Algoritmos e Programação I: Aula 08*

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
79070-900 Campo Grande, MS
https://ava.ufms.br

Sumário

1 Conceitos Básicos de Lógica

A Lógica tem aplicações em várias áreas da computação. No estudo de algoritmos, utilizamos conceitos básicos de lógica em quatro formas distintas e importantes:

a. escrever pré- e pós-condições e outras afirmações a respeito do comportamento da correção de programas;

b. o uso de expressões lógicas em programação;

c. argumentação a respeito da correção de programas;

d. e no projeto de circuitos usados em computadores.

*Este material é para o uso exclusivo da disciplina de Algoritmos e Programação I da FACOM/UFMS e utiliza as referências bibliográficas da disciplina

Em Algoritmos e Programação I, utilizaremos mais as duas primeiras formas. A terceira forma é vista mais em correção de programas e a quarta na disciplina de Introdução a Sistemas de Computação.

Conceitos mais detalhados em Lógica serão vistos nas disciplinas de Fundamentos de Teoria da Computação e Introdução a Sistemas de Computação. Na nossa disciplina descreveremos as definições necessárias para o entendimento da solução algorítmica de problemas.

Recordamos que a **lógica proposicional** é um sistema para argumentação e execução de cálculos com proposições. Lógica proposicional teve início com o trabalho de George Boole entre 1847 e 1854. Boole observou a similaridade entre as propriedades das operações lógicas **and** e **or** e as operações aritméticas de multiplicação e adição. Ele desenvolveu um sistema para manipulação de afirmações lógicas que era tão preciso para estas afirmações quanto é a aritmética para a manipulação de números.

A estratégia básica de Boole foi o de estudar padrões de argumentos lógicos, usando símbolos para representar tanto as afirmações individuais como os próprios padrões. Esta abordagem na lógica é conhecida como lógica proposicional (ou lógica simbólica). Uma afirmação (ou proposição) em lógica proposicional pode ter somente um de dois valores, verdadeiro (**True**) ou falso (**False**).

Definição: Uma *proposição* é uma expressão qualquer que pode ser formada pelas seguintes regras:

- Regra 1: **True** e **False** são proposições.
- Regra 2: Qualquer variável do tipo {**True**,**False**} é uma proposição.
- Regra 3: Se p é uma proposição, então $(\sim p)$ é uma proposição.
- Regra 4: Se p e q são proposições, então $(p \vee q)$, $(p \wedge q)$, $(p \Rightarrow q)$ e $(p \Leftrightarrow q)$ são proposições.

A Tabela 1 lista os operadores lógicos que são usados para escrever proposições, junto com sua representação equivalente em *Python* e seus significados. O Python usa `True` para uma variável booleana com valor verdadeiro e `False` para uma variável booleana com valor falso. A atribuição de uma variável booleana em Python pode ser feita com:

```
limite = True
```

No caso da função `bool(str)`, para qualquer string `str`, diferente da string vazia, a função retorna `True` e `False` para `bool()`. Para qualquer inteiro (ou de ponto flutuante) `x`, temos que `bool(x)` é `True` para qualquer `x != 0` e `bool(0)` é `False`. Em função disso, para lermos um valor booleano temos que lê-lo como um número inteiro e convertê-lo para booleano.

```
limite = bool(int(input()))
```

Nem sempre é possível representar simbolicamente sentenças arbitrárias usando proposições. Isto é, muitas sentenças não são afirmações declarativas e desta forma não possuem um valor verdade. Por exemplo, "Feche a porta!" é um comando, e "Você assistiu a aula de Estatística hoje?" é uma pergunta. Contudo, existem muitas afirmações que podem ser representadas por proposições, e muitas mais que podem ser representadas por *predicados*.

Definição - Um *predicado* é uma expressão que é **True** ou **False** para cada estado de suas variáveis. Ele pode incluir constantes (naturais, inteiros, reais e valores booleanos **True**

Representação Matemática	Equivalência Python	Significado
$\sim$	`not`	Negação
$\vee$	`or`	Disjunção
$\wedge$	`and`	Conjunção
$\Rightarrow$	(nenhum)	Implicação
$\Leftrightarrow$	(nenhum)	Equivalência

Tabela 1: Os Operadores Lógicos

Representação Matemática	Equivalência Python	Significado
$<$	`<`	Menor que
$\leq$	`<=`	Menor ou igual que
$>$	`>`	Maior que
$\geq$	`>=`	Maior ou igual que
$=$	`==`	Igual
$\neq$	`!=`	Diferente

Tabela 2: Os Operadores Relacionais

ou **False**); variáveis e referências a funções aritméticas e Booleanas; operadores aritméticos ($+$,$*$, etc.); operadores relacionais ($<, \leq, \geq, >, =, \neq$), e operadores lógicos ($\vee, \wedge, \sim$).

Essa extensão da noção de lógica proposicional para uma ideia mais ampla como lógica dos predicados, permite-nos usar lógica em uma grande variedade de formas construtivas em computação. Em computação, a maior parte das instruções que desejamos representar de uma forma simbólica são declarativas, tendo um valor verdade que não é ambíguo, e podem ser expressas por proposições ou predicados.

Em `Python`, estes tipos de predicados são conhecidos como expressões relacionais, e eles tem muito uso em programação. Os operadores relacionais, contudo, tem diferentes representações em `Python` que em matemática (veja Tabela 2).

Uma proposição é definida com o uso de regras de sintaxe precisas. O mesmo pode ser feito para predicados; contudo, vamos inicialmente ver alguns exemplos. Começamos por observar que toda proposição é também um predicado.

O uso da lógica de proposicional e de predicados nos auxilia na formulação mais precisa de pré e pós-condições e também no projeto de algoritmos. A estrutura de controle condicional utiliza a teoria dessas lógicas para a formulação de passos dos nossos algoritmos.

2 Lógica e Solução de Problemas Algorítmicos

Como uma aplicação de lógica proposicional (expressões booleanas), considere circuito lógico da Figura 1, chamado **semi-somador binário**, o qual pode ser utilizado para somar dois números binários de dois dígitos (lembramos que um estudo detalhado dessa área será objeto da disciplina de Introdução a Sistemas de Computação).

Este circuito é projetado para aceitar duas entradas, **A** e **B**, e produzir duas saídas, **Soma** e **Vai_Um**. Ele contém quatro componentes eletrônicos básicos chamados **portas**, duas portas **AND**, uma porta **OR** e uma porta **NOT** (*inversor*). A entrada dessas portas são pulsos de corrente aplicadas as linhas que chegam às portas, e as saídas são pulsos de correntes nas linhas que saem das portas. No caso de uma porta **AND**, um pulso de

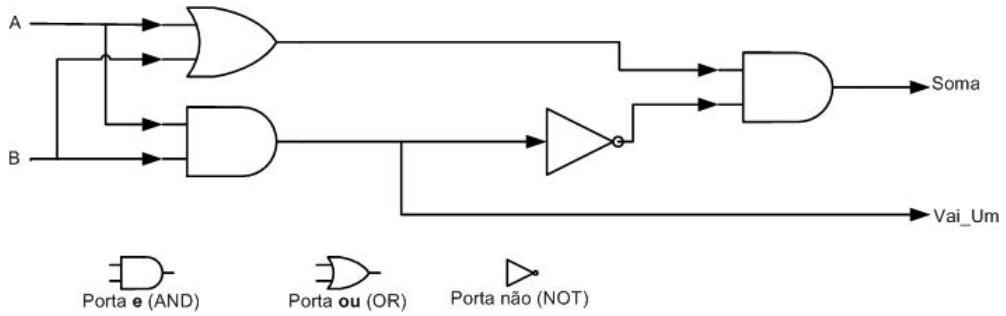

Figura 1: Semi-Somador Binário.

saída somente é produzido se existirem pulsos em ambas as linhas de entrada. Uma porta **OR** somente produz um pulso de saída se existir pelo menos um pulso numa das linhas de entrada. Uma porta **NOT** é projetada para produzir um pulso de saída somente quando não houver pulso de entrada. Se associarmos a expressão booleana "um pulso está presente" com cada linha, isto é, se interpretarmos `True` como a presença de um pulso e `False` como a ausência de um pulso, então os operadores booleanos `and`, `or` e `not` do Python podem ser utilizadas para modelar as portas **AND**, **OR** e **NOT**, respectivamente.

Desta forma, expressões booleanas podem ser utilizadas para modelar circuitos lógicos. Por exemplo, a saída **Soma** pode ser representada pela expressão booleana

```
(A or B) and  not(A and B)
```

e a saída `Vai_Um` por

```
A and B
```

e com isso podemos resolver o problema de somar dois números binários de um dígito.

```
0 + 0 = 00
0 + 1 = 01
1 + 0 = 01
1 + 1 = 10
```

Também nesse caso, a descrição do problema é imediata.

```
# Este programa lê dois números binários de um dígito em um
# circuito lógico que representa um semi-somador binário e calcula a
# saída do circuito.
```

Temos que as especificações de entrada e saída ficam:

Entrada	**Saída**
a e b, 2 números binários binários com um dígito.	Um número binário de dois dígitos igual a adição binária de a com b.

