



**Apostila**  
**Algoritmos e Programação**  
**- Linguagem Python -**

**Profa. Flávia Pereira de Carvalho**

**Agosto de 2018**

<b>1 INTRODUÇÃO.....</b>	<b>4</b>
<b>2 FORMAS DE REPRESENTAÇÃO DE ALGORITMOS.....</b>	<b>6</b>
2.1 DIAGRAMA NASSI-SHNEIDERMAN .....	6
2.2 FLUXOGRAMA .....	7
2.3 PORTUGUÊS ESTRUTURADO .....	7
<b>3 CONCEITOS IMPORTANTES.....</b>	<b>9</b>
3.1 CONSTANTES.....	9
3.2 VARIÁVEIS .....	9
3.3 ATRIBUIÇÃO.....	10
<b>4 ALGORITMOS SEQUENCIAIS .....</b>	<b>11</b>
<b>5 INSTRUÇÃO ESCREVER.....</b>	<b>13</b>
5.1 EXEMPLOS PRÁTICOS - ALGORITMOS .....	13
<b>6 OPERADORES ARITMÉTICOS.....</b>	<b>14</b>
<b>7 PROGRAMAS E PROGRAMAÇÃO.....</b>	<b>15</b>
<b>8 LINGUAGEM DO COMPUTADOR .....</b>	<b>16</b>
<b>9 LINGUAGENS DE ALTO NÍVEL .....</b>	<b>17</b>
9.1 LINGUAGEM PYTHON.....	18
<b>10 INSTALANDO PYTHON.....</b>	<b>19</b>
10.1 USANDO O INTERPRETADOR PYTHON.....	20
10.2 EXEMPLOS PRÁTICOS – PROGRAMAS EM PYTHON .....	21
<b>11 CONVERSÃO DO PROGRAMA-FONTE.....</b>	<b>22</b>
11.1 PROGRAMA COMPILADO .....	22
11.2 PROGRAMA INTERPRETADO .....	22
<b>12 SOFTWARES QUE VOCÊ PRECISA TER OU CONHECER.....</b>	<b>23</b>
12.1 SISTEMA OPERACIONAL .....	23
12.2 EDITOR DE TEXTOS .....	23
12.3 COMPILADOR/INTERPRETADOR .....	23
<b>13 CUIDADOS AO DIGITAR PROGRAMAS EM PYTHON .....</b>	<b>24</b>
<b>14 COMENTÁRIOS EM PYTHON .....</b>	<b>25</b>
14.1 COMENTÁRIO DE UMA LINHA .....	25
14.2 COMENTÁRIO DE VÁRIAS LINHAS .....	25
<b>15 TIPOS DE DADOS.....</b>	<b>27</b>
15.1 CONVERSÃO DE TIPOS NUMÉRICOS.....	29
15.2 MARCADORES E FORMATAÇÃO DE NÚMEROS .....	29
<b>16 INSTRUÇÃO LER .....</b>	<b>31</b>
<b>17 HORIZONTALIZAÇÃO DE FÓRMULAS (LINEARIZAÇÃO).....</b>	<b>31</b>
<b>18 FUNÇÕES NUMÉRICAS PREDEFINIDAS .....</b>	<b>32</b>
18.1 FUNÇÕES MATEMÁTICAS .....	32
18.2 FUNÇÕES DE NÚMEROS RANDÔMICOS .....	33
<b>19 EXPRESSÕES ARITMÉTICAS.....</b>	<b>34</b>
<b>20 ALGORITMOS COM SELEÇÃO.....</b>	<b>34</b>
20.1 ESTRUTURA DE SELEÇÃO ANINHADA.....	35
20.2 ESTRUTURA DE SELEÇÃO CONCATENADA.....	36
<b>21 OPERADORES RELACIONAIS.....</b>	<b>37</b>
<b>22 SELEÇÃO EM PYTHON.....</b>	<b>38</b>

22.1 FORMA SIMPLES: IF .....	38
22.2 FORMA COMPOSTA: IF - ELSE .....	39
<b>23 OPERADORES LÓGICOS .....</b>	<b>40</b>
23.1 EXPRESSÕES LÓGICAS E PRIORIDADES DAS OPERAÇÕES .....	41
<b>24 ALGORITMOS COM REPETIÇÃO .....</b>	<b>43</b>
24.1 ESTRUTURA DE REPETIÇÃO: ENQUANTO .....	44
<b>25 REPETIÇÕES EM PYTHON .....</b>	<b>45</b>
25.1 ESTRUTURA DE REPETIÇÃO: WHILE.....	45
<b>26 DIZER SIM PARA CONTINUAR OU NÃO PARA FINALIZAR O LAÇO .....</b>	<b>45</b>
<b>27 CONTADORES E ACUMULADORES .....</b>	<b>46</b>
27.1 CONTADORES .....	46
27.2 ACUMULADORES (OU SOMADORES) .....	47
<b>28 ESTRUTURA DE REPETIÇÃO: PARA (FOR EM PYTHON).....</b>	<b>48</b>
<b>29 FUNÇÃO RANGE .....</b>	<b>52</b>
29.1 COMO FUNCIONA A FUNÇÃO RANGE ( ) : .....	52
<b>30 DETERMINAÇÃO DO MAIOR E/OU MENOR VALOR EM UM CONJUNTO DE VALORES .....</b>	<b>54</b>
<b>31 RESPOSTAS DOS EXEMPLOS PRÁTICOS DESTA APOSTILA.....</b>	<b>55</b>
<b>REFERÊNCIAS.....</b>	<b>57</b>

## 1 Introdução

---

Nesta apostila é apresentado o estudo de **Lógica de Programação** e, para isto, é importante ter uma visão geral do processo de desenvolvimento de programas (softwares), visto que o objetivo final é ter um bom embasamento para a prática da programação de computadores [MAR03].

São apresentados conteúdos teóricos com exemplos e exercícios que também são executados na prática utilizando a **Linguagem de Programação Python**.

Para o desenvolvimento de qualquer programa, deve-se seguir basicamente as seguintes etapas, conhecidas como Ciclo de Vida do Sistema [BUF03]:

- 1) Estudo da viabilidade do software (Estudos Iniciais)
- 2) Análise detalhada do Sistema (Projeto Lógico)
- 3) Projeto preliminar do Sistema (Projeto Físico)
- 4) Projeto detalhado do Sistema (**Algoritmo**)
- 5) **Implementação** ou Codificação do Sistema (na Linguagem de Programação escolhida)
- 6) Testes do Sistema
- 7) Instalação e Manutenção do Sistema

No desenvolvimento de um sistema, quanto mais tarde um erro é detectado, mais dinheiro e tempo se gasta para repará-lo. Assim, a responsabilidade do programador é maior na criação dos algoritmos do que na sua própria implementação, pois quando bem projetados não se perde tempo tendo que refazê-los, reimplantá-los e retestá-los, assegurando assim um final feliz e no prazo previsto para o projeto [BUF03].

Pode-se encontrar, na literatura em informática, várias formas de representação das etapas que compõem o ciclo de vida de um sistema. Essas formas de representação podem variar tanto na quantidade de etapas quanto nas atividades a serem realizadas em cada fase [MAR03].

Como pode-se observar, nesse exemplo de ciclo de vida de um sistema (com sete fases) apresentado acima, os algoritmos fazem parte da quarta etapa do desenvolvimento de um programa. Na verdade, os algoritmos estão presentes no nosso dia-a-dia sem que saibamos, pois, uma receita culinária, as instruções de uso de um equipamento ou as indicações de um instrutor sobre como estacionar um carro, por exemplo, nada mais são do que algoritmos.

Um algoritmo pode ser definido como um conjunto de regras (instruções), bem definidas, para solução de um determinado problema. Segundo o dicionário Michaelis, o conceito de algoritmo é a "**utilização de regras para definir ou executar uma tarefa específica ou para resolver um problema específico**".

A partir desses conceitos de algoritmos, pode-se perceber que a palavra algoritmo não é um termo computacional, ou seja, não se refere apenas à área de informática. É uma definição ampla que agora que você já sabe o que significa, talvez a utilize no seu cotidiano naturalmente.

Na computação, o algoritmo é o "**projeto do programa**", ou seja, antes de se fazer um programa (software) na Linguagem de Programação desejada (Python, Pascal, C, Java etc.) deve-se fazer o algoritmo do programa. Então um programa, é um algoritmo escrito numa forma compreensível pelo computador (através de uma Linguagem de Programação), onde todas as ações a serem executadas devem ser especificadas nos mínimos detalhes e de acordo com as regras de sintaxe<sup>1</sup> da linguagem escolhida.

Um algoritmo não é a solução de um problema, pois, se assim fosse, cada problema teria apenas um algoritmo possível. **Um algoritmo é um 'caminho' para a solução de um problema e, em geral, existem muitos caminhos que levam a uma solução satisfatória**, ou seja, para resolver o mesmo problema pode-se obter vários algoritmos diferentes [ORT01].

Nesta apostila são apresentados os passos básicos e as técnicas para a construção de algoritmos através de três métodos para sua representação, que são alguns dos mais conhecidos. O objetivo ao final da apostila, é que você tenha adquirido capacidade de transformar qualquer problema em um algoritmo de boa qualidade, ou seja, a intenção é que você aprenda a Lógica de Programação dando uma base teórica e prática suficientemente boa, para que você domine os algoritmos e esteja habilitado a aprender uma Linguagem de Programação posteriormente [BUF03].

Para resolver um problema no computador é necessário que seja primeiramente encontrada uma maneira de descrever este problema de uma forma clara e precisa. É preciso encontrar uma sequência de passos que permitam que o problema possa ser resolvido de maneira automática e repetitiva. Esta sequência de passos é chamada de algoritmo [GOM04].

**A noção de algoritmo é central para toda a computação.** A criação de algoritmos para resolver os problemas é uma das maiores dificuldades dos iniciantes em programação em computadores [GOM04].

**Uma das formas mais eficazes de aprender algoritmos é através da execução de “muitos” exercícios.** Veja na Tabela 1 algumas dicas de como aprender e como não aprender algoritmos:

Algoritmos <b>não</b> se aprende	Algoritmos <b>se</b> aprende
Copiando algoritmos	Construindo algoritmos
Estudando algoritmos prontos	Testando algoritmos

**Tabela 1:** Dicas de como aprender e como *não* aprender algoritmos

**O aprendizado da Lógica é essencial para a formação de um bom programador, servindo como base para o aprendizado de todas as Linguagens de Programação**, estruturadas ou não. De um modo geral esses conhecimentos serão de supra importância, pois ajudarão no cotidiano, desenvolvendo um raciocínio rápido [COS04].

<sup>1</sup> *Sintaxe*: segundo o dicionário Aurélio, é a parte da gramática que estuda a disposição das palavras na frase e a das frases no discurso, bem como a relação lógica das frases entre si. Cada Linguagem de Programação tem a sua sintaxe (forma de escrever as instruções, os comandos) que deve ser seguida corretamente para que o programa funcione. O conjunto de palavras e regras que definem o formato das sentenças válidas chama-se de sintaxe da linguagem.

## 2 Formas de Representação de Algoritmos

Os algoritmos podem ser representados de várias formas, como por exemplo:

- a) Através de uma língua (português, inglês etc.): forma utilizada nos manuais de instruções, nas receitas culinárias, bulas de medicamentos etc.
- b) Através de uma linguagem de programação (Python, C, Delphi etc.): esta forma é utilizada por alguns programadores experientes, que "pulam" a etapa do projeto do programa (algoritmo) e passam direto para a programação em si.
- c) Através de representações gráficas: são bastante recomendáveis, já que um "desenho" (diagrama, fluxograma etc.) muitas vezes substitui, com vantagem, várias palavras.

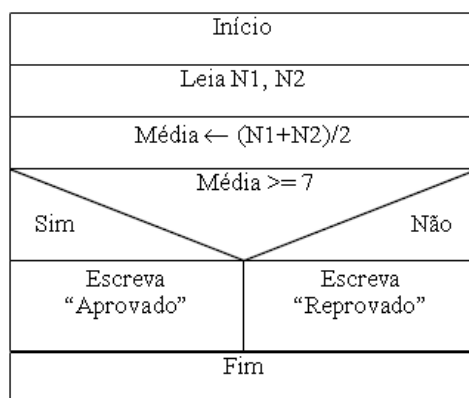
Cada uma dessas formas de representar um algoritmo tem vantagens e desvantagens, cabe à pessoa escolher a forma que melhor lhe convir. Nesta disciplina serão apresentadas três formas de representação de algoritmos (que são algumas das mais utilizadas), são elas:

- Diagrama de Nassi-Shneiderman (Diagrama de Chapin)
- Fluxograma (Diagrama de Fluxo)
- Português Estruturado (Pseudocódigo, Portugol ou Pseudolinguagem)

Não existe consenso entre os especialistas sobre qual é a melhor maneira de representar um algoritmo. Nesta apostila será incentivada a utilização do Diagrama Nassi-Shneiderman, mais conhecido como Diagrama de Chapin (lê-se “chapã”), por acreditar que é uma das formas mais didáticas de aprender e representar a lógica dos problemas. Mas, fica a critério de cada um escolher a forma que achar mais conveniente ou mais fácil de entender. Nos próximos capítulos são apresentadas breves explicações sobre cada uma dessas três formas de representar algoritmos e alguns exemplos.

### 2.1 Diagrama Nassi-Shneiderman

Os Diagramas Nassi-Shneiderman, também conhecidos como Diagramas de Chapin, surgiram nos anos 70 [YOU04] [SHN03] [CHA02] [NAS04] como uma maneira de ajudar nos esforços da abordagem de programação estruturada. Um típico diagrama Nassi-Shneiderman é apresentado na Figura 1 abaixo. Como pode-se observar, o diagrama é fácil de ler e de entender, pois cada "desenho" representa uma ação (instrução) diferente.



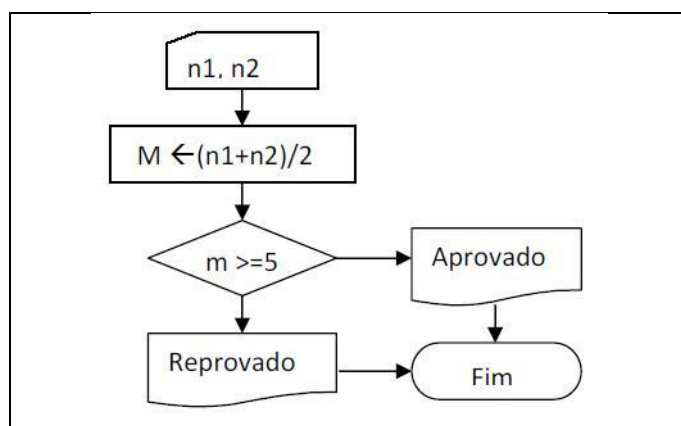
**Figura 1:** Exemplo de Diagrama Nassi-Shneiderman

A ideia básica deste diagrama é representar as ações de um algoritmo dentro de um único retângulo, subdividindo-o em retângulos menores, que representam os diferentes blocos de sequência de ações do algoritmo. Para saber mais sobre o histórico desses diagramas e conhecer os seus criadores acesse o site: <http://www.cs.umd.edu/hcil/members/bshneiderman/nsd/>. Para ter acesso ao primeiro artigo elaborado pelos autores do Diagrama de Chapin, escrito em 1973, acesse o seguinte endereço onde você pode fazer o download do artigo: <http://fit.faccat.br/~fpereira/p12-nassi.pdf>.

## 2.2 Fluxograma

Os Fluxogramas (ou Diagramas de Fluxo) são uma representação gráfica que utilizam formas geométricas padronizadas ligadas por setas de fluxo, para indicar as diversas ações (instruções) e decisões que devem ser seguidas para resolver o problema em questão.

Eles permitem visualizar os caminhos (fluxos) e as etapas de processamento de dados possíveis e, dentro destas, os passos para a resolução do problema. A seguir, na Figura 2, é apresentado um exemplo de fluxograma [GOM04] [MAR03].



**Figura 2:** Exemplo de Fluxograma

## 2.3 Português Estruturado

O Português Estruturado é uma forma especial de linguagem bem mais restrita que a Língua Portuguesa e com significados bem definidos para todos os termos utilizados nas instruções (comandos).

Essa linguagem também é conhecida como Portugol (junção de Português com Algol [ALG96] [PRO04]), Pseudocódigo ou Pseudolinguagem. **O Português Estruturado na verdade é uma simplificação extrema da língua portuguesa**, limitada a pouquíssimas palavras e estruturas que têm significado pré-definido, pois deve-se seguir um padrão de escrita. **Emprega uma linguagem intermediária entre a linguagem natural e uma linguagem de programação, para descrever os algoritmos.**

A sintaxe do Português Estruturado não precisa ser seguida tão rigorosamente quanto a sintaxe de uma linguagem de programação, já que o algoritmo não será executado como um programa [TON04].

