Introdução à Implementação de Algoritmos

Neste conteúdo veremos inicialmente uma introdução sobre algoritmos, e nos aprofundaremos um pouco acerca de sua implementação utilizando como linguagem de apoio o Phyton. Para isso, abordaremos assuntos como: tipos de variáveis, operadores lógicos e aritméticos, relações condicionais e Iterações. Esses temas são fundamentais para quem está iniciando na programação.

Sumário

Cap. 1 Definição de Algoritmo

Segundo o Dicionário Oxford, a definição matemática de Algoritmo consiste em uma sequência finita de regras, raciocínios ou operações que, aplicada a um número finito de dados, permite solucionar classes semelhantes de problemas. Enquanto a definição em informática é descrita por um conjunto de regras e procedimentos lógicos perfeitamente definidos que levam à solução de um problema em um número finito de etapas.

A seguir é apresentado um exemplo de algoritmo escrito em Pseudocódigo e em Python.

Exemplo: “Somar dois números quaisquer”

Em Pseudocódigo:

1. Escreva o primeiro número e salve na posição A.
2. Escreva o segundo número e salve na posição B.
3. Some o número da posição A com número da posição B e coloque o resultado na posição C.
4. Imprima o número da posição C.

Em Phyton:

```
A = int(input('Digite o primeiro número: '))
B = int(input('Digite o segundo número: '))
C = A + B
print('A soma é:',C)
```

Através da comparação entre o Pseudocódigo e o Código em Python apresentado acima podemos verificar os seguintes pontos sobre essa linguagem de programação:

- Para atribuir um valor à uma variável utiliza-se o símbolo “=”;
- “input()” é aplicado para que um item possa ser inserido ao programa, no caso esse item é um valor que será atribuído a variável “A”. De maneira análoga outro item será atribuido a variável “B”;
- Todo input é lido pelo Python como um texto, então para que o item inserido seja entendido como um número é necessário transformá-lo em um número, e o comando “int()” é responsável por transformar um item em um número inteiro;
- O operador utilizado para soma de dois números em Python é o símbolo “+”;
- Para apresentar um número na tela é utilizado o comando “print()”.

Ao executar o script acima e inserir os itens “5” e “4” obtemos a seguinte resposta:

```
Digite o primeiro número: 5
Digite o segundo número: 4
A soma é: 9
```

Essas poucas linhas de algoritmo podem assustar a primeira vista, parecer muita informação a ser entendida de uma vez, mas ao longo desse conteúdo explicaremos a fundo cada um desses comandos até que fique tão natural compreender, como um texto em português.

Um problema pode ser solucionado de uma infinidade de maneiras diferentes, essa variedade estende-se também aos algoritmos, que podem ser formulados de muitas maneiras diferentes e mesmo assim levar a mesma resposta.

Desenvolver algoritmos é um exercício baseado na criatividade e experiência de seu desenvolvedor. Vários algoritmos diferentes podem ao mesmo tempo estar certo, e levar a mesma resposta correta. Entretanto, o algoritmo mais “bem escrito” será aquele mais objetivo e conciso, de fácil entendimento para outros desenvolvedores.

Cap. 2 Desenvolvendo Algoritmos

Inicialmente vamos utilizar um pseudocódigo para descrever os algoritmos, que se baseia na lista de tarefas que devem ser executadas para atingir o objetivo do código. E então vamos traduzir esse pseudocódigo para a linguagem de programação Python, para que o computador possa entender essa lista de tarefas a ser executadas.

Diferentemente de uma linguagem de programação não existe um formalismo rígido de como deve ser escrito o algoritmo, ou pseudocódigo, este deve ser fácil de se interpretar e fácil de codificar, ou seja, ele corresponde ao intermediário entre uma linguagem falada e uma linguagem de programação.

Como dito anteriormente, para escrever Algoritmos é necessário uma especificar tarefas, ou seja, é necessário criar instruções / comandos a serem executados. Comandos/Instruções podem ser definidos como frases que indicam as ações a serem executadas, são compostas de um verbo no imperativo, ou no infinitivo, mais um complemento.

Analisando o exemplo da seção anterior:

Em Pseudocódigo:

1. Escreva o primeiro número e salve na posição A.
2. Escreva o segundo número e salve na posição B.
3. Some o número da posição A com número da posição B e coloque o resultado na posição C.
4. Imprima o número da posição C.

Todas as linhas correspondem a uma ação que deverá ser executada pelo computador.

Cap 3. Tipos de Variáveis

Uma variável é um espaço reservado na memória do computador para armazenar um tipo de dado determinado. Variáveis devem receber nomes para poderem ser referenciadas e modificadas quando necessário. Um programa deve conter declarações que especificam de que tipo são as variáveis que ele utilizará e as vezes um valor inicial. Tipos podem ser por exemplo: inteiros, reais, caracteres, etc. As expressões combinam variáveis e constantes para calcular novos valores.

Nesta seção serão discutidos os tipos de variáveis em Python. Uma maneira de categorizar esses tipos básicos de variáveis é apresentada abaixo nesses 4 grupos:

- **Numérico**: int (inteiro), float (ponto flutuante)
- **Texto**: str (string)
- **Sequencia**: list (listas, podem ser vetores ou matrizes)
- **Booleana**: bool, True (verdadeiro) ou False (falso)
- **Dicionário**: dict (dicionário, composto por chave e valor)

É importante frisar que em Python não é necessário o desenvolvedor especificar qual o tipo de variável será utilizada, o próprio Python irá vincular o tipo de variável. Diferente de outras Linguagens de mais baixo nível como C++ ou Fortran onde é necessário especificar o tipo de variável.

Outro ponto igualmente importante a se ressaltar é que Python é uma linguagem de programação “fracamente tipada”, o quê significa que a variável pode mudar seu tipo ao longo da execução do script. Ao contrário do que ocorre em linguagens “fortemente tipadas” como C++ e Java.

