# Guião 01

# 1. Tarefas preliminares

## O Ambiente de Desenvolvimento

Esta primeira parte pretende essencialmente garantir uma utilização fluida do ambiente de trabalho adoptado na UC. Isso pressupõe a utilização do `github` (em particular, do repositório atribuído ao grupo de trabalho), e do ambiente `Python` (versão 3.10 ou superior). Atente, em particular, às instruções presentes na secção "ORGANIZAÇÃO DO REPOSITÓRIO".

Nota: Sempre que solicitado, preserve o nome do programa e a ordem dos argumentos. Os programas submetidos serão testados automáticamente numa primeira fase de avaliação do Guião.

## Instalação de bibliotecas de suporte

### Cryptography - https://cryptography.io/en/stable/

A biblioteca criptográfica que iremos usar maioritariamente no curso é `cryptography`. Trata-se de uma biblioteca para a linguagem Python bem desenhada e bem documentada que oferece uma API de alto nível a diferentes "Serviços Criptográficos" (*recipes*). No entanto, e no âmbito concreto deste curso, iremos fazer um uso "menos standard" dessa biblioteca, na medida em que iremos recorrer directamente à funcionalidade de baixo nível.

Instalação:

Sugere-se o método de instalação baseado no `pip` (ver https://cryptography.io/en/latest/installation/).

```
pip3 install --upgrade pip
pip3 install cryptography
```

#### Validação da instalação

Para verificar a instalação, sugere-se executar o *snippet* de código apresentado na página inicial da biblioteca cryptography. Neste momento pretende-se apenas validar que são apresentados os *resultados esperados*, sendo que ao longo das próximas aulas iremos ter uma percepção mais abrangente da funcionalidade criptográfica que está a ser executada nesse *snippet*.

## Questões de Exemplo

Nas respostas às questões colocadas nos guiões, é muito importante:

- seguir estritamente as indicações do enunciado referentes ao nome do programa; opções da linha de comando; formato dos dados de entrada/saida; etc.
- testar/validar o programa com o(s) exemplo(s) fornecidos.

### QUESTÃO: Q1

Qual a versão da biblioteca `cryptography` instalada?

Note que pode executar o comando apresentado abaixo para verificar a versão da biblioteca instalada:

```
$ python3  -c "import cryptography; print(cryptography.__version__)"
```

> **Note**
>
> Deverá ser algo próximo de `42.0.2`

# 2. Cifras clássicas

## Cifra de César

A **cifra de César** caracteriza-se por realizar um deslocamento do alfabeto com que é escrita a mensagem. A título de exemplo, se considerarmos o texto-limpo "OLAMUNDO" [1] e um deslocamento de 2, obteríamos o criptograma "QNCOWPFQ". A seguinte tabela explica a transformação:

| **texto limpo** | **A** | **B** | **C** | **D** | **...** | **L** | **...** | **M** | **N** | **O** | **...** | **U** | **...** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| criptograma | C | D | E | F | ... | N | ... | O | P | Q | ... | W | ... |

Note que nas cifras clássicas é normal considerar-se unicamente um alfabeto com letras maiúsculas (sem qualquer pontuação). Se desejar, pode considerar a seguinte função que converte letras em maiúsculas e "filtra" todos os restantes caracteres.

```
    def preproc(str):
       l = []
       for c in str:
           if c.isalpha():
               l.append(c.upper())
       return "".join(l)
```

### PROG: `cesar.py`

Escrever o programa `cesar.py` que receba 3 argumentos:

- o tipo de operação a realizar: `enc` ou `dec`
- a chave secreta: `A`, `B`, ..., `Z`
- a mensagem a cifrar, por exemplo "Cartago esta no papo".

Apresenta-se de seguida um exemplo de utilização para este programa, através do terminal:

```
$ python3 cesar.py enc G "CartagoEstaNoPapo"
IGXZGMUKYZGTUVGVU
$ python3 cesar.py dec G "IGXZGMUKYZGTUVGVU"
CARTAGOESTANOPAPO
```

> **Note**
>
> Para converter a letra da chave na sua posição do alfabeto, podem usar a função ord que recebe um carater e dá o seu valor ASCII. Depois podem converter esse valor ASCII para a posição dessa letra no alfabeto ao subtrair o

ASCII da primeira letra (atenção que maiúsculas e minúsculas têm valores ASCII diferentes). A função chr faz exatamente o oposto.