Embora o Português Estruturado seja uma linguagem bastante simplificada, ela possui todos os elementos básicos e uma estrutura semelhante à de uma linguagem de programação de computadores. Portanto, resolver problemas com português estruturado pode ser uma tarefa tão complexa quanto a de escrever um programa em uma linguagem de programação qualquer só não tão rígida quanto a sua sintaxe, ou seja, o algoritmo não deixa de funcionar porque esquecemos de colocar um ';' (ponto-e-vírgula) por exemplo, já um programa não funcionaria. A Figura 3 apresenta um exemplo de algoritmo na forma de representação de Português Estruturado.

```
Início
  Ler N1
  Ler N2
  M = (N1+N2)/2
  Escrever M
  Se M >= 6 Então
    Escrever "Aprovado"
  Senão
    Escrever "Reprovado"
  FimSe
Fim
```

**Figura 3:** Exemplo de Português Estruturado

O uso de uma linguagem algorítmica antes de aprender uma linguagem de programação, além de ser menos traumático, tem-se como premissa ensinar um conceito de cada vez e **o objetivo desta apostila é ensinar uma forma de expressar algoritmos e não uma sintaxe de uma linguagem de programação** [ORT01].



### 3 Conceitos Importantes

---

Neste capítulo são apresentados e explicados três conceitos fundamentais para a construção de algoritmos, são eles: Constante, Variável e Atribuição.

#### 3.1 Constantes

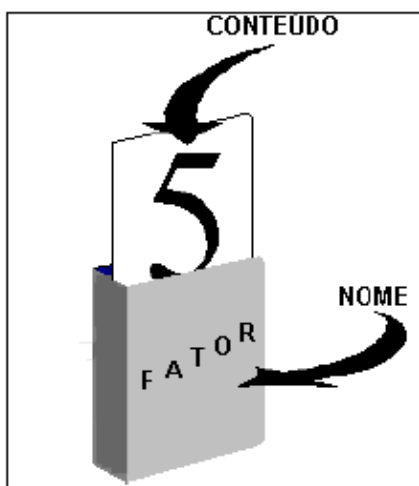
São chamadas de constantes, as informações (os dados) que **não variam com o tempo**, ou seja, permanecem sempre com o mesmo conteúdo, é um **valor fixo** (invariável). Como exemplos de constantes pode-se citar: números, letras, palavras, frases.

#### 3.2 Variáveis

O bom entendimento do conceito de variável é fundamental para elaboração de algoritmos e, consequentemente de programas. Uma variável é um **espaço da memória do computador** que se "reserva" **para guardar informações** (dados). Como o próprio nome sugere, as variáveis, podem conter valores diferentes a cada instante de tempo, ou seja, seu conteúdo pode variar de acordo com as instruções do algoritmo.

As variáveis são referenciadas através de um nome (identificador) criado pelo próprio programador durante o desenvolvimento do algoritmo. Exemplos de nomes de variáveis: produto, idade, a, x, nota1, peso, n1. O conteúdo de uma variável pode ser alterado, consultado ou apagado quantas vezes forem necessárias durante o algoritmo. Mas, ao alterar o conteúdo da variável, a informação anterior é perdida (sobrescreve o conteúdo), ou seja, sempre "vale" a última informação armazenada na variável. **Uma variável armazena 'apenas' um conteúdo de cada vez!**

Uma variável pode ser vista como uma caixa com um rótulo (nome) colado nela, que em um dado momento guarda um determinado objeto. O conteúdo desta caixa não é algo fixo, permanente. Na verdade, essa caixa pode ter seu conteúdo alterado diversas vezes. No exemplo abaixo, a caixa (variável) rotulada como FATOR, contém o valor 5 (a constante 5). Em outro momento, essa caixa poderá conter qualquer outro valor numérico. Entretanto, a cada instante, ela conterá um, e somente um, valor [TON04].



**Figura 4:** Ilustração de Variável [TON04]

### 3.3 Atribuição

A atribuição é uma notação utilizada para **atribuir um valor a uma variável**, ou seja, para **armazenar** um determinado conteúdo em uma variável. A operação de atribuição, normalmente, é representada por uma seta apontando para a esquerda (  $\leftarrow$  ), mas existem outros símbolos para representar a atribuição, como o sinal de igualdade ( = ), por exemplo. A notação a ser usada para representar a atribuição, depende da forma de representação do algoritmo escolhida. Já, nas linguagens de programação, cada uma tem a sua notação descrita pelas regras de sintaxe da linguagem. Na Linguagem Python, é o sinal de igualdade que representa a atribuição. Na Tabela 2 a seguir, são apresentados alguns exemplos de atribuições possíveis:

Atribuições Possíveis	Exemplos
variável $\leftarrow$ constante	idade $\leftarrow$ 12 (lê-se: idade <i>recebe</i> 12)
variável $\leftarrow$ variável	peso $\leftarrow$ valor
variável $\leftarrow$ expressão	A $\leftarrow$ B + C

**Tabela 2:** Exemplos de Atribuições

Uma observação importante a ser feita em relação a atribuições é que na “parte esquerda” (a que vai “receber” algo) não pode haver nada além da variável, ou seja, é só **a variável que “recebe” algum conteúdo**. Não é possível ter um cálculo, por exemplo, ou uma constante, recebendo alguma coisa. Veja, por exemplo, essa notação abaixo:

**nota1 + nota2  $\leftarrow$  valor** (notação **ERRADA!!!**)

**Atenção:** Esta operação apresentada acima não é possível, não está correta esta atribuição!

## 4 Algoritmos Sequenciais

---

Nada é mais básico em computação, do que o conceito de Algoritmo (como já apresentado no capítulo 1 Introdução): “*Um conjunto finito de regras, bem definidas, para a solução de um problema em um tempo finito.*”

Considere o seguinte problema: Dadas 2 notas de um aluno em uma determinada disciplina, calcule a média aritmética desse aluno.

As tarefas a serem executadas para a solução desse problema, podem ser descritas da seguinte maneira:

- 1) Iniciar o algoritmo
- 2) Obter as duas notas do aluno
- 3) Calcular a média aritmética usando a fórmula  $\frac{\text{nota1}+\text{nota2}}{2}$
- 4) Comunicar o resultado obtido
- 5) Terminar o algoritmo

Essa sequência de tarefas, especificadas passo-a-passo, é um algoritmo, pois é formada por um conjunto finito de regras que podem ser consideradas bem definidas e cuja execução produz a solução do problema proposto, após um tempo finito.

Na prática não é importante ter-se apenas um algoritmo, mas ter-se um bom algoritmo. O mais importante com respeito a um algoritmo, é a sua correção, ou seja, se ele resolve realmente o problema proposto e o faz exatamente.

Alguns outros critérios que devem ser considerados para ter um bom algoritmo são: o tempo necessário para sua execução, a adaptabilidade de algoritmo para a solução de outros problemas similares, sua simplicidade, sua elegância, sua legibilidade etc.

Para se ter um algoritmo é necessário:

- 1) Que se tenha um número finito de passos.
- 2) Que cada passo esteja precisamente definido, sem ambiguidade.
- 3) Que exista uma ou mais entradas.
- 4) Que exista uma ou mais saídas.
- 5) Que o conjunto de passos leve a execução de uma tarefa útil.
- 6) Que exista uma condição de fim, num tempo finito.

Um **método para a construção de algoritmos** pode, simplificadaamente, ser descrito pelos seguintes passos:

- 1) Ler atentamente o enunciado do problema proposto.
- 2) Descobrir no enunciado os dados de entrada (que serão fornecidos).
- 3) Descobrir no enunciado os dados de saída (resultados que devem ser produzidos).
- 4) Determinar o que deve ser feito para transformar as entradas nas saídas desejadas.
- 5) Construir o algoritmo (tarefas passo-a-passo).
- 6) Executar verificando se produz o resultado desejado.

Voltando ao exemplo anterior: Dadas 2 notas de um aluno em uma determinada disciplina, calcule a média aritmética desse aluno. **O algoritmo para resolver esse problema, escrito em Português Estruturado seria:**

```
Início
  Ler nota1
  Ler nota2
  Média = (nota1+nota2)/2
  Escrever Média
Fim
```

Nos próximos capítulos, todas essas instruções algorítmicas (comandos) serão vistas detalhadamente, com exemplos [ORT01].

## 5 Instrução Escrever

---

Existem basicamente duas instruções principais em algoritmos (e em programação em geral) que são: Escrever e Ler. Ou seja, cada Linguagem de Programação tem um comando para representar a Entrada de Dados (leitura) e outro para a Saída de Dados (escrita). Neste capítulo será apresentada a instrução *Escrever*.

A instrução *Escrever* é utilizada quando deseja-se **mostrar informações na tela do computador**, ou seja, é um comando de **Saída de Dados**. Resumindo: usa-se a instrução *Escrever*, quando necessita-se que o algoritmo (programa) mostre alguma informação para o usuário, pois o comando Escrever é para **“escrever na tela do computador”**.

Tanto no Diagrama de Chapin quanto em Português Estruturado representa-se a saída de dados através da própria palavra ***Escrever*** (ou Escreva). Já em Fluxogramas, a representação da saída de dados é feita através de uma forma geométrica específica [GOM04] [MAR03]. Na Linguagem Python o comando para escrever na tela do computador é o **print**.

### 5.1 Exemplos Práticos - Algoritmos

Exemplos de algoritmos para serem feitos em aula, no caderno e corrigidos no quadro.

1) Escreva um **algoritmo** para armazenar o valor 20 em uma variável X e o valor 5 em uma variável Y. A seguir, armazenar a soma do valor de X com o de Y em uma variável Z. Escrever (na tela) o valor armazenado em X, em Y e em Z (*Capítulo 31: Respostas dos Exemplos – Pág.55*).

2) Escreva um **algoritmo** para armazenar o valor 4 em uma variável A e o valor 3 em uma variável B. A seguir, armazenar a soma de A com B em uma variável C e a subtração de A com B em uma variável D. Escrever o valor de A, B, C e D e também escrever a mensagem “Fim do Algoritmo”.

### Observação:

Note que quando deseja-se **escrever alguma mensagem** literalmente **na tela** (letra, frase, número etc.), deve-se utilizar **aspas** para identificar o que será escrito, pois *o que estiver entre aspas no programa, será exatamente o que aparecerá na tela do computador*. Diferente de quando necessita-se escrever o conteúdo de uma variável, pois neste caso não se utiliza aspas. Veja os exemplos abaixo:

Escrever A

Esta instrução faz com que o algoritmo escreva o conteúdo da variável A na tela

Escrever “A”

Esta instrução faz com que o algoritmo escreva a letra A na tela

## 6 Operadores Aritméticos

Muitas vezes, ao desenvolver algoritmos, é comum utilizar expressões matemáticas para a resolução de cálculos. Neste capítulo são apresentados os operadores aritméticos necessários para determinadas expressões. Veja a Tabela 3 a seguir.

Operação	Símbolo	Prioridade de Execução
Multiplicação (Produto)	*	1ª.
Divisão	/	1ª.
Adição (Soma)	+	2ª.
Subtração (Diferença)	-	2ª.

**Tabela 3:** Operadores Aritméticos e Prioridades de Execução

Nas linguagens de programação e, portanto, nos exercícios de algoritmos, as expressões matemáticas sempre obedecem às regras matemáticas comuns, ou seja:

- As expressões dentro de parênteses são sempre resolvidas antes das expressões fora dos parênteses. Quando existem vários níveis de parênteses, ou seja, um parêntese dentro de outro, a solução sempre inicia do parêntese mais interno até o mais externo (“de dentro para fora”).
- Quando duas ou mais expressões tiverem a mesma prioridade, a solução é sempre iniciada da expressão mais à esquerda até a mais à direita (“da esquerda para a direita”).

Desta forma, veja os seguintes exemplos e os respectivos resultados:

**ExemploA:**  $2 + (6 * (3 + 2)) = 32$

**ExemploB:**  $2 + 6 * (3 + 2) = 32$

## 7 Programas e Programação

---

O *hardware* do computador, constituído de placas e dispositivos mecânicos e eletrônicos, precisa do *software* para lhe dar vida: programas, com finalidades bem determinadas, que façam o que os usuários querem ou precisam. Há programas para editar textos, para fazer cálculos, jogos e milhares de outras finalidades. Alguns programas maiores, como processadores de textos, planilhas eletrônicas e navegadores da Internet, são de fato agrupamentos de dezenas de programas relacionados entre si (são sistemas).

Programas são constituídos de instruções e comandos que o processador do computador entende, do tipo: faça isso, faça aquilo. Esses comandos devem estar representados em uma linguagem. Talvez você não esteja ciente de que está familiarizado com vários tipos de linguagem. Além do Português, há linguagens para inúmeras finalidades: sinais, faixas e placas de trânsito, gestos com a mão e com a cabeça, o Braille, a linguagem dos surdos-mudos etc. Até para falar com bebês temos formas específicas! Também há formas de linguagem mais simples para nos comunicarmos com máquinas, como a televisão, o videocassete, a calculadora. Ninguém "chama" verbalmente um elevador, nem "diz" à TV qual canal sintonizar; se você não fizer algo que os aparelhos entendam, não vai conseguir o que quer. Assim é o computador: você deve comandá-lo de uma forma que ele entenda e faça o que você está solicitando.

Para que algo aconteça no computador, não basta um programa; os comandos do programa devem ser executados. Programas de computador são como filmes: uma coisa é a película em um rolo, contendo uma sequência de imagens. Outra coisa é colocar o filme em um projetor e assisti-lo na telona. Os programas ficam guardados em arquivos no Disco Rígido ou CD, DVD, enfim, até que seja comandada (acionada) sua execução. São então carregados pelo Sistema Operacional para a memória e só então acontece (exatamente) o que foi programado, e você pode perceber o que o programa faz.

Uma diferença entre máquinas em geral e o computador é que este pode fazer muito mais coisas, portanto precisa de uma variedade maior de comandos. E outra diferença fundamental: o computador pode armazenar os comandos, agrupados em programas, para execução posterior.

Programar um computador é, portanto, produzir comandos agrupados em programas, em uma linguagem que o computador entenda e que, quando executados, façam o computador produzir algum resultado desejado. Um bom programador é treinado em algumas habilidades, sendo o objetivo desta apostila desenvolver na prática essas habilidades, para isso usando uma linguagem de programação de computadores chamada **Python**.

## 8 Linguagem do Computador

A atividade básica de um computador consiste em executar instruções, através de um microprocessador ou simplesmente processador, às vezes também chamado de CPU (*Central Processing Unit* – Unidade Central de Processamento). O processador, em última análise, recebe instruções na forma de impulsos elétricos: em um determinado circuito, pode estar ou não fluindo corrente. Representa-se cada impulso por 1 ou 0, conforme passe corrente ou não. Esta é a menor unidade de informação que pode ser representada em um computador, e é chamada **bit**.

A CPU recebe instruções e dados na forma de bits agrupados de 8 em 8; **cada conjunto de 8 bits é chamado byte**. De uma forma simplificada, um byte, ou seja, um conjunto de 8 impulsos elétricos (ou sua ausência), constitui uma instrução ou um dado (na verdade, uma instrução pode ocupar mais de um byte). Essa unidade de informação é representada pelo número correspondente no sistema decimal. Dessa forma, ao byte 00000001 associa-se o número 1, ao byte 00000011 associa-se o número 3, ao 00000100 o número 4 e assim por diante. Um byte pode armazenar, portanto, um número de 0 a 255 (11111111).

A memória RAM (*Random Access Memory* - Memória de Acesso Aleatório) de um computador é constituída de uma sequência de milhares ou milhões de bytes, cada um identificado por um número que constitui o seu endereço (veja a Tabela 4). O processador tem a capacidade de buscar o conteúdo da memória e executar instruções ali armazenadas. A CPU também contém algumas unidades de memória, chamadas registradores, identificados por nomes como AX, CS e IP, que também armazenam números e servem a várias funções.

1	2	3	4	5	6	...								
56	23	0	247	154	87	...								

**Tabela 4:** Esquema Simplificado da Memória RAM - endereços e respectivos valores.

Os números armazenados na memória podem representar dados ou instruções. Quando representando instruções, têm significados específicos para a CPU. Por exemplo, a CPU em geral recebe comandos do tipo (mas não na forma):

*“Armazene 9 no registrador DS”*

*“Armazene 204 no endereço de memória 1.234.244”*

*“Some 5 ao conteúdo do registrador AL”*

*“Se a última instrução deu resultado 0, passe a executar as instruções a partir do endereço de memória 457.552”*



## 9 Linguagens de Alto Nível

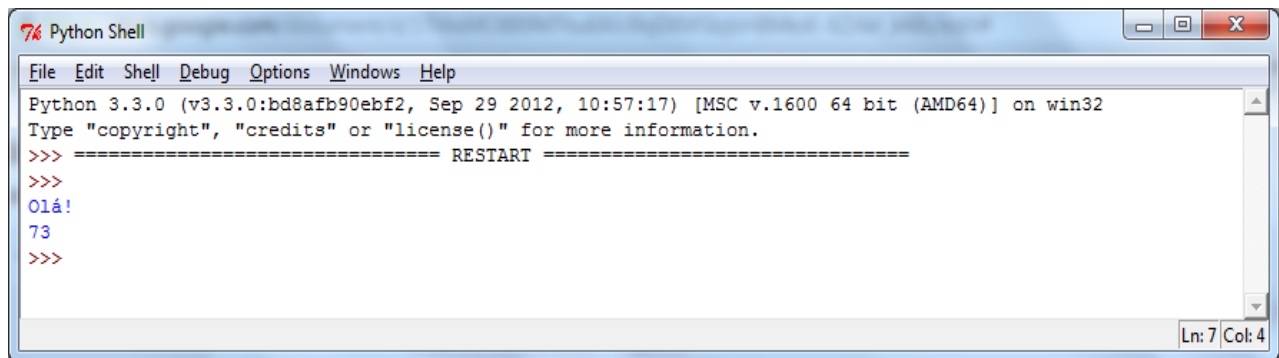
Escrever programas de computador em linguagem de máquina, apesar de os programas produzidos serem rápidos, pode ser muito difícil e trabalhoso, além de exigir conhecimentos profundos sobre o computador.

