se forem iguais, retornará: Autenticação concluída.

Foto de Exemplo:





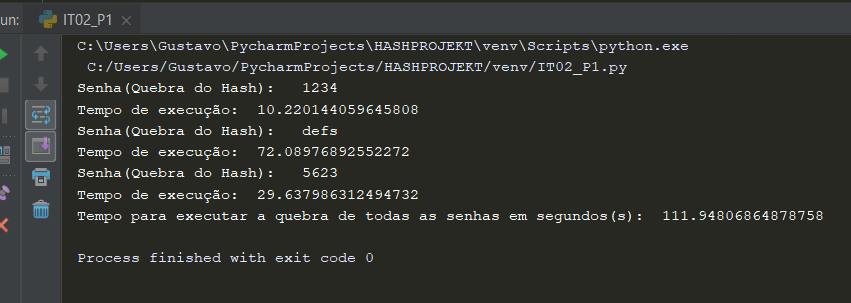
O item 02 P1 é responsável por fazer a quebra, encontrar o HASH. Ele utiliza as seguintes funções do arquivo Funções\_IT02.py :

- crack -> utiliza a biblioteca itertools para fazer permutações entre os possíveis caracteres, também utiliza as funções: execute, hexdigest e md5 da biblioteca hashlib para gerar códigos HASH. Depois de fazer as comparações, a função retorna a senha que corresponde com o código HASH fornecido.

- executeCrack -> utiliza a função crack, retorna as senhas de um arquivo e o tempo para decifrá-las, também retorna o tempo de todas as execuções de senhas do arquivo somadas.

O código para realizar a quebra das senhas HASH foi encontrado na internet, parcialmente, pronto. Tive que adaptá-lo para funcionar no ambiente proposto.

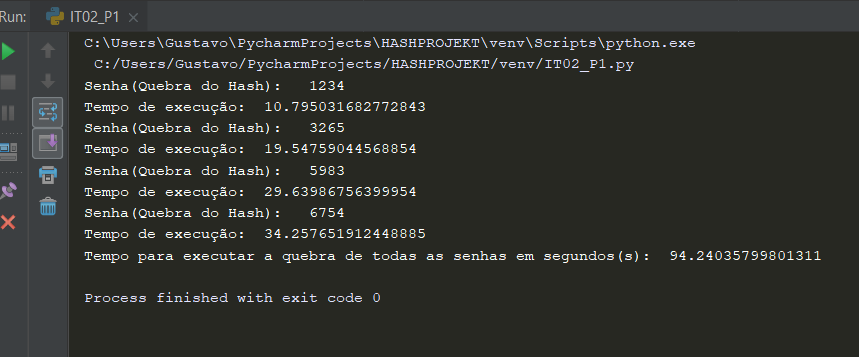
Foto de Exemplo:



Inclusive o tempo para quebrar o HASH da senha de 4 usuários varia em torno de :

30 segundos a 35 segundos

Foto de Exemplo:



Nesse caso a soma dos tempos foi de 94.24 segundos.

Veja o arquivo: Hash tempo 4 usuários no excel, lá poderá ver um gráfico com 10 tentativas de quebra de 4 senhas, ou seja, 40 tempos de execução. As informações tiradas para fazer o gráfico resultaram do meu próprio programa, elas estão armazenadas no arquivo: tempo 4 usuários.txt . E a pasta tempo 4 usuários contém cada arquivo gerado para realizar a estimativa de tempo.

O item 03 P1 executa função: mainCall2 que executa, do arquivo Funções\_IT03\_P1.py, as seguintes funções:

- cadastrar2 -> não difere de a função cadastrar

- banco\_De\_Dados2 -> não difere da função banco\_De\_Dados

Only\_hash2 -> não difere da função Only\_hash

O que teremos de diferente é que agora essas funções executarão suas propriedades e as salvarão em um arquivo com o sufixo “2BF”, o porquê de fazer isso é a nova função hash\_encrypt: hash\_encrypt2. Ela implementa a solução para diminuir a possibilidade de sucesso de encontrar uma senha que seja condizente com um HASH, pois, ela faz o HASH da senha dada 3 vezes, ou seja, HASH(HASH(HASH(KEY))). Uma vez que cada código HASH é uma combinação de 32 caracteres e que a biblioteca itertools implementada faz a permutação de 32 hexdigits. Ou seja 32 na potência de 32 possibilidades equivalente a dizer 2 elevados a 160 possibilidades. Isso tudo para resolver o primeiro HASH. O mesmo se aplicaria as outras duas vezes e só então teríamos encontrado a senha.

Para deixar claro, o sufixo 2BF não influencia em nada na nova função, serve apenas para separar os arquivos e evitar que haja dois arquivos iguais. O motivo de repetir todo o código do arquivo Funções\_IT01.py é prover um ambiente em que essa ocasião não possa vir a acontecer. Por isso deve-se ao menos cadastrar um usuário quando for utilizar o código do item 03 pela primeira vez.

Fazendo as estimativas baseadas na potência do meu PC, para encontrar o HASH correspondente a um caractere, levou um tempo entre 0,0003 e 0,00008 segundos. A média entre os dois extremos corresponde a 0,00019 segundos. Calculando a regra de três entre o número de possibilidades e o tempo teríamos: 2,77 \* 10^(46) segundos, que é equivalente a 8,805 \* 10^(38) anos. Claro esse tempo todo só será necessário se precisar testar todas as combinações, mas é o suficiente para dar uma ideia de quanto tempo levaria para encontrar o código equivalente.

Eu preparei um código para quebrar esse código do item 03 P1, ele está localizado no arquivo item 03 P2. Mas obviamente jamais proverá algum resultado, devido ao tempo requerido para tal.