Podemos usar variáveis simples do tipo booleano (`bool`) para armazenar os valores dos números binários `a` e `b` e os resultados da adição `soma` e `vaiUm`. Quando da implementação do algoritmo pode surgir a necessidade de utilizarmos variáveis auxiliares para armazenar valores intermediários.

```
# descrição das variáveis utilizadas
# bool a, b, soma, vaiUm
```

e as pré e pós-condições ficam:

```
# pré: a b | a, b em 0, 1

# pós: vaiUm == a and b  soma == (a or b) and not(a and b)
```

Exemplo:

```
# formato da entrada
1 + 0 = 01

# estado das variáveis após a leitura
a == 1
b == 0
soma == indefinido
vaiUm == indefinido

# estado das variáveis após a soma
a == 1
b == 0
coma == 1
vaiUm == 0
```

Vamos agora descrever os passos do algoritmo necessários para solucionar o problema.

```
Algoritmo: SemiSomador
# Passo 1. Leia dois números binários de 1 dígito
# Passo 2. Calcule a soma de dois dígitos binários
# Passo 2.1. Calcule o dígito menos significativo da soma
# Passo 2.2. Calcule o dígito mais significativo da soma;
# Passo 3. Imprima os dois dígitos da soma;
# fim Algoritmo
```

Podemos agora descreve a codificação da solução em Python:

```
1  # -*- coding: utf-8 -*-
2  # Programa: semisomador.py
3  # Programador:
4  # Data: 30/03/2011
5  # Este programa lê dois números binários de um dígito em um
6  # circuito lógico que representa um semi-somador binário e calcula e
7  # imprime a soma dos dois dígitos binários.
8  # início do módulo principal
9  # descrição das variáveis utilizadas
10 # bool a, b, soma, vaiUm
```

```
11
12 # pré: a b | a, b em {0, 1}
13
14 # Passo 1. Leia dois números binários de 1 dígito
15 print('Entre com dois valores binários: ')
16 a = bool(int(input()))
17 b = bool(int(input()))
18 # Passo 2. Calcule a soma de dois dígitos binários
19 # Passo 2.1. Calcule o dígito menos significativo da soma
20 soma = (a or b) and not(a and b)
21 # Passo 2.2. Calcule o dígito mais significativo da soma
22 vaiUm = a and b
23 # Passo 3. Imprima a soma de dois números binários de 1 dígito
24 print('{0:1d} + {1:1d} = {2:1d}{3:1d}'.format(a, b, vaiUm, soma))
25
26 # pós: vaiUm == a and b  soma == (a or b)  and not(a and b)
27 # fim do módulo principal
```

Podemos testar e verificar o programa. Use o interpretador `Python` para interpretar e executar o programa:

```
$ python semisomador.py
```

Exemplo 1

```
# formato da entrada
Entre com dois valores binários: 0 0

# formato da entrada
0 + 0 = 00
```

Exemplo 2

```
# formato da entrada
Entre com dois valores binários: 0 1

# formato da entrada
0 + 1 = 01
```

Exemplo 3

```
# formato da entrada
Entre com dois valores binários: 1 0

# formato da entrada
1 + 0 = 01
```

Exemplo 4

```
# formato da entrada
Entre com dois valores binários: 1 1

# formato da entrada
1 + 1 = 10
```

3 Regra De Morgan

Quando projetamos o fluxo lógico de um programa, muitas vezes temos a situação na qual o operador *not* está na frente de uma operação entre parenteses. É muito mais natural ler e entender uma lógica afirmativa do que uma lógica negativa. Portanto, nesses casos, se quisermos transformar a expressão para lógica afirmativa com a retirada dos parenteses, devemos aplicar o operador *not* diretamente a cada um dos operandos. A Regra de De Morgan estabelece como o complemento dos operadores nesta situação. Esta regra é definida como:

Quando retiramos os parenteses numa expressão lógica precedida por um operador `not`, devemos aplicar o operador `not` para cada expressão enquanto mudamos os operadores lógicos - isto é, mudando o *and* para `or` e mudando o `or` para `and`.

Considere a expressão abaixo:

`not(x and y)` $\rightarrow$ `not x or not y`
`not(x or y)` $\rightarrow$ `not x and not y`

Se a expressão for complementada apropriadamente de acordo com as regras de De Morgan, então o resultado da primeira expressão *true* ou *false* será o mesmo que o resultado da segunda expressão para qualquer conjunto de valores para `x` ou `y`. Lembramos que para situações mais complexas, as regras de precedência do `Python` podem afetar as regras de De Morgan.

4 A Estrutura de Seleção

As expressões booleanas, tornam possível a utilização de uma estrutura de seleção nos programas. Essa estrutura seleciona e executa uma (ou mais) instrução(ões) entre duas ou mais alternativas no fluxo de instruções do programa. Isto proporciona ao programador a introdução de pontos de decisão em um programa, isto é, a decisão é feita durante a execução do programa para seguir uma entre diversas possibilidades de ação. Inicialmente vamos considerar uma estrutura de seleção simples, na qual existem duas possibilidades para definir o fluxo de ações.

Primeiro, vamos analisar uma estrutura de seleção na qual uma sequência de instruções é executada ou ignorada dependendo se uma dada expressão booleana é **True** ou **False**. O diagrama da Figura 2 ilustra essa situação.

O Python implementa esta estrutura de seleção usando a instrução `if`. Essa instrução avalia uma expressão booleana, e se a expressão booleana for verdadeira (`True`), o Python executa a(s) instrução(ões) da linha seguinte que esteja(m) indentada(s) com um recuo com relação a instrução `if`. Essa é a forma como o Python usa para definir as instruções que serão executadas numa estrutura de seleção. Caso a expressão booleana seja falsa (`False`), o Python executará a próxima instrução que esteja indentada (alinhada) com o `if`. O exemplo abaixo descreve como o Python implementa o diagrama da Figura 2:

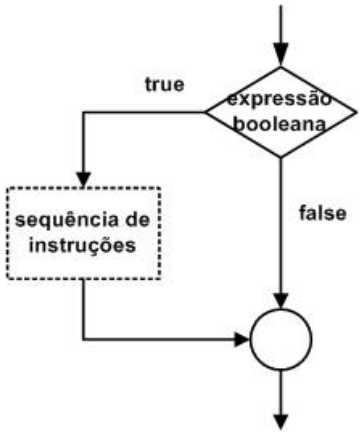

Figura 2: Estrutura de Seleção `if`.

```
1 if expressão_booleana: # True vá p/ linha 2 - False vá p/ linha 6
2    instrução-1 # vá p/ linha 3
3    instrução-2 # vá p/ linha 4
4    ...
5    instrução-n # vá p/ linha 6
6 instrução
```

Se a expressão booleana é `True`, as linhas 2 a 5 são executadas, caso contrário, ela são ignoradas, e a execução continua com a execução da próxima instrução no programa no mesmo nível de indentação do `if`, a linha 6. Por exemplo, no trecho do programa

```
1 if media >= 6.0: # True vá p/ linha 2 - False vá p/ linha 4
2    print('Parabéns!') # vá p/ linha 3
3    print('Aprovado') # vá p/ linha 4
4 instrução
```

a expressão booleana `média >= 6.0` é avaliada, e se ela for verdadeira (`True`), as instruções com um nível de recuo (indentadas) com relação a instrução `if` são executadas, primeiro a instrução da linha 2, `print('Parabéns!')` e a seguir a instrução da linha 3, `print('Aprovado')`. Após executar a instrução da linha 3, como não há mais nenhuma instrução no bloco indentado da instrução `if`, a instrução da linha 4 será executada. Caso contrário, no caso da expressão booleana ser falsa (`False`), as instruções das linhas 2 e 3 serão ignoradas e o programa "desvia" a execução para a instrução da linha 4, que está no mesmo nível de indentação do `if`. Quando a sequência de instruções se resumir a uma única instrução, a instrução condicional em `Python` fica:

```
1 if media >= 6.0: # True vá p/ linha 2 - False vá p/ linha 3
2    print('Aprovado') # vá p/ linha 3
3 instrução
```

Também nesse caso, a expressão booleana `media >= 6.0` é avaliada, e se ela for `verdadeira` (`True`), instrução `print('Aprovado')` é executada e como não há mais nenhuma indentado "dentro" da instrução `if`, a instrução da linha 3 será executada. Caso contrário, no caso da expressão booleana ser falsa (`False`), a instrução da linha 2 será ignorada e o programa

"desvia" a execução para a instrução da linha 3, que está no mesmo nível de indentação do `if`.

