

# Algoritmos e Lógica de Programação com Python



```
def menu(principal):
    if principal == 1:
        print("1 - Cadastrar Usuário")
        print("2 - Consultar Usuário")
        print("3 - Alterar Usuário")
        print("4 - Excluir Usuário")
        print("5 - Sair do Sistema")
    else:
        print("1 - Cadastrar Usuário")
        print("2 - Consultar Usuário")
        print("3 - Alterar Usuário")
        print("4 - Excluir Usuário")
        print("5 - Sair do Sistema")
    op = int(input("Digite a opção desejada: "))
    return op

def cadastrar_usuario():
    nome = input("Digite o nome: ")
    email = input("Digite o e-mail: ")
    senha = input("Digite a senha: ")
    cursor.execute("INSERT INTO usuarios (nome, email, senha) VALUES (?, ?, ?)", (nome, email, senha))
    connection.commit()
    print("Usuário cadastrado com sucesso!")

def consultar_usuario():
    id = int(input("Digite o ID do usuário: "))
    cursor.execute("SELECT * FROM usuarios WHERE id = ?", (id,))
    resultado = cursor.fetchone()
    if resultado:
        print(f"ID: {resultado[0]}, Nome: {resultado[1]}, E-mail: {resultado[2]}, Senha: {resultado[3]}")
    else:
        print("Usuário não encontrado!")

def alterar_usuario():
    id = int(input("Digite o ID do usuário que deseja alterar: "))
    nome = input("Digite o novo nome: ")
    email = input("Digite o novo e-mail: ")
    senha = input("Digite a nova senha: ")
    cursor.execute("UPDATE usuarios SET nome = ?, email = ?, senha = ? WHERE id = ?", (nome, email, senha, id))
    connection.commit()
    print("Usuário alterado com sucesso!")

def excluir_usuario():
    id = int(input("Digite o ID do usuário que deseja excluir: "))
    cursor.execute("DELETE FROM usuarios WHERE id = ?", (id,))
    connection.commit()
    print("Usuário excluído com sucesso!")

def sobre():
    print("Este é um aplicativo de gerenciamento de usuários em Python.")

def main():
    while True:
        op = menu(1)
        if op == 1:
            cadastrar_usuario()
        elif op == 2:
            consultar_usuario()
        elif op == 3:
            alterar_usuario()
        elif op == 4:
            excluir_usuario()
        elif op == 5:
            sobre()
            break
        else:
            print("Opção inválida! Tente novamente.")

if __name__ == "__main__":
    main()
```

Um guia básico para o programador iniciante

Bruno Luvizotto Carli

© 2017 by Bruno Luvizotto Carli



**Esta obra é licenciada sob a Licença Creative Commons Atribuição–Não Comercial 2.5 Brasil. Você pode compartilhar e adaptar o conteúdo deste material desde que os devidos créditos sejam dados ao autor do trabalho.**

Para ver uma cópia desta licença, visite  
<http://creativecommons.org/licenses/by-ncsa/2.5/br/> ou  
envie uma carta para Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

---

**Índices para catálogo sistemático:**

- 1. Computadores: Programação;**
  - 2. Programação de computadores;**
  - 3. Algoritmos;**
  - 4. Linguagem de Programação;**
-

## **1° Edição**

---

**CAPA: Bruno L. Carli.**

**Adaptado de: <https://pixabay.com/pt/animal-a-fotografia-animal-close-up-1853944/>**

---

**Contato: brunolcarli@gmail.com**

**BRUNO LUVIZOTTO CARLI**

# **Algoritmos e lógica de programação com Python**

**1° Edição**

**Curitiba, PR  
Edição do Autor  
2017**

*Para meu pai, que me disse  
que computação era besteira...*

# Sumário

APRESENTAÇÃO.....	8
Capítulo 1.....	10
CONCEITO DE ALGORITMO.....	10
ALGORITMOS PARA A LOGICA DE PROGRAMAÇÃO.....	13
TIPOS DE ALGORITMO .....	14
PYTHON.....	21
Capítulo 2.....	28
CONSTRUINDO ALGORITMOS COM PYTHON .....	28
VARIAVEIS E TIPOS DE DADOS.....	32
VARIÁVEIS EM PYTHON.....	38
CONSTANTES.....	50
EXPRESSÕES.....	50
FUNÇÕES INTRINSECAS.....	56
ENTRADA DE DADOS.....	59
EXERCICIOS ELABORADOS.....	61
Capítulo 3.....	69
COMENTÁRIOS.....	69
ESTRUTURA CONDICIONAL.....	70
ESTRUTURA CONDICIONAL SIMPLES.....	71
ESTRUTURA CONDICIONAL COMPOSTA.....	74
ESTRUTURA CONDICIONAL ANINHADA.....	84
EXERCICIOS ELABORADOS.....	86
Capítulo 4.....	92
ESTRUTURAS DE REPETIÇÃO.....	92
ESTRUTUTURA FOR .....	93
ESTRUTUTURA WHILE.....	99
ESTRUTUTURAS DE REPETIÇÃO ANINHADAS.....	104
EXERCICIOS ELABORADOS.....	106
Capítulo 5.....	115
ESTRUTURAS DE DADOS.....	115
ESTRUTURAS DE DADOS UNIDIMENSIONAIS.....	116
ESTRUTURAS DE DADOS MULTIDIMENSIONAIS .....	128
TUPLAS.....	137

DICIONÁRIOS.....	139
EXERCÍCIOS ELABORADOS.....	149
Capítulo 6.....	155
SUB-ROTINAS E PROGRAMAÇÃO COM ARQUIVOS.....	155
SUB-ROTINAS.....	155
PROCEDIMENTOS.....	156
FUNÇÕES.....	162
PARÂMETROS.....	165
ESCOPO DAS VARIÁVEIS.....	168
RECURSIVIDADE.....	171
ARQUIVOS.....	172
BIBLIOTECAS E MÓDULOS.....	190
REFERÊNCIAS .....	198

## **APRESENTAÇÃO**

Olá amigo leitor, nesta obra pretendo elucidar conceitos básicos, porém fundamentais ao aprendizado da lógica de programação e o entendimento acerca da elaboração de algoritmos. Você será inicialmente apresentado à alguns exemplos de tipos de algoritmos e imerso na conceituação implícita do que é um algoritmo. Vamos ser apresentados à poderosa linguagem de programação Python, nossa principal ferramenta para implementação de algoritmos no decorrer deste livro.