Então alguma variável que era inicialmente numérica, do tipo “int”, por exemplo, pode facilmente tornar-se uma variável do tipo string, basta vinculá-la ao tipo “str”. No exemplo apresentado no Capítulo1 nota-se que o inverso desse exemplo é apresentado, em que uma variável do tipo “str” (todo “input” é inicialmente do tipo “str”) é transformada no tipo numérico utilizando “int”. Conforme esquematizado abaixo:

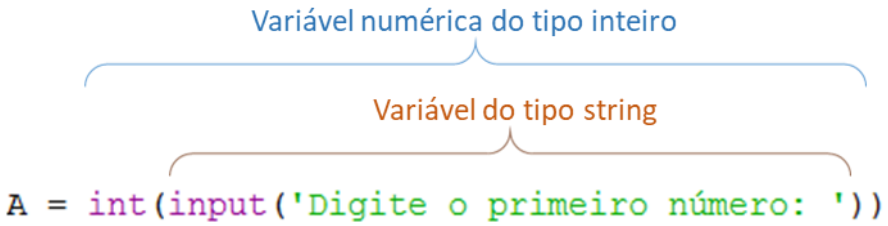


Figura 1 – Código Python: Somar dois números quaisquer

Variáveis numéricas

Esses tipos de dados são bem definidos e representam valores numéricos. Esses valores podem ser valores inteiros, valores racionais ou até números complexos. Nas seções subsequentes serão detalhados os dois primeiros tipos.

Variáveis do tipo Inteiro - int

O tipo de int lida com valores inteiros. Isso significa que esse tipo representa valores como 0, 1, -5 e -10, e não números como 0.4, 3.02, -12.8, etc.

Em Python, o código a seguir irá concluir que a é um inteiro e irá vincular o tipo int ao tipo do dado.

```
>>> x = 5
>>> type(x)
<class 'int'>
```

Nós poderíamos ser mais específicos, com a função int(), para garantir que o Python entendeu nosso 5 como inteiro:

```
>>> x = int(5)
>>> type(x)
<class 'int'>
```

O Python trata qualquer sequência de números (sem prefixos) como um número decimal. E o valor desse inteiro, é “sem limites”. Diferente de outras linguagens como Java ou C++, o valor de uma variável do tipo int não tem um valor máximo.

O sys.maxsize pode soar contra-intuitivo, se for pensado como o máximo valor que um inteiro pode ter, pois em Python não é.

```
>>> x = sys.maxsize
>>> x
2147483647
```

Nota-se que obtemos um valor inteiro binário de 32-bit, mas veremos o que ocorre se vincularmos um valor maior que este a x:

```
>>> x = sys.maxsize
>>> x+1
2147483648
```

Na verdade, nós conseguimos ir ainda mais longe:

```
>>> y = sys.maxsize + sys.maxsize
>>> y
4294967294
```

O único limite real para quão grande um inteiro pode ser em Python, é a memória da máquina em que o código está sendo executado.

Variáveis do tipo ponto flutuante - float

Esse tipo de variável pode representar números decimais com até 15 casas decimais. O que significa que ele consegue representar números como: 0.3, 2.9, 9.4273849723, etc. É importante ressaltar que esse tipo de variável pode representar também números inteiros.

Números que possuem mais de 15 números depois da vírgula serão truncados na décima quinta casa. Por exemplo, Python não terá dificuldades em entender as seguintes definição como float:

```
>>> y = 2.3
>>> type(y)
<class 'float'>
>>> y = 5/4
>>> type(y)
<class 'float'>
```

Entretanto, se quisermos considerar, por exemplo, 5 como float, será necessário declarar isso, uma vez que 5 também pode ser do tipo int, como mencionado anteriormente.

```
>>> y = 5.0
>>> type(y)
<class 'float'>
>>> y = float(5)
>>> type(y)
<class 'float'>
```

Esse tipo de variável pode ser utilizado também para representar alguns casos especiais de números como “NaN” (“Not a Number – não número”), +/- inf (+/- infinito), e expoentes:

```
>>> y = float('-infinity')
>>> y
-inf
>>> y = float(5e-3)
>>> y
0.005
>>> y = float('nan')
>>> y
nan
```

Variáveis texto

Variáveis do tipo string - str

Strings são sequências de dados de caracteres. O tipo de variável string em Python é chamado str.

As strings podem ser delimitadas utilizando ambas: aspas simples ou duplas. Todos os caracteres entre os delimitadores de abertura e fechamento são parte da string.

```
>>> print("Eu sou uma string.")
Eu sou uma string.
>>> type("Eu sou uma string.")
<class 'str'>
>>> print('Eu sou uma string também')
Eu sou uma string também
>>> type('Eu sou uma string também')
<class 'str'>
```

Uma string em Python pode conter muitos caracteres, conforme o desejo do programador. O único fator limitante é a fonte de memória da máquina na qual o programa está sendo executado. Uma string pode também ser vazia:

```
>>> ''
''
```

Como poderíamos incluir um caractere aspas ‘ como parte de uma string? Nosso primeiro impulse é tentar algo como o exemplo abaixo:

```
>>> print('Essa string contém caractere (') aspas simples.')
SyntaxError: invalid syntax
```

Como pode-se notar, essa forma não funcionou muito bem. A string nesse exemplo abre com a aspas simples e presume que a próxima aspas simples, aquela após o parênteses, a qual era para ser parte da string, é o delimitador de fechamento da string. E a aspas simples final acaba abrindo uma outra string causando o erro de sintaxe apresentado.