Na estrutura de seleção anterior, a seleção é feita entre (1) executando uma dada sequência de instruções e (2) ignora estas instruções. Na seleção de duas formas apresentada na Figura 3, a seleção é feita entre (1) executando uma sequência de instruções e (2) executando uma outra sequência de instruções.

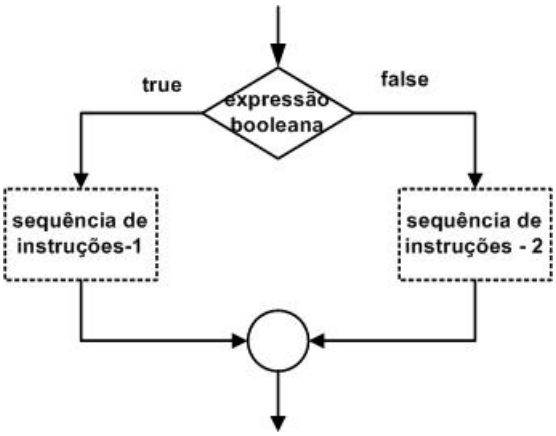

Figura 3: Estrutura de Seleção `if - else`.

```
1 if expressão booleana: # True vá p/ linha 2 - False vá p/ linha 3
2    sequência de instruções-1 # vá p/ linha 5
3 else: # vá p/ linha 4
4    sequência de instruções-2 # vá p/ linha 5
5 instrução
```

Se a `expressão booleana` for `True`, a `sequência de instruções-1` será executada e `sequência de instruções-2` será ignorada. Caso contrário, se a expressão booleana for `red`, a `sequência de instruções-1` será ignorada e a `sequência de instruções-2` será executada. Em qualquer caso, a execução continua com a instrução da linha 5, após o final do bloco `if-else`, no mesmo nível de indentação que `if-else`, a menos que a execução seja finalizada ou transferida para uma outra parte do programa por uma das instruções na sequência de instruções selecionada.

Essa estrutura de seleção `if-else` permite ao programador não somente especificar a sequência de instruções selecionadas para a execução quando a expressão booleana for `True` (verdadeira) mas também para indicar uma sequência de instruções alternativa para a execução quando ela for `False` (falsa). No exemplo

```
1 if media >= 6.0: # True vá p/ linha 2 - False vá p/ linha 4
2    print('Parabéns!') # vá p/ linha 3
3    print('Aprovado') # vá p/ linha 6
4 else: # vá p/ linha 5
5    print('Reprovado') # vá p/ linha 6
6 instrução
```

onde a expressão booleana `media >= 6.0` é avaliada, e se ela for verdadeira (`True`), as instruções das linhas 2 e 3 (`print('Parabéns!')` e `print('Aprovado')`) serão executadas e após a execução o programa será "desviado" para a linha 6, ignorando as linhas 4 e 5. Caso

contrário, se `media >= 6.0` for falsa (`False`) o programa será "desviado" para linha 4 e a instrução da linha 5, `print('Reprovado')`, será executada. Como não há mais instruções ni bloco `else` o programa executa a linha 6, que está no mesmo nível de indentação que `if-else`.

5 Solucionando Problemas Numéricos com a Estrutura de Seleção

5.1 Valor Absoluto

Um exemplo bem simples da utilização da instrução condicional é o cálculo do valor absoluto de um número real. A definição de $|x|$ é dada por:

$$|x| = \begin{cases} x \text{ se } x \geq 0 \\ -x \text{ se } x < 0 \end{cases}$$

Exemplos

```
# formato da entrada
27

# formato da saída
|27| == 27
```

```
# formato da entrada
-55

# formato da saída
|-55| == 55
```

O cálculo do valor absoluto de um número consiste em ler um número do tipo `float` e verificar se o número é maior ou igual a zero ou menor que zero. Se o número for maior ou igual a zero, o valor absoluto do número é o próprio número e no caso se o número for negativo, o valor absoluto será o número multiplicado por -1.

Para esse problema, o descrição do problema é imediata:

```
# Este programa computa o valor absoluto de um número do tipo float. O
# programa lê um número do tipo float e verifica se o número é maior ou
# igual a zero. Em caso afirmativo, o valor absoluto do número é o próprio
# número, caso contrário, o valor absoluto do número é o valor do número
# multiplicado por -1. O programa imprime o número e o valor absoluto de
# número.
```

Neste problema, a entrada consiste da leitura de um número do tipo `float`. A saída a ser gerada consiste do cálculo do valor absoluto no número lido e a impressão desse valor de acordo com um formato especificado.

As especificações de entrada e saída para o problema são as seguintes:

Entrada	Saída
Um número do tipo `float`.	Valor absoluto do número lido

Os identificadores abaixo indicam as variáveis e seus respectivos tipos para armazenar as informações de entrada e saída.

```
# descrição das variáveis utilizadas
# float x, absx
```

e as pré e pós-condições ficam:

```
# pré: x

# pós: absx == x and x >= 0 or absx == -x
```

Exemplo 1:

```
# formato da entrada
27 # valor de x

# estado da variável após a leitura
x == 27

# cálculo de |x|
absx = x

# estado das variáveis após o cálculo
x == 27
absx == 27

# formato da saída
|27| == 27
```

Exemplo 2:

```
# formato da entrada
-55 # valor de x

# estado da variável após a leitura
x == -55

# cálculo de |x|
absx = -x

# estado das variáveis após o cálculo
x == -55
absx == 55
```

```
# formato da saída
|-55| == 55
```

Podemos solucionar esse problema com os seguintes passos do algoritmo:

```
Algoritmo: Valor Absoluto
# Passo 1. Leia um número real
# Passo 2. Calcule o valor absoluto do número lido
# Passo 3. Imprima o número e seu valor absoluto
# fim do Algoritmo
```

O Passo 2 pode ser implementado diretamente com o uso da função `abs(x)`. Na nossa solução vamos explorar a funcionalidade da instrução `if else`. Na computação do valor absoluto, temos que verificar se o valor da entrada `x` é maior ou igual a 0 (`x >= 0.0`), se o resultado da comparação for verdadeiro, o valor absoluto de `x` é o próprio `x`, caso contrário o valor absoluto de `x` será `-x`. Isso pode ser implementado em Python usando a instrução `if else`:

```
# Passo 2. Calcule o valor absoluto do número lido
# Passo 2.1. Se x >= 0:
if x >= 0.0:
   absx = x
# Passo 2.2. Caso contrário
else:
   absx = -x
```

Exemplo 1

```
# formato da entrada
27

# cálculo do valor absoluto
if x >= 0.0:
   absx = x
else:
   absx = -x

# como x == 27, a expressão x >= 0 é verdadeira - True
# e será executada a instrução depois de ``:'' e antes do else.

if 27 >= 0.0: # True
   absx = x
```

Exemplo 2

```
# formato da entrada
-55
```

```
# cálculo do valor absoluto
if x >= 0.0:
   absx = x
else:
   absx = -x

# como x == -55, a expressão if x >= 0 é falsa - False
# e será executada a instrução depois do else:.

if -55 >= 0.0: # False
else:
   absx = -x
```

Como já destacamos anteriormente, a linguagem Pyhon usa tipagem dinâmica e não exige a declaração das variáveis no programa. Mesmo assim, é importante descrever as variáveis que serão utilizadas no programa. No nosso caso, utilizaremos variáveis do tipo `float` para armazenar os valores de `x` e `absx`. Quando da implementação do programa pode surgir a necessidade da utilizarmos outras variáveis para armazenar valores intermediários.

O programa `valorabasoluto.py` apresenta uma implementação em Python:

```
# -*- coding: utf-8 -*-
# Programa: valorabsoluto.py
# Programador:
# Data: 02/09/2018
# Este programa computa o valor absoluto de um número do tipo float. O
# programa lê um número do tipo float e verifica se o número é maior ou
# igual a zero. Em caso afirmativo, o valor absoluto do número é o próprio
# número, caso contrário, o valor absoluto do número é o valor do número
# multiplicado por -1. O programa imprime o número e o valor absoluto de
# número.
# início do módulo principal
# descrição das variáveis utilizadas
# float x, absx

# pré: x

# Passo 1. Leia um número real
x = float(input())
# Passo 2. Calcule o valor absoluto do número lido
# Passo 2.1. Se x >= 0
if x >= 0.0:
   absx = x
# Passo 2.2. Caso contrário
else:
   absx = x
# Passo 3. Imprima o número e seu valor absoluto
print('|{0:.1f}| == {1:.1f}'.format(x, absx))

# pós: absx == x and x >= 0 or absx == -x
```

```
30 # fim do módulo principal
```

Vamos agora simular duas execuções do programa `valorabsoluto.py`. Lembramos, que com exceção da linha 01, todas as demais linhas que iniciam com `#` serão ignoradas pelo interpretador. Verificação e Teste:

Exemplo 1

```
# execução do programa
# vamos assumir que o valor da leitura será 27
# o valor em azul será digitado pelo usuário, seguido de um enter
27 # linha 18

# estado das variáveis após a leitura dos dados (após a linha 18)
x == 27.0

# as linhas 20 a 23 computam o valor absoluto de x
# na execução da linha 20 temos
if x >= 0.0:
if 27.0 >= 0.0:
# a instrução if avalia a expressão 27.0 >= 0.0, cujo resultado é True,
# e com isso executa a instrução da linha 21
   x = x
# e passa a executar o programa a partir da linha 24, as linhas
# 22 e 23 são ignoradas

# na execução da linha 25 será impresso na tela o valor de x e absx
|27.0| == 27.0

# e o programa é encerrado
```

Exemplo 2

```
# execução do programa
# vamos assumir que o valor da leitura será -55
# o valor em azul será digitado pelo usuário, seguido de um enter
-55 # linha 18

# estado das variáveis após a leitura dos dados (após a linha 18)
x == -55.0

# as linhas 20 a 23 computam o valor absoluto de x
# na execução da linha 20 temos
if x >= 0.0:
if -55.0 >= 0.0: # False
# a instrução if avalia a expressão -55.0 >= 0.0, cujo resultado
# é False, e com isso não executa a instrução da linha 21 e
# passa a executar as linhas 22 e 23
else:
   absx = -x
```

```
# e continua a executar o programa a partir da linha 24

# na execução da linha 25 será impresso na tela o valor de x e absx
|-55.0| == 55.0

# e o programa é encerrado
```

5.2 Poluição I

O nível de poluição do ar da cidade de Cachorro Sentado é medido por um índice de poluição. As leituras são feitas às 12:00 P.M. em três locais: Olaria do Cerrado, no centro da cidade na esquina da Avenida Lassie com a Rua Dálmata, e em uma localização aleatoriamente selecionada em uma área residencial. A média aritmética das três leituras é o índice de poluição, e valores de 50.0 ou maiores indicam uma condição perigosa, enquanto valores menores que 50.0 indicam uma condição segura. Visto que esse cálculo deve ser feito diariamente, O centro de Estatísticas Ambientais de Cachorro Sentado deseja um programa que calcule o índice de poluição e então determine qual a condição da poluição, segura ou perigosa.
Exemplo:

```
# formato da entrada
LIMITE = 50.0 # constante
44 50 50 # leituras das medidas

# formato da saída
Condição Segura
```

As informações relevantes consistem de três leituras de poluição e um valor limite usado para distinguir entre condição segura e condição perigosa. A solução para o problema consiste na computação do índice de poluição, comparação com o valor limite e a impressão de uma mensagem indicando a condição da poluição do ar. A solução pode ser generalizada para qualquer valor de corte, e não apenas para o valor 50.0, nesse caso descrição detalhado do problema pode ser como abaixo:

```
# Este programa lê três medidas de poluição, leitura1, leitura2, leitura3,
# dadas por números inteiros, o valor limite (LIMITE) e calcula um índice
# de poluição, um número real. Se o valor do índice for menor que LIMITE,
# uma mensagem indicando 'Condição Segura' será impressa; caso contrário,
# uma mensagem indicando 'Condição Perigosa será impressa.
```

Para este problema, a entrada consiste de três leituras de poluição e um valor limite que distingue entre a condição segura e condição perigosa. A saída a ser produzida consiste de um índice de poluição, que é a média aritmética das três leituras, e uma mensagem indicando a condição apropriada.

As especificações de entrada e saída para o problema são as seguintes:

Entrada	Saída
3 números inteiros representado as leituras de as leituras de poluição e uma constante `leitura1`, `leitura2`, `leitura3`, `LIMITE`	Uma MSG informando "condição segura" ou "condição perigosa". `msg`

Os identificadores abaixo indicam as variáveis e constantes e seus respectivos tipos para armazenar as informações de entrada e saída.

```
# descrição das variáveis e constantes utilizadas
# LIMITE = 50.0
# int leitura1, leitura2, leitura3
# float indice
# str msg
```

e as pré e pós-condições ficam:

```
# pré: LIMITE leitura1 leitura2 leitura3

# pós: (Soma i em {1,2,3}: leitura[i])/3.0 < LIMITE and
#      msg == 'Condição Segura' or  msg == 'Condição Perigosa'
```

Exemplo 1:

```
# formato da entrada
LIMITE = 50.0 # constante
44 50 50 # leituras das medidas

# estado das variáveis após a leitura
LIMITE == 50.0
leitura1 == 44
leitura2 == 50
leitura3 == 50

# cálculo do índice
indice = (leitura1 + leitura2 + leitura3)/3.0

# estado da variável após o cálculo
indice == 48.0

# formato da saída
Condição Segura
```

Exemplo 2:

```
# formato da entrada
LIMITE = 50.0 # constante
44 50 60 # leituras das medidas
```

```
# estado das variáveis após a leitura
LIMITE == 50.0
leitura1 == 44
leitura2 == 50
leitura3 == 60

# cálculo do índice
indice = (leitura1 + leitura2 + leitura3)/3.0

# estado da variável após o cálculo
indice == 51.333333

# formato da saída
Condição Perigosa
```

Vamos agora descrever o algoritmo para solucionar o problema. O primeiro passo em um algoritmo para resolver este problema é o de obter os valores para os itens de entrada - as três leituras de poluição e o valor limite. O próximo passo é calcular o índice de poluição computando a média aritmética das três medidas. Após o cálculo do índice de poluição devemos tomar uma decisão e computar a mensagem que será impressa, uma mensagem indicando condição segura, ou uma mensagem indicando condição perigosa.

```
Algoritmo: Poluição
# Passo 1. Leia as três leituras de poluição
# Passo 2. Calcule o índice de poluição
# Passo 3. Compute a mensagem relativa ao índice de poluição
# Passo 4. Imprima a mensagem
fim do Algoritmo
```

Temos que detalhar um pouco mais o Passo 3. Após o cálculo do índice, temos que comparar o valor obtido com o valor limite. Se o valor do índice for menor que o valor limite, atribuiremos `'Condição Segura'` a variável mensagem, caso contrário, atribuiremos `'Condição Perigosa'`.

```
# Passo 3. Compute a mensagem relativa ao índice de poluição
# Passo 3.1. Se indice < LIMITE
# Passo 3.1.1. Atribua 'Condição Segura'
# Passo 3.2. Caso contrário
# Passo 3.2.1. Atribua 'Condição Perigosa'
```

O Passo 3 pode ser implementado na linguagem Python usando a instrução `if-else` e a atribuição de string:

```
# Passo 3. Compute a mensagem relativa ao índice de poluição
# Passo 3.1. Se indice < LIMITE
if indice < LIMITE:
   msg = 'Condição Segura'
# Passo 3.2. Caso contrário
else:
   msg = 'Condição Perigosa'
```

Exemplo 1

```
# valores da entrada
LIMITE = 50.0
44 50 50

# cálculo do índice
indice = (44 + 50 + 50)/3.0

if indice < LIMITE:
   msg = 'Condição Segura'
else:
   msg = 'Condição Perigosa'

# como indice == 48.0, a expressão indice < LIMITE é verdadeira - True
# e será executada a instrução depois de ``:´´ e antes do else.

if 48.0 < 50.0: # True
   msg = 'Condição Segura'
```

Exemplo 2

```
# valores da entrada
LIMITE = 50.0
44 50 60

# cálculo do índice
indice = (44 + 50 + 60)/3.0

if indice < LIMITE:
   msg = 'Condição Segura'
else:
   msg = 'Condição Perigosa'

# como indice == 51.333333, a expressão indice < LIMITE é falsa - False
# e será executada a instrução depois do else:.

if 51.333333 < 50.0: # False
else:
   msg = 'Condição Perigosa'
```

Com observamos anteriormente, a linguagem Pyhon usa tipagem dinâmica e não exige a declaração das variáveis no programa. Mesmo assim, é importante descrever as constantes e variáveis que serão utilizadas no programa. No nosso caso, definimos uma constante `LIMITE = 50.0` e utilizaremos variáveis do tipo `int` para armazenar os valores de `leitura1`, `leitura2` e `leitura3` e uma variável do tipo `float` para armazenar o `indice` de poluição. Além disso, necessitamos de uma variável do tipo `string`, `mensagem` para armazenar a mensagem correspondente. Quando da implementação do programa pode surgir a necessidade da utilizarmos outras variáveis para armazenar valores intermediários.