Saliento desde já que esta obra não pretende ser mais um livro para o ensino da linguagem de programação Python, pretendo com este manuscrito apresentar os conceitos básicos e fundamentais dos algoritmos e lógica de programação como os tipos de dados, características de um algoritmo, estruturas de dados laços de repetição sub-rotinas e manipulação de arquivos permanentes, ao mesmo tempo em que possa induzir você leitor a conhecer e utilizar o Python para implementação dos nossos programas, e se você caro leitor se identificar positivamente com o Python, insisto que busque na farta

bibliografia literária disponível abordando os mais diversos conceitos técnicos sobre Python.

Este livro objetiva orientar e colaborar no estudo dos algoritmos e lógica de programação de iniciantes no estudo da computação, *hobbyistas* iniciantes e alunos de cursos técnicos e graduações que estejam começando nesta magnifica área que é a programação de computadores. Ressalto que este não é um livro orientado ao ensino de Python e espera-se de você leitor que compreenda os fundamentos da lógica de programação podendo facilmente adaptar os exemplos demonstrado nesta obra em outras linguagens de programação sem muita dificuldade.

- O autor.

## CAPITULO 1

---

...

### CONCEITO DE ALGORITMO

Quando falamos em algoritmo é comum pensar em algo relacionado à matemática, o que de certa forma não está errado, pois quando pequenos na escola primária éramos incitados a resolver diversas contas utilizando algoritmos matemáticos necessários à resolução dos exercícios. Puga e Rissetti (2010, p. 9) comentam que “A matemática clássica é, em grande parte o estudo de determinados algoritmos”, isso pode a uma primeira vista assustar iniciantes no estudo da lógica de programação, mas vale salientar que não necessariamente o estudo de algoritmos envolve o estudo da matemática.

Para Puga e Rissetti (Op cit.) um algoritmo nada mais é que “uma sequência lógica de instruções que devem ser seguidas para a resolução de um problema ou execução de uma tarefa”. Ascencio e Campos (2010, p.2) vão ainda mais profundamente na conceituação de algoritmos citando diversas fontes definindo o que são algoritmos, dentre algumas pode-se citar: “sequência de passos que visa atingir um objetivo bem

definido” (FORBELLONE, 1999 apud ASCENCIO; CAMPOS, 2010), “sequência finita de instruções ou operações cuja execução, em tempo finito, resolve um problema computacional, qualquer que seja sua instância” (SALVETTI, 1999 apud ASCENCIO & CAMPOS, 2010).

Etimologicamente, de acordo com Abbagnano (2007, p. 27), a palavra Algoritmo deriva do nome de Mohammed al-Khuwarizmi que foi um astrólogo e matemático árabe do século IX, responsável por introduzir o sistema de numeração indiano no Ocidente, esta notação durante a Idade Média tornou-se conhecida como algorísmos, sendo conhecida hoje pelo sistema numérico decimal que utilizamos no nosso dia a dia.

Para finalizar a conceituação de algoritmo de uma forma, simples e entendível, um algoritmo é um passo a passo ou ainda, uma sequência de instruções para se chegar a um determinado objetivo, simples assim. Para exemplificar vamos demonstrar um algoritmo de uma tarefa corriqueira que todos fazemos diariamente: Beber água.

*Passo a passo para beber água:*

1. Pegar um copo ou recipiente;

2. Colocá-lo sob a torneira ou filtro de água;
3. Abrir a torneira/filtro;
4. Quando o copo estiver suficientemente cheio, fechar a torneira/filtro;
5. Beber a água do copo/recipiente;

Simples, não é? Pode-se perceber que não temos nada de muito matemático nisso, é um simples passo a passo, ou ainda, uma sequência finita de instruções que são seguidas para realizar uma tarefa (beber água). Vale-se ressaltar que não existe uma única forma de beber água, poderia-se beber água da fonte ou de um rio utilizando as mãos ou outros procedimentos, e isso é válido para todos os algoritmos, pois podem haver diferentes formas de realizar uma mesma tarefa desde que respeitem a lógica do processo, a exemplo disso pode-se dizer que não seria coerente fechar a torneira para depois colocar o copo abaixo da torneira para encher. Vamos demonstrar um outro exemplo de algoritmo: Uma ligação telefônica.

*Passo a passo para realizar uma ligação:*

1. Pegar o telefone;

2. Inserir o número para quem deseja-se telefonar;
3. Aguardar ser atendido;
4. Depois de realizar a conversação, colocar o telefone no gancho (ou apertar o botão de desligar no caso de alguns telefones);

## **ALGORITMOS PARA A LOGICA DE PROGRAMAÇÃO**

Certo, já sabemos o que são algoritmos e de onde essa palavra surgiu. Mas e a relação entre os algoritmos e a programação de computadores?

Os computadores não compreendem instruções humanas como estas citadas anteriormente, os computadores só conhecem um tipo de linguagem: binária (zeros e uns) ou como define Tanenbaum (2010, p. 1) **linguagem de máquina**, mas se acalme, não há necessidade de se tornar um expert em linguagem binária para programar um computador, para isso existem as **linguagens de alto nível** ou **linguagens de programação**, que de acordo com Pressman (2010, p. 677) “são veículos de comunicação entre os seres humanos e os computadores”, as quais permitem que o programador possa elaborar um roteiro de instruções passo a passo (algoritmo)

para que o computador execute uma determinada tarefa, estas linguagens de alto nível ou popularmente conhecidas como linguagens de programação, das quais pode-se citar o C, C++, Java, Cobol, Fortran, Python, dentre muitas outras linguagens existentes. Cada linguagem possui sua própria estrutura sintática porém sua essência reside na coerência lógica em que as instruções são fornecidas ao computador, sendo assim, o algoritmo escrito em uma linguagem de programação recebe o nome de **programa** (TANENBAUM, 2010).

## **TIPOS DE ALGORITMO**

Os algoritmos podem ser descritos em diferentes formas, Ascencio e Campos (2010, p.3-4) apresentam a Descrição Narrativa, o Fluxograma e o Pseudocódigo. Cada um desses será analisado nesta seção:

### *Descrição Narrativa*

A descrição narrativa já foi apresentada no início deste capítulo, nos exemplos de beber água e no exemplo de realizar uma ligação telefônica. Consiste em descrever em linguagem natural a sequência de eventos que se sucederão. Vamos

exemplificar com mais um algoritmo em descrição narrativa:  
Somar dois números:

1. Definir o primeiro número;
2. Definir o segundo número;
3. Realizar a operação de soma entre os dois números;
4. Obter e demonstrar o resultado;

Esse tipo de algoritmo nos lembra muito de receitas de bolo por exemplo, porém possui uma grande desvantagem, por se tratar de uma linguagem natural, suas instruções podem ser ambíguas abrindo espaço para diferentes interpretações, dificultando sua transcrição em programa(ASCENCIO; CAMPOS, 2010), fazendo com que este modelo de algoritmo não seja muito utilizado em programação.