## PROG: `cesar_attack.py`

Escreva o programa `cesar_attack.py` que realize um "ataque" à cifra de César. O programa deverá esperar os seguintes argumentos:

- o cryptograma considerado;
- uma sequência não vazia de palavras que poderão ser encontradas no texto-limpo (das palavras apresentadas, assume-se que pelo menos uma ocorra no texto-limpo).
  Como resultado, o programa deverá produzir:
  *resposta vazia no caso de não conseguir encontrar uma chave que faça* match *com uma das palavras fornecidas;*
  duas linhas no caso de sucesso
  - A primeira linha com a chave usada na cifra;
  - segunda linha com o texto limpo completo

```
$ python3 cesar_attack.py "IGXZGMUKYZGTUVGVU" BACO TACO
$ python3 cesar_attack.py "IGXZGMUKYZGTUVGVU" BACO PAPO
G
CARTAGOESTANOPAPO
```

# Cifra de Vigenère

A cifra de Vigenère aplica uma sequência de cifras de César: ao primeiro carácter é aplicada a primeira cifra; e assim sucessivamente até ao tamanho de chave considerado (número de cifras de César), reinicializando-se o processo (o carácter seguinte será novamente cifrado com a primeira cifra de César). A título de exemplo, considere-se a chave "BACO" (que corresponde à sequência de cifras de César com deslocamento 1, 0, 2 e 14). A cifra do texto limpor "OLAMUNDO" será "PLCAVNFC", conforme se ilustra na tabela abaixo:

| **ptxt** | **A** | **B** | **C** | **D** | **...** | **L** | **M** | **N** | **O** | **...** | **U** | **...** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ctxt-0 (B) | B | C | D | E | ... | M | N | O | `P` | ... | `V` | ... |
| ctxt-1 (A) | A | B | C | D | ... | `L` | M | `N` | O | ... | O | ... |
| ctxt-2 (C) | `C` | D | E | `F` | ... | N | O | P | Q | ... | W | ... |
| ctxt-3 (O) | O | P | Q | R | ... | Z | `A` | B | `C` | ... | I | ... |

## PROG: `vigenere.py`

Crie um novo programa com o nome `vigenere.py` que se comporta de forma similar ao programa anterior. A única diferença é que a chave pode agora ser uma palavra (e não um carácter). Exemplos de utilização:

```
$ python3 vigenere.py enc BACO "CifraIndecifravel"
DIHFBIPRFCKTSAXSM
$ python3 vigenere.py dec BACO "DIHFBIPRFCKTSAXSM"
CIFRAINDECIFRAVEL
```

## PROG: `vigenere_attack.py`

Atacar a cifra de Vigenère é consideravelmente mais desafiante. O problema é normalmente abordado considerando dois sub-problemas: 1) determinar tamanhos prováveis para a chave; e 2) atacar isoladamente cada uma das "fatias" do criptograma cifradas por cada carácter da chave. O primeiro destes problemas já recorre a uma sofisticação considerável, pelo que nos focaremos no segundo: atacar as várias cifras de César usadas.

Escreva o programa `vigenere_attack.py` que realize um "ataque" à cifra de Vigenère. O programa deverá esperar os seguintes argumentos:

- o tamanho da chave;
- o cryptograma considerado;
- uma sequência não vazia de palavras que poderão ser encontradas no texto-limpo (das palavras apresentadas, assume-se que pelo menos uma ocorre no texto-limpo).
  Como resultado, o programa deverá produzir:
  *resposta vazia no caso de não conseguir encontrar uma chave que faça* match *com uma das palavras fornecidas;*
  duas linhas no caso de sucesso
  - A primeira linha com a chave usada na cifra;
  - segunda linha com o texto limpo completo

Exemplo:

```
$ python3 vigenere_attack.py 3 "PGRGARHSFHPRGCVHOJHWEPZRSCJFIVSOFRWUTBKPZGGOZPZLHWKPBR" PAPO PRAIA
POR
ASARMASEOSBAROESASSINALADOSQUEDAOCIDENTALPRAIALUSITANA
```

> **Tip**
>
> Deverá ser útil considerar quais as letras mais frequentes no Português. Considere para o efeito que as letras mais frequentes são `A, E, O, S, ...` (por essa ordem -- ver [1-1]). Note no entanto que a probabilidade de ocorrências de algumas delas (em particular o `A` e `E`) são muito próximas.