Se o objetivo é incluir tanto aspas simples quanto duplas dentro de uma string, a maneira mais simples de expressar essa string é utilizando o outro delimitador não contido na string, conforme apresentado abaixo.

```
>>> print("Essa string contém caractere (') aspas simples.")
Essa string contém caractere (') aspas simples.

>>> print('Essa string contém caractere (") aspas duplas.')
Essa string contém caractere (") aspas duplas.
```

Outra solução é suprimir o significado do caractere especial utilizando a barra invertida "\" na frente do caractere aspas na string. Com isso o Python vai suprimir o significado usual desse caráter e vai interpretar as aspas de forma literal:

```
>>> print('Essa string contém caractere (\') aspas simples.')
Essa string contém caractere (') aspas simples.
```

Da mesma maneira para as aspas duplas:

```
>>> print("Essa string contém caractere (\") aspas dupla.")
Essa string contém caractere (") aspas dupla.
```

Para inserir uma barra invertida literalmente dentro de uma string, devemos:

```
>>> print('teste\\teste')
teste\teste
```

Agora imagine que nós queremos criar uma string com um caractere tab, como poderíamos fazer? Alguns editores de texto podem permitir inserir um caractere tab em seu código. Mas muitos programadores consideram uma prática pobre, devido várias razões:

O computador consegue distinguir entre um caractere tab e uma sequência de caracteres espaços, mas o programador não. Para uma pessoa lendo o código, caracteres tab e espaço são visualmente indistinguíveis.

Alguns editores de texto são configurados para eliminar automaticamente o caractere tab e expandí-lo para um número adequado de caracteres espaço.

Alguns ambientes REPL de Python não inserem tabs em seus códigos.

Para Python e para quase todas as linguagens de programação comum, um caractere tab pode ser especificado pela sequência \t:

```
>>> print ('oi\toi')
oi      oi
```

A barra invertida antes do t, faz com que o t perca seu significado usual, que ´´e o literal t. A seguir é apresentada uma lista de sequências em que a barra invertida altera o significado literal dos caracteres.

Código	Interpretação
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\N{<name>}	Unicode com um dado <nome>
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\uxxxx	Unicode com 16-bit hex value xxxx
\Uxxxxxxxx	Unicode com 32-bit hex value xxxxxxxx
\v	ASCII Vertical Tab (VT)
\ooo	Caractere com valor octal ooo
\xhh	Caractere com valor hex hh

Veja alguns exemplos da Tabela acima aplicadas em Python:

```
>>> print('oi\toi')
oi      oi
>>> print('oi\noi')
oi
oi
>>> print('\u2192 \N{rightwards arrow}')
→ →
>>> print('a\t\141\t\x61')
a       a       a
```

De maneira oposta podemos especificar que não queremos que o caractere especial seja traduzido, para tanto utilizamos uma "raw string", em que, inserimos o r minúsculo ou maiúsculo antes de definir nossa string, conforme nos exemplos apresentados abaixo.

```
>>> print('oi\noi')
oi
oi
>>> print(r'oi\noi')
oi\noi
>>> print('oi\\oi')
oi\oi
>>> print(R'oi\\oi')
oi\\oi
```

Há ainda uma outra maneira de delimitar strings em Python: aspas triplas. Esse método permite que novas linhas possam ser incluídas, além disso permite a inclusão de ambas aspas simples e duplas.

```
>>> print(''' Essa string tem aspas simples (')
tem muitas linhas e
tem aspas duplas (")''')
 Essa string tem aspas simples (')
tem muitas linhas e
tem aspas duplas (")
```

E por último, mas não menos importante, vamos falar sobre as string formatadas, que são aplicadas ao adicionar "f" antes da string. O exemplo a seguir explicita a vantagem de utilizar esse formato, em que, com ela é apresentado apenas os números de casas decimais requisitadas pelo usuário.

```
>>> divisão = 2/3
>>> print('A resposta é ', divisão)
A resposta é  0.6666666666666666
>>> print(f'A resposta é {divisão:.2f}')
A resposta é 0.67
```

Variáveis sequência

Variáveis do tipo lista - list

Uma variável do tipo list corresponde a uma sequência de itens. É um dos tipos de variáveis mais utilizadas em Python e ela é bastante flexível. Todos os itens em uma lista não precisam necessariamente ser do mesmo tipo. Para declarar uma variável

deste tipo basta utilizar colchetes para delimitar o início e o fim da variável e separar seus itens utilizando vírgula.

```
var = [4, 1.2, 'python é legal']
```

É importante ressaltar que as variáveis list sempre se iniciam com índice 0 em python, então a variável declarada acima contém 3 itens com índices que vão de 0 a 2.

Até agora utilizamos somente o ambiente de comandos do Python. Para exemplificar essa seção utilizaremos o editor, em que vamos escrever um código previamente e depois executá-lo.

Código – Editor:

```
var = [2,11,13,21,26,32,36,40]

# var[2] = 13
print("var[2] =", var[2])

# var[0:3] =  [2, 11, 13]
print("var[0:3] =", var[0:3])

# var[5:] =  [32, 36, 40]
print("var[5:] =", var[5:])
```

Saída:

```
var[2] =  13
var[0:3] =  [2, 11, 13]
var[5:] =  [32, 36, 40]
```

As listas são mutáveis, ou seja, os valores dos elementos de uma lista podem ser facilmente alterados.

Código – Editor:

```
var = [1, 2, 3]
var[2] = 10
print(var)
```

Saída:

```
[1, 2, 10]
```

Variáveis Booleana

Variáveis do tipo booleana - bool

Python proporciona o uso de variáveis do tipo booleanas. Variáveis desse tipo podem ter um dos dois valores: verdadeiro ou falso.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Além das variáveis Booleanas, outros tipos de variáveis podem ser estimados também em um contexto Booleano para verificar a validade de uma condição. No Capítulo 6 falaremos mais sobre essa questão das relações condicionais.

Variáveis Dicionário

Variáveis do tipo dicionário - dict

Esse tipo de variável corresponde a uma coleção de pares: chave-valor. É geralmente utilizada quando utilizamos uma grande quantidade de dados. Esses tipo de variável em Pyton é otimizado para salvar esses dados. E para salvar algum valor é necessário saber sua chave.

Para definir esse tipo de variável são utilizadas as chaves {}, onde cada sendo uma par na forma chave : valor, os quais podem ser de qualquer tipo (numérico, texto...).

```
>>> var = {1:'valor','chave':2}
>>> type(var)
<class 'dict'>
```

Nós utilizamos a chave para salvar o valor, mas não o contrário. Faremos esse teste utilizando o editor e apresentamos o código e sua saída abaixo.