### *Fluxograma*

O fluxograma é uma forma de representar a sequência de instruções através de figuras geométricas, sendo elas:

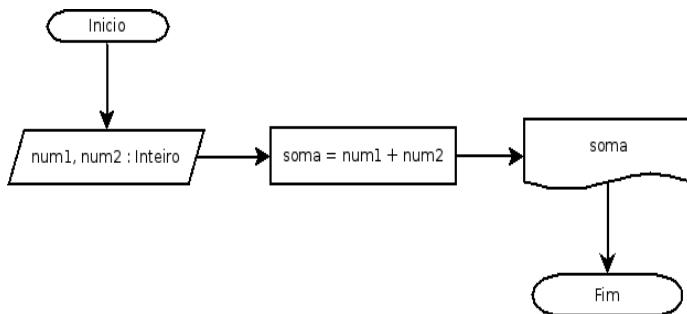
Tabela 1 – Símbolos utilizados no fluxograma e seus significados

	Símbolo utilizado para indicar o início e o fim do algoritmo.
	Permite indicar o sentido do fluxo de dados. Serve exclusivamente para conectar os símbolos ou blocos existentes.
	Símbolo utilizado para realizar cálculos e atribuições de valores.
	Símbolo utilizado para representar a entrada de dados.
	Símbolo utilizado para representar a saída de dados.
	Símbolo utilizado para indicar que deve ser tomada uma decisão, apontando a possibilidade de desvio.

Fonte: Ascencio & Campos (2010, p. 4)

Vamos exemplificar a utilização do fluxograma em um algoritmo cujo objetivo é realizar a soma de dois números, o mesmo algoritmo apresentado anteriormente:

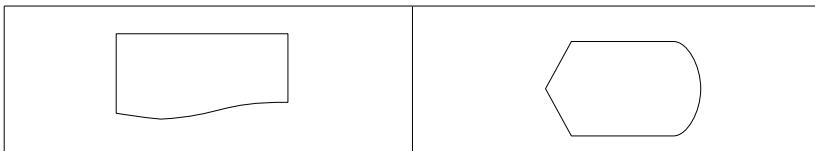
Figura 1. Fluxograma representando um algoritmo para soma de dois números



Fonte: O autor

As desvantagens do fluxograma segundo Ascencio e Campos(2010) são a necessidade de aprender os significados dos símbolos do fluxograma, que muitas vezes podem diferir, como por exemplo podemos ver na Figura 2 duas representações diferentes para o símbolo de saída de dados. Outra desvantagem apresentada é que o fluxograma não apresenta muitos detalhes, o que também dificulta a implementação deste algoritmo em um programa de computador.

Figura 2: Representações para a Saída de Dados



Fonte: Figura do lado esquerdo, Ascencio & Campos (2010). Figura do lado direito, Puga & Rissetti (2010).

### *Pseudocódigo*

O Pseudocódigo, também conhecido como Portugol (ASCENCIO; CAMPOS, 2010) ou Português Estruturado (PUGA; RISSETTI, 2010) é um modelo de algoritmo muito semelhante a um código escrito em linguagem de programação de alto nível, tornando sua implementação em um programa uma tarefa muito simples para o programador. Para Puga e Rissetti (2010, p. 11) este é uma representação de algoritmo em uma linguagem intermediária entre linguagem de programação e a linguagem natural humana, mais especificamente, o nosso Português. De acordo com Puga & Rissetti (Op. Cit.) o nome “Pseudocódigo” significa “falso código”, justamente por ser extremamente parecido com um algoritmo implementado em linguagem de alto nível. Vamos ver o mesmo exemplo anterior (soma de dois números) representado em Pseudocódigo:

**ALGORITMO:** soma

**VAR**

    num1, num2, soma : **Inteiro**

**INICIO**

**LEIA**(num1)

**LEIA**(num2)

        soma ← num1 + num2

**ESCREVA**(soma)

**FIM.**

Este exemplo pode ser facilmente transcrito em qualquer linguagem de programação e executado por um computador. Vamos analisar a estrutura do pseudocódigo:

Tabela 2 – Estrutura do pseudocódigo

<b>ALGORITMO:</b> soma	Esta primeira seção declara o nome do algoritmo
<b>VAR</b> num1, num2, soma : inteiro	Esta seção é chamada de declaração de variáveis, onde todas as variáveis que serão utilizadas no algoritmo devem ser fornecidas, neste caso precisamos de três delas, o primeiro número, o segundo número e o resultado da soma. Todas elas são do tipo inteiro então declaramos uma após a outra, separando-as por vírgulas. Veremos mais sobre variáveis mais adiante.
<b>INICIO</b>  <b>LEIA</b> (num1)  <b>LEIA</b> (num2)	Esta seção é o corpo principal do algoritmo, onde todas as instruções serão declaradas. Neste caso temos duas instruções, <b>LEIA</b> ( ) e <b>ESCREVA</b> ( ). A instrução <b>LEIA</b> ( ) é um comando de

<pre>soma ← num1 + num2 ESCREVA(soma) <b>FIM.</b></pre>	<p>entrada de dados, o que significa que um valor será fornecido para a variável num1 e num2.</p> <p>Então temos a atribuição do resultado para a variável soma, esta atribuição é representada pelo simbolo “←”.</p> <p>O comando ESCRVA( ) representa uma saída de dados, que neste caso demonstra a exibição do valor guardado na variável “soma”.</p> <p>FIM. Indica que chegamos ao fim do algoritmo.</p>
---	--

Fonte: O autor.

Algumas desvantagens do pseudocódigo apresentadas por Ascencio e Campos (2010) é que deve-se aprender as regras do pseudocódigo, da mesma forma que se aprenderia os significados dos símbolos do fluxograma ou as regras sintáticas de uma linguagem de programação, além do mais não é possível implementar programas reais em pseudocódigo, por estes motivo vamos conceber a linguagem de programação Python como forma de implementação e aprendizado dos nossos algoritmos por sua facilidade de compreensão, ao final deste livro você não somente saberá os princípios básicos da construção e interpretação de algoritmos assim como terá os conhecimentos básicos de uma poderosa linguagem de programação e estará escrevendo seus próprios programas de computador.