> **Note**
>
> A maior dificuldade ao atacar a cifra de Vigenère traduz-se no facto de poderem ser necessário criptogramas maiores para se ter sucesso no ataque (mais informação disponível para o ataque...).

## PROG: `otp.py`

A cifra **One-Time-Pad** (OTP) pode ser identificada com a cifra de Vigenère quando a chave é perfeitamente aleatória e com tamanho não inferior ao texto-limpo. No entanto, é normalmente apresentada no alfabeto binário, em que chave e texto-limpor são sequência de bits, e a operação para cifrar/decifrar é o **XOR** (ou exclusivo).

“otp.png” não foi encontrado.

Crie o programa `otp.py` que se comporta da seguinte forma:

- caso o primeiro argumento do program seja `setup`, o segundo argumento será o número de bytes aleatórios a gerar e o terceiro o nome do ficheiro para se escrever os bytes gerados.
- caso o primeiro argumento seja `enc`, o segundo será o nome do ficheiro com a mensagem a cifrar e o terceiro o nome do ficheiro que contém a chave one-time-pad. O resultado da cifra será guardado num ficheiro com nome do ficheiro da mensagem cifrada com sufixo `.enc`.

- caso o primeiro argumento seja `dec`, o segundo será o nome do ficheiro a decifrar e o terceiro argumento o nome do ficheiro com a chave. O resultado será guardado num ficheiro cujo nome adiciona o sufixo `.dec` ao nome do ficheiro com o criptograma.

Exemplo:

```
$ python otp.py setup 30 otp.key
$ echo "Mensagem a cifrar" > ptxt.txt
$ python otp.py enc ptxt.txt otp.key
$ python otp.py dec ptxt.txt.enc otp.key
$ cat ptxt.txt.enc.dec
Mensagem a cifrar
```

> **Important**
>
> A qualidade dos números aleatórios usados em criptografia é de extrema > importância. Na definição do seu programa, atente à observação
> disponibilizada em https://cryptography.io/en/stable/random-numbers/.

## PROG: `bad_otp.py`

Retomando o tema da geração de números aleatórios, já se fez referência que aplicações cripográficas **devem** recorrer a geradores de números pseudo-aleatórios que garantam *segurança criptográfica* (CSPRG). Isso exclui a utilização de geradores de números aleatórios encontrados na generalidade das bibliotecas standard das linguagens de utilização corrente (e.g. random do Python)[2].

Neste problema iremos "simular" um ataque ao gerador de números aleatórios, substituindo o gerador de números aleatórios seguro pela utilização do seguinte fragmento de código:

```
import random

def bad_prng(n):
    """ an INSECURE pseudo-random number generator """
    random.seed(random.randbytes(2))
    return random.randbytes(n)
```

> **Note**
>
> Note que a sequência de bytes gerada será necessariamente uma de $2^{16}$ possibilidades (resultantes das diferentes *sementes* possíveis).

Escreva o programa `bad_otp.py` que substitua o gerador de números aleatórios para utilizar a função `bad_prng`.

## QUESTÃO: Q1

- Consegue observar diferenças no comportamento dos programas `otp.py` e `bad_otp.py`? Se sim, quais?

## PROG: `bad_otp_attack.py`

Pretende-se agora tirar partido da vulnerabilidade introduzida na geração de números aleatórios. Escreva um programa `bad_otp_attack.py` que receba como primeiro argumento o nome do ficheiro com o criptograma, seguido de uma lista de palavras que poderão ser encontradas no texto limpo. Deve retornar o texto-limpo da mensagem cifrada.

```
$ python3 bad_otp_attack.py ptxt.txt.enc texto cifrar
Mensagem a cifrar
```

## QUESTÃO: Q2

- O ataque realizado no ponto anterior não entra em contradição com o resultado que estabelece a "segurança absoluta" da cifra *one-time pad*? Justifique.

---

1. Pode consultar a lista completa em https://en.wikipedia.org/wiki/Letter_frequency ↩ ↩
2. Para mais detalhes em como atacar este gerador, consultar https://www.schutzwerk.com/en/blog/attacking-a-rng/ ↩