Código – Editor:

```
d = {1:'valor','chave':2}
print(type(d))

print("d[1] = ", d[1]);

print("d['chave'] = ", d['chave']);

# Gera um erro, não podemos usar um valor para chamar a chave:
print("d[2] = ", d[2]);
```

Saída:

```
<class 'dict'>
d[1] =  valor
d['chave'] =  2
Traceback (most recent call last):
  File "C:/Users/thiag/AppData/Local/Programs/Python/Python38-32/a.py", line 9, in <module>
    print("d[2] = ", d[2]);
KeyError: 2
```

Cap. 4 - Operadores

Os operadores são utilizados de maneira a incrementar, decrementar, comparar e avaliar dados dentro de um computador. Em Linguagem de Programação há três tipos desses operadores:

- Operadores Aritméticos
- Operadores Relacionais
- Operadores Lógicos

Nas seções subsequentes explicaremos sobre cada um desses tipos, e apresentaremos alguns exemplos de sua utilização.

Operadores Aritméticos

Os operadores aritméticos são os utilizados para obter resultados numéricos. Além da adição, subtração, multiplicação e divisão, podemos utilizar também o operador para exponenciação. Em Python, os símbolos utilizados para representar esses operadores aritméticos são:

- Adição: +
- Subtração: -
- Multiplicação: *
- Divisão: / (resto da divisão: %)
- Exponenciação: **

Código – Editor:

```
# Adição: +
adição = 1 + 1
print('1 + 1 = ', adição)

# Subtração: -
subtração = 1 - 1
print('1 - 1 = ', subtração)

# Multiplicação: *
multiplicação = 2 * 3
print('2 * 3 = ', multiplicação)

# Divisão: /
divisão = 10 / 5
print('10 / 5 = ', divisão)

# Resto da Divisão: %
módulo = 3 % 2
print('3 % 2 =', módulo)

# Exponenciação: **
exponencial = 2 ** 3
print('2 ** 3 =', exponencial)
```

Saída:

```
1 + 1 =  2
1 - 1 =  0
2 * 3 =  6
10 / 5 =  2.0
3 % 2 = 1
2 ** 3 = 8
```

Operadores Relacionais

Os operadores relacionais compararam String de caracteres e números. Os valores a serem comparados podem ser caracteres ou números. Estes operadores sempre retornam valores booleanos (True ou False), para estabelecer prioridades em relação ao que executar primeiro, são utilizados parêntese. Os operadores relacionais estão apresentados na tabela abaixo:

Descrição	Operador
Igual a	==

Maior que	>
Menor que	<
Maior ou igual a	>=
Menor ou igual a	<=
Diferente de	!=
Está dentro de	In

Como vimos anteriormente, o símbolo “=” é utilizado para atribuir um item a uma variável. Então para comprar dois itens é necessário utilizar outro símbolo, o qual corresponde a “==”.

Para obtermos a relação entre 2 membros, temos que utilizar a seguinte estrutura:

<membro a esquerda> **OPERADOR RELACIONAL** <membro a direita>

É importante observar que a inversão dos membros (para os operadores >, <, >=, e <=) ocasiona na inversão do resultado da expressão, isto é, se o membro que estiver à esquerda for para a direita e vice-e-versa, a relação entre eles será o contrário, ou seja, a >= b, assim como b <= a.

Abaixo são apresentados alguns exemplos, utilizando esses operadores relacionais.

```
>>> print(1 > 2)
False
>>> print('a' < 'b')
True
>>> print(5 < 10)
True
>>> print(200 != 200)
False
>>> print('oi' == 'oi')
True

>>> 2 in (1, 2, 3, 4, 5)
True
>>> 2 in (10, 11, 12)
False
```

Operadores Lógicos

Os operadores lógicos servem para combinar resultados de expressões, retornando se o resultado final é verdadeiro ou falso. Os operadores lógicos são:

- and (e) – Uma expressão and é verdadeira se todas as condições forem verdadeiras.

Tabela Verdade: Operador AND

Condição A	Condição B	AND (A.B)
True	True	True
True	False	False
False	True	False
False	False	False

```
>>> x = 1
>>> A = x > 0
>>> print(A)
True
>>> B = x < 2
>>> print(B)
True
>>> A and B
True
```

No exemplo acima vimos que as duas condições são verdadeiras, então a conjunção delas é verdadeira.

- or (ou) – Uma expressão or é verdadeira se pelo menos uma condição for verdadeira.

Tabela Verdade: Operador OR

Condição A	Condição B	OR (A + B)
True	True	True
True	False	True
False	True	True
False	False	False

```
>>> x = 0
>>> y = 1
>>> A = x == 1
>>> print(A)
False
>>> B = y == 1
>>> print(B)
True
>>> A or B
True
```

No exemplo acima vimos que as apenas uma das condições é verdadeira, o que é suficiente para a disjunção ser verdadeira.

- not (não) – Uma expressão not (não) inverte o valor da expressão ou condição, se verdadeira inverte para falsa e vice-versa.

Tabela Verdade: Operador NOT

Condição A	NOT (~A)
True	False
False	True

```
>>> x = 1
>>> A = x > 0
>>> print(A)
True
>>> not A
False
```

Cap. 5 – Relações Condicionais

As Relações condicionais são baseadas nos comandos de decisão os quais são utilizados para direcionar o fluxo de sua execução. Se determinada condição for satisfeita pelo comando if (se) então execute determinado comando, pode-se adicionar outras possibilidades a serem satisfeitas com o comando elif, e para todos os outros casos utilize else.

Comando de Seleção IF

O comando **if** serve para **selecionar** que parte de um código será executado.

O exemplo a seguir avalia se um aluno está aprovado na matéria ou não, sabendo que a média para aprovação é 6.0.