## PYTHON

Python é uma linguagem extremamente eficiente: seus programas farão mais com menos linhas de código, se comparado ao que muitas outras linguagens exigiriam. A sintaxe de Python também ajudará você a escrever um código “limpo”. Seu código será fácil de ler, fácil de depurar, fácil de estender e de expandir, quando comparados com outras linguagens. (MATTHES, 2016. p. 28)

Para a metodologia deste livro, pretende-se utilizar a Linguagem Python como forma de aprendizado de algoritmos, visto que as formas mais populares de algoritmos apresentam desvantagens relacionadas ao aprendizado das regras próprias de cada modelo, utilizaremos a Linguagem de Programação Python por sua alta legibilidade, o que quer dizer que um programa escrito nesta linguagem é facilmente compreendido, sem haver muito esforço da parte do programador em entender seus comandos, além do mais, um algoritmo implementado em Python é facilmente executado em qualquer sistema operacional com um interpretador Python instalado. Isto possibilitará aos iniciantes na lógica de programação não somente a compreender a implementação dos algoritmos mas também a familiarizar-se com as regras de uma simples, porém extremamente poderosa linguagem de programação de propósito geral, permitindo que escreva programas eficientes com poucas linhas de comando em qualquer ambiente

operacional. Finalizando esta justificativa, é fundamental no aprendizado de algoritmos que se pratique muito, e nada melhor do que ver resultados instantâneos através de atividades práticas, logo o Python se demonstrará uma excelente ferramenta para se iniciar no mundo da programação, então vamos colocar a mão na massa.

Primeiramente será necessário a instalação de um Interpretador Python no seu computador, o download pode ser feito pelo website oficial do Python: <<https://www.python.org/downloads/>>.

O Python é disponibilizado em duas versões, Python 2.x e Python 3.x, não tendo grandes diferenças entre si, porém para fins de praticidade recomenda-se que o leitor utilize a mesma versão apresentada no livro, neste caso utilizaremos a versão Python 3.x, mais especificamente 3.5.2. Qualquer versão do Python 3.x poderá ser utilizada para executar os algoritmos deste livro.

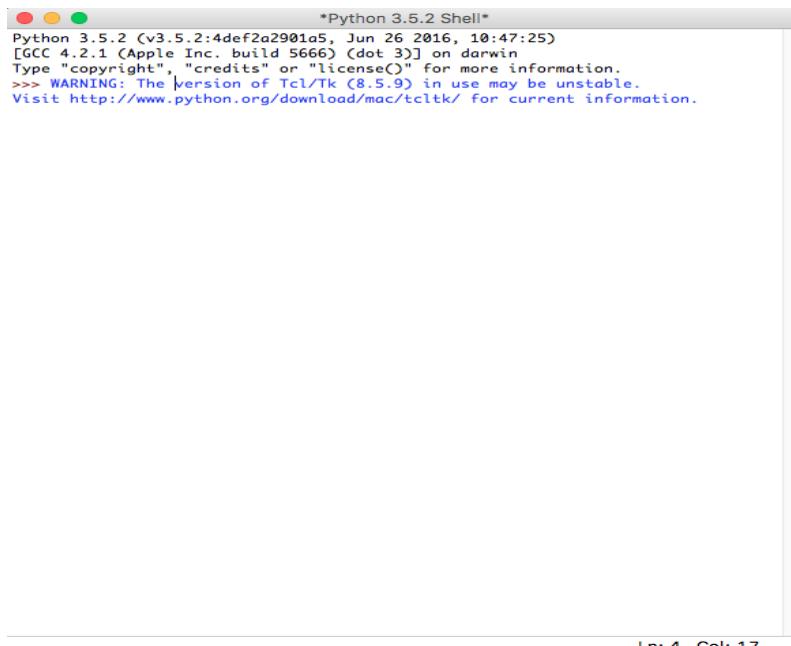
Após fazer o download e instalação do Python 3 para o seu Sistema Operacional você notará que um editor chamado “IDLE” foi instalado juntamente com sua versão do Python.

Quando você instala o Python 3, também terá o IDLE, o ambiente simples – mas surpreendentemente útil – de desenvolvimento integrado do Python. O IDLE inclui um editor de destaque da sintaxe, um depurador, o Python Shell e uma

cópia completa do conjunto de documentação online do Python 3. (BARRY, 2015)

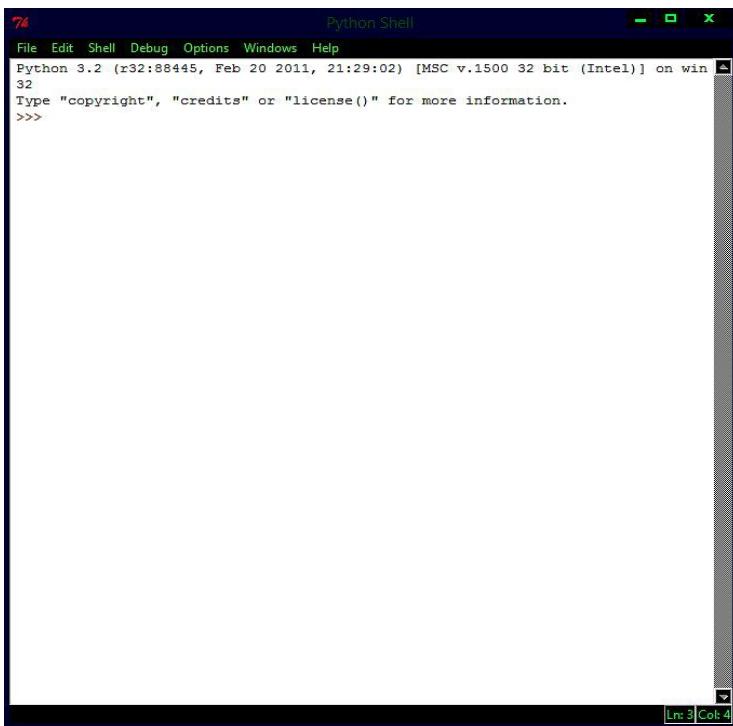
Este é um editor padrão do Python porém você também poderá escrever os programas em um editor de sua preferência, para este livro, por motivos didáticos será utilizado o próprio IDLE.

Figura 3 – Aparência do IDLE no Mac Os.



Fonte: O autor.

Figura – 4 – Aparência do IDLE no Windows.