O programa poluica01.py descreve implementação em Python para o problema:

```
1  # -*- coding: utf-8 -*-
2  # Programa: poluicao01.py
3  # Programador:
4  # Data: 30/03/2011
5  # Este programa lê três medidas de poluição, leitura1, leitura2, leitura3,
6  # dadas por números inteiros, o valor limite (LIMITE) e calcula um índice
7  # de poluição,um número real. Se o valor do índice for menor que LIMITE,
8  # uma mensagem indicando 'Condição Segura' será impressa; caso contrário,
9  # uma mensagem indicando 'Condição Perigosa será impressa.
10 # início do módulo principal
11 # descrição das variáveis e constantes utilizadas
12 # LIMITE = 50.0
13 # int leitura1, leitura2, leitura3
14 # float indice
15 # str msg
16
17 # pré: LIMITE leitura1 leitura2 leitura3
18
19 # Passo 1. Leia a entrada
20 # Passo 1.1. Defina a constante do valor limite
21 LIMITE = 50
22 # Passo 1.2. Leia três medidas de poluição
23 leitura1 = int(input('Entre com a primeira medida de poluição: '))
24 leitura2 = int(input('Entre com a segunda medida de poluição: '))
25 leitura3 = int(input('Entre com a terceira medida de poluição: '))
26 # Passo 2. Calcule o índice de poluição
27 indice = (leitura1 + leitura2 + leitura3)/3.0
28 # Passo 3. Compute a mensagem relativa ao índice de poluição
29 if indice < LIMITE:
30    msg = 'Condição Segura'
31 else:
32    msg = 'Condição Perigosa'
33 # Passo 4. Imprima a mensagem
34 print(msg)
35
36 # pós: (Soma i em {1,2,3}: leitura[i])/3.0 < LIMITE and
37 #      msg == 'Condição Segura' or  msg == 'Condição Perigosa'
38 # fim do módulo principal
```

Vamos agora simular duas execuções do programa poluicao01.py. Lembramos, que com exceção da linha 01, todas as demais linhas que iniciam com # serão ignoradas pelo interpretador. Verificação e Teste:

Exemplo 1

```
# execução do programa
# a linha 21 é executada e o valor 50.0 é atribuído a constante LIMITE
# vamos assumir que o valor das leituras será 44 50 50
# os valores em azul serão digitados pelo usuário, seguido de um enter
```

```
Entre com a primeira medida de poluição: 44 # linha 23
Entre com a segunda medida de poluição: 50 # linha 24
Entre com a terceira medida de poluição: 50 # linha 25

# estado das variáveis após a leitura dos dados (após a linha 25)
LIMITE == 50.0
leitura1 == 44
leitura2 == 50
leitura3 == 50

# a linha 27 computa o índice de poluição
indice = (leitura1 + leitura2 + leitura3)/3.0
indice = (44 + 50 + 50)/3.0
indice = 144/3.0
indice = 48.0

# estado das variáveis após a linha 27
LIMITE == 50.0
leitura1 == 44
leitura2 == 50
leitura3 == 50
indice = 48.0

# na execução da linha 29 temos
if indice < LIMITE:
if 48.0 < 50.0:
# a instrução if avalia a expressão 48.0 < 50.0, cujo resultado é True,
# e com isso executa a instrução da linha 30
   msg = 'Condição Segura'
# e passa a executar o programa a partir da linha 33, as linhas
# 31 e 32 são ignoradas

# na execução da linha 34 será impresso na tela o valor da variável msg
Condição Segura

# e o programa é encerrado
```

Exemplo 2

```
# execução do programa
# a linha 21 é executada e o valor 50.0 é atribuído a constante LIMITE
# vamos assumir que o valor das leituras será 44 50 60
# os valores em azul serão digitados pelo usuário, seguido de um enter
Entre com a primeira medida de poluição: 44 # linha 23
Entre com a segunda medida de poluição: 50 # linha 24
Entre com a terceira medida de poluição: 60 # linha 25

# estado das variáveis após a leitura dos dados (após a linha 25)
LIMITE == 50.0
leitura1 == 44
```

```
leitura2 == 50
leitura3 == 60

# a linha 27 computa o índice de poluição
indice = (leitura1 + leitura2 + leitura3)/3.0
indice = (44 + 50 + 60)/3.0
indice = 154/3.0
indice = 51.333333

# estado das variáveis após a linha 27
LIMITE == 50.0
leitura1 == 44
leitura2 == 50
leitura3 == 60
indice = 51.333333

# na execução da linha 29 temos
if indice < LIMITE:
if 51.333333 < 50.0:
# a instrução if avalia a expressão 51.333333 < 50.0, cujo resultado
# é False, e com isso não executa a instrução da linha 30 e
# passa a executar as linhas 31 e 32
else:
   msg = 'Condição Perigosa'

# e continua a executar o programa a partir da linha 33

# na execução da linha 34 será impresso na tela o valor da variável msg
Condição Perigosa

# e o programa é encerrado
```

6 Solucionando Problemas Computacionais com Texto

6.1 Cifrador de César II

Vimos anteriormente que o *Cifrador de César* é um esquema de codificação na qual cada letra é substituída por uma letra diferente do alfabeto, digamos a terceira letra seguindo-a. Desta forma a mensagem

```
algoritmos
```

é codificada no Cifrador de César usando o deslocamento igual a 3 como

```
dojrulwprv
```

Com a estrutura de seleção podemos ampliar o escopo desse problema e analisar situações onde os caracteres na palavra não são letras. Vamos agora descrever uma forma de como

codificar uma dada palavra com 5 caracteres usando o Codificador de César. Dada uma palavra com 5 caracteres visíveis do alfabeto ASCII e um número inteiro entre 0 e 25, codificaremos a palavra convertendo cada caractere da palavra que pertença ao alfabeto {`'A',..,'Z'`} para o seu código numérico ASCII correspondente e adicionamos o inteiro dado. O valor numérico obtido nessa operação será novamente convertido para o caractere correspondente do código ASCII. Consideramos o alfabeto de forma circular, ou seja, `'Z'+1 == 'A'`. Caso o caractere pertença ao alfabeto {`'a',..,'z'`}, antes da conversão, o caractere será convertido para maiúsculo (`'a'` para `'A'`, `'b'` para `'B'`, `..`, `'z'` para `'Z'`). Os caracteres da palavra que não pertencerem ao alfabeto {`'A', .., 'Z', 'a', .., 'z'`}, permanecerão inalterados na codificação.

Exemplo:

```
# formato da entrada
AuLA1 # palavra da entrada
10 # valor do deslocamento

# formato da saída
KEVK1
```

A entrada é dada por uma string com 5 caracteres ASCII visíveis e um inteiro no intervalo $[0, 25]$. A descrição do problema define com mais clareza alguns detalhes do problema.

```
# Este programa lê uma palavra composta por 5 caracteres visíveis e um
# número inteiro no intervalo [0,25]. O programa codifica os caracteres da
# palavra que pertencem ao alfabeto {'A', .., 'Z', 'a', .., 'z'} usando o
# deslocamento dos caracteres da string de entrada. Os caracteres do
# alfabeto são transformados em maiúsculos e são tratados de forma
# circular, ou seja, 'Z' + 1 == 'A'. Os demais caracteres permanecem
# inalterados. A codificação da palavra é feita usando os códigos ASCII
# dos caracteres. O programa imprime a palavra codificada.
```

As especificações de entrada e saída ficam:

Entrada	**Saída**
Uma palavra com 5 caracteres do ASCII visíveis e um inteiro no intervalo $[0, 25]$	Uma palavra codificada com o Cifrador de César usando o deslocamento dos caracteres

Os identificadores abaixo representam as variáveis do tipo `float` para armazenar as informações da entrada e saída:

```
# descrição das variáveis utilizadas
# str palavra, codificada
# int deslocamento
```

Exemplo:

```
# formato da entrada
AuLA1 # entrada da string
```

```
10 # valor do deslocamento

# estado das variáveis após a leitura
palavra == 'AuLA1'
deslocamento == 10

# codificação da palavra, caractere a caractere
caractere = 'A'
# converter o valor para o intervalo [0,25]
codigo = ord('A') - ord('A')

# somar o deslocamento
codigo = codigo + deslocamento
codigo = 0 + 10
codigo = 10

# manter no intervalo [0,25]
codigo = codigo % 26

# converter o valor para o intervalo [65,90]
codigo  = 10 + ord('A')
codigo = 10 + 65
codigo = 75

# computar o caractere correspondente
caractereC = chr(codigo)
caractereC = chr(75)
caractereC = 'K'

caractere = 'u'
# converter para maiúscula
caractere = 'U'
# converter o valor para o intervalo [0,25]
codigo = ord('U') - ord('A')

# somar o deslocamento
codigo = codigo + deslocamento
codigo = 20 + 10
codigo = 30

# manter no intervalo [0,25]
codigo = codigo % 26
codigo = 30 % 26
codigo = 4

# converter o valor para o intervalo [65,90]
codigo  = 4 + ord('A')
codigo = 4 + 65
codigo = 69
```

```
# computar o caractere correspondente
caractereC = chr(codigo)
caractereC = chr(69)
caractereC = 'E'

...

caractere = '1'
# não pertence ao alfabeto {'A', .., 'Z', 'a', .., 'z'}
caractereC = '1'

# juntando os passos
# se caractere pertence a {'A', .., 'Z'}
caractereC = chr((ord(caractere) - ord('A') + codificador)%26 + ord('A'))
# se caractere pertence a {'a', .., 'z'}
caractere = caractere.upper()
caractereC = chr((ord(caractere) - ord('A') + codificador)%26 + ord('A'))
# se caractere não pertence ao alfabeto {'A', .., 'Z', 'a', .., 'z'}
caractereC = caractere
```