Código – Editor:

```
a = input("Digite a nota do aluno: ")
nota = float (a)

if nota >=6.0:
    print ("Parabéns! O aluno está aprovado.")

if nota < 6.0:
    print ("Que pena... O aluno foi reprovado.")
```

Testando o código acima com valores de 0 a 10, temos:

Saídas:

```
Digite a nota do aluno: 0
Que pena... O aluno foi reprovado.
>>>
Digite a nota do aluno: 1
Que pena... O aluno foi reprovado.
>>>
Digite a nota do aluno: 2
Que pena... O aluno foi reprovado.
.
.
.
Digite a nota do aluno: 6
Parabéns! O aluno está aprovado.
>>>
.
.
.
Digite a nota do aluno: 10
Parabéns! O aluno está aprovado.
```

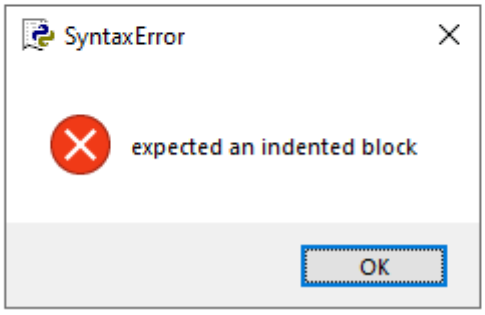
No código implementado no editor note que foi adicionado um TAB ao comando que

está na linha imediatamente após cada IF, o que é obrigatório em Python e denomina-se identação. Se o autor do script não incluir esse tab um erro ocorrerá. Verifique a ocorrência desse erro no script a seguir.

```
a = input("Digite a nota do aluno: ")
nota = float (a)

if nota >=6.0:
print ("Parabéns! O aluno está aprovado.")

if nota < 6.0:
    print ("Que pena... O aluno foi reprovado.")
```



A linha assinalada em vermelho não contém um TAB no seu início, o que ocasiona um erro na execução.

Comando de Seleção IF-ELSE

Para evitar uma sequência longa de IFs é possível agregar o comando **else** a um comando **if**. Desta forma, o programa que avalia ou não a aprovação do aluno pode ser escrito de uma maneira mais “limpa” e obtendo a mesma saída, conforme mostrado abaixo.

```
a = input("Digite a nota do aluno: ")
nota = float (a)

if nota >=6.0:
    print ("Parabéns! O aluno está aprovado.")
else:
    print ("Que pena... O aluno foi reprovado.")
```

Importante salientar que as linhas dos comandos if e else marcadas em laranja devem ser terminadas pelo sinal de **dois-pontos**.

Blocos de Comandos

As linhas de comando de if / else, terminadas pelo símbolo dois pontos ":", delimitam o início de uma estrutura denominada bloco.

Esta estrutura é usada quando é preciso "separar" um ou mais comandos, de outro grupo de comandos do código.

Nos exemplos acima e abaixo, o que se deseja é separar o **bloco** que será executado no IF, daquele **bloco** que será executado no ELSE e daqueles comandos serão executados fora do if/else.

Além do símbolo **dois-pontos** no final da linha, deve-se colocar todos os comandos que pertencem a um mesmo bloco **deslocados/alinhados** à direita, usando a tecla TAB.

No exemplo a seguir, o bloco do IF possui dois comandos, já o bloco do ELSE, três.

Código – Editor:

```
a = input("Digite sua idade : ")
idade = float (a)

if idade >= 18:
    print("Você já pode Trabalhar.")  # primeira linha do bloco IF
    print("Procure um emprego, se não tiver um!") # segunda linha do bloco IF
else:
    print("Você ainda é muito novo pra trabalhar") # primeira linha  do bloco do ELSE
    tempo_faltante = 18 - idade # segunda linha do bloco do ELSE
    print("Faltam ", tempo_faltante, "anos" ) # terceira linha  do bloco do ELSE
```

Testando o código acima com valores 10 e 19, temos:

Saídas:

```
Digite sua idade : 10
Você ainda é muito novo pra trabalhar
Faltam  8.0 anos
>>>
Digite sua idade : 19
Você já pode Trabalhar.
Procure um emprego, se não tiver um!
```

Aninhamento de IFs

Quando é preciso refinar uma condição, tomando uma decisão depois que outra já foi tomada, usa-se o aninhamento de IFs.

No exemplo a seguir, caso a pessoa tenha mais de 18 anos ela já poderá trabalhar, caso contrário, não.

Código – Editor:

```
a = input("Digite sua idade : ")
idade = float (a)

if idade >= 18:
    print("Você já pode Trabalhar.")  # primeira linha do bloco IF
    print("Procure um emprego, se não tiver um!") # segunda linha do bloco IF
else:
    print("Você ainda é muito novo pra trabalhar") # primeira linha  do bloco do ELSE
    tempo_faltante = 18 - idade # segunda linha do bloco do ELSE
    print("Faltam ", tempo_faltante, "anos" ) # terceira linha  do bloco do ELSE
```

Testando o código acima com valores 10, 8, 4 e 3 temos:

Saídas:

```
Digite a nota do aluno: 10
Parabéns! O aluno está aprovado.
Aluno com nota excelente!
Boas Férias!
Processo finalizado
>>>

Digite a nota do aluno: 8
Parabéns! O aluno está aprovado.
Boas Férias!
Processo finalizado
>>>

Digite a nota do aluno: 4
O aluno está de recuperação.
Deverá vir na próxima semana
Processo finalizado
>>>

Digite a nota do aluno: 3
Que pena... O aluno está reprovado!
Ele não poderá nem ir para recuperação.
Processo finalizado
```

Comando ELIF

Quando for necessário fazer um novo teste após um **ELSE**, como no exemplo anterior, é possível simplificar o comando usando um ELIF.