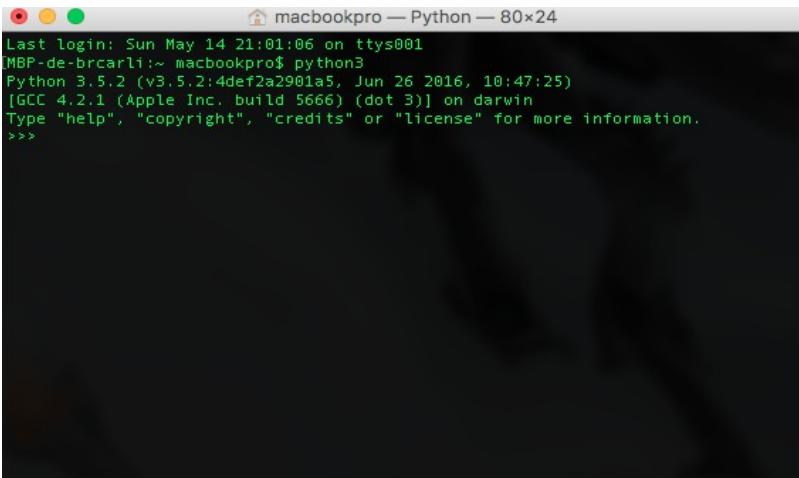


Fonte: O autor.

As Figuras 3 e 4 apresentam a aparência do IDLE no Sistema Operacional Mac Os e Windows. “Quando você iniciar o IDLE pela primeira vez, será apresentado ao *prompt* com 'três divisas' (>>>) no qual você insere o código e imediatamente a executa para você, exibindo qualquer resultado produzido na

tela” (BARRY, 2015. p. 4). Outro modo de chamar o Python é abrir seu Terminal (no Windows é muito conhecido como *prompt* ou *cmd*) e escrever o comando **python** e se as três divisas aparecerem significa que o Python está corretamente instalado no seu computador e você está pronto para escrever programas de verdade.

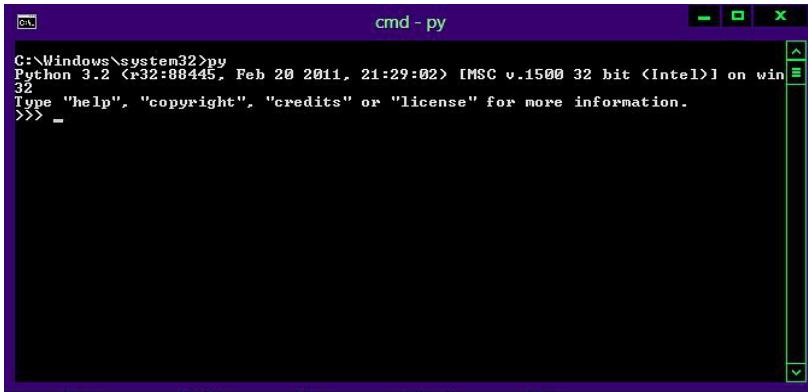
Figura 5 – Chamar o Python pelo terminal no Mac Os



```
Last login: Sun May 14 21:01:06 on ttys001
MBP-de-brcarli:~ macbookpro$ python3
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Fonte: O autor

Figura 6 – Chamando Python no terminal Windows



A screenshot of a Windows Command Prompt window titled "cmd - py". The window shows the Python 3.2 interpreter starting. The text in the window reads:

```
C:\Windows\system32>py
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Fonte: O autor.

Uma observação importante é que, caso sua máquina já possua outras versões do Python instalada (como no MacOs e Linux onde o Python 2 já vem instalado por padrão) o comando necessário para chamar o Python pelo Terminal pode vir a ser diferente, dentre alguns exemplos estão **python**, **python3**, **py**, **python3.3**, **python3.5** (ou a versão específica que você instalou). Agora que você já sabe o que são algoritmos e já tem o Python 3 instalado no seu computador podemos dar continuidade ao nosso estudo da Lógica de Programação.

Os algoritmos em Python apresentados nesta obra poderão ser encontrados através deste link:  
<https://github.com/brunolcarli/AlgoritmosELogicaDeProgramacaoComPython>

## CAPÍTULO 2

---

...

### CONSTRUINDO ALGORITMOS COM PYTHON

Antes de efetivamente construir nossos algoritmos é importante sabermos como elaborar um algoritmo. Ascencio e Campos (2010, p. 3) definem um método para construção de algoritmos, sendo estruturado a partir de alguns elementos básicos, dentre eles:

- a) Compreender o problema a ser resolvido;
- b) Definir os dados de entrada que deverão ser fornecidos;
- c) Definir o processamento, quais operações deverão ser executadas sobre os dados;
- d) Definir a saída final, exibindo o resultado obtido através do processamento dos dados;
- e) Construir o algoritmo;
- f) Testar o algoritmo através de simulações;

Vamos realizar cada uma destas etapas e comparar o último exemplo de algoritmo fornecido (soma de dois números) em Pseudocódigo e Python para ter uma boa visão de como seria um algoritmo implementado em Python 3:

- a) **Problema a ser resolvido:** Somar dois números;
- b) **Dados de entrada:** numero1 e numero2;
- c) **Processamento:** Somar o numero1 com o numero2;
- d) **Saída:** Exibir o resultado da soma;
- e) **Construção do algoritmo:**

Pseudocódigo	Python 3
<b>ALGORITMO:</b> soma <b>VAR</b> num1, num2, soma : <b>inteiro</b> <b>INICIO</b> LEIA(num1) LEIA(num2) soma ← num1 + num2 <b>ESCREVA(soma)</b> <b>FIM.</b>	num1 = <b>input</b> ("Insira o primeiro numero ") num2 = <b>input</b> ("Insira o segundo numero ") soma = <b>int</b> (num1) + <b>int</b> (num2) <b>print</b> ("O resultado da soma é ", soma)

Para testar o algoritmo abra seu IDLE e selecione na parte superior a opção *File* então *New File*, uma nova janela se abrirá para que você possa escrever o algoritmo, escreva o algoritmo Python apresentado (recomenda-se fortemente que você escreva os comandos ao invés de copiar e colar) então selecione a opção *Run* e então *Run Module* na caixa de ferramentas na parte superior da tela, ao executar o algoritmo

no IDLE será solicitado que você salve seu programa, então selecione um diretório de sua preferência e salve o arquivo com a extensão `.py` (uma sugestão é que salve este primeiro algoritmo como `soma.py`). Após salvar o arquivo, o Python irá executar seu algoritmo e se tudo ocorrer bem sua saída deve ser parecida com a da Figura 5:

Figura 7 – Saída do algoritmo `soma.py`

```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/macbookpro/Documents/Untitled.py =====
Insira o primeiro numero 2
Insira o segundo numero 3
0 resultado e 5
>>>
```

Fonte: O autor

Agora vamos entender os comandos utilizados nesse algoritmo Python, primeiro criamos uma variável `num1` (vamos ver mais sobre variáveis adiante) e atribuímos a ela o valor de entrada recebido em `input()`. O comando `input()` em Python 3 permite que o usuário entre com um valor que será tratado como *string* (veremos mais sobre tipos de dados adiante), este comando também permite que um texto seja passado como argumento e exibido na tela, neste caso passamos o texto

“Insira o primeiro numero ” como argumento. Quando o interpretador Python ler esta instrução ele vai primeiro exibir o texto ao usuário e ficará aguardando uma entrada, logo que o usuário do sistema fornecer um valor de entrada e confirmar, o interpretador executará a próxima instrução do algoritmo, que será, neste caso, a atribuição de mais um valor de entrada para a variável num2, da mesma forma como fizemos na primeira instrução.