Após um melhor entendimento do problema, temos que as pré e pós-condições ficam:

```
# pré: palavra deslocamento and palavra[i] caractere visível

pós: codificada and codificada[i] caractere visível and
     para i em {0,..,4}:codificada[i] == (palavra[i] + deslocamento
     and palavra[i] em {'A',..,'Z'}) or (codificada[i]==palavra[i].upper()
     + deslocamento and palavra[i] em {'a', .., 'z'}) or codificada[i] ==
     palavra[i]
```

Nas especificações de entrada e saída, usamos a descrição dos caracteres de uma string. Como vimo anteriormente, o Cifrador de César utilizam funções que avaliam os códigos ASCII dos caracteres de uma string. Vamos agora descrever os passos do algoritmo para a solução do problema.

```
Algoritmo: Cifrador de César 01
# Passo 1. Leia uma palavra com 5 caracteres visíveis e o deslocamento
# Passo 1.1. Leia uma palavra
# Passo 1.2. Leia o deslocamento
# Passo 2. Codifique a palavra
# Passo 3. Imprima a palavra codificada
# fim do algoritmo
```

O Passo 2 necessita um refinamento. Como vimos acima, a codificação da palavra é feita caractere a caractere.

```
# Passo 2. Codifique a palavra
# Passo 2.1. Codifique o primeiro caractere
# Passo 2.2. Codifique o segundo caractere
```

```
# Passo 2.3. Codifique o terceiro caractere
# Passo 2.4. Codifique o quarto caractere
# Passo 2.6. Codifique o quinto caractere
# Passo 2.7. Concatene os caracteres
```

Mesmo com esse refinamento, temos que detalhar como será feito a codificação de cada caractere, pois ela depende se o caractere é do alfabeto {`'A', .., 'Z', 'a', .., 'z'`}, e se é maiúsculo ou minúsculo.

```
# Passo 2.1. Codifique o primeiro caractere
# Passo 2.1.1. Se o caractere for do alfabeto
# Passo 2.1.1.1. Transforme o caractere para maiúsculo
# Passo 2.1.1.2. Codifique o caractere
# Passo 2.1.2. Caso contrário
# Passo 2.1.2.1. Mantenha o caractere
```

Uma vez finalizado o refinamento, vamos codificar a solução na linguagem Python.

Os Passos 1 e 3 são facilmente implementados em Python. Para implementarmos o Passo 2 temos que entender como o Python avalia cada um dos caracteres da string `palavra` para ver se são letras ou não. Considerando que a string de entrada é composta por 5 caracteres, como vimos anteriormente, o acesso a cada um dos caracteres de `palavra` pode ser feito da seguinte maneira:

```
palavra = 'AuLA1'
palavra[0] == 'A'
palavra[1] == 'u'
palavra[2] == 'L'
palavra[3] == 'A'
palavra[4] == '1'
```

onde `palavra[i]`, $0 \leq i \leq 4$ armazena o i-ésimo caractere da string `palavra`. Como vimos anteriormente, se o caractere for uma letra maiúscula, a sua codificação pode ser feita com:

```
caractereC = chr((ord(caractere) - ord('A') + codificador)%26 + ord('A'))
```

sendo que para letras minúsculas, antes da codificação elas serão transformadas para letras maiúsculas antes da codificação. Para os demais caracteres (visíveis) a codificação é o próprio caractere.

Lembramos mais uma vez que o tipo string é imutável, ou seja, não é possível atribuir ou alterar caracteres individuais no tipo string. Como no exemplo do programa `codifica00.py`, vamos codificar cada um dos cinco caracteres da string e ao final concatená-los para obter a palavra codificada.

Exemplo

```
# formato da entrada
AuLA1
10

# estado das variáveis após a leitura
palavra == 'AuLA1'
```

```
deslocamento == 10

# avaliando caractere a caractere
# caractereC = chr((ord(caractere) - ord('A') + codificador)%26 + ord('A'))
# codifica os 5 caracteres da variável palavra
# verifica se o caractere é uma letra
if palavra[0].isalpha():
# True - transforme para maiúscula
   char0 = palavra[0].upper()
   char0 = chr((ord(char0) - ord('A') + codificador)%26 + ord('A'))
# False - não é necessário codificar o caractere
else:
   char0 = palavra[0]
if palavra[1].isalpha():
# True - transforme para maiúscula
   char1 = palavra[1].upper()
   char1 = chr((ord(char1) - ord('A') + codificador)%26 + ord('A'))
# False - não é necessário codificar o caractere
else:
   char1 = palavra[1]
if palavra[2].isalpha():
# transforme para maiúscula
   char2 = palavra[2].upper()
   char2 = chr((ord(char2) - ord('A') + codificador)%26 + ord('A'))
# False - não é necessário codificar o caractere
else:
   char2 = palavra[2]
if palavra[3].isalpha():
# True - transforme para maiúscula
   char3 = palavra[3].upper()
   char3 = chr((ord(char3) - ord('A') + codificador)%26 + ord('A'))
# False - não é necessário codificar o caractere
else:
   char3 = palavra[3]
if palavra[4].isalpha():
# True - transforme para maiúscula
   char4 = palavra[4].upper()
   char4 = chr((ord(char4) - ord('A') + codificador)%26 + ord('A'))
# False - não é necessário codificar o caractere
else:
   char4 = palavra[4]
# concatena os caracteres codificados
codificada = char0 + char1 + char2 + char3 + char4

# estado da variável após a codificação
codifica == 'KEVK1'
```

Em função da codificação de cada um dos caracteres da variável `palavra`, necessitamos de 5 novas variáveis para armazenar cada um dos caracteres. No Python, um caractere é

considerado uma string. Com isso as variáveis necessárias para o programa são:

```
# Descrição das variáveis utilizadas
# str palavra, codificada
# str char0, char1, char2, char3. char4
# int deslocamento
```

O programa `cifrador01.py` abaixo implementa o algoritmo dessa versão do Cifrador de César.

```
# -*- coding: utf-8 -*-
# Programa: cifrador01.py
# Programador:
# Data: 27/04/2016
# Este programa lê uma palavra composta por 5 caracteres visíveis e um
# número inteiro no intervalo [0,25]. O programa codifica os caracteres da
# palavra que pertencem ao alfabeto {'A', .., 'Z', 'a', .., 'z'} usando o
# deslocamento dos caracteres da string de entrada. Os caracteres do
# alfabeto são transformados em maiúsculos e são tratados de forma
# circular, ou seja, 'Z' + 1 == 'A'. Os demais caracteres permanecem
# inalterados. A codificação da palavra é feita usando os códigos ASCII
# dos caracteres. O programa imprime a palavra codificada.
# início do módulo principal
# Descrição das variáveis utilizadas
# str palavra, codificada
# str char0, char1, char2, char3, char4, char
# int deslocamento

# pré: palavra deslocamento and palavra[i] caractere visível

# Passo 1. Leia uma palavra com 5 caracteres visíveis e o deslocamento
# Passo 1.1. Leia uma palavra
palavra = input('Entre com uma palavra com 5 caracteres visíveis: ')
# Passo 1.2. Leia o deslocamento
codificador = int(input('Entre com um inteiro ente 0 e 25: '))
# Passo 2. Codifique os caracteres da palavra
# Passo 2.1. Codifique o primeiro caractere
# Passo 2.1.1. Se o caractere for do alfabeto
if palavra[0].isalpha():
# Passo 2.1.1.1. Transforme o caractere para maiúsculo
   char0= palavra[0].upper()
# Passo 2.1.1.2. Codifique o caractere
   char0 = chr((ord(char0) - ord('A') + codificador)%26 + ord('A'))
# Passo 2.1.2. Caso contrário
else:
# Passo 2.1.2.1. Mantenha o caractere
   char0 = palavra[0]
# Passo 2.2. Codifique o segundo caractere
# Passo 2.2.1. Se o caractere for do alfabeto
if palavra[1].isalpha():
```

```
41 # Passo 2.2.1.1. Transforme o caractere para maiúsculo
42    char1 = palavra[1].upper()
43 # Passo 2.2.1.2. Codifique o caractere
44    char1 = chr((ord(char1) - ord('A') + codificador)%26 + ord('A'))
45 # Passo 2.2.2. Caso contrário
46 else:
47 # Passo 2.2.2.1. Mantenha o caractere
48    char1 = palavra[1]
49 # Passo 2.3. Codifique o terceiro caractere
50 # Passo 2.3.1. Se o caractere for do alfabeto
51 if palavra[2].isalpha():
52 # Passo 2.3.1.1. Transforme o caractere para maiúsculo
53    char2 = palavra[2].upper()
54 # Passo 2.3.1.2. Codifique o caractere
55    char2 = chr((ord(char2) - ord('A') + codificador)%26 + ord('A'))
56 # Passo 2.3.2. Caso contrário
57 else:
58 # Passo 2.3.2.1. Mantenha o caractere
59    char2 = palavra[2]
60 # Passo 2.4. Codifique o quarto caractere
61 # Passo 2.4.1. Se o caractere for do alfabeto
62 if palavra[3].isalpha():
63 # Passo 2.4.1.1. Transforme o caractere para maiúsculo
64    char3 = palavra[3].upper()
65 # Passo 2.4.1.2. Codifique o caractere
66    char3 = chr((ord(char3) - ord('A') + codificador)%26 + ord('A'))
67 # Passo 2.4.2. Caso contrário
68 else:
69 # Passo 2.4.2.1. Mantenha o caractere
70    char3 = palavra[3]
71 # Passo 2.5. Codifique o quinto caractere
72 # Passo 2.5.1. Se o caractere for do alfabeto
73 if palavra[4].isalpha():
74 # Passo 2.5.1.1. Transforme o caractere para maiúsculo
75    char4 = palavra[4].upper()
76 # Passo 2.5.1.2. Codifique o caractere
77    char4 = chr((ord(char4) - ord('A') + codificador)%26 + ord('A'))
78 # Passo 2.5.2. Caso contrário
79 else:
80 # Passo 2.5.2.1. Mantenha o caractere
81    char4 = palavra[4]
82 # Passo 2.6. Concatene os caracteres
83 codificada = char0 + char1 + char2 + char3+ char4
84 # Passo 3. Imprima a mensagem codificada
85 print('Codificada: {0:s}'.format(codificada))
86
87 pós: codificada and codificada[i] caractere visível and
88      para i em {0,..,4}:codificada[i] == (palavra[i] + deslocamento
89      and palavra[i] em {'A',..,'Z'}) or (codificada[i]==palavra[i].upper()
90      + deslocamento and palavra[i] em {'a', .., 'z'}) or codificada[i] ==
```

```
91      palavra[i]
92  # fim do módulo principal
```

