Este documento apresenta uma revisão dos principais assuntos abordados em Programação 1.

1 Fundamentos da programação estruturada

Qual a diferença entre **programa**, **linguagem de programação**, **código fonte** e **agoritmo**?

Para o computador, um **programa** é apenas uma sequência de instruções (código de máquina) composto por 0's e 1's. Máquinas não entendem linguagens humanas (português, inglês, *etc*) e humanos, geralmente, não entendem código de máquina. Uma **linguagem de programação** é um meio termo entre humanos e máquinas. Possui um conjunto de regras para descrever um funcionamento de um programa. Um **código-fonte** contém **algoritmos** escritos em alguma linguagem de programação. É preciso haver uma **tradução** de código-fonte escrito em uma certa linguagem de programação para que ele seja transformado em um programa executável por um computador.

Um **paradigma de programação** define o "estilo" de uma linguagem de programação. Existem diversos paradigmas de programação diferentes. Exemplos: *paradigma estruturado*, *paradigma funcional*, *paradigma orientado a objetos*. Python é multiparadigma: é possível escrever código em Python usando o paradigma estruturado, funcional e OO. Entretanto, estudamos apenas o paradigma estruturado nesta disciplina. A programação estruturada é composta, basicamente, por:

- Sequências de instruções:

```
1 x = 10
2 y = 30
3 z = x+y
4 print(z)
```

  O computador irá sempre executar as instruções na ordem em que aparecem no código, a não ser que haja algum comando que o faça desviar para outro trecho do código.

- Entrada e saída (*Input/Output*, ou apenas I/O): Um programa pode interagir com o usuário de diversas formas. Existem dispositivos de entrada de dados, para que o usuário passe comandos ao programa (como *mouse* e teclado), e os dispositivos de saída, para que ele receba informações do programa (como o monitor, uma impressora ou até mesmo um HD). Dizemos que o teclado é a entrada de dados padrão (*standard input*) e o monitor é a saída padrão (*standard output*). Em Python, a entrada de dados padrão é feita pela função **input** e a saída padrão é feita pela função **print**:

```
1 # Lendo dois numeros e transformando-os em inteiros:
2 x = int(input("Digite um numero: "))
3 y = int(input("Digite outro numero: "))
4
5 # Imprimindo a soma dos dois numeros:
6 print(x, "+", y, "=", x+y)
7
8 # Outra formatacao possivel para a mesma impressao:
9 print("{} + {} = {}".format(x, y, x+y))
```

  Se os número digitados forem $x = 45$ e $y = 3$, por exemplo, o trecho de código acima irá imprimir na tela:

```
45 + 3 = 48
45 + 3 = 48
```

- **Decisões**: O comando **if** em Python desvia o código para trechos diferentes dependendo do resultado de uma condição, geralmente expressa por uma expressão booliana (verdadeiro ou falso):

```
1 if x>y:
2     print(z)
3 else:
4     print(x+y)
```

- **Repetições** (iterações): Repetições em Python podem ser realizadas com os comandos **while** ou **for**. O primeiro executa um trecho de código **enquanto** a condição for verdadeira. O segundo executa uma iteração **para cada** elemento em uma lista (veja mais sobre listas na Seção 2). Chamamos o bloco a ser repetido de laço (ou *loop*, em inglês):

```
1 while x<y:
2     print("Oi!")
3     x = x+1
4
```

```
5  frutas = ["pera", "uva", "maca"]
6  for fruta in frutas:
7      print(fruta)
8
9  for n in range(10):
10     # Lista com os 10 primeiros inteiros a partir de 0
11     print(n)
```

Cuidado: não confunda **interação** (quando o usuário troca informações com o programa que está em execução) com **iteração** (cada vez que um trecho de código é repetido).

- Sub-rotinas (funções, métodos ou procedimentos): blocos de comandos que realizam tarefas específicas e atendem a alguma necessidade, independente de quando ou qual trecho de código a solicitar. Uma **função** é uma sub-rotina que recebe um conjunto de parâmetros (que pode ser vazio), calcula um resultado e retorna-o para o trecho de código em que foi chamada. O fatorial de um número, por exemplo, é definido por:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

Matematicamente, podemos definir a função fatorial como:

$$n! = \begin{cases} n \times (n-1)!, & \text{se } n > 0 \\ 1, & \text{caso contrário.} \end{cases}$$

A linguagem Haskell (que utiliza o paradigma funcional), define funções de forma bem semelhante às definições matemáticas. A função fatorial em Haskell, por exemplo, seria:

```
1  fat 0 = 1
2  fat n = n * fat (n - 1)
```

Em Python, podemos definir a função de duas formas:

```
1  def fat(n):
2      if n > 0:
3          return n * fat(n-1)
4      else:
5          return 1
6
7  def fat2(n):
8      x = 1
```

```
9     while n > 0:
10        x = x * n
11        n = n - 1
12    return x
```

A primeira função, mais semelhante à definição matemática, é **recursiva** (ou seja, contém uma chamada à própria função fatorial). A segunda função é **iterativa**, ou seja, utiliza um comando de repetição (neste caso, um **while**) para acumular o resultado do cálculo de $n \times (n-1) \times (n-2) \times \cdots \times 1$. As duas funções recebem apenas um parâmetro (o número n) e retornam o fatorial desse número como resultado. O retorno de uma função é definido pelo comando **return**.