A terceira instrução do algoritmo *soma.py* irá realizar a operação de processamento que soma num1 e num2 e armazena seu resultado em uma variável chamada soma, para isto utilizamos um método chamado **int( )** que irá dizer ao interpretador Python que durante este processamento as variáveis num1 e num2 devem ser tratadas como números inteiros<sup>1</sup>.

Por fim a instrução **print( )** diz ao Python para escrever na tela, este é um comando de saída de dados, e exibirá na tela os argumentos que forem fornecido à instrução **print( )**. Neste

---

<sup>1</sup> Uma observação importante é que, em Python 3 o comando `input()` vai receber sempre os dados em formato de cadeia de caracteres (string), sendo tratada como um texto, por este motivo a necessidade de converter para números os valores inseridos através da instrução `int()`, em Python 2 o comando `input()` receberia apenas valores numéricos e caso o usuário entrasse com uma letra ou símbolo o Python retornaria um erro. Em python 2 há um comando específico para receber entradas de texto chamado `raw_input()`. Não se preocupe com isso por enquanto, mas para frente quando você estiver mais familiarizado com os tipos de dados isto se tornará mais simples de compreender.

caso fornecemos um argumento em forma de texto dizendo “O resultado da soma é ” e adicionamos uma vírgula (,) para separar o próximo argumento que foi a variável soma, desta forma o comando de saída exibirá o texto fornecido e o valor guardado na variável soma.

## **VARIÁVEIS E TIPOS DE DADOS**

Variáveis em Algoritmos e Lógica de Programação são valores que podem sofrer alterações no decorrer do algoritmo. Ascencio e Campos (2010, p. 7) afirma que “Um algoritmo e, posteriormente, um programa, recebem dados, que precisam ser armazenados no computador para serem posteriormente utilizados no processamento. Esse armazenamento é feito na memória”, logo, sempre que declaramos uma variável em nosso algoritmo o computador irá separar um espaço na memória para que um dado possa ser armazenado ali. Segundo Ascencio e Campos(Op. Cit) uma variável “possui nome e tipo, e seu conteúdo pode variar ao longo do tempo, durante a execução de um programa. Embora uma variável possa assumir diferentes valores, ela só pode armazenar um valor a cada instante”.

Agora vamos fazer uma pequena abstração, imagine um garoto chamado Pedrinho. Pedrinho possui uma variável chamada idade que neste exato momento possui o valor 7, representando que Pedrinho tem 7 anos. No seu próximo aniversário Pedrinho completará mais um ano de vida então o valor da sua variável idade será incrementado em um, passando agora a representar o valor 8. Esta pequena analogia demonstra a mutabilidade dos valores guardados em uma variável, que recebe este nome justamente por seus valores variarem ao longo de um algoritmo.

As variáveis sempre guardam valores de um respectivo **tipo de dado**, que de acordo com Ascencio e Campos (2010, p. 8) os mais comuns são numéricos, lógicos e literais.

### *Numéricos*

Os dados do tipo numérico são divididos em duas categorias: **Inteiros** e **Reais**. “Os números inteiros podem ser positivos ou negativos e não possuem parte fracionária”(ASCENCIO; CAMPOS, 2010), como exemplo de números inteiros temos:

12

-7

154

0

-98

Os números Reais são aqueles que possuem uma parte fracionária e podem ser positivos ou negativos, como por exemplo:

3.14

2.9876

-1.98

0.765

Uma observação importante deixada por Ascencio e Campos (2010) é que “os números reais seguem a notação da língua inglesa, ou seja, a parte decimal é separada da parte inteira por um . (ponto) e não por uma , (vírgula)” é importante compreender isso desde o início pois poderá evitar complicações futuras quando implementar seu algoritmo.

### *Lógicos*

Os valores lógicos, também chamados de booleanos (ASCENCIO; CAMPOS, 2010), somente podem assumir dois valores: **verdadeiro** ou **falso**. Estes são utilizados muitas vezes

em comparações lógicas e verificações condicionais (veremos mais sobre isto nos próximos capítulos).

### *Literais*

Os dados do tipo literal são formados por sequências de caracteres (letras maiúsculas, minúsculas e símbolos) ou por um único caractere(ASCENCIO; CAMPOS, 2010). Este tipo de dado é popularmente chamado de string, sendo representado pelo texto envolto por “aspas”, como no exemplo a seguir:

“aluno”

“Carro”

“Minha casa é azul”

“fulano@email.com”

“12 x 10 =”

“F\$0c!3^y

Perceba que mesmo os numerais que estiverem entre aspas serão identificados como dados literais, e Puga e Rissetti (2010, p. 37) afirmam que “Os números armazenados em uma variável cujo tipo de dado é literal não poderão ser utilizadas para cálculo”, somente sendo possível realizar operações matemáticas com as variáveis do tipo numérico, desta forma é importante conhecer os tipos de dados que estamos trabalhando.

Definir o tipo de dado mais adequado para ser armazenado em uma variável é uma questão de grande importância para garantir a resolução do problema. Ao desenvolver um algoritmo, é necessário que se tenha conhecimento prévio do tipo de informação (dado) que será utilizado para resolver o problema proposto. Daí, escolhe-se o tipo adequado para a variável que representa esse valor. (PUGA; RISSETTI, 2010, p. 36)

As variáveis também possuem identificadores, que nada mais são do que o seu próprio nome. Sempre que definimos uma variável a declaramos com um nome, este nome chama-se identificador, praticamente todos os comandos possuem identificadores, o **print( )** é um identificador para uma rotina de exibição de dados por exemplo.

É importante saber que existem algumas regras para formação dos identificadores, segundo Ascencio e Campos(2010, p. 9), sendo elas:

- Somente podem ser utilizadas letras maiúsculas, minúsculas, números e o caracter sublinhado ( \_ ) no nome das variáveis;
- O caractere inicial deve obrigatoriamente ser uma letra ou o caractere sublinhado, não se deve começar o nome da variável por números;
- Não são permitidos espaços em branco nem caracteres

- especiais (!@#\$%^&\*+-<>=...) com exceção do sublinhado (\_);
- Não podemos utilizar palavras reservadas, estas são nomes iguais a outros identificadores como comandos da linguagem de programação que você estiver utilizando, nomes de variáveis já declaradas no escopo, etc.