Exemplo

```
# execução do programa
# vamos assumir que os valores lidos são
# palavra == 'Aula1'
# deslocamento == 10
# o valor em azul será digitado pelo usuário, seguido de um enter
# formato da entrada
Entre com uma palavra com 5 caracteres visíveis: AuLA1 # linha 23
Entre com um inteiro ente 0 e 26: 10 # linha 25

# estado das variáveis após a leitura
palavra == 'AuLA1'
deslocamento == 10

# as linhas 26 a 81 codificam os caracteres palavra de entrada
# na execução da linha 29 temos
if palavra[0].isalpha():
if 'A'.isapha():
# a instrução if avalia a expressão 'A'.isapha(), cujo resultado é True,
# e com isso executa as instruções do programa das linhas 30 a 33
   char0 = 'A'.upper() # linha 31
   char0 = 'A' # linha 31
   char0 = chr((ord('A')-ord('A')+10) % 26 + ord('A')) # linha 33
   char0 = chr((65 - 65 + 10) % 26 + 65) # linha 33
   char0 = chr(10 % 26 + 65) # linha 33
   char0 = chr(75) # linha 33
   char0 = 'K' # linha 33
# e passa a executar o programa a partir da linha 38, as linhas
# 34 a 37 são ignoradas
# na execução da linha 40 temos
if palavra[1].isalpha():
if 'u'.isapha():
# a instrução if avalia a expressão 'u'.isapha(), cujo resultado é True,
# e com isso executa as instruções do programa das linhas 41 a 44
   char1 = 'u'.upper() # linha 42
   char1 = 'U' # linha 42
   char1 = chr((ord('U')-ord('A')+10) % 26 + ord('A')) # linha 44
   char1 = chr((85 - 65 + 10) % 26 + 65) # linha 44
   char1 = chr(30 % 26 + 65) # linha 44
   char1 = chr(69) # linha 44
   char1 = 'E' # linha 44
# e passa a executar o programa a partir da linha 49, as linhas
# 45 a 48 são ignoradas
# na execução da linha 51 temos
if palavra[2].isalpha():
if 'L'.isapha():
```

```
# a instrução if avalia a expressão 'L'.isapha(), cujo resultado é True,
# e com isso executa as instruções do programa das linhas 52 a 55
   char2 = 'L'.upper() # linha 53
   char2 = 'L' # linha 53
   char2 = chr((ord('L')-ord('A')+10) % 26 + ord('A')) # linha 55
   char2 = chr((76 - 65 + 10) % 26 + 65) # linha 55
   char2 = chr(21 % 26 + 65) # linha 55
   char2 = chr(86) # linha 55
   char2 = 'V' # linha 55
# e passa a executar o programa a partir da linha 60, as linhas
# 56 a 59 são ignoradas
# na execução da linha 62 temos
if palavra[3].isalpha():
if 'A'.isapha():
# a instrução if avalia a expressão 'A'.isapha(), cujo resultado é True,
# e com isso executa as instruções do programa das linhas 63 a 66
   char3 = 'A'.upper() # linha 64
   char3 = 'A' # linha 64
   char3 = chr((ord('A')-ord('A')+10) % 26 + ord('A')) # linha 66
   char3 = chr((65 - 65 + 10) % 26 + 65) # linha 66
   char3 = chr(10 % 26 + 65) # linha 66
   char3 = chr(75) # linha 66
   char3 = 'K' # linha 66
# e passa a executar o programa a partir da linha 71, as linhas
# 67 a 70 são ignoradas
# na execução da linha 73 temos
if palavra[4].isalpha():
if '1'.isapha():
# a instrução if avalia a expressão '1'.isapha(), cujo resultado é False,
# e com isso executa desvia a execução do programa para a linha 78 e
# executa a linha 79, e após else:
   char4 = '1' # linha 81
# e passa a executar o programa a partir da linha 82
codificada = char0 + char1 + char2 + char3 + char4 # linha 83
codificada = 'K' + 'E' + 'V' + 'K' + '1' # linha 83
# na execução das linhas 85 será gerada a saída
KEVK1

# e o programa é encerrado
```

7 Solucionando Problemas Computacionais com Ambiente Gráfico

Na Aula 05 usamos a biblioteca `stddraw.py` para imprimir um reticulado numa janela gráfica para um jogo da velha. Além disso, imprimimos também na janela um texto com informações sobre o jogo. Uma forma mais "amigável" de entrar com os movimento do jogo, é por meio do mouse. Em função disso, temos que entender como o "`click`" do mouse pode ser lido na janela gráfica. Vamos explorar esse recurso na nossa próxima versão do jogo da

velha.

7.1 Jogo da Velha - II

Vamos dar continuidade no desenvolvimento do nosso Jogo da Velha. Agora vamos incorporar a utilização do mouse para entrar com os movimentos do jogo. A biblioteca `stddraw.py` possui funções (métodos) para verificar se o mouse foi pressionado sobre a janela gráfica e "ler" as coordenadas (x, y) onde o mouse foi pressionado. Outras bibliotecas gráficas possuem funções/métodos equivalentes para "ler" os movimentos do mouse.

```
mousePressed() # verifica se o mouse foi pressionado
mouseX() # lê a coordenada x da posição que o mouse foi pressionado
mouseY() # lê a coordenada x da posição que o mouse foi pressionado
```

Na Aula 05 geramos um tabuleiro do jogo da velha com dois movimentos, um `x` e um `o`. No exemplo, as coordenadas dos movimentos eram fixas. No exemplo desta aula vamos usar o mouse para fornecer as coordenadas dos dois movimentos no tabuleiro. A descrição e as especificações de entrada e saída deste problema são as mesmas que do problema da Aula 05 (`jogodavelha00.py`). A diferença na subdivisão e implementação é a forma de como serão dadas as duas coordenadas.