O trecho apresentado a seguir classifica uma pessoa de acordo sua idade as classificações definidas estão citadas abaixo:

- 0-1: Bebê
- 2-13: Criança
- 14-18: Adolescente
- 18-59: Adulto
- >=60: Idoso

Código – Editor:

```
a = input("Digite a idade: ")
idade = int(a)

if idade <=1:
    print("Bebê")
else:
    if idade < 14:
        print("Criança")
    else:
        if idade < 18:
            print("Adolescente")
        else:
            if idade < 60:
                print("Adulto")
            else:
                print("Idoso")

print("Processo Finalizado")
```

Testando o código acima com valores 1, 2, 14, 18 e 67 temos:

Saídas:

```
Digite a idade: 1
Bebê
Processo Finalizado
>>>

Digite a idade: 2
Criança
Processo Finalizado
>>>

Digite a idade: 14
Adolescente
Processo Finalizado
>>>

Digite a idade: 18
Adulto
Processo Finalizado
>>>

Digite a idade: 67
Idoso
Processo Finalizado
```

Para evitar o deslocamento excessivo de blocos à direita, como ocorre no exemplo acima, é possível usar o comando ELIF.

Desta forma, todos os ELIFs ficam alinhados com o primeiro IF. O ELSE do final, é executado caso nenhuma das condições anteriores sejam verdadeiras.

Código – Editor:

```
a = input("Digite a idade: ")
idade = int(a)

if idade <=1:
    print("Bebê")
elif idade < 14:
    print("Criança")
elif idade < 18:
    print("Adolescente")
elif idade < 60:
    print("Adulto")
else:
    print("Idoso")

print("Processo Finalizado")
```

Testando o código acima com valores 1, 2, 14, 18 e 67 temos exatamente as mesmas saídas que no programa anterior.

Saídas:

```
Digite a idade: 1
Bebê
Processo Finalizado
>>>

Digite a idade: 2
Criança
Processo Finalizado
>>>

Digite a idade: 14
Adolescente
Processo Finalizado
>>>

Digite a idade: 18
Adulto
Processo Finalizado
>>>

Digite a idade: 67
Idoso
Processo Finalizado
```

Cap. 6 Iteração

Iterar significa uma ação de repetir algo. Em linguagem de programação, iteração significa repetir de um conjunto de instruções por uma quantidade finita de vezes ou, enquanto uma condição seja aceita.

Alguns sinônimos para descrever iterações são:

- Iteração
- Estrutura de Repetição
- Looping
- Laço
- Repetição

Todos as nomenclaturas citadas acima são utilizadas descrever as estruturas das linguagens que tem o propósito de repetir um mesmo bloco de código, por uma quantidade finita de vezes, ou enquanto uma condição for verdadeira.

Em Python temos algumas ferramentas para implementar essa estrutura, dentre elas, as mais utilizadas são:

- Comando “For”
- Comando “While”
- Recursividade

A seguir apresentaremos um exemplo onde há a necessidade de implementarmos uma iteração, que se não utilizássemos seria um código “estático” que só serviria pra um tipo de input.

Vamos supor que temos um mercadinho e nele vendemos 4 itens com os preços listados abaixo:

- R$ 13,00
- R$ 14,00
- R$ 16,00
- R$ 11,00

Agora, vamos supor que necessitados aplicar a correção da inflação sobre esses preços e que o índice IPCA neste período foi de 10%.

Sem utilizarmos iterações, um código para realizar a operação descrita ficaria da seguinte forma:

Código – Editor:

```
preços = [13, 14, 16, 11]

IPCA = 0.1

preços[0] = preços[0]*(1 + IPCA)
preços[1] = preços[1]*(1 + IPCA)
preços[2] = preços[2]*(1 + IPCA)
preços[3] = preços[3]*(1 + IPCA)

print(f'R$ {preços[0]:.2f}')
print(f'R$ {preços[1]:.2f}')
print(f'R$ {preços[2]:.2f}')
print(f'R$ {preços[3]:.2f}')
```

Saída:

```
R$ 14.30
R$ 15.40
R$ 17.60
R$ 12.10
```

Utilizando “for”

Vamos agora imaginar que nosso mercadinho está indo “de vento em poupas” e depois de um ano estamos vendendo 100 produtos e é a hora de utilizarmos nosso programa novamente para atualizar o preço dos produtos de acordo com o IPCA.

Note que da maneira que formulamos nosso código acima, realizar as operações com os cem produtos ficaria algo lusitano, sem falar que seria um código “feio”. A solução para esse problema é a implementação de iterações. Em que, um bloco de comando irá repetir a mesma operação para cada um dos valores de uma lista, conforme apresentado no programa abaixo.

Código – Editor:

```
preços = [5, 11, 11, 1, 7, 16, 13, 11, 11, 5, 11, 17, 4, 13, 2, 16, 19, 20, 1, 3, 11,
          2, 12, 3, 5, 1, 14, 10, 4, 20, 2, 15, 5, 5, 5, 12, 6, 20, 3, 15, 20, 17, 5,
          9, 18, 19, 18, 2, 12, 1, 4, 12, 9, 7, 3, 15, 4, 20, 18, 9, 11, 3, 5, 14, 8,
          14, 12, 9, 4, 13, 2, 19, 8, 7, 1, 14, 15, 10, 11, 7, 4, 6, 19, 18, 11, 18,
          17, 5, 5, 12, 13, 4, 15, 16, 10, 19, 8, 11, 8, 8]

IPCA = 0.1

for i in range(100):
    preços[i] = preços[i]*(1 + IPCA)
    print(f'R$ {preços[i]:.2f}')
```