Tabela 3 – Identificadores válidos e inválidos

<b>Forma Válida</b>	<b>Forma Inválida</b>		
Joao12	Ok	8C	Não pode começar com números
cpf	Ok	nome usuario	Não pode conter espaços em branco
_nome	Ok	True	Não pode ser nome de palavra reservada (True é uma palavra reservada do python)
registro_usuario	Ok	id-paciente	Não pode conter caractere especial
NOTA	Ok	bot@o	Não pode conter caractere especial
x5	Ok	print	Não pode ser igual a nomes de outros identificadores.

Fonte: O autor.

## VARIÁVEIS EM PYTHON

As variáveis em Python são bem flexíveis, se adaptando ao tipo de dado e alocando espaço dinamicamente na memória. Para atribuir um valor à uma variável utilizamos o sinal de = desta forma:

```
nome = "Edgar Morin"  
idade = 22  
peso = 62.24
```

Diferentemente do pseudocódigo e de algumas linguagens de programação não precisamos declarar o tipo de variável para que o Python a reconheça. Vamos a um exemplo, abra seu IDLE e crie um novo arquivo (*File* então *New File*), insira o pequeno trecho a seguir:

```
mensagem = "Olá Mundo"  
print(mensagem)
```

Salve seu arquivo e execute, sua saída deve ser parecida com esta:

```
===== RESTART: /Users/macbookpro/Documents/exemplo1.py =====
Ola Mundo
>>>
```

Tome cuidado ao escrever o identificador da variável, pois o Python diferencia letras maiúsculas e minúsculas, assim a variável **Carro** é diferente de **carro**. Da mesma forma, caso o identificador esteja escrito incorretamente, nosso interpretador retornará um erro. Tente reescrever o programa anterior omitindo uma letra da variável mensagem na instrução **print()**, desta forma:

```
mensagem = "Ola Mundo"
print(mesagem)
```

Perceba que criamos uma variável chamada mensagem, mas passamos para a instrução **print( )** um identificador chamado mesagem. O interpretador Python irá retornar um erro parecido com este:

```
===== RESTART: /Users/macbookpro/Documents/exemplo1.py =====
Traceback (most recent call last):
  File "/Users/macbookpro/Documents/exemplo1.py", line 2, in <module>
    print(mesagem)
NameError: name 'mesagem' is not defined
>>>
```

Não se preocupe com os erros, eles irão aparecer muitas vezes nas suas simulações de algoritmo. O Python é muito eficiente ao informar para o programador onde está o erro e

que tipo de erro foi encontrado. Veja, na primeira linha temos um *Traceback*, que segundo Matthes(2016, p. 52) “um *traceback* é um registro do ponto em que o interpretador se deparou com problemas quando tentou executar seu código”. Na próxima linha, o interpretador indica qual foi o arquivo e a linha do arquivo em que o programa parou. Neste caso vemos que o arquivo *exemplo1.py* possui um erro na linha 2. A terceira linha tenta nos mostrar aproximadamente a instrução onde o Python encontrou o erro, que foi na instrução `print(mesage)`. Por fim, na quarta linha da mensagem de erro temos o tipo de erro encontrado, que neste caso é um **NameError** que é um erro comum gerado quando tentamos operar um identificador não existente. Como não declaramos nenhuma variável chamada *mesagem* o interpretador retornou um erro avisando (em trocadilhos) “Olha Sr. Programador, o sr. me pediu para escrever o conteúdo identificado por 'mesagem' mas eu não encontrei nada na memória com esse identificador, tem certeza que é isso mesmo?”. Se o Python pudesse falar ele diria algo como isto, mas as informações que ele mostra já são suficientes pra nos informar onde foi que erramos<sup>2</sup>.

Ao declarar uma variável em Python, também devemos

---

<sup>2</sup> No Python você verá que os erros são chamados de *exceptions*, todos eles estão listados na documentação oficial do Python: <<https://docs.python.org/2/library/exceptions.html>>

respeitar as palavras reservadas da linguagem, ou seja os identificadores já existentes. Segundo a documentação do Python temos dispostos alguns exemplos de palavras reservadas (*keywords*):

Figura 8 – Palavras reservadas em Python:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Fonte: <[https://docs.python.org/3.5/reference/lexical\\_analysis.html#identifiers](https://docs.python.org/3.5/reference/lexical_analysis.html#identifiers)>

Vamos exercitar mais um pouco e criar uma variável para cada tipo de dado que acabamos de conhecer:

```
texto = "mensagem com texto"
numero_inteiro = 9
fracionario = 1.29
dado_logico = True

print(texto, numero_inteiro, fracionario, dado_logico)
```

A saída seria:

```
===== RESTART: /Users/mac  
mensagem com texto 9 1.29 True  
>>>
```

Os dados do tipo literal em Python podem ser declarados entre 'aspas simples' ou “aspas duplas”, desde que se abra e feche as mesmas aspas, não sendo possível abrir um aspa simples e fechar com uma dupla ('exemplo') e nem ao contrário (“exemplo’), mas pode-se abrir uma aspa simples dentro de uma aspa dupla e vice-versa (“Assim 'por exemplo' ” ou 'Assim “por exemplo” '). “Python chama qualquer número com um ponto decimal de *ponto flutuante (float)*. Esse termo é usado na maioria das linguagens de programação e refere-se ao fato de um ponto decimal poder aparecer em qualquer posição de um número” (MATTHES, 2016, p. 63), ressalta-se aqui a importância de declarar os números reais (*float*) utilizando-se o ponto ( . ) para representar a parte fracionária e não a vírgula ( , ) do contrário o Python irá retornar um erro.

Os dados lógicos ou booleanos devem ser declarados com a primeira letra maiúscula (**True** ou **False**), se você inserir o valor true, ou inserir o valor false, o Python não irá

reconhecer e irá enviar um *traceback* indicando um **NameError** como vimos anteriormente, sua desculpa é que nenhum identificador com o nome 'true' ou 'false' foi definido, isto porque assim como muitas outras linguagens, o Python é *Case Sensitive*, o que significa que ele diferencia letras maiúsculas e minúsculas, logo, *True* e *true* são duas coisas completamente distintas para nosso interpretador, sempre que escrevermos um algoritmo devemos ser muito claros e específicos.

Podemos passar as variáveis separadas por vírgula para a instrução **print( )** como no exemplo anterior, ou podemos fazer uma instrução para cada variável:

```
print(texto)
print(numero_inteiro)
print(fracionario)
print(dado_logico)
```

E a saída seria:

```
=====
mensagem com texto
9
1.29
True
>>>
```

Há outras formas de imprimir na tela pulando linhas, e

formas mais eficientes de se trabalhar com os dados de forma que o desempenho do algoritmo seja superior, porém em nível de aprendizado estas instruções demonstram muito bem o comportamento do interpretador Python.