A subdivisão do nosso problema fica:

```
# Passo 1. Gere o tabuleiro
# Passo 1.1. Defina a cor laranja do fundo do painel
# Passo 1.2. Gere um retângulo de fundo laranja
# Passo 1.3. Defina a cor para gerar as bordas do tabuleiro
# Passo 1.4. Gere as bordas do tabuleiro
# Passo 1.5. Gere um reticulado 3 x 3
# Passo 1.5.1. Gere as linhas horizontais
# Passo 1.5.2. Gere as linhas verticais
# Passo 1.6. Gere as mensagens
# Passo 1.6.1. Defina o tipo de letra para as mensagens
# Passo 1.6.2. Defina o tamanho da fonte da letra
# Passo 1.6.3. Defina a cor para as mensagens
# Passo 1.6.4. Gere as mensagens
# Passo 2. Imprima o tabuleiro do jogo na tela
# # Passo 3. Leia os movimentos do mouse
# Passo 3.1. Defina a cor para os movimentos
# Passo 3.2. Se o mouse foi pressionado
# Passo 3.2.1. Leia a coordenada x
# Passo 3.2.2. Leia a coordenada y
# Passo 3.2.3. Gere x nas coordenadas (x,y)
# Passo 3.2.4. Imprima o tabuleiro do jogo na tela com o movimento x
# Passo 3.3. Se o mouse foi pressionado
# Passo 3.3.1. Leia a coordenada x
# Passo 3.3.2. Leia a coordenada y
# Passo 3.3.3. Gere o nas coordenadas (x,y)
# Passo 3.2.4. Imprima o tabuleiro do jogo na tela com o movimento o
```

A subdivisão acima usa as funcionalidades da biblioteca `stddraw`. A implementação em `Python` fica:

```
# -*- coding: utf8 -*-
# Programa: jogodavelha01.py
# Programador:
# Data: 23/02/2020
# Este programa utiliza a classe stddraw. Ele gera e imprime um reticulado
# 3 x 3 com mensagens. O programa lê dois movimentos, um para X e outro
# para O. Os movimentos são lidos por meio do mouse. O programa gera e
# imprime o tabuleiro (reticulado) com as mensagens e os dois movimentos
# na tela.
# Declaração das classes utilizadas
import stddraw
# início do módulo principal
# Descrição das variáveis e constantes utilizadas
# float x1, y1, x2, y2

# pré: dimensões e posição do tabuleiro e reticulado e conteúdo
#      das mensagens e x1, y1, x2, y2

# Passo 1. Gere o tabuleiro
# Passo 1.1. Defina a cor laranja do fundo do painel
stddraw.setPenColor(stddraw.ORANGE)
# Passo 1.2. Gere um retângulo de fundo laranja
stddraw.filledRectangle(0.01, 0.48, 0.98, 0.34)
# Passo 1.3. Defina a cor para gerar as bordas do tabuleiro
stddraw.setPenColor(stddraw.BLACK)
# Passo 1.4. Gere as bordas do tabuleiro
stddraw.line(0.01, 0.82, 0.99, 0.82)
stddraw.line(0.01, 0.48, 0.99, 0.48)
stddraw.line(0.01, 0.48, 0.01, 0.82)
stddraw.line(0.99, 0.48, 0.99, 0.82)
stddraw.line(0.39, 0.48, 0.39, 0.82)
# Passo 1.5. Gere um reticulado 3 x 3
# Passo 1.5.1. Gere as linhas horizontais
stddraw.line(0.05, 0.8, 0.35, 0.8)
stddraw.line(0.05, 0.7, 0.35, 0.7)
stddraw.line(0.05, 0.6, 0.35, 0.6)
stddraw.line(0.05, 0.5, 0.35, 0.5)
# Passo 1.5.2. Gere as linhas verticais
stddraw.line(0.05, 0.5, 0.05, 0.8)
stddraw.line(0.15, 0.5, 0.15, 0.8)
stddraw.line(0.25, 0.5, 0.25, 0.8)
stddraw.line(0.35, 0.5, 0.35, 0.8)
# Passo 1.6. Gere as mensagens
# Passo 1.6.1. Defina o tipo de letra para as mensagens
stddraw.setFontFamily('Courier')
# Passo 1.6.2. Defina o tamanho da fonte da letra
stddraw.setFontSize(16)
# Passo 1.6.3. Defina a cor para as mensagens
stddraw.setPenColor(stddraw.BLUE)
```

```
# Passo 1.6.4. Gere as mensagens
stddraw.text(0.7,0.75, 'Bem Vindo ao Jogo da Velha! ')
stddraw.text(0.7,0.70, 'Clique numa célula para o   ')
stddraw.text(0.7,0.65, 'próximo movimento, ou clique')
stddraw.text(0.7,0.60, 'fora do reticulado para     ')
stddraw.text(0.7,0.55, 'finalizar o jogo.           ')
# Passo 2. Imprima o tabuleiro do jogo na tela
stddraw.show(1000.0)
# # Passo 3. Leia os movimentos do mouse
# Passo 3.1. Defina a cor para os movimentos
stddraw.setPenColor(stddraw.RED)
# Passo 3.2. Se o mouse foi pressionado
if stddraw.mousePressed():
# Passo 3.2.1. Leia a coordenada x
   x1 = stddraw.mouseX()
# Passo 3.2.2. Leia a coordenada y
   y1 = stddraw.mouseY()
# Passo 3.2.3. Gere x nas coordenadas (x,y)
   stddraw.text(x1,y1, 'x')
# Passo 3.2.4. Imprima o tabuleiro do jogo na tela com o movimento x
   stddraw.show(1000.0)
# Passo 3.3. Se o mouse foi pressionado
if stddraw.mousePressed():
# Passo 3.3.1. Leia a coordenada x
   x2 = stddraw.mouseX()
# Passo 3.3.2. Leia a coordenada y
   y2 = stddraw.mouseY()
# Passo 3.3.3. Gere o nas coordenadas (x,y)
   stddraw.text(x2,y2, 'o')
# Passo 3.3.4. Imprima o tabuleiro do jogo na tela com o movimento o
   stddraw.show()

# pós: Um tabuleiro com um jogo da velha e a impressão de
#      dois movimentos.
# fim do módulo principal
```

Edite o programa e execute. Dependendo de onde for pressionado o mouse, a saída deve ser como:

Para uma implementação funcional de um jogo da velha, vários pontos deverão ser abordados, tais como evitar que mais de um movimento seja registrado numa célula, término do jogo, possível ganhador, etc.

Vamos agora descrever as funcionalidade da leitura dos movimentos de um mouse no tabuleiro. Para que possamos ler os movimentos do mouse no tabuleiro, ele deve estar "ativo" e "lendo" eventos. Existem vários formas de fazer isso, uma delas é definir um valor em milisegundos na impressão do tabuleiro na janela gráfica. A instrução

```
# Passo 2. Imprima o tabuleiro do jogo na tela
stddraw.show(1000.0)
```

imprime o tabuleiro e fica lendo dentro de uma dada faixa de tempo se ocorre algum evento (por exemplo, se o mouse foi pressionado na região). A instrução do Passo 3.1 simples-

Figura 4: Monitorando o Jogo da Velha.

mente muda a cor que será usada no próxima impressão. No Passo 3.2., a instrução `mousePressed()` retorna verdadeiro `True` se o mouse foi pressionado dentro do intervalo de tempo que a janela está ativa (
tt Passo 2). Se o mouse foi pressionado e o movimento captado, serão executadas as instruções dos Passos 3.2.1 a 3.2.4. A instrução do Passo 3.2.1., `mouseX()`, lê o valor da coordenada `x` do local onde o mouse foi pressionado e atribui o valor para a variável `x1`. De maneira análoga, o Passo 3.2.2. atribui o valor da coordenada `y` para a variável `y1`.

```
# # Passo 3. Leia os movimentos do mouse
# Passo 3.1. Defina a cor para os movimentos
stddraw.setPenColor(stddraw.RED)
# Passo 3.2. Se o mouse foi pressionado
if stddraw.mousePressed():
# Passo 3.2.1. Leia a coordenada x
   x1 = stddraw.mouseX()
# Passo 3.2.2. Leia a coordenada y
   y1 = stddraw.mouseY()
```

sendo que o Passo 3.2.3. gera um texto "`x`" na posição $(x1, y1)$ da janela gráfica e o Passo 3.2.4. imprime a janela gráfica na tela com todas as componentes geradas até o momento (de forma cumulativa).

```
# Passo 3.2.3. Gere x nas coordenadas (x,y)
   stddraw.text(x1,y1, 'x')
# Passo 3.2.4. Imprima o tabuleiro do jogo na tela com o movimento x
   stddraw.show(1000.0)
```

Se o Passo 3.2. for `False`, o programa é desviado para a instrução do Passo 3.3.

O Passo 3.3. repete a execução do Passo 3.2.

```
# Passo 3.3. Se o mouse foi pressionado
if stddraw.mousePressed():
# Passo 3.3.1. Leia a coordenada x
   x2 = stddraw.mouseX()
# Passo 3.3.2. Leia a coordenada y
   y2 = stddraw.mouseY()
# Passo 3.3.3. Gere o nas coordenadas (x,y)
   stddraw.text(x2,y2, 'o')
# Passo 3.3.4. Imprima o tabuleiro do jogo na tela com o movimento o
   stddraw.show()
```

Se o Passo 3.3. for `False`, o programa é desviado para a instrução após o Passo 3.3.4. e nesse caso, o programa é finalizado.

Usando um editor de programas, edite o programa `jogodavelha01`. Execute o programa pressionado o mouse em várias regiões da janela gráfica. Note que o programa não analise a região onde o mouse foi pressionado para verificar se o movimento é válido ou não.