Uma função também pode fazer chamada à outra função e utilizar o valor retornado por ela. Uma combinação simples, por exemplo, é dada por:

$$C_{(n,p)} = \frac{n!}{p! \times (n-p)!}$$

Em Python, podemos defini-la como:

```
1 def comb(n, p):
2     num = fat(n)
3     den = fat(p) * fat(n-p)
4     return num / den
```

Quando fazemos chamada a alguma função, podemos salvar o seu valor de retorno em alguma variável (segunda linha da função **comb**), usá-lo diretamente em outra expressão (terceira linha da função **comb**) ou até mesmo imprimir diretamente o valor.

Um **procedimento** é uma sub-rotina que não possui um valor de retorno. Um procedimento em Python, assim como uma função, é definido pelo comando **def**, mas não possui um valor de retorno. Podemos, por exemplo, criar um procedimento que imprime os n primeiros múltiplos de k:

```
1 def multiplos(n, k):
2     i = 1
3     print("Os {} primeiros multiplos de {} sao:".format(n, k))
4     while i <= n:
5         print(i*k)
6         i = i+1
```

A saída do procedimento para $n = 10$ e $k = 3$ seria:

```
Os 10 primeiros multiplos de 3 sao:
3
```

```
6
9
12
15
18
21
24
27
30
```

De forma parecida, imprimimos a seguir todos os múltiplos de k até n:

```
1 def nMultiplos(n, k):
2     i = 1
3     print("Os multiplos de {} ate {} sao:".format(k, n))
4     while i*k <= n:
5         print(i*k)
6         i = i+1
```

A saída do procedimento para $n = 10$ e $k = 3$ seria:

```
Os multiplos de 3 menores ou iguais a 10 sao:
3
6
9
```

Note que, como um procedimento não possui um valor de retorno, a chamada de um procedimento não deve ser armazenada em uma variável. Exemplo:

```
1 # Retorno da funcao input sendo transformado para inteiro
2 # e depois sendo salvo na variavel x:
3 x = int(input("Digite um numero: "))
4
5 # Retorno da funcao fat sendo impresso diretamente:
6 print("{}! = {}".format(x, fat(x))
7
8 # Imprimindo os 5 primeiros multiplos de x:
9 multiplos(5, x)
```

Por fim, um **método** é uma sub-rotina que pertence aos objetos de uma classe em uma linguagem orientada a objetos (assunto de outra disciplina).

2 Listas e Matrizes

Cada variável do programa armazena um único valor (inteiro, double, string, *etc*), que pode ser acessado ou modificado ao longo do código. Uma **lista** é uma estrutura que permite trabalhar com um conjunto de dados e permite o acesso a cada um deles.

É possível alterar os valores, removê-los ou adicionar novos elementos a ela. Como exemplo, podemos ter uma lista de convidados para uma festa. Em Python, uma lista pode ser criada da seguinte forma:

```
1 # Criando uma lista de strings:
2 convidados = ["Enzo", "Valentina", "Pedro", "Maria"]
```

Em uma lista, os elementos são **indexados** (começando pela posição zero). Na lista acima, os índices de cada item são:

Índice:	0	1	2	3
Elemento:	"Enzo"	"Valentina"	"Pedro"	"Maria"

Tabela 1: Índices dos elementos em uma lista

Os índices permitem que cada elemento seja acessado individualmente:

```
1  # Alterando o valor do terceiro elemento,
2  # que se encontra na posicao 2:
3  convidados[2] = "Ana"
4
5  # Adicionando um novo elemento ao final da lista,
6  #sem modificar os demais:
7  convidados.append("Rita")
8
9  # Imprimindo a lista:
10 for pessoa in convidados:
11     print(pessoa)
```

A saída da impressão acima seria:

```
Enzo
Valentina
Ana
Maria
Rita
```

Note que, na linha 3, alteramos o terceiro elemento da lista, que deixou de ter o valor "Pedro" e passou a ter o valor "Ana". Depois, na linha 6, adicionamos um novo

elemento ao final da lista, sem modificar os demais. Por fim, na linha 9, iteramos sobre os elementos da lista. A cada iteração, escolhemos identificar cada elemento através da variável chamada *pessoa*, mas poderíamos ter utilizado qualquer outro nome de variável.