Para não ter que programar nessa linguagem difícil, foram desenvolvidas as **linguagens de alto nível**. Estas permitem descrever o que se deseja que o computador faça, utilizando instruções mais próximas da linguagem humana. Além de facilitarem as descrições dos processos a serem executados, as linguagens de alto nível simplificam a utilização da memória do computador, diminuindo a quantidade de detalhes com os quais deve ocupar-se o programador. Assim, ao invés de lidar com bits, bytes, endereços de memória e uma infinidade de outros detalhes, o programador pode pensar em "limpar a tela", "escrever uma linha de texto", somar e subtrair variáveis como na matemática e tomar uma decisão na forma "se...então".

Veja um exemplo específico: as instruções abaixo, escritas na linguagem Python, determinam que seja escrito na tela a palavra **Olá!** e, logo após, uma soma seja efetuada e o resultado desta também mostrado na tela:

```
print("Olá!")  
print(23+44+6)
```

O resultado na tela seria o seguinte (usando IDLE<sup>2</sup> que é a IDE básica do Python):



**Figura 5:** Tela da IDE básica do Interpretador Python

Uma linguagem de programação de alto nível possui várias características em comum com a linguagem humana. Elas possuem um *alfabeto* (letras, números e outros símbolos) e *palavras*. *Frases* podem ser construídas com as palavras, respeitando-se certas regras. O conjunto de regras de construção de palavras e frases numa linguagem de alto nível, assim como nas linguagens comuns, chama-se **sintaxe**. Cada Linguagem de Programação tem a sua sintaxe que deve ser seguida para que os programas sejam executados corretamente.

<sup>2</sup> IDLE: é um ambiente de desenvolvimento integrado (IDE - *Integrated Development Environment*) para a Linguagem de Programação Python.

## 9.1 Linguagem Python

Por que foi escolhida a Linguagem Python para ensinar programação de computadores nesta apostila? Toda programação de computadores é feita através de uma ou mais linguagens de programação, portanto para aprender a programar é necessário aprender ao menos uma linguagem de programação. **O objetivo desta apostila não é ensinar uma linguagem específica, mas sim ensinar a programar de uma forma geral. Ou seja, a linguagem é apenas uma ferramenta para esse aprendizado. O mais importante é aprender a lógica de programação – Algoritmos!**

Existem várias linguagens de programação, como C, C++ e Java, por exemplo. Essas linguagens são mais complexas, então, por isso, optou-se pelo uso de Python, devido à sua simplicidade e clareza.

A linguagem Python foi escolhida por ser uma linguagem muito versátil, usada não só no desenvolvimento Web, mas em muitos outros tipos de aplicações. Embora simples, é também uma linguagem poderosa, podendo ser usada para administrar sistemas e desenvolver grandes projetos. É uma linguagem clara e objetiva, pois vai direto ao ponto, sem rodeios. Na página oficial do Python no Brasil (<https://wiki.python.org.br/EmpresasPython>), é possível verificar algumas empresas que utilizam a linguagem. Então o leitor desta apostila estará aprendendo a programar em uma linguagem que poderá utilizar na prática e não ficará apenas na teoria.

O nome Python é uma homenagem ao grupo humorístico inglês Monty Python, adorado por *geeks* (nerds) de todo o mundo.

Apesar de sua sintaxe simples e clara, Python oferece muitos recursos disponíveis também em linguagens mais complexas como Java e C++, como por exemplo: programação orientada a objetos, recursos avançados de manipulação de textos, listas e outras estruturas de dados, possibilidade de executar o mesmo programa sem modificações em várias plataformas de hardware e sistemas operacionais.

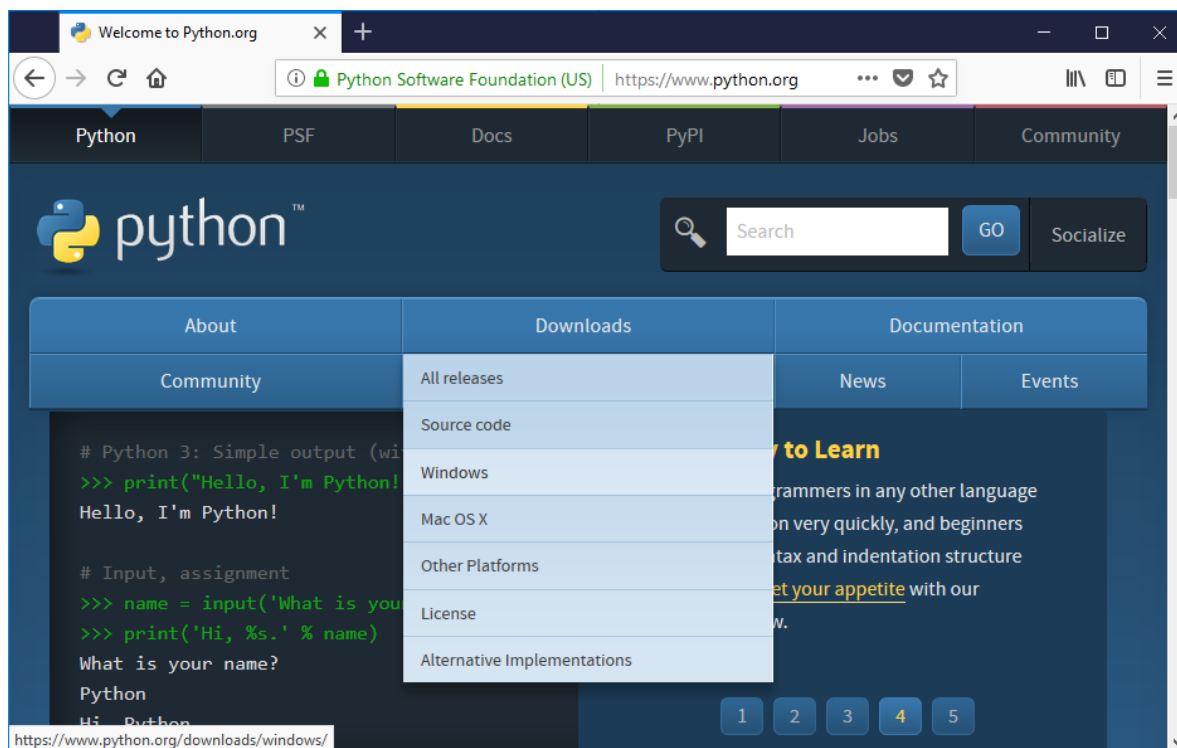
Em resumo, Python oferece uma sintaxe tão simples quanto a linguagem PHP, por exemplo, mas é mais versátil. Permite explorar vários recursos de Java e C++ de uma forma mais acessível. Por esses motivos acredita-se que seja a melhor escolha para quem quer começar a programar hoje (<https://wiki.python.org.br/AprendaProgramar>).

Python é software livre, ou seja, pode ser utilizada gratuitamente, graças ao trabalho da Python Foundation (<http://www.python.org/>) e de inúmeros colaboradores. Python pode ser utilizada em praticamente qualquer arquitetura de computadores ou sistema operacional, como Linux, FreeBSD, Microsoft Windows ou Mac OS X.

## 10 Instalando Python

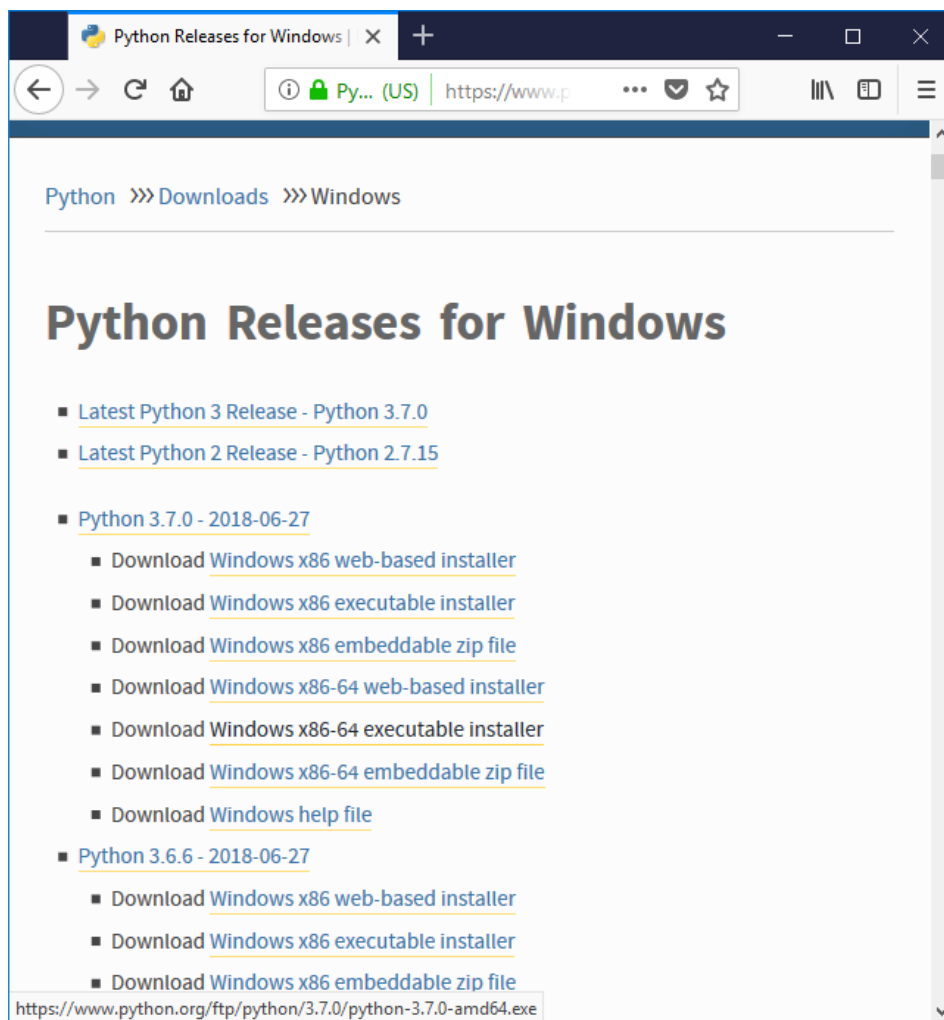
É necessário instalar o **Interpretador da Linguagem Python**. O interpretador Python não vem instalado com o Microsoft Windows, ou seja, deve-se fazer download da Internet. No Mac OS X ou no Linux, provavelmente já esteja instalado juntamente com o Sistema Operacional.

Como Python é Software Livre, pode ser baixado tranquilamente e gratuitamente no site <http://www.python.org>. Na Figura 6 pode ser observada a opção Downloads no menu superior, na página web oficial da linguagem.

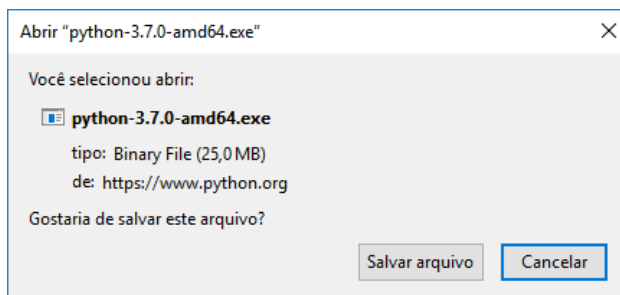


**Figura 6:** Site da Python Foundation – Site Oficial do Python

Você deve fazer o download, de acordo com o Sistema Operacional do seu computador. Não esqueça também, de verificar se o sistema é 32 ou 64 bits, antes de fazer a instalação (no Windows, essa informação está em Painel de Controle – Sistema).



**Figura 7:** Página de Download do Python



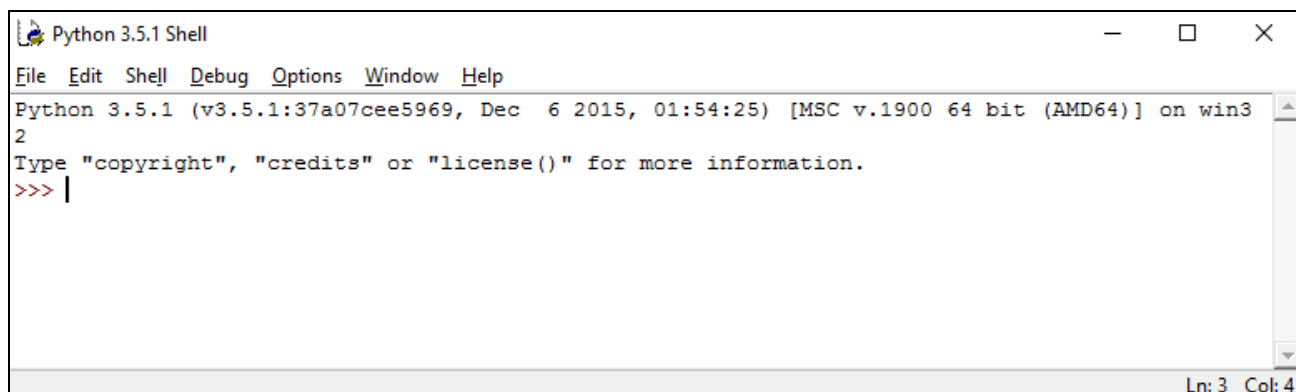
**Figura 8:** Arquivo de Instalação do Python (versão 3.7.0 para amd64 bits)

## 10.1 Usando o Interpretador Python

IDLE é uma interface gráfica para o interpretador Python que permite também a edição e execução dos programas.

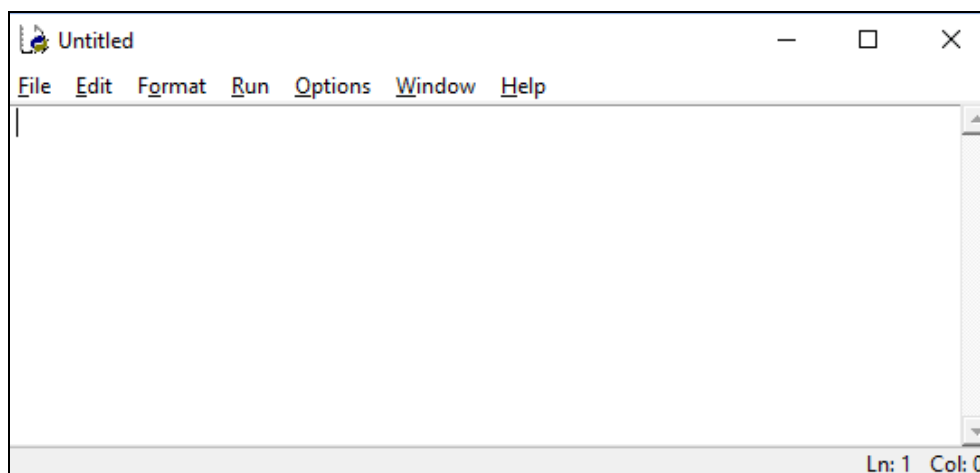
- No **Windows**, deve ter uma pasta no menu **Iniciar > Programas > Python 3.7 > IDLE**
- No **Linux**, abra o terminal e digite: **idle-python3.7 &**
- No **Mac OS X**, abra o terminal e digite: **IDLE3.7 &**

A janela inicial do IDLE versão 3.5.1, no Windows 8 – 64 bits, é mostrada na Figura 9. No Mac OS X, no Linux ou em outra versão do Windows, talvez não seja igual a janela da figura, mas será parecida.