Saída:

```
R$ 5.50     R$ 3.30     R$ 19.80    R$ 14.30    R$ 16.50
R$ 12.10    R$ 5.50     R$ 2.20     R$ 2.20     R$ 17.60
R$ 12.10    R$ 1.10     R$ 13.20    R$ 20.90    R$ 11.00
R$ 1.10     R$ 15.40    R$ 1.10     R$ 8.80     R$ 20.90
R$ 7.70     R$ 11.00    R$ 4.40     R$ 7.70     R$ 8.80
R$ 17.60    R$ 4.40     R$ 13.20    R$ 1.10     R$ 12.10
R$ 14.30    R$ 22.00    R$ 9.90     R$ 15.40    R$ 8.80
R$ 12.10    R$ 2.20     R$ 7.70     R$ 16.50    R$ 8.80
R$ 12.10    R$ 16.50    R$ 3.30     R$ 11.00
R$ 5.50     R$ 5.50     R$ 16.50    R$ 12.10
R$ 12.10    R$ 5.50     R$ 4.40     R$ 7.70
R$ 18.70    R$ 5.50     R$ 22.00    R$ 4.40
R$ 4.40     R$ 13.20    R$ 19.80    R$ 6.60
R$ 14.30    R$ 6.60     R$ 9.90     R$ 20.90
R$ 2.20     R$ 22.00    R$ 12.10    R$ 19.80
R$ 17.60    R$ 3.30     R$ 3.30     R$ 12.10
R$ 20.90    R$ 16.50    R$ 5.50     R$ 19.80
R$ 22.00    R$ 22.00    R$ 15.40    R$ 18.70
R$ 1.10     R$ 18.70    R$ 8.80     R$ 5.50
R$ 3.30     R$ 5.50     R$ 15.40    R$ 5.50
R$ 12.10    R$ 9.90     R$ 13.20    R$ 13.20
R$ 2.20     R$ 19.80    R$ 9.90     R$ 14.30
R$ 13.20    R$ 20.90    R$ 4.40     R$ 4.40
```

No código acima verifique a utilização do comando “for”, em que, para cada valor de “i” em um range de 0 a 99, “range(100)”, serão executadas todas as linhas de comando que estão identadas abaixo do “for”. Note que análogo ao apresentado no Capítulo anterior para “if”, para separar os blocos de instruções é também utilizado o TAB (identação).

Há ainda mais uma melhoria que pode ser implementada no código acima de forma a permitir que o número de produtos na lista seja “dinâmico”, ou seja, sem a necessidade de saber quantos produtos temos em nossa lista, deixemos que o próprio programa descubra essa informação, e desta forma script funciona para qualquer tamanho de lista inserido.

Código – Editor:

```
preços = [5, 11, 11, 1, 7, 16, 13, 11, 11, 5, 11, 17, 4, 13, 2, 16, 19, 20, 1, 3, 11,
          2, 12, 3, 5, 1, 14, 10, 4, 20, 2, 15, 5, 5, 5, 12, 6, 20, 3, 15, 20, 17, 5,
          9, 18, 19, 18, 2, 12, 1, 4, 12, 9, 7, 3, 15, 4, 20, 18, 9, 11, 3, 5, 14, 8,
          14, 12, 9, 4, 13, 2, 19, 8, 7, 1, 14, 15, 10, 11, 7, 4, 6, 19, 18, 11, 18,
          17, 5, 5, 12, 13, 4, 15, 16, 10, 19, 8, 11, 8, 8]

IPCA = 0.1

for i in range(len(preço)):
    preços[i] = preços[i]*(1 + IPCA)
    print(f'R$ {preços[i]:.2f}')
```

A saída deste código é exatamente a mesma que a anterior, mas note que agora não foi necessário inserir que a nossa lista tem 100 itens, uma vez que o comando “len(preço)” estima esse valor para nós.

Utilizando “while”

No código a seguir, temos o mesmo código implementado acima utilizando a estrutura de repetição `while`, que numa tradução livre, significa, `enquanto`.

Código – Editor:

```
preços = [5, 11, 11, 1, 7, 16, 13, 11, 11, 5, 11, 17, 4, 13, 2, 16, 19, 20, 1, 3, 11,
          2, 12, 3, 5, 1, 14, 10, 4, 20, 2, 15, 5, 5, 5, 12, 6, 20, 3, 15, 20, 17, 5,
          9, 18, 19, 18, 2, 12, 1, 4, 12, 9, 7, 3, 15, 4, 20, 18, 9, 11, 3, 5, 14, 8,
          14, 12, 9, 4, 13, 2, 19, 8, 7, 1, 14, 15, 10, 11, 7, 4, 6, 19, 18, 11, 18,
          17, 5, 5, 12, 13, 4, 15, 16, 10, 19, 8, 11, 8, 8]

IPCA = 0.1

count = 0

while(True):
    preços[count] = preços[count]*(1 + IPCA)
    print(f'R$ {preços[count]:.2f}')
    count = count + 1
    if count >= len(preços):
        break
```

Essa é apenas mais uma das maneiras de escrever um código com a mesma saída, pois a saída deste código é exatamente a mesma que a anterior. É importante ressaltarmos alguns pontos deste código para um melhor entendimento:

- Na estrutura while inserimos um contador (variável “count”) para manter controle de qual item da lista estamos em nosso script;
- Utilizamos a condição “while(True)”, que deve ser acompanhada de alguma condição de parada (“break”) que no caso a condição foi quando o contador atingir o número de itens de nossa lista de preços (“if count >= len(preços)”);
- caso não colocássemos essa condição de parada “break”, o programa entraria em um loop infinito.

Outra maneira de escrevermos esse mesmo código utilizando o comando “while” é inserindo a condição de parada na mesma linha do comando, assim, não é necessário acrescentar o break, conforme apresentado a seguir:

Código – Editor:

```
preços = [5, 11, 11, 1, 7, 16, 13, 11, 11, 5, 11, 17, 4, 13, 2, 16, 19, 20, 1, 3, 11,
          2, 12, 3, 5, 1, 14, 10, 4, 20, 2, 15, 5, 5, 5, 12, 6, 20, 3, 15, 20, 17, 5,
          9, 18, 19, 18, 2, 12, 1, 4, 12, 9, 7, 3, 15, 4, 20, 18, 9, 11, 3, 5, 14, 8,
          14, 12, 9, 4, 13, 2, 19, 8, 7, 1, 14, 15, 10, 11, 7, 4, 6, 19, 18, 11, 18,
          17, 5, 5, 12, 13, 4, 15, 16, 10, 19, 8, 11, 8, 8]

IPCA = 0.1

count = 0

while count >= len(preços):
    preços[count] = preços[count]*(1 + IPCA)
    print(f'R$ {preços[count]:.2f}')
    count += 1
```

A saída, mais uma vez é exatamente a mesma apresentada anteriormente.

Outro ponto interessante a ressaltar é que no programa acima incrementamos o contador utilizando a forma concisa de escrever, então a linha de comando “count = count +1” significa exatamente a mesma coisa que “count += 1”.