Também é possível acessar os elementos através de índices negativos. Neste caso, contamos a partir de -1, da direita para a esquerda:

Índice:	-5	-4	-3	-2	-1
Elemento:	"Enzo"	"Valentina"	"Ana"	"Maria"	"Rita"

Tabela 2: Índices negativos para os elementos em uma lista

Também podemos acessar um intervalo de índices na lista. Se acessarmos *convidados*[1 : 4], por exemplo, estaremos criando uma sublista com os elementos de *convidados* do índice 1 ao índice 4 (exceto ele). Exemplo:

```
1 print(convidados[1:4])
```

A saída da impressão acima seria:

```
['Valentina', 'Ana', 'Maria']
```

Se omitirmos o primeiro índice do intevalo, os elementos serão acessados a partir do início da lista. Se o último índice for omitido, os elementos serão acessados até o final da lista. Por fim, podemos definir um terceiro parâmetro que indica de quantos em quantos elementos queremos acessar. Exemplo:

```
1 numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
2 print(numeros[2:8:2])
3 print(numeros[::2])
```

A saída do trecho acima é:

```
[3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

A função a seguir recebe uma lista l e um número x como parâmetros e retorna uma nova lista com todos os elementos de l que forem múltiplos de x:

```
1 def buscarMultiplos(x, l):
2     l2 = []
3     for y in l:
4         if y%x == 0:
5             l2.append(y)
6
7     return l2
```

A função a seguir cria uma lista com o fatorial de todos os números até n:

```
1 from fat import fat
2
3 def fatoriais(n):
4     l = []
5     i = 1
6     while i <= n:
7         l.append(fat(i))
8         i += 1
9     return l
```

Uma lista pode conter elementos de quaiquer tipos, inclusive outras listas. Portanto, podemos utilizar listas de listas para implementarmos uma matriz em Python. Um elemento da lista contém uma linha da matriz, que por sua vez corresponde a uma lista com os elementos da coluna da matriz. A seguir, armazenamos uma matriz 3×4 como uma lista que contém 3 listas, sendo que cada uma delas contém 4 elementos:

```
1 matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ]
```

Para acessarmos o elemento $a_{3,1}$ de uma matriz $A_{5\times6}$, por exemplo, usamos $A[2, 0]$ (lembrando que os elementos de uma lista são indexados a partir de zero).

Matrizes são estruturas bidimensionais (tabelas) com m linhas por n colunas muito importantes na matemática, utilizadas por exemplo para a resolução de sistemas de equações e transformações lineares.

Podemos criar uma matriz utilizando um *loop*, no qual cada iteração irá criar uma linha da matriz:

```
1 M = []
2 for i in range(5):
3     linha = []
4     for j in range(7):
5         linha.append(0)
6     M.append( linha )
```

No exemplo acima, temos uma matriz $M_{5\times7}$ nula (todos os elementos iguais a zero). O loop interno pode ser simplificado (o externo, não):

```
1 M = []
2 for i in range(5):
3     M.append( 7*[0] )
```

A seguir, criamos uma função que cria uma matriz nula de m linhas e n colunas, e outra função para impressão da matriz:

```
1 def cria_matriz(m, n):
2     M = []
```

```
3       for i in range(m):
4           M.append( n*[0] )
5       return M
6
7   def imprime_matriz(M):
8       for linha in M:
9           for elemento in linha:
10              print(M[i][j], end='\t')
11          print()
```

A seguir, verificamos se uma matriz M é uma matriz identidade:

```
1   def verifica_identidade(M):
2       nLinhas = len(M)
3       i = 0
4       while i < nLinhas:
5           nElem = len(M[i])
6           if nLinhas != nElem: return False
7           j = 0
8           while j < nElem:
9               if i == j and M[i][j] != 1: return False
10              if i != j and M[i][j] != 0: return False
11              j += 1
12          i += 1
13      return True
```

A função a seguir recebe como parâmetros duas matrizes A e B de mesmo tamanho e retorna uma terceira matriz com o resultado de $A + B$:

```
1   def soma_matrizes(A, B):
2       nLinhas = len(A)
3       nColunas = len(A[0])
4       C = cria_matriz(nLinhas, nColunas)
5
6       for i in range(nLinhas):
7           for j in range(nColunas):
8               C[i][j] = A[i][j] + B[i][j]
9
10      return C
```

Por fim, considerando que M é uma matriz de 3 linhas e 3 colunas, calculamos o determinante de $M_{3\times3}$:

```
1   def det(M):
2       diagPrincipal1 = M[0][0] * M[1][1] * M[2][2]
```

```
3       diagPrincipal2 = M[0][1] * M[1][2] * M[2][0]
4       diagPrincipal3 = M[0][2] * M[1][0] * M[2][1]
5       soma1 = diagPrincipal1 + diagPrincipal2 + diagPrincipal3
6
7       diagSecundaria1 = M[0][2] * M[1][1] * M[2][0]
8       diagSecundaria2 = M[0][0] * M[1][2] * M[2][1]
9       diagSecundaria3 = M[0][1] * M[1][0] * M[2][2]
10      soma2 = diagSecundaria1 + diagSecundaria2 + diagSecundaria3
11
12      return soma1 - soma2
```