**Figura 9:** Janela Inicial IDLE Python

É possível usar qualquer Editor de Textos para fazer um programa na Linguagem Python e depois apenas usar o Interpretador para Executar o programa, ou então, pode-se usar o próprio Editor do Interpretador, da seguinte forma: na janela do Interpretador, conforme Figura 9 (acima), clique no menu **File >> New File** (ou Ctrl + N). Aparecerá a janela a seguir:



**Figura 10:** Janela do Editor de Textos do IDLE Python

Essa janela do Editor de Textos é parecida com a do Interpretador, mas tem alguns menus diferentes. Neste ambiente serão digitados os códigos-fontes dos programas e, depois de **Salvar** o arquivo (File >> Save ou Ctrl + S), serão executados através da opção **Run >> Run Module** (ou F5) do menu. Ao salvar um arquivo estando neste editor, deve-se escolher apenas o nome do arquivo, pois a extensão aparecerá automaticamente. Os programas em Python tem a extensão **.py**

## 10.2 Exemplos Práticos – Programas em Python

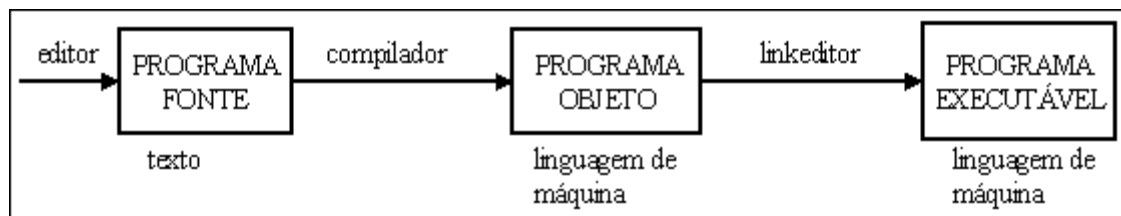
Passar os algoritmos dos exemplos 1 e 2 (página 13) para a Linguagem de Programação Python, transformando em programas para serem executados no computador (*Capítulo 31: Respostas dos Exemplos – Pág.55*).

## 11 Conversão do Programa-Fonte

Como o computador não entende as instruções de um programa-fonte (ou código-fonte), para que este possa ser executado, precisa ser convertido para a linguagem que o computador reconhece, a **linguagem de máquina**. Uma instrução em uma linguagem de alto nível pode corresponder a centenas ou até milhares de instruções em linguagem de máquina. A conversão é feita por programas apropriados, e pode ser feita **antes ou durante** a execução do programa.

### 11.1 Programa Compilado

Quando o **programa-fonte** é todo convertido em linguagem de máquina *antes* da execução, esse processo é chamado de **compilação**, e os programas que o fazem são chamados compiladores. O programa resultante é chamado **programa-objeto** (ou código-objeto), que contém instruções em linguagem de máquina, mas ainda não pode ser executado; para que isso ocorra, é necessária outra etapa chamada "linkedição" ou ligação, efetuada também por um programa apropriado chamado "linkeditor" ou ligador. Na linkedição, são juntados ao código-objeto do programa outros códigos-objeto necessários à sua execução. Após esta etapa, a conversão está completa e produz finalmente um **programa executável** pelo computador, observe a Figura 11.



**Figura 11:** Etapas para Construção de um Programa Executável pelo Computador

### 11.2 Programa Interpretado

Quando a conversão é feita *durante* a execução, o processo é chamado de **interpretação**, e o programa conversor, **interpretador**. Neste caso, o interpretador permanece na memória, junto com o programa-fonte, e converte cada instrução e a executa, antes de converter a próxima, ou seja, vai interpretando e executando linha a linha do programa.

A Linguagem Python é interpretada, mas também pode ser compilada, se necessário. Como visto no capítulo 10.1, a IDLE é a interface gráfica do interpretador Python.

## 12 Softwares que você precisa ter ou conhecer

---

Alguns programas são necessários às atividades de um programador. Abaixo são citados e explicados brevemente alguns dos principais softwares necessários para o programador desenvolver seus programas:

### 12.1 Sistema Operacional

Como programador, é necessário ter um mínimo de informações sobre o sistema ou ambiente operacional em que vai trabalhar, como por exemplo:

- a) Como executar programas
- b) Os nomes que pode dar aos arquivos
- c) Como organizar seus arquivos em pastas ou subdiretórios
- d) Como listar, excluir, renomear e copiar arquivos

### 12.2 Editor de Textos

Um programa em Python (ou em qualquer outra linguagem de programação) é um texto simples, sem caracteres de controle do tipo negrito, tamanho de fonte, paginação etc. Você vai precisar de um editor de textos para digitar seu programa no computador. Editores que produzem tais textos são o Edit, que vem com o MS-DOS, o Bloco de Notas (NotePad), que vem com o Windows, e vários outros. Textos formatados pelo Word ou WordPad, por exemplo, não servem. Os compiladores e interpretadores comerciais normalmente trazem um editor; usá-los será normalmente mais prático.

### 12.3 Compilador/Interpretador

Uma vez digitado o texto do programa (no editor de textos), na linguagem de programação escolhida, é preciso convertê-lo para linguagem de máquina. Para isso, necessita-se de um compilador ou um interpretador (depende da linguagem usada, como visto em capítulos anteriores). Para a linguagem Python que é a apresentada nesta apostila, existem várias IDE's para desenvolvimento. Na página oficial do Python no Brasil encontra-se uma lista com breves explicações sobre cada uma para poder escolher: <https://wiki.python.org.br/IdesPython>.

### 13 Cuidados ao Digitar Programas em Python

---

Em Python, deve-se tomar cuidado com os seguintes itens:

1. **Letras maiúsculas e minúsculas são diferentes.** Assim, `print` e `Print` são completamente diferentes, causando um erro caso digite o `P` maiúsculo. Quando isso acontece, diz-se que a linguagem é *case sensitive*<sup>3</sup>, ou em português, “sensível à caixa” (com relação à caixa baixa e caixa alta, para minúsculas e maiúsculas).
2. **Aspas** são muito importantes e não devem ser esquecidas. Toda vez que abrir aspas, não esqueça de fechá-las. Se esquecer, o programa não funcionará. Observe que o IDLE muda a cor do texto que está entre aspas, facilitando essa verificação.
3. **Parênteses** não são opcionais em Python. Não remova os parênteses dos programas e preste a mesma atenção dada às aspas para abri-los e fechá-los. Todo parêntese aberto deve ser fechado.
4. **Espaços são muito importantes.** A linguagem Python se baseia na quantidade de espaços em branco antes do início de cada linha para realizar diversas operações. O IDLE também ajuda nesses casos, avisando sobre problemas de alinhamento (**recuos**). Os recuos delimitam blocos de comandos a serem executados. Se estiver escrevendo seu programa na IDLE do Python, verá que automaticamente será feito um recuo ao dar Enter após colocar `:` ao final de um comando.

---

<sup>3</sup> Python é *case sensitive*, ou seja, faz diferença entre maiúsculas e minúsculas!



## 14 Comentários em Python

Nas linguagens de programação, normalmente existe alguma forma de comentar uma linha ou um conjunto de linhas de código para documentar o programa em questão. É importante documentar o código-fonte dos programas, pois assim facilita o entendimento caso outra pessoa precise estudar esse código ou até mesmo se o próprio programador, algum tempo depois, necessitar entender ou lembrar do programa. Normalmente usa-se comentários ao longo do código para explicar o que tal comando ou conjunto de comandos faz, para que serve, o que está acontecendo no programa naquele instante, enfim, documentar o que o programa faz e como funciona.

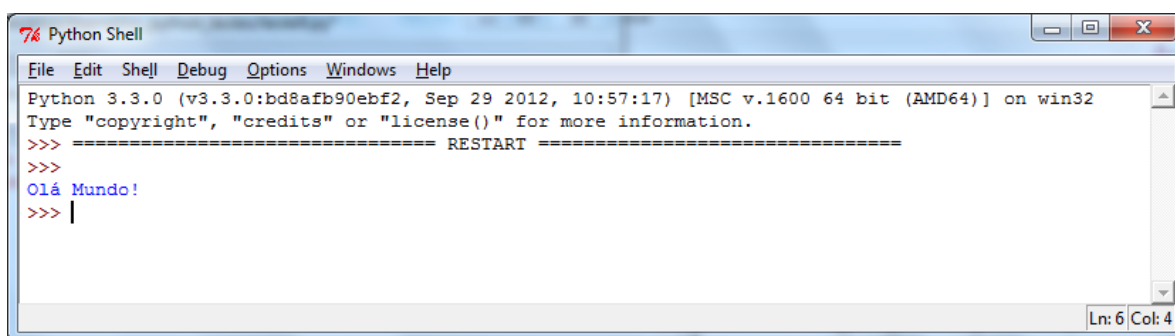
Em Python, tem duas maneiras de incluir comentários ao longo dos programas: pode-se usar três apóstrofes ( ' ' ' ) ou o caractere sustenido ( # ). Uma observação importante: os comentários usados nos códigos de programas, só são vistos por quem analisar o código-fonte daquele programa, ou seja, ao executar o programa não se vê os comentários!

### 14.1 Comentário de Uma Linha

Para fazer comentário de apenas uma linha na Linguagem Python, inicia-se com o caractere # (sustenido, também conhecido como “cerquinha” ou *hashtag*). Então, tudo que estiver depois do caractere # será ignorado pelo interpretador, portanto considerado como comentário do código-fonte do programa (o conteúdo que estiver logo após o caractere # não será executado no programa). O fechamento do comentário acaba quando acabar a linha do interpretador, ou seja, não é necessário “fechar o comentário” de uma linha, ele é fechado automaticamente com a quebra de linha (ao mudar de linha no interpretador). Com isso, cada linha nova de comentário deve-se indicar novamente o caractere #.

Abaixo, exemplo de código-fonte com comentários de uma linha na linguagem Python:

```
#comentário de uma linha pode ser usado no início da linha  
print ("Olá Mundo!") #mas também pode ser acrescentado depois de um comando
```



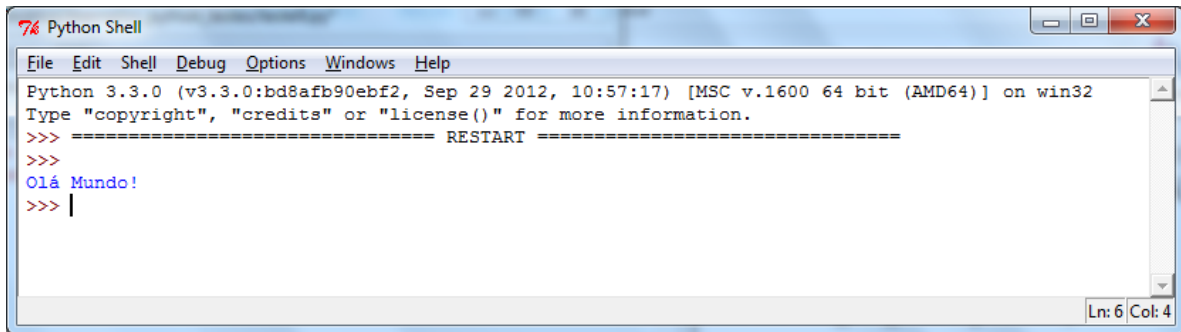
**Figura 12:** Resultado da Execução – Exemplo Comentário de Uma Linha

### 14.2 Comentário de Várias Linhas

Na linguagem Python, também é possível fazer comentários no código-fonte, usando várias linhas, ou seja, um comentário longo, para isso usa-se três apóstrofes para iniciar o comentário e mais três apóstrofes para fechar, finalizar o comentário. Com isso, tudo que estiver entre esse “conjunto” de apóstrofes, não será executado no programa (não será interpretado pelo interpretador Python), será “ignorado”.

A seguir, exemplo de código-fonte com comentários de várias linhas na linguagem Python:

```
'''  
Comentário usando várias linhas em Python, pode-se escrever o que  
quiser e utilizar quantas linhas forem necessárias.  
'''  
print ("Olá Mundo!")
```



**Figura 13:** Resultado da Execução – Exemplo Comentário de Várias Linhas

## 15 Tipos de Dados

Em muitas linguagens de programação é necessário saber e se preocupar com os tipos de dados existentes. Já em Python, não é necessário se preocupar “tanto” com os tipos de dados, pois não é preciso declarar os tipos juntamente com as variáveis. Em Python as variáveis são implicitamente declaradas, ou seja, o tipo delas é o tipo do valor que estão recebendo.

As linguagens de programação mais populares têm diversos tipos de dados integrados e Python não é diferente em relação a isso. Por exemplo, a linguagem de programação C possui tipos de número inteiro e ponto flutuante. Isso significa que em um programa em C, é possível escrever `int i = 100` para criar e inicializar uma variável de número inteiro. O mesmo é possível na linguagem Java e C#. Já em Python faz-se isso de forma mais direta: `i = 100`.

A seguir é apresentada uma tabela com alguns tipos básicos de dados nativos de Python:

Tipo de Dado	Descrição	Exemplo da Sintaxe
<b>NÚMEROS</b>		
<code>int</code>	<b>Inteiros:</b> números inteiros de precisão fixa. São positivos ou negativos sem ponto decimal.	10, 100, -786, 48
<code>float</code>	<b>Ponto flutuante</b> (tem parte fracionária): decimal. Valores de ponto flutuante reais, representam números reais e são escritos com um ponto decimal dividindo as partes inteira e fracionária. Podem também estar na notação científica, com E ou e indicando a potência de 10 ( $2.5e2 = 2.5 \times 10^2 = 250$ ).	3.1415927, -10.5, 0.0, 15.20
<code>long</code>	Inteiros Longos: são inteiros de tamanho ilimitado, escritos como números inteiros e seguido pela letra L, maiúscula ou minúscula.	10897, 51924361L, -1221
<code>complex</code>	Números complexos: são da forma de $a + bj$ , onde $a$ e $b$ são <i>floats</i> e J (ou j) representa a raiz quadrada de -1 (que é um número imaginário). $a$ é a parte real do número, e $b$ é a parte imaginária. Os números complexos não são muito utilizados na programação Python.	3+2j, 20J, 3.14j
<b>CARACTERES</b>		
<code>str</code>	<b>String:</b> uma cadeia de caracteres imutável (palavra, frase etc.) entre aspas.	“frase ou palavra”
<b>BOOLEAN</b>		
<code>bool</code>	Booleano: é um tipo de dado primitivo <sup>4</sup> que possui dois valores, que podem ser considerados como 0 ou 1, verdadeiro ou falso.	True ou False

**Tabela 5:** Alguns Tipos Básicos de Dados Nativos de Python

Fonte: [http://www.tutorialspoint.com/python/python\\_numbers.htm](http://www.tutorialspoint.com/python/python_numbers.htm)

**Atenção:** em Python, é necessário **declarar o tipo**, quando se **lê do teclado** alguma informação!

<sup>4</sup> *Tipos Primitivos:* são aqueles já embutidos no núcleo da linguagem.

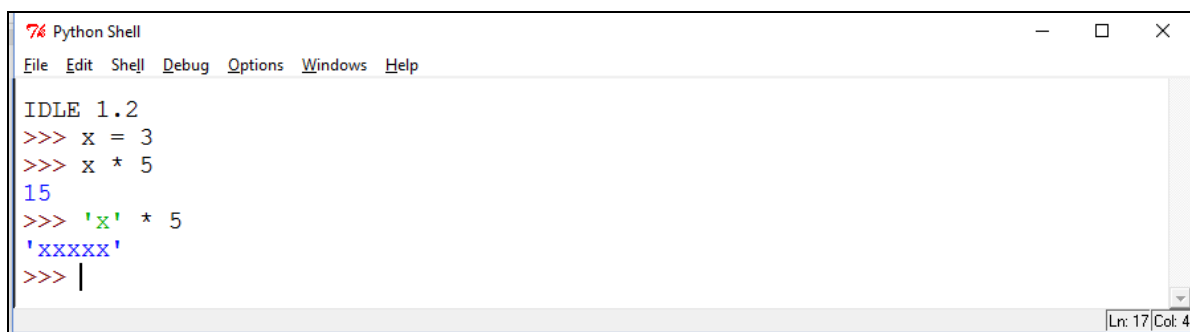
O tipo de uma variável pode mudar conforme o valor atribuído a ela (int, float, str etc.).

### Exemplo:

```
>>> a = "1"
>>> b = 1
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

**Observação importante:** a linguagem Python permite usar L minúsculo com os números do tipo *long*, mas é recomendado usar apenas **L maiúsculo** para evitar confusão com o número 1. Python exibe os inteiros longos com L maiúsculo.

Outra observação importante se dá quanto ao uso de aspas. Por exemplo, o operador **+** realiza uma soma quando aplicado a dados numéricos, mas quando aplicado a dados do tipo string, o sinal **+** faz uma operação de concatenação (junção de duas sequências). Veja o seguinte exemplo:



```
Python Shell
File Edit Shell Debug Options Windows Help
IDLE 1.2
>>> x = 3
>>> x * 5
15
>>> 'x' * 5
'xxxxx'
>>> |
```

Note que **x** e **'x'** são coisas totalmente diferentes. **x** é o nome de uma variável que neste momento se refere ao valor 3 (um int). O resultado de **x\*5** é 15 (outro int). Já **'x'** é uma string com um caractere. Quando o sinal **\*** é aplicado entre uma string e um número inteiro, Python realiza uma operação de repetição. Como pode-se notar, os operadores **+** e **\*** fazem coisas diferentes dependendo dos tipos de dados fornecidos na expressão. Python é uma linguagem muito coerente. Observe:

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> '12' * 3
'121212'
>>> [1,2] * 3
[1, 2, 1, 2, 1, 2]
>>>
```

No primeiro exemplo, o operador **+** concatena duas listas. Os outros dois exemplos mostram a operação de repetição. Note que **12** não é um número, mas uma *string* composta pelos caracteres **1** e **2**. Para Python, strings e listas têm muito em comum: ambas são sequências de itens. Enquanto strings são sequências de caracteres, listas são sequências de itens quaisquer. Nos dois casos, concatenação e repetição funcionam de forma logicamente idêntica.