Utilizando Recursividade

Recursividade significa a invocação de uma função por ela mesma, ou seja, é toda função que invoca a si mesma. A grande vantagem desta prática é o tempo gasto para executar um programa, que normalmente é menor que as outras estruturas de repetição.

Mas essa é uma prática perigosa de trabalho, principalmente para programadores menos experientes, pois pode facilmente travar nossos programas, ou então, estourar a pilha de execução, isto é, podemos obter o famoso erro `stack overflow error` que, nada mais é do que o limite máximo de vezes que uma função pode chamar a si mesma.

Primeiramente iremos fornecer uma pequena explicação sobre a implementação de funções em Python, elas são estruturadas pela seguinte linha de comando:

```
def nome_funçao(input)
```

Em que são definidos o nome da função e os inputs da função entre parênteses. De maneira análoga ao “for”, “while” e “if” é necessário incluir TAB para todos os comandos a serem executados pela função.

Abaixo é presentado um programa para resolver o mesmo problema tratado anteriormente utilizando-se a ferramenta de recursividade.

Código – Editor:

```
def main():
    preços = [5, 11, 11, 1, 7, 16, 13, 11, 11, 5, 11, 17, 4, 13, 2, 16, 19, 20,
          2, 12, 3, 5, 1, 14, 10, 4, 20, 2, 15, 5, 5, 5, 12, 6, 20, 3, 15, 20, 1
          9, 18, 19, 18, 2, 12, 1, 4, 12, 9, 7, 3, 15, 4, 20, 18, 9, 11, 3, 5, 1
          14, 12, 9, 4, 13, 2, 19, 8, 7, 1, 14, 15, 10, 11, 7, 4, 6, 19, 18, 11,
          17, 5, 5, 12, 13, 4, 15, 16, 10, 19, 8, 11, 8, 8]

    IPCA = 0.1
    count = 0
    preços_IPCA(preços, IPCA, count)

def preços_IPCA(preços, IPCA, count):

    preços[count] = preços[count]*(1 + IPCA)
    print(f'R$ {preços[count]:.2f}')
    if count + 1 < len(preços):
        preços[count + 1] = preços_IPCA(preços, IPCA, count + 1)

main()
```

A saída deste programa continua sendo a mesma saída apresentada anteriormente.

No programa acima vimos a implementação de 2 funções a primeira é a função principal “main()”, em que são incluídas a lista de preços, valor de IPCA bem como o contador, e por último, mas não menos importante, nela é chamada a função “preços_IPCA()”. Na

função principal não há nenhum input como entrada, já na outra função, há 3 inputs, são eles: a lista de preços, o valor do IPCA e o valor do contador.

Ao analisar dentro da função “preços_IPCA()” vimos que ela calcula o valor do preço atualizado para o IPCA e após verificar a condição de parada, chama ela mesmo novamente, o que ocasiona um laço, sem a necessidade de implementação nem dos comando “for” e nem “while”.

Após o entendimento e aplicação dos conceitos apresentados neste conteúdo, você já está pronto para se tornar um programador de sucesso. Mas lembre-se a experiência e a prática destes conceitos são fundamentais neste caminho.

Testes

Cap. 1 - Definição de Algoritmo

Assinale a alternativa que melhor define o significado de algoritmo.

a) Maneira de solucionar um problema.
b) Sequência de passos finitos, ou seja, um conjunto de instruções para se obter um determinado objetivo.
c) É composto por operações lógicas, relacionais e aritméticas e possui variáveis.
d) Ações que executamos em um computador.
e) É um código implementado em Python.

Resposta correta: b

Cap. 2 – Desenvolvendo Algoritmos

Assinale a alternativa que melhor define o conceito de Pseudocódigo

a) Composto por uma organização das situações, em um algoritmo.
b) Série de comandos coerentes que obedecem a certas convenções e regras.
c) Significa o encadeamento uma série de instruções para que se possa chegar à solução de um problema.
d) Característico da maneira rigorosa de raciocinar.
e) Forma genérica de escrever um algoritmo, utilizando uma linguagem simples (nativa a quem escreve, de forma a ser entendida por qualquer pessoa) sem a necessidade de conhecer a sintaxe de uma linguagem de programação específica.

Resposta correta e

Cap. 3 – Tipos de Variáveis

O seguinte código em Python imprime na tela qual o tipo de variável?

```
valor = input('Digite um número entre 1 a 5: ')
print(type(valor))
```

a) <class 'int'>
b) <class 'float'>
c) <class 'char'>
d) <class 'str'>
e) O código gera um erro de sintaxe.

Resposta correta: d

Cap. 4 – Operadores

Qual o operador lógico que presente no seguinte programa?

```
a = 10
b = 3

print("Operadores:")
resultado = (a < b) or (a > b)
print(a, " <", b, " ou ",a, " >", b, "==", resultado)
```

a) <
b) >
c) or
d) a
e) Não há operador lógico no programa

Resposta correta: c

Cap. 5 – Relações condicionais

Qual das seguintes linhas de comando utilizando if irá gerar um erro quando executada?

a)
```
if (1, 2):
    print('foo')
```

b)
```
if (1, 2):
                print('foo')
```

c)
```
if (1, 2):

    print('foo')
```

d)
```
if (1, 2):
print('foo')
```

e) Nenhuma das opções irá gerar um erro.

Resposta correta: d

Cap. 6 – Iterações

O programa a seguir pode conter um erro de sintaxe em Python, qual das opções abaixo identifica esse erro?

```
preços = [5, 11, 11, 1]
IPCA = 0.1

for i in range(len(preços)):
    preços[i] = preços[i]*(1 + IPCA)
    print(f'R$ {preços[i]:.2f}')
```

a) Falta “;” na definição das variáveis
b) Deve-se remover o TAB nas duas linhas de comando após o for
c) Está faltando parênteses na linha de comando do “for”
d) A lista, variável “preços” deve ter seus itens separados por “;” ao invés de “,”
e) Não há erros neste código

Resposta correta: e