## 15.1 Conversão de Tipos Numéricos

Python converte números internamente, em uma expressão contendo tipos mistos, para um tipo comum. Mas, às vezes, é necessário converter um número explicitamente de um tipo para outro, para satisfazer as exigências de um parâmetro de operador ou função. Abaixo estão alguns exemplos de conversões:

<code>int(x)</code>	Converte <code>x</code> para um inteiro
<code>long(x)</code>	Converte <code>x</code> para um inteiro longo
<code>float(x)</code>	Converte <code>x</code> para um ponto flutuante
<code>complex(x)</code>	Converte <code>x</code> para um número complexo, com <code>x</code> sendo a parte real e 0 a parte imaginária
<code>complex(x, y)</code>	Converte <code>x</code> e <code>y</code> para número complexo, sendo <code>x</code> a parte real e <code>y</code> a parte imaginária

**Tabela 6:** Exemplos de Conversões de Tipos Numéricos em Python

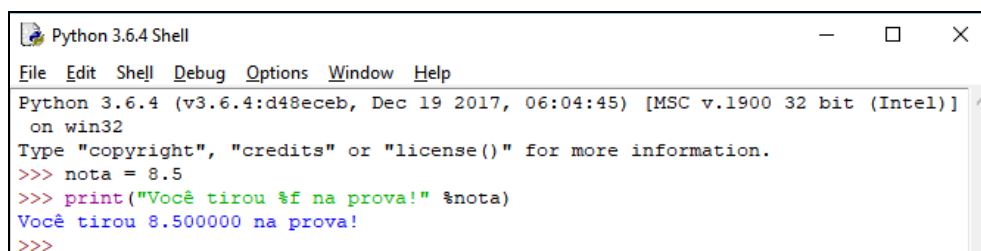
## 15.2 Marcadores e Formatação de Números

Python suporta diversas operações com marcadores. A tabela 7 apresenta os principais tipos de marcadores. Veja que os marcadores são diferentes de acordo com o tipo de variável que está sendo utilizada.

Marcador em Python	Tipo de Dado
<b>%d</b>	Números <b>Inteiros</b>
<b>%f</b>	Números Decimais <sup>5</sup> ( <b>Float</b> )
<b>%s</b>	<b>Strings</b> (alfanumérico: números, letras e sinais)

**Tabela 7:** Marcadores em Python

**Atenção:** além desse uso, **%** é um marcador de posição, ou seja, serve para sinalizar onde Python deverá inserir um número quando deseja-se imprimir uma mensagem na tela que contenha números ou variáveis numéricas. **Exemplo:**

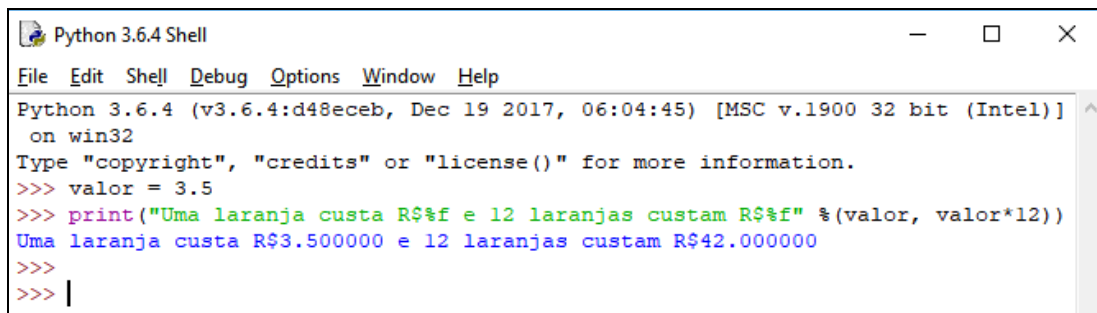


```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> nota = 8.5
>>> print("Você tirou %f na prova!" %nota)
Você tirou 8.500000 na prova!
>>>
```

Entendendo o que aconteceu nesse exemplo: Python substituiu o **marcador %f** pelo valor da variável **nota**. É assim que funciona: a partir de uma string (frase, palavra) com marcas de posição e um ou mais valores, o operador **%** produz uma nova string com os valores inseridos nas respectivas posições. Veja agora um exemplo com dois marcadores (dois valores – duas variáveis numéricas):

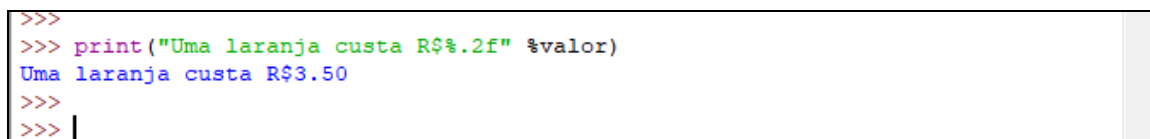
<sup>5</sup> **Números Decimais:** indicam os números que não são inteiros. Geralmente após o algarismo das unidades, usa-se uma vírgula (em Python usa-se **ponto**), indicando que o algarismo a seguir pertence à ordem das casas decimais (parte fracionária).

### Outro Exemplo: com dois marcadores na string



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> valor = 3.5
>>> print("Uma laranja custa R${:f} e 12 laranjas custam R${:f}" .format(valor, valor*12))
Uma laranja custa R$3.500000 e 12 laranjas custam R$42.000000
>>>
>>> |
```

Note que as variáveis `valor` e `valor*12` estão entre parênteses. Isso é obrigatório quando deseja-se passar mais de um valor para o operador `%`. O símbolo `%f` serve para informar ao Python que o valor a ser inserido naquela posição é um **Float**. Se quiser limitar o número de casas após o ponto decimal, basta usar um formato, ou seja, tem que formatar o número, da seguinte maneira:



```
>>>
>>> print("Uma laranja custa R${:.2f}" .format(valor))
Uma laranja custa R$3.50
>>>
>>> |
```

Após o marcador `%`, a indicação `.2` determina que devem aparecer duas casas decimais após o ponto (3.50).

## 16 Instrução Ler

Como visto no capítulo 5 *Instrução Escrever*, existem basicamente duas instruções principais em algoritmos (e em programação em geral) que são: Escrever e Ler. No capítulo 5, foi apresentada a instrução *Escrever*. Agora, neste capítulo, será apresentado o funcionamento da instrução *Ler*.

A instrução *Ler* é utilizada quando deseja-se **obter informações do teclado do computador**, ou seja, é um comando de **Entrada de Dados**. Resumindo: usa-se a instrução *Ler*, quando necessita-se que **o usuário do programa digite alguma informação**. O comando Ler é para “ler informações do teclado do computador”.

Tanto no Diagrama de Chapin quanto em Português Estruturado representa-se a entrada de dados através da palavra *Ler* (ou Leia). Já em Fluxogramas a representação da entrada de dados é feita através de uma forma geométrica específica [GOM04] [MAR03].

### **Exemplo Prático 3:** (feito em aula)

Escreva um algoritmo para **ler** dois valores e armazenar cada um em uma variável. A seguir, armazenar a soma dos dois valores lidos em uma terceira variável. Escrever o resultado da soma efetuada.

## 17 Horizontalização de Fórmulas (Linearização)

Para o desenvolvimento de algoritmos que possuam cálculos matemáticos, as expressões aritméticas devem estar horizontalizadas, ou seja, linearizadas e, para isso, deve-se utilizar os operadores corretamente. Na Tabela 8 a seguir, é apresentado um exemplo de uma expressão aritmética na forma tradicional e como deve ser utilizada nos algoritmos e em programação em geral (linearmente).

Matemática Tradicional	Algoritmo (ou Programa)
$M = \frac{N1 + N2}{2}$	$M \leftarrow (N1 + N2) / 2$

**Tabela 8:** Exemplo de Horizontalização de Expressões Aritméticas

As expressões matemáticas na forma horizontalizada não são apenas utilizadas em algoritmos, mas também na maioria das linguagens de programação.

## 18 Funções Numéricas Predefinidas

A Linguagem de Programação Python (o interpretador Python) oferece várias funções predefinidas, como funções numéricas, matemáticas, de conversão etc. que facilitam e colaboram com o desenvolvimento de programas. Em inglês, são chamadas de *built-in functions*.

### 18.1 Funções Matemáticas

Python inclui as seguintes funções que executam cálculos matemáticos (alguns exemplos):

Função em Python	Descrição do Retorno (resposta da função)
<code>abs(x)</code>	O valor absoluto de $x$ : a distância (positiva) entre $x$ e zero
<code>exp(x)</code>	O exponencial de $x$ : $e^x$
<code>log(x)</code>	O logaritmo natural de $x$ , para $x > 0$
<code>log10(x)</code>	O logaritmo base 10 de $x$ , para $x > 0$
<code>max(x1, x2, ...)</code>	O maior dos seus argumentos: o valor mais próximo de infinito positivo
<code>min(x1, x2, ...)</code>	O menor dos seus argumentos: o valor mais próximo de infinito negativo
<code>pow(x, y)</code>	O valor de $x^{**}y$ , ou seja: $x^y$
<code>round(x [, n])</code>	$x$ arredondado para $n$ dígitos a partir do ponto decimal. Python arredonda longe do zero como um <i>tie-breaker</i> (desempate): <code>round(0.5)</code> é 1.0 e <code>round(-0.5)</code> é -1.0
<code>sqrt(x)</code>	Raiz quadrada de $x$ , para $x > 0$

**Tabela 9:** Exemplos de Funções Matemáticas em Python



## 18.2 Funções de Números Randômicos

Números randômicos são usados para jogos, simulações, testes etc. Python inclui algumas funções comumente usadas, tais como:

Função em Python	Descrição do Retorno (resposta da função)
<code>choice(seq)</code>	Um item aleatório de uma lista, tupla ou string
<code>randrange([start,] stop [,step])</code>	Um elemento selecionado aleatoriamente a partir de um intervalo (iniciar, parar, passo)
<code>random()</code>	Um <i>float</i> aleatório $r$ , tal que 0 é menor ou igual a $r$ e $r$ é menor que 1
<code>shuffle(lst)</code>	Randomiza os itens de uma lista (no lugar). Não tem retorno.
<code>uniform(x, y)</code>	Um <i>float</i> aleatório $r$ , de tal modo que $x$ é menor ou igual a $r$ e $r$ é menor que $y$

**Tabela 10:** Exemplos de Funções de Números Randômicos em Python

Na página oficial do Python encontra-se uma lista completa das funções predefinidas da linguagem com explicações detalhadas e exemplos: <http://docs.python.org/2/library/functions.html>.

## 19 Expressões Aritméticas

As expressões aritméticas são escritas *linearmente*, usando-se a notação matemática. A Tabela 11 apresenta dois exemplos de representação de expressões aritméticas em Python:

Expressão Aritmética	Representação em Python
$\sqrt{Px(P-A)x(P-B)x(P-C)}$	<code>sqrt ( P * ( P-A ) * ( P-B ) * ( P-C ) )</code>
$A-Bx\left(C+\frac{D}{E-1}-F\right)+G$	<code>A-B * ( C+D / ( E-1 ) -F ) +G</code>

**Tabela 11:** Exemplo de Representação de Expressão Aritmética em Python  
(Horizontalização)

## 20 Algoritmos com Seleção

Os exemplos vistos até agora, nesta apostila, eram puramente sequenciais, ou seja, todas as instruções eram executadas seguindo a ordem do algoritmo (normalmente, de cima para baixo). Neste capítulo, serão apresentadas algumas estruturas de seleção.

Uma estrutura de seleção, como o próprio nome já diz, permite que determinadas instruções sejam executadas ou não, dependendo do resultado de uma condição (de um teste), ou seja, o algoritmo terá mais de uma saída, mais de uma opção que será executada de acordo com o teste realizado.

### **Exemplo Prático 4:** (feito em aula)

Escreva um algoritmo para ler um valor. **Se** o valor lido for igual a 6, escrever a mensagem “Valor lido é o 6”, caso contrário escrever a mensagem “Valor lido não é o 6”.

Quando se está utilizando algoritmos com seleção, pode-se utilizar dois tipos de estruturas diferentes, dependendo do objetivo do algoritmo, chamadas de "Seleção Múltipla", cujos tipos são: Estrutura Aninhada e Estrutura Concatenada. Os capítulos 20.1 e 20.2 a seguir, apresentam essas duas estruturas com suas respectivas características.

20.1 Estrutura de Seleção Aninhada

A estrutura de Seleção Aninhada normalmente é utilizada quando é necessário fazer várias *comparações (testes) sempre com a mesma variável*. Esta estrutura é chamada de aninhada porque na sua representação (tanto em Chapin quanto em Português Estruturado) coloca-se *uma seleção dentro de outra seleção*.

Como exemplo destes dois tipos de estruturas, será utilizado o exercício número 35 da Apostila de Exercícios de Algoritmos e Programação (pág.8). Abaixo é apresentada a **resposta do exercício 35**, representada em Diagrama de Chapin, utilizando a estrutura Aninhada:

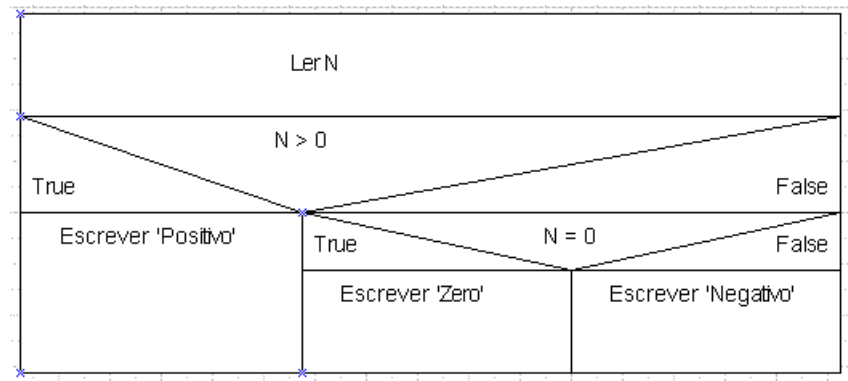
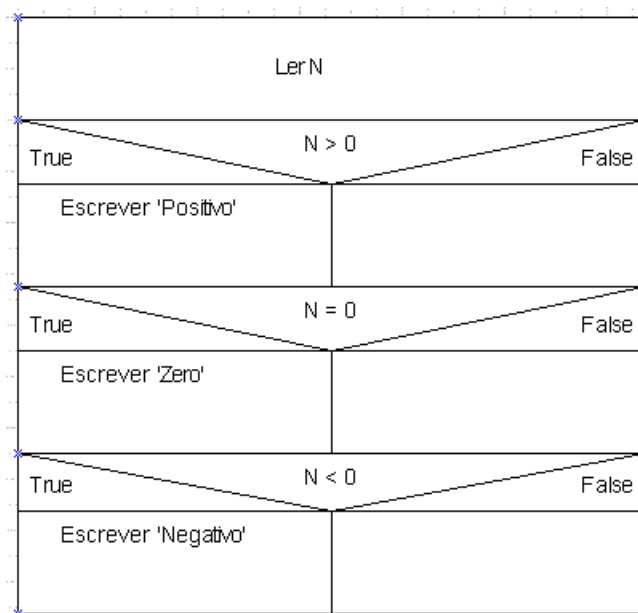


Figura 14: Exemplo de Estrutura de Seleção Aninhada - Resposta do Exercício 35

## 20.2 Estrutura de Seleção Concatenada

A estrutura de Seleção Concatenada normalmente é utilizada quando se está *comparando (testando) variáveis diferentes*, ou seja, independentes entre si. Esta estrutura é chamada de concatenada porque na sua representação (tanto em Chapin quanto em Português Estruturado) as seleções ficam separadas uma da outra, e nesse caso, não existe o lado "falso" do Chapin, ou o "Senão" do Português Estruturado.

Abaixo é apresentada a **resposta do exercício número 35** da Apostila de Exercícios de Algoritmos e Programação (pág.8), representada em Diagrama de Chapin, utilizando a estrutura de seleção Concatenada:



**Figura 15:** Exemplo de Estrutura de Seleção Concatenada - Resposta do Exercício 35

Como pode ser observado nessas duas respostas apresentadas para o exercício 35 (estrutura aninhada e estrutura concatenada, nas figuras 14 e 15, respectivamente), existe uma grande diferença entre as duas estruturas, ou seja, uma *característica de execução* do algoritmo. Você saberia dizer qual é esta diferença?<sup>6</sup>

<sup>6</sup> *Resposta:* A execução da estrutura aninhada é mais rápida que a concatenada, ou seja, é mais eficaz. Pois a estrutura concatenada, segue sendo executada mesmo depois de ter encontrado a resposta. A forma aninhada é mais otimizada!

## 21 Operadores Relacionais

Operações relacionais são as **comparações** permitidas entre valores, variáveis, expressões e constantes. A Tabela 12 a seguir, apresenta os operadores relacionais utilizados nos algoritmos e na programação em geral:

Símbolo do Operador na Linguagem Python	Significado da Operação	Símbolo Matemático
>	maior que	>
<	menor que	<
==	igual	=
>=	maior ou igual	≥
<=	menor ou igual	≤
!=	diferente (não igual)	≠

**Tabela 12:** Operadores Relacionais – símbolos da Linguagem Python

A seguir, na Tabela 13, são apresentados alguns **exemplos de comparações válidas**, ou seja, possíveis testes utilizando os operadores relacionais:

Comparações Válidas	Exemplos
variável e constante	X == 3
variável e variável	A != B
variável e expressão	Y == W + J
expressão e expressão	(X + 1) < (Y + 4)

**Tabela 13:** Exemplos de Comparações Válidas

**Atenção:** o símbolo do operador, pode variar de acordo com a sintaxe da linguagem de programação utilizada.

## 22 Seleção em Python

---

Estrutura de seleção (ou estrutura condicional) é uma estrutura de desvio do fluxo de controle, presente em linguagens de programação, que realiza diferentes ações dependendo se a seleção (ou condição) é verdadeira ou falsa. Ou seja, nem sempre deseja-se que todas as linhas de código de um programa sejam executadas. Muitas vezes é necessário decidir quais partes do programa devem ser executadas, ou não, dependendo do resultado de uma condição (um teste).

Na linguagem Python, a estrutura de decisão é o **if**, sendo que pode-se apresentar de duas formas: simples ou composta. Nos capítulos a seguir essa estrutura é apresentada em detalhes.

### 22.1 Forma Simples: if

O formato da estrutura condicional **if** e a sintaxe em Python é:

```
if condição:
    bloco de código verdadeiro
```

O comando **if**, significa “se”, traduzindo para o português, então as linhas de código descritas acima devem ser lidas da seguinte forma: “**se a condição for verdadeira, execute os seguintes comandos**” (se o teste for verdadeiro, execute os comandos a seguir). Se o teste for falso, os comandos serão ignorados. Para a condição (teste) usa-se os Operadores Relacionais (capítulo 21).

Os dois-pontos ( : ), após a condição, indicam que um bloco de linhas de código virá a seguir (pode ser apenas uma linha ou várias linhas de código). Em Python, um bloco é representado deslocando-se o início da linha para a direita (reco da linha indica se está dentro do comando ou fora dele). O bloco continua até a primeira linha com deslocamento diferente, ou seja, todos os comandos que estiverem com o mesmo deslocamento para a direita, fazem parte do mesmo bloco de código. O bloco só será finalizado, quando houver uma linha com deslocamento diferente (o deslocamento, ou seja, o reco de um bloco de código, determina o início e o fim do bloco). Veja o exemplo a seguir:

```
a = int(input("Digite um valor: "))
b = int(input("Digite outro valor: "))
if a < b :
    print ("O primeiro valor é menor que o segundo valor digitado")
print ("Tchau!")
```

O deslocamento necessário para delimitar um bloco de código em Python, é feito através da endentação<sup>7</sup> das linhas de código, ou seja, todos os comandos que fazem parte do mesmo bloco, devem estar com o mesmo reco da margem.

Python é uma das poucas linguagens de programação que utiliza o **deslocamento do texto à direita (reco) para marcar o início e o fim de um bloco de comandos**. Outras linguagens utilizam palavras ou caracteres especiais para isso como **begin** e **end**, em Pascal ou as famosas chaves ( { } ), em C e em Java.

---

<sup>7</sup> O reco de uma ou várias linhas de código, serve para delimitar um “bloco de código”, na linguagem de programação Python. Esse reco é denominado *indentation*, em inglês. Em português alguns descrevem como “identação”, outros como endentação, que significa entalhe, recorte dentado.

## 22.2 Forma Composta: if - else

Quando há uma situação em que deseja-se que o programa faça tal coisa, se o teste for verdadeiro, mas senão, se o teste for falso, deve fazer outra coisa, então usa-se o comando **else**, como apresentado no exemplo a seguir:

```
a = int(input("Digite um valor: "))
b = int(input("Digite outro valor: "))
if a < b :
    print ("O primeiro valor é menor que o segundo valor digitado")
else:
    print ("Tchau!")
```

Nesse exemplo acima, lê-se: se a é menor que b então **escreve** na tela "O primeiro valor é menor...", **senão**, escreve na tela "Tchau!". **Else** significa senão, em português.

Quando forem necessários vários **if**'s, Python apresenta duas maneiras de usá-los, apresentadas a seguir (resposta do exercício 35).

### 22.2.1 Usando if - else de forma aninhada:

```
numero = int(input("Digite um número: "))
if numero < 0:
    print("O número digitado é Negativo")
else:
    if numero > 0:
        print("O número digitado é Positivo")
    else:
        print ("O número digitado é o Zero")
```

### 22.2.2 Usando o comando elif:

Pode haver zero ou mais partes **elif**, e a parte do **else** é opcional. A palavra-chave 'elif' é uma abreviação para 'else if', e é útil para evitar o uso excessivo de recuos (*identificações*). Uma sequência **if ... elif ... elif ...** é um substituto, usado na Linguagem Python, para os comandos **switch** ou **case** usados em algumas linguagens de programação.

#### Exemplo usando o comando elif:

```
numero = int(input("Digite um número: "))
if numero < 0:
    print("O número digitado é Negativo")
elif numero > 0:
    print("O número digitado é Positivo")
elif numero == 0:
    print ("O número digitado é o Zero")
```

## 23 Operadores Lógicos

Os operadores lógicos permitem que mais de uma condição seja testada em uma única expressão, ou seja, pode-se fazer mais de uma comparação (mais de um teste) ao mesmo tempo. A Tabela 14 a seguir, apresenta os operadores lógicos utilizados na construção de algoritmos e na programação em geral:

Operação	Operador no Algoritmo	Operador em Python
Negação	Não ( $\sim$ )	<code>not</code>
Conjunção	E ( $\wedge$ )	<code>and</code>
Disjunção (não-exclusiva)	Ou ( $\vee$ )	<code>or</code>

**Tabela 14:** Operadores Lógicos

Note que a Tabela 14 acima, apresenta os operadores lógicos já ordenados de acordo com suas prioridades, ou seja, se na mesma expressão tiver o operador **ou** e o operador **não**, por exemplo, primeiro será executado o **não** e depois o **ou**.

De uma forma geral, os resultados possíveis para os operadores lógicos podem ser vistos na Tabela 15 abaixo, conhecida como **Tabela Verdade**:

Variáveis		Operações Lógicas		
A	B	A e B	A ou B	não A
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

**Tabela 15:** Possíveis Resultados dos Testes Lógicos - Tabela Verdade

### Exemplos de Testes utilizando Operadores Lógicos:

➔ Exemplos usando operadores lógicos [KOZ06]: exemplos em português, usados no dia-a-dia, para entender o conceito dos operadores.

Expressão	Quando eu não saio?
Se chover <u>e</u> relampejar, eu não saio.	Somente quando chover e relampejar ao mesmo tempo (apenas 1 possibilidade).
Se chover <u>ou</u> relampejar, eu não saio.	Somente quando chover, somente quando relampejar ou quando chover e relampejar ao mesmo tempo (3 possibilidades).



**Exemplos A, B e C abaixo são apresentados em Português Estruturado:**

- A)** Se (**salario** > 180) **e** (**salário** < 800) Então  
 Escrever “Salário válido para financiamento”  
 Senão  
 Escrever “Salário fora da faixa permitida para financiamento”  
 FimSe
- B)** Se (**idade** < 18) **ou** (**idade** > 95) Então  
 Escrever “Você não pode fazer carteira de motorista”  
 Senão  
 Escrever “Você pode possuir carteira de motorista”  
 FimSe
- C)** Se (**idade** >= 18) **e** (**idade** <= 95) **e** (**aprovado\_exame** == “sim”) Então  
 Escrever “Sua carteira de motorista estará pronta em uma semana”  
 Senão  
 Escrever “Você não possui idade permitida ou não passou no exame”  
 FimSe

**23.1 Expressões Lógicas e Prioridades das Operações**

Os operadores lógicos podem ser combinados em expressões lógicas mais complexas. Quando uma expressão tiver mais de um operador lógico, avalia-se o operador **not** (não) primeiramente, seguido do operador **and** (e) e finalmente **or** (ou). Ou seja, existe uma prioridade de execução entre os operadores lógicos, como mostrado na Tabela 16 a seguir:

Operador Lógico no Algoritmo	Operador Lógico em Python	Prioridade de Execução
Não	not	1 <sup>a</sup> .
E	and	2 <sup>a</sup> .
Ou	or	3 <sup>a</sup> .

**Tabela 16:** Prioridades das Operações Lógicas

**Atenção:** Vários níveis de **parênteses** também podem ser usados em expressões lógicas para fixar entre os operadores uma ordem de execução, diferente da indicada pela prioridade dada na Tabela 16. Lembrando que as operações entre parênteses sempre têm prioridade máxima. Então, se quiser uma sugestão: use parênteses para definir a ordem de execução das operações e durma tranquilo! 😊

A seguir são apresentados mais alguns exemplos de expressões lógicas.

Para **A = V**, **B = F** e **C = F**, as expressões abaixo fornecem os seguintes resultados:

Expressões	Resultados
a) não A	não V = F
b) A e B	V e F = F
c) não (B ou C)	não (F ou F) = não F = V
d) não B ou C	(não F) ou F = V ou F = V

### Exemplo Prático utilizando Operadores Lógicos para ser resolvido em aula:

5) Escreva um algoritmo para ler um valor e escrever se ele está entre os números 1 e 10, ou não está.

Ao verificar as respostas (errada e correta) do exemplo número 5 acima, pode-se constatar que quando necessita-se fazer mais de um teste ao mesmo tempo, deve-se utilizar os operadores lógicos apresentados neste capítulo na Tabela 14.

**Resumo das Prioridades de Operações:** se houver vários tipos de operadores em uma mesma expressão, a ordem de execução é a descrita na Tabela 17 a seguir.

Prioridades de Execução	
Tipos de Operações	Operadores em Python
1ª. Aritméticas	* / + -
2ª. Relacionais	> < == >= <= !=
3ª. Lógicas	not and or

**Tabela 17:** Resumo das Prioridades de Operações

**Exemplo:** Supondo que `salario = 100` e `idade = 20`, resolva a expressão abaixo:

**`salario > 1000 and idade > 18`**

Como resultado tem-se:

`100 > 1000` and `20 > 18`  
False and True  
**False**

A resposta final desse exemplo é **False**, pois executa-se primeiramente as operações relacionais (sublinhadas apenas para destacar a prioridade de execução), ou seja, os dois testes de maior ( `>` ) para, depois, executar a operação lógica, **and** ( e ).

## 24 Algoritmos com Repetição

---

Nos exemplos e exercícios vistos até agora, sempre foi possível resolver os problemas com uma sequência de instruções que eram *executadas apenas uma vez*. Existem três estruturas básicas para a construção de algoritmos, que são: algoritmos sequenciais, algoritmos com seleção e algoritmos com repetição. A combinação dessas três estruturas permite a construção de algoritmos para a resolução de problemas extremamente complexos [MAR03]. Nos capítulos anteriores viu-se a estrutura puramente sequencial e algoritmos com seleção (capítulo 20). Neste capítulo, serão apresentadas as estruturas de repetição possíveis em algoritmos e existentes na maioria das linguagens de programação.

Uma estrutura de repetição permite que uma sequência de instruções (de comandos) seja executada várias vezes, até que uma condição, um teste seja satisfeito, ou seja, repete-se um conjunto de instruções sem que seja necessário escrevê-lo várias vezes. As estruturas de repetição também são chamadas de Laços ou Loops [MAR03].

Para saber quando utilizar uma estrutura de repetição, basta analisar se uma instrução ou uma sequência de instruções precisa ser executada várias vezes, se isto se confirmar, então deve-se utilizar uma estrutura de repetição. As estruturas de repetição permitem que um trecho do algoritmo (conjunto de instruções) seja repetido um número determinado (ou indeterminado) de vezes, sem que o código correspondente, ou seja, sem que as instruções a serem repetidas tenham que ser escritas mais de uma vez [TON04].

### **Exemplo Prático 6:** (feito em aula)

Escreva um algoritmo para comer um cacho de uva.

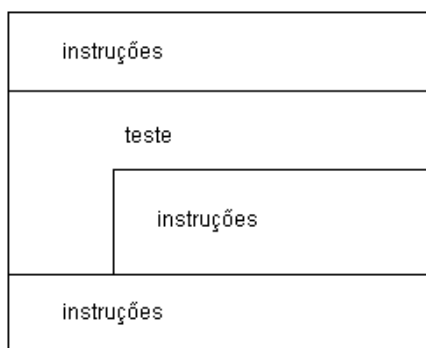
➔ A resposta apresentada em aula, no quadro, (em Chapin e/ou Português Estruturado) está correta?

Na primeira solução do exemplo 6 apresentada em aula, não foi utilizada uma ‘estrutura de repetição’, por isto **o algoritmo não está correto**. Nesse exemplo número 6 é necessária uma estrutura de repetição, pois a instrução "*comer 1 uva*" precisa ser repetida várias vezes.

Na linguagem Python existem dois tipos de estruturas de repetição: **Enquanto** (`while`) e **Para** (`for`), cada uma com suas peculiaridades e apropriada para cada tipo de problema. Muitas vezes, é possível resolver um mesmo problema usando qualquer uma das estruturas de repetição, mas, na maioria das situações, haverá uma mais adequada. Neste capítulo serão apresentadas as características de cada uma dessas estruturas.

## 24.1 Estrutura de Repetição: Enquanto

Na estrutura **Enquanto** as instruções a serem repetidas podem não ser executadas nenhuma vez, pois o *teste fica no início da repetição*, então a execução das instruções, que estão "dentro" da repetição, depende do teste. Nesta estrutura, a repetição é finalizada quando o teste é Falso (F), ou seja, **enquanto o teste for Verdadeiro as instruções serão executadas e, quando o teste for Falso, o laço é finalizado.** Veja nas Figuras 16 e 17 abaixo a forma geral da estrutura Enquanto em Chapin e em Português Estruturado.



**Figura 16:** Estrutura Enquanto em Diagrama de Chapin

```

início
  instruções
  enquanto teste faça
    instruções
  fim_enquanto
  instruções
fim

```

**Figura 17:** Estrutura Enquanto em Português Estruturado

### **Observações da estrutura de repetição ENQUANTO:**

- 1) A repetição (o laço) **encerra** quando a condição for **falsa** (quando o teste for falso).
- 2) As instruções a serem repetidas **podem nunca ser executadas**, porque o teste é no início da repetição (só entra no laço, quando o teste for verdadeiro).

**Atenção:** Resolva o exemplo 6 anterior, utilizando a estrutura de repetição Enquanto!

## 25 Repetições em Python

---

Nos capítulos a seguir serão apresentadas as estruturas de repetição Enquanto (`while`) e Para (`for`) na linguagem de programação Python, com alguns exemplos de usos e sintaxes.

### 25.1 Estrutura de Repetição: `while`

O **`while`** é um comando que manda um bloco de código ser executado **enquanto** uma condição for satisfeita (enquanto o teste for verdade, executa o laço). Assim, permite que sejam criados loops de execução. O **`while`** é um comando muito útil, mas pode ser perigoso, pois, se não tratarmos corretamente o critério de parada, o laço pode não ter fim, e o programa não faz o que deveria e pode entrar em *loop infinito*, como é chamado.

Exemplo do uso do laço **`while`** em Python:

```
x = 100
while x > 0:
    print ("x > 0")
    x = x - 1
```

Neste código, será mostrada a mensagem “**`x > 0`**” até que `x` seja igual a zero (ou seja, “enquanto `x` for maior que zero”, executa o laço). Quando o valor de `x` passar para 0, a verificação `x > 0` retornará falso e o programa sai do laço (o laço é finalizado).

Ao executar o programa acima, ele escreverá 100 vezes na tela a mensagem “`x > 0`”.

## 26 Dizer SIM para Continuar ou NÃO para Finalizar o Laço

---

### Exemplo Prático 7: (feito em aula)

Escreva um algoritmo para ler dois valores. Após a leitura, deve-se calcular a soma dos valores lidos e armazená-la em uma variável. Após o cálculo da soma, escrever o resultado e escrever também a pergunta “Novo Cálculo (S/N)?”. Deve-se ler a resposta e, se a resposta for “S” (sim), deve-se repetir todos os comandos (instruções) novamente, mas se a resposta for “N” (não), o algoritmo deve ser finalizado escrevendo a mensagem “Fim dos Cálculos”.

## 27 Contadores e Acumuladores

---

Em algoritmos com estruturas de repetição (Enquanto ou Para) é comum surgir a necessidade de utilizar variáveis do tipo contador e/ou acumulador. Neste capítulo são apresentados esses conceitos detalhadamente.

### 27.1 Contadores

Um contador é utilizado para contar o número de vezes que um evento (instrução, comando) ocorre, ou seja, contar a quantidade de vezes que uma instrução é executada (ou um conjunto de instruções).

Forma Geral: **VARIÁVEL**  $\leftarrow$  **VARIÁVEL** + **CONSTANTE**

Exemplo:  $X \leftarrow X + 1$

Explicação: um contador é uma variável (qualquer) que recebe ela mesma mais um valor (uma constante), no caso do exemplo acima, a variável X está recebendo o valor dela mesma mais 1. Normalmente a constante que será somada no contador é o valor 1, para contar de 1 em 1, mas pode ser qualquer valor, como por exemplo 2, se quisermos contar de 2 em 2.

#### Observações dos Contadores:

1) A variável (do contador) deve possuir um valor inicial conhecido, isto é, ela deve ser inicializada. Normalmente inicializa-se a variável do contador com zero, ou seja, zera-se a variável antes de utilizá-la. Para zerar uma variável basta atribuir a ela o valor zero: **VARIÁVEL**  $\leftarrow$  **0**

2) A constante (que é geralmente o valor 1) determina o valor do incremento da variável (do contador), ou seja, o que será somado (acrescido) a ela. Normalmente usa-se o valor 1 para esse incremento, mas pode ser qualquer valor, dependendo do problema a ser resolvido.

#### Exemplo 8 – usando Contador: (feito em aula)

Escreva um algoritmo para perguntar ao usuário se ele quer digitar um número. Enquanto o usuário responder que “sim”, o algoritmo deve ler o número que o usuário digitou e **contar** quantos números ele vai digitar, ou seja, quantas vezes ele vai dizer que “sim”, quer digitar um número.

## 27.2 Acumuladores (ou Somadores)

Um acumulador, também conhecido como Somador, é utilizado para obter somatórios ( $\Sigma$ ).

Forma Geral: **VARIÁVEL1**  $\leftarrow$  **VARIÁVEL1** + **VARIÁVEL2**

Exemplo: **X**  $\leftarrow$  **X** + **Y**

Explicação: um acumulador (somador) é uma variável (qualquer) que recebe ela mesma mais uma outra variável, no caso do exemplo acima, a variável X está recebendo o valor dela mesma mais o valor da variável Y. A variável Y representa o valor a ser somado, acumulado na variável X, que é o somador.

### Observações dos Acumuladores:

1) A variável1 (do acumulador) deve possuir um valor inicial conhecido, isto é, ela deve ser inicializada. Normalmente inicializa-se a variável do acumulador com zero, ou seja, zera-se a variável antes de utilizá-la. Para zerar uma variável basta atribuir a ela o valor zero: **VARIÁVEL1**  $\leftarrow$  **0**

2) A variável2 indica o valor a ser acumulado, somado e armazenado na variável1.

### Exemplo 9 – usando Acumulador:

Altere o exemplo 8 para calcular também a soma de todos os números lidos, ou seja, o somatório dos números que o usuário digitou.

**ATENÇÃO:** Normalmente inicializa-se as variáveis que serão utilizadas como *contador* ou como *acumulador* com o valor *zero*, mas pode-se inicializá-las com um valor qualquer de acordo com a necessidade do problema a ser resolvido.

## 28 Estrutura de Repetição: Para (For em Python)

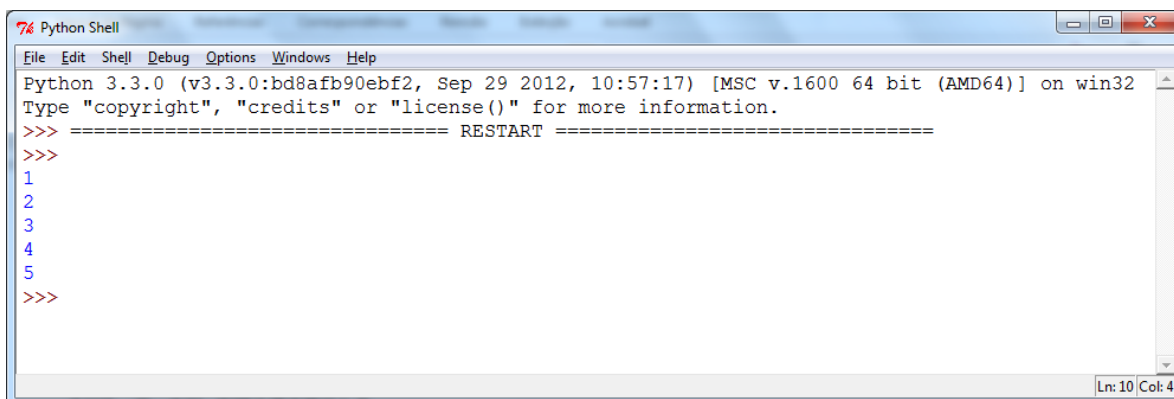
A laço `for` em Python é usado de maneira um pouco diferente do que em outras linguagens, como C ou Pascal. Ao invés de sempre ocorrer uma iteração sobre uma progressão aritmética de números (como em Pascal), ou dar ao usuário a possibilidade de definir o passo da iteração ou de travar a condição (como em C), em Python, a instrução `for` itera sobre os itens de qualquer sequência (uma lista ou uma *string*), na ordem em que eles aparecem na sequência.

O `for` percorre uma sequência em ordem, a cada ciclo substituindo a variável especificada por um dos elementos.

Em outras linguagens, determina-se o valor inicial da variável que controla o laço, seu limite e o passo de iteração. Em Python isso não acontece. Aqui, o `for` somente itera sobre **listas**!

### Exemplo A)

```
l = [1,2,3,4,5]
for i in l:
    print (i)
```



**Figura 18:** Tela que mostra o resultado do exemplo acima – Exemplo A

Explicação do ‘Exemplo A’: a cada iteração<sup>8</sup>, a variável `i` recebe o valor de um elemento da lista (da sequência).

O código do ‘Exemplo A’ acima, também poderia ser escrito da seguinte maneira:

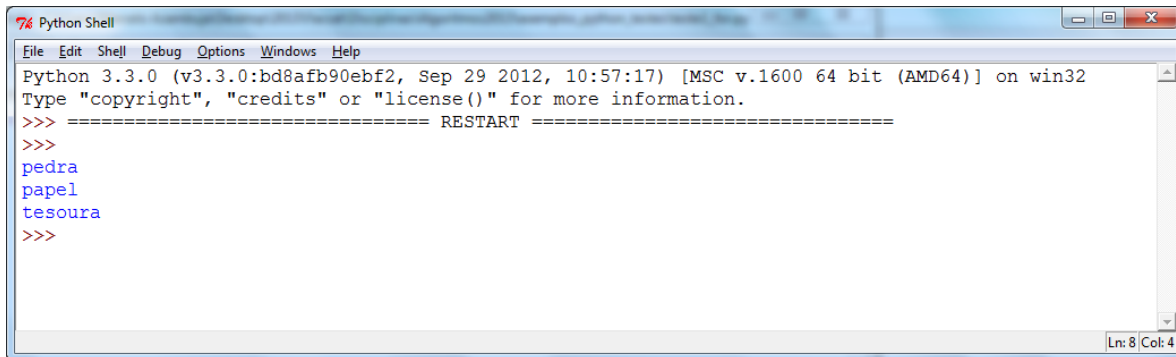
```
for i in [1,2,3,4,5]:
    print (i)
```

<sup>8</sup> *Iteração*: cada execução do laço.



**Exemplo B)**

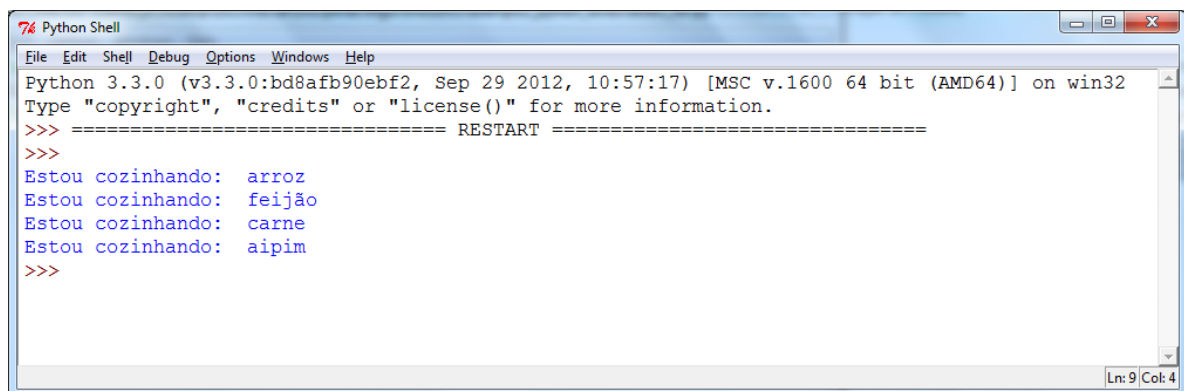
```
jogo = ["pedra", "papel", "tesoura"]
for item in jogo:
    print (item)
```



**Figura 19:** Tela que mostra o resultado do exemplo acima – Exemplo B

**Exemplo C)**

```
comidas = ["arroz", "feijão", "carne", "aipim"]
for x in comidas:
    print ("Estou cozinhando: ", x)
```

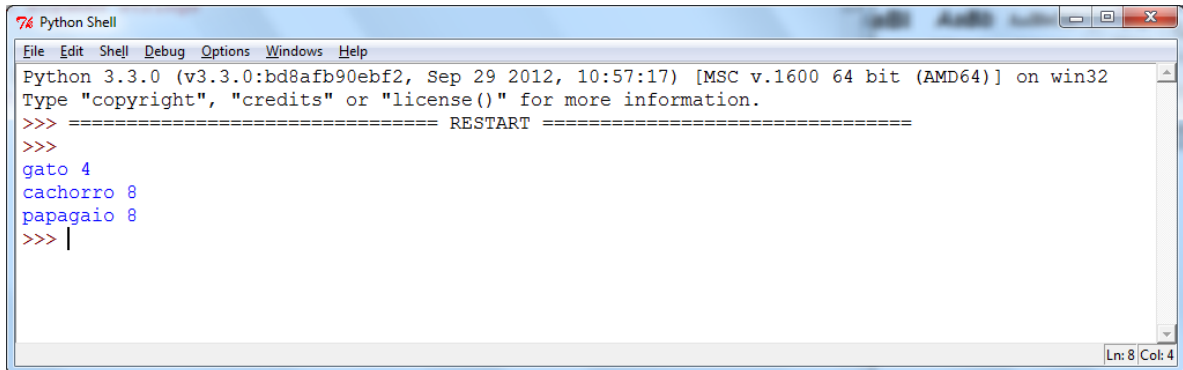


**Figura 20:** Tela que mostra o resultado do exemplo acima – Exemplo C

**Exemplo D)**

```
# Medindo algumas strings

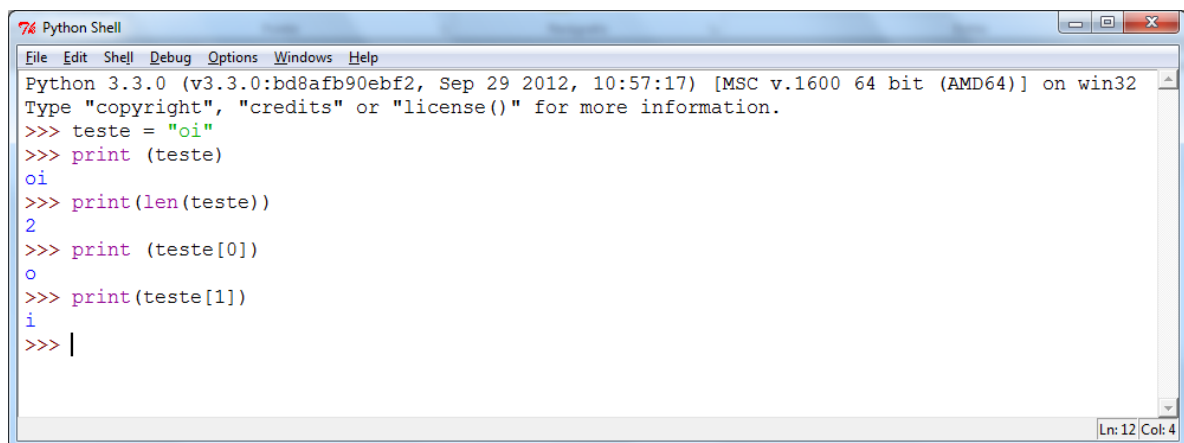
palavras = ["gato", "cachorro", "papagaio"]
for x in palavras:
    print (x, len(x))
```



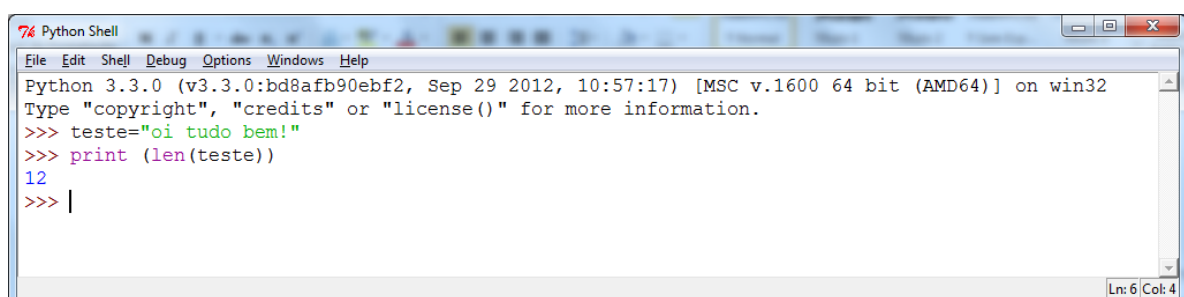
```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
gato 4
cachorro 8
papagaio 8
>>> |
```

**Figura 21:** Tela que mostra o resultado do exemplo acima – Exemplo D

A função **len ()** retorna o **tamanho de uma string** (palavra, frase etc.), ou seja, ela mede o tamanho da string e informa esse valor (tamanho da string equivale a quantidade de caracteres, incluindo espaços). Veja alguns exemplos:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> teste = "oi"
>>> print (teste)
oi
>>> print(len(teste))
2
>>> print (teste[0])
o
>>> print (teste[1])
i
>>> |
```

**Figura 22:** Exemplos com strings e usando a função **len ()**

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> teste="oi tudo bem!"
>>> print (len(teste))
12
>>> |
```

**Figura 23:** Mais um Exemplo com string usando a função **len ()**

**Atenção:** uma string é formada por várias posições, ou seja, quando armazenamos uma palavra ou uma frase em uma variável, cada caractere dessa string fica armazenado em uma posição, cada caractere tem o seu índice (que é um número correspondente a posição do caractere na string). Pode-se imaginar como se a variável do tipo string fosse dividida em várias “casinhas”, cada uma com um número correspondente a sua posição. Por isso, é possível trabalhar cada caractere de uma string, separadamente, de acordo com a posição de cada um.

## 29 Função Range

Se você precisa repetir uma sequência de números, a função `range()`, da linguagem Python, vem a calhar. Ela gera listas contendo progressões aritméticas. Exemplo de uso da função `range` junto com o laço `for`:

```
for a in range(10):  
    print (a)
```

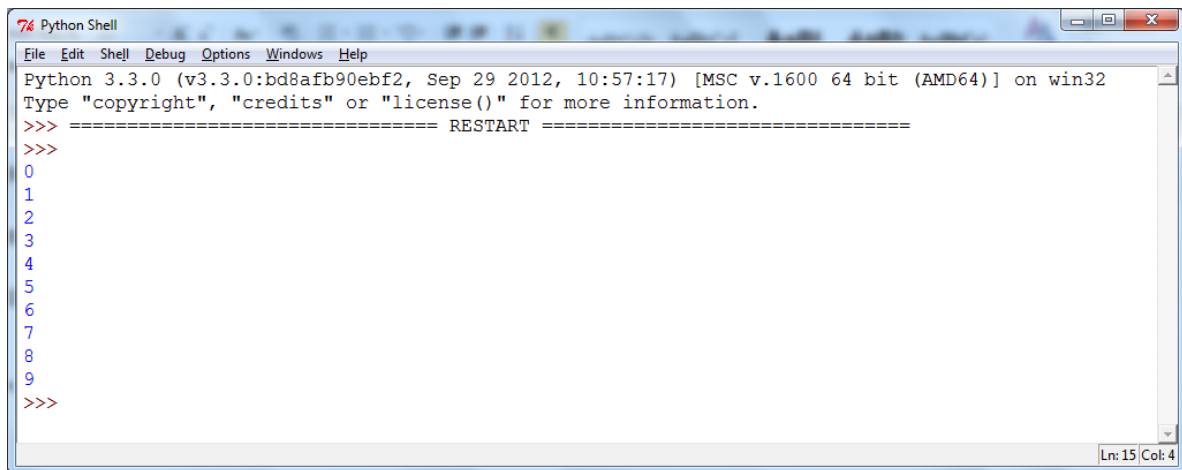


Figura 24: Resultado do Exemplo acima – uso da função `range()`

O valor final não faz parte da sequência gerada; `range(10)` gera 10 valores, uma sequência de comprimento 10. Normalmente inicia em zero (de 0 a 9), mas é possível começar em outro número, ou também especificar um incremento diferente (mesmo negativo), o incremento também é chamado de *step* (passo).

### 29.1 Como funciona a função `range()`:

`range(10) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`: quando coloca-se apenas um argumento, no caso o 10, a contagem começa em zero e vai até o número informado *menos um*, ou seja,  $10 - 1 = 9$  (de 0 a 9). O número final não faz parte da sequência gerada, no caso o 10, porque indica que são 10 números, iniciando em 0 (zero).

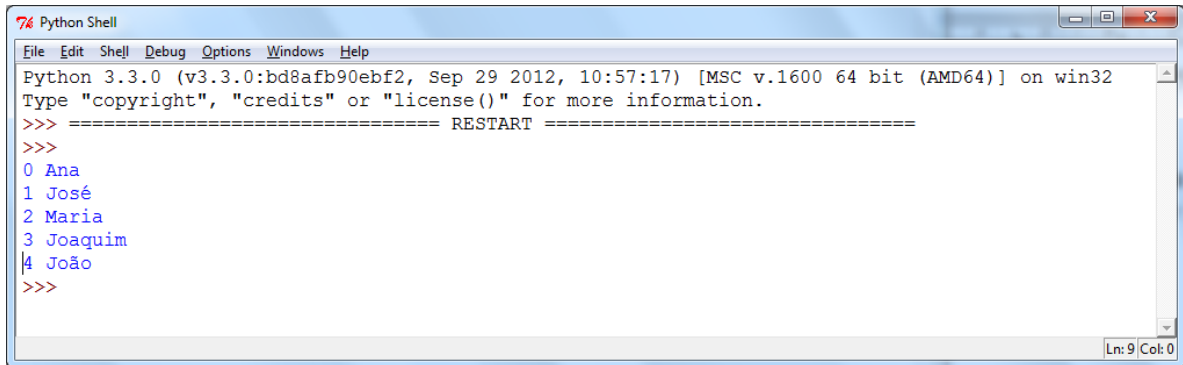
`range(5, 10) = [5, 6, 7, 8, 9]`: também é possível informar o número inicial, no caso inicia em 5 e vai até  $10 - 1$  (de 5 a 9).

`range(1, 20, 3) = [1, 4, 7, 10, 13, 16, 19]`: existe também um terceiro argumento, que indica o “passo” (em inglês *step*), ou seja, neste exemplo, inicia em 1, vai até  $20 - 1$ , pulando de 3 em 3. Também é possível especificar um incremento até mesmo negativo. Exemplo:

`range(-10, -100, -30) = -10, -40, -70`

Para iterar sobre os índices de uma sequência, você pode combinar `range()` e `len()` como segue:

```
a = ['Ana', 'José', 'Maria', 'Joaquim', 'João']
for i in range(len(a)):
    print(i, a[i])
```

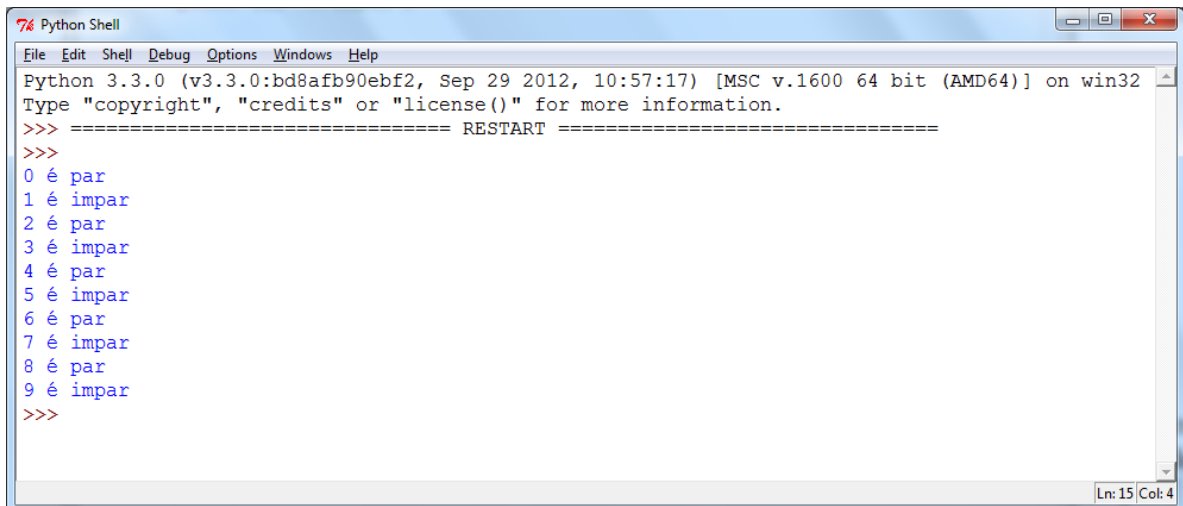


```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
0 Ana
1 José
2 Maria
3 Joaquim
4 João
>>>
```

**Figura 25:** Resultado do Exemplo acima – uso da função `range()`

Mais um exemplo usando a função `range()`:

```
for i in range(10):
    if i%2 == 0:
        print(i, "é par")
    else:
        print(i, "é impar")
```



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
0 é par
1 é impar
2 é par
3 é impar
4 é par
5 é impar
6 é par
7 é impar
8 é par
9 é impar
>>>
```

**Figura 26:** Resultado do Exemplo acima – uso da função `range()`

O operador `%` significa **mod**, e retorna o valor do **resto da divisão**. Nesse exemplo acima, está testando em 10 números (de 0 a 9), quais são pares e quais são ímpares, ou seja, testa se o resto da divisão de um número por 2 for igual a 0, quer dizer que ele é par, senão ele é ímpar.

### **30 Determinação do MAIOR e/ou MENOR valor em um Conjunto de Valores**

---

Em muitos algoritmos surge a necessidade de determinar qual o maior ou o menor valor dentro de um conjunto de valores e, para isto, não existe uma estrutura especial, apenas utiliza-se os conhecimentos já vistos nos capítulos anteriores, como mostrado no exemplo a seguir:

#### **Exemplo 10:** (feito em aula)

Escreva um algoritmo para ler a nota de 10 alunos e escrever a nota mais alta, ou seja, a maior nota entre as 10 notas lidas.

**IMPORTANTE:** Quando sabe-se os **limites** dos valores possíveis, ou seja, por exemplo com as notas sabe-se que os valores serão de 0 a 10, então sabe-se quais são os valores limites (o valor mínimo e o valor máximo), não terá nota menor que 0 e nem nota maior que 10. Nesses casos é mais fácil descobrir o maior ou o menor valor, pois pode-se inicializar a variável Maior, por exemplo, com o valor 0 e a variável Menor com o valor 10 que funcionará perfeitamente. Acontece que, se os valores dos limites não são conhecidos, complica um pouco, pois não sabe-se com que valor inicializar as variáveis para comparação. Então, nesse caso, tem que inicializar tanto a variável Maior quanto a Menor com o “**primeiro valor lido**” e depois ir comparando os próximos valores lidos com o primeiro já lido. Os nomes “Maior” e “Menor”, são apenas exemplos, pode-se denominar as variáveis que serão usadas para os testes como quiser.

**31 Respostas dos Exemplos Práticos desta Apostila**

Neste capítulo são apresentadas as respostas de todos os exemplos encontrados no decorrer desta apostila. As respostas dos Algoritmos são apresentadas em Diagrama de Chapin e/ou Português Estruturado e também os Programas, através da Linguagem Python.

**Exemplo 1: Algoritmo em Chapin**

X ← 20
Y ← 5
Z ← X + Y
Escrever X
Escrever Y
Escrever Z

Escrever X, Y, Z

**Exemplo 1: Programa em Python**

```
ExemploPratico1Apostila.py ...
File Edit Format Run Options Window Help
x = 20
y = 5
z = x + y
print(x)
print(y)
print(z)
Ln: 8 Col: 0
```

**Exemplo 2: Algoritmo em Chapin**

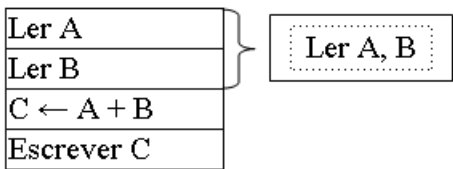
A = 4
B = 3
C = A + B
D = A - B
Escrever A, B, C, D
Escrever "Fim do Algoritmo"

**Exemplo 2: Programa em Python**

```
ExemploPratico2Apostila.py ...
File Edit Format Run Options Window Help
a=4
b=3
c=a+b
d=a-b
print(a,b,c,d)
print("Fim do Algoritmo")
Ln: 7 Col: 0
```

**Exemplo 3:**

Algoritmo Resposta em Chapin



Algoritmo Resposta em Português Estruturado

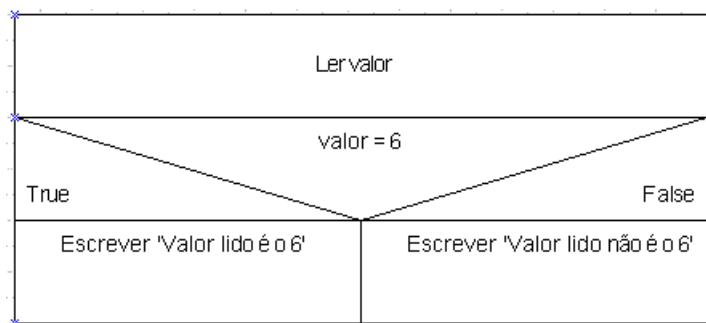
Início  
Ler A, B  
C ← A + B  
Escrever C  
Fim

**Algoritmo Resposta em Fluxograma**

```
graph TD
    Inicio([Início]) --> ReadAB[/A, B/]
    ReadAB --> CalcC[C ← A + B]
    CalcC --> WriteC[/C/]
    WriteC --> Fim([Fim])
```

**Programa Resposta em Python**

```
*ExemploPratico3Apostila.py - C:/Users/flavi/Desktop...
File Edit Format Run Options Window Help
a = int(input("Digite um valor: "))
b = int(input("Digite outro valor: "))
c = a + b
print("Soma: ", c)
Ln: 6 Col: 0
```

**Exemplo 4:**Resposta em ChapinResposta em Português Estruturado

```

Início
  Ler valor
  Se valor = 6 Então
    Escrever 'Valor lido é o 6'
  Senão
    Escrever 'Valor lido não é o 6'
  FimSe
Fim

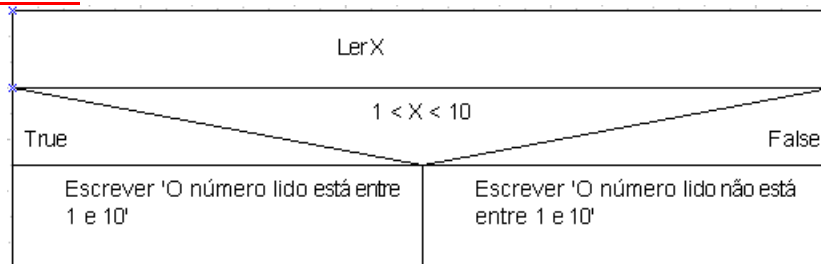
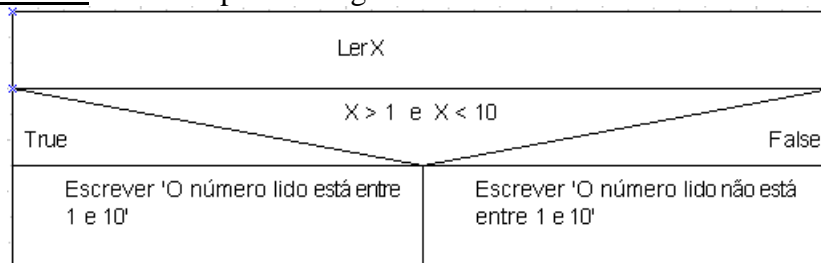
```

Programa em Python

```

ExemploPratico4Apostila.py - C:/Users/flavi/Desktop/S...
File Edit Format Run Options Window Help
valor = int(input("Digite um valor: "))
if valor == 6:
    print("Valor lido é o 6.")
else:
    print("Valor lido não é o 6.")
Ln: 8 Col: 0

```

**Exemplo 5:**Resposta Errada:Resposta Correta: usando operador lógico.Programa em Python

```

ExemploPratico5Apostila.py - C:/Users/flavi/Desktop/SamsungJan2018...
File Edit Format Run Options Window Help
x = int(input("Digite um valor: "))
if x > 1 and x < 10:
    print("Valor lido está entre 1 e 10.")
else:
    print("Valor lido não está entre 1 e 10.")
Ln: 7 Col: 0

```



## Referências

---

Para elaboração e construção desta apostila foram consultados vários tipos de materiais, como por exemplo: livros, outras apostilas, páginas web etc. Algumas das referências consultadas estão apresentadas neste capítulo, mas grande parte do material disponibilizado na apostila, como exemplos e exercícios, foram utilizados das aulas da disciplina de Algoritmos e Programação da faculdade de Análise de Sistemas, da UCPel - Universidade Católica de Pelotas, com o **Prof. Ricardo Andrade Cava**.

- [ALG96] **The ALGOL Programming Language**. Disponível em:  
<http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html>. Acesso em: Jun. 2006.
- [BUF03] BUFFONI, Salete. **Apostila de Algoritmo Estruturado - 4ª edição**. Disponível em:  
<http://www.saletebuffoni.hpg.ig.com.br/algoritmos/Algoritmos.pdf>. Acesso em: Mar. 2004.
- [CAR03] CARBONI, Irenice de Fátima. **Lógica de Programação**. São Paulo, Pioneira Thomson Learning, 2003.
- [CHA70] CHAPIN, Ned. **Flowcharting with the ANSI Standard: A Tutorial**. ACM Computing Surveys, Volume 2, Number 2 (June 1970), pp. 119-146.
- [CHA74] CHAPIN, Ned. **New Format for Flowcharts, Software - Practice and Experience**. Volume 4, Number 4 (October-December 1974), pp. 341-357.
- [CHA02] CHAPIN, Ned. **Maintenance of Information Systems**. Disponível em:  
<http://www.iceis.org/iceis2002/tutorials.htm>. Acesso em: Jun. 2006.
- [COS04] COSTA, Renato. **Apostila de Lógica de Programação - Criação de Algoritmos e Programas**. Disponível em: <http://www.meusite.pro.br/apostilas2.htm>. Acesso em: Jun. 2006.
- [GOM04] GOMES, Abel. **Algoritmos, Fluxogramas e Pseudo-código - Design de Algoritmos**. Disponível em: <http://mail.di.ubi.pt/~programacao/capitulo6.pdf>. Acesso em: Jun. 2006.
- [KOZ06] KOZAK, Dalton V. **Técnicas de Construção de Algoritmos**. Disponível em:  
[http://minerva.ufpel.edu.br/~rossato/ipd/apostila\\_algoritmos.pdf](http://minerva.ufpel.edu.br/~rossato/ipd/apostila_algoritmos.pdf). Acesso em: Jun. 2006.
- [MAR03] MARTINS, Luiz E. G.; ZÍLIO, Valéria M. D. **Apostila da Disciplina Introdução à Programação**. Disponível em: <http://www.unimep.br/~vmdzilio/apostila00.doc>. Acesso em: Jun. 2006.
- [MEN00] MENEZES, Nilo N. C. **Introdução à Programação com Python: Algoritmos e Lógica de Programação para Iniciantes**. São Paulo, Editora Novatec, 2010.
- [NAS73] NASSI, Ike; SHNEIDERMAN, Ben. **Flowchart Techniques for Structured Programming**. ACM SIGPLAN Notices, Volume 8, Number 8 (August 1973), pp.12-26.
- [NAS04] NASSI, Ike. **Ike Nassi's Home Page**. Disponível em: <http://www.nassi.com/ike.htm>. Acesso em: Jun. 2006.
- [ORT01] ORTH, Afonso Inácio. **Algoritmos e Programação com Resumo das Linguagens Pascal e C**. Porto Alegre, Editora AIO, 2001.
- [PRO04] **Programming Languages**. Disponível em:  
[http://www.famed.ufrgs.br/disciplinas/inf\\_med/prog\\_ling.htm](http://www.famed.ufrgs.br/disciplinas/inf_med/prog_ling.htm). Acesso em: Mar.2004.

- [SAN04] SANTANA, João. **Algoritmos & Programação**. Disponível em:  
<http://www.iesam.com.br/paginas/cursos/ec/1ano/aulas/08/joao/APunidade-1.pdf>. Acesso em: Mar. 2004.
- [SOU11] SOUZA, Marco Antonio Furlan de [et al.]. **Algoritmos e Lógica de Programação** - 2ª edição revista e ampliada. São Paulo, Cengage Learning, 2011.
- [SHN03] SHNEIDERMAN, Ben. **A short history of structured flowcharts (Nassi-Shneiderman Diagrams)**. Disponível em: <http://www.cs.umd.edu/~ben/> Acesso em: Jun. 2006.
- [TON04] TONET, Bruno; KOLIVER, Cristian. **Introdução aos Algoritmos**. Disponível em:  
<http://dein.ucs.br/napro/Algoritmo/manuais/Manual 20Visualg.pdf>. Acesso em: Mar. 2004.
- [YOU04] YOURDON, Ed. **Ed Yourdon's Web Site**. Disponível em:  
<http://www.yourdon.com/books/msa2e/CH15/CH15.html>. Acesso em: Mar. 2004.