Vimos no inicio deste capítulo que uma variável somente pode guardar um valor por vez, isso quer dizer que se eu declarar uma variável chamada surpresa e atribuir diferentes valores para ela, cada vez que um novo valor for atribuído, o anterior será esquecido, veja:

```
surpresa = 9
surpresa = 16.78
surpresa = 'doce de goiaba'
surpresa = 'CWB City Rocks'

print(surpresa)
```

Atribuímos inicialmente o o valor inteiro 9 para a variável surpresa, então logo em seguida atribuímos o valor real 16.78, então a próxima instrução diz ao Python para atribuir uma string à mesma variável, e em seguida pede para atribuir outra string. No final dizemos ao Python para escrever o valor da variável surpresa.

```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/macbookpro/Documents/exemplo1.py =====
CWB City Rocks
>>> |
```

O valor exibido na tela foi o último valor a ser atribuído à variável, todos os outros valores foram desconsiderados, isso porque uma variável somente pode guardar um valor de cada vez. Esta sequência de instruções também demonstra como o Python consegue atribuir diferentes tipos de dados em uma mesma variável, o que não aconteceria em muitas outras linguagens de programação.

Lembra-se do algoritmo *soma.py* que construímos no início do capítulo? Neste algoritmo o a instrução **input( )** irá esperar o usuário entrar com um tipo de dado, porém independente do tipo de dado fornecido, o Python irá armazenar a entrada como uma *string* (dado do tipo literal), então para realizar a soma tivemos que fazer uma conversão utilizando uma instrução **int( )**. Quando passamos um tipo de dado para a instrução **int( )** o Python tentará realizar uma conversão do tipo de dado fornecido para um tipo inteiro. Vamos ver um exemplo:

```
numero = '12'  
print(numero)  
numero = int(numero)  
print(numero)
```

Primeiro declaramos uma variável chamada numero e pedimos para mostrá-la na tela, então realizamos a conversão para inteiro e pedimos para mostrar na tela. Na saída não conseguimos identificar a diferença, observe o resultado:

```
----- RESTART:  
12  
12  
>>>
```

Mas se tentarmos realizar uma operação matemática, como a soma, em variáveis do tipo literal o que acontece é um evento chamado concatenação, e não a adição em si, veja o exemplo:

```
numero = '12'  
numero2 = '3'  
print(numero + numero2)  
  
numero = int(numero)  
numero2 = int(numero2)  
print(numero + numero2)
```

Primeiro declaramos duas strings e tentamos exibir a soma de ambas, depois convertemos as *strings* para inteiro e exibimos a soma das variáveis:

```
===== RESTART:  
123  
15  
>>>
```

Observe os resultados, na primeira linha obtivemos o resultado da concatenação, que nada mais é do que juntar a segunda string ao final da primeira, resultando não na soma, mas na junção das duas variáveis literais em um único texto. Já no segundo resultado pudemos obter o resultado da operação de adição, pois a instrução **int( )** converteu os dados literais para numéricos e o Python compreendeu que deveria realizar uma operação aritmética com estes dados.

Mas o que aconteceria se você tentasse converter uma letra para número?

```
fruta = 'banana'  
banana = int(fruta)  
print(fruta)|
```

Ao criarmos uma variável literal com caracteres não numéricos e forçarmos uma conversão, o Python logo retornará um erro, pois não é possível converter letras para números, o mesmo aconteceria se tentássemos converter um dado do tipo real ou lógico.

```
===== RESTART: /Users/macbookpro/Documents/exemplo1.py =====  
Traceback (most recent call last):  
  File "/Users/macbookpro/Documents/exemplo1.py", line 2, in <module>  
    banana = int(fruta)  
ValueError: invalid literal for int() with base 10: 'banana'  
>>>
```

O erro retornado é um **ValueError**, dizendo que os valores expressados na tentativa de execução não são válidos, por isto devemos tomar muito cuidado ao trabalhar com os tipos de dados para não levantar erros inesperados.

Caso você tenha dúvida quanto ao tipo de dado que você está lidando, o Python tem a instrução **type( )**. Quando você passar uma variável para a instrução **type(variavel)** ele irá

informar o tipo de dado dessa variável, como no exemplo a seguir:

```
fruta = 'banana'  
numero = 78  
God = False  
PI = 3.1415  
  
print(type(fruta))  
print(type(numero))  
print(type(God))  
print(type(PI))
```

Saída:

```
===== RESTART: /i  
<class 'str'>  
<class 'int'>  
<class 'bool'>  
<class 'float'>  
>>>
```

Vemos que cada tipo de dado pertence a uma classe que representa um tipo de dado específico: Literal (*str*), Inteiro (*int*), Lógico (*bool*) e Real (*float*).<sup>3</sup>

---

<sup>3</sup> O Python também tem os tipos de dados complexos, mas não será abordado nesta obra, você pode conferir o modelo de dados do Python em <<https://docs.python.org/3/reference/datamodel.html#>>.

## CONSTANTES

Da mesma forma que possuímos valores variáveis, também temos valores constantes, ou seja, que não se modificam ao longo do seu algoritmo. No Python, por questões de boas práticas, sempre definimos uma constante com todas as letras do identificador em CAIXA ALTA, ou seja, letras maiúsculas, dessa forma:

```
PI = 3.1415  
ALTURA = 600  
LARGURA = 800
```

## EXPRESSÕES

Quando estivermos elaborando nossos algoritmos, muitas vezes teremos que utilizar expressões aritméticas para processar os dados, as expressões são formadas por operadores aritméticos e possuem, assim como na matemática, uma prioridade de execução, denominada precedência. Segue um quadro com os operadores em Python:

Tabela 4 – Operadores aritméticos mais comuns em Python e sua precedência

OPERADOR	OPERAÇÃO	PRECEDÊNCIA	DESCRIÇÃO
+	Adição	0	Realiza a soma dos operandos. Ex: $a + b$
-	Subtração	0	Realiza a subtração dos operandos. Ex: total - desconto
/	Divisão	1	Realiza a divisão do operando a esquerda pelo operando a direita. Ex: $12 / 2$
//	Divisão inteira	1	Retorna a parte inteira da divisão Ex: $3 // 3$ (resulta em 1)
*	Multiplicação	1	Multiplica os operandos. Ex: $2 * a$
**	Exponenciação	2	Eleva o operando a esquerda à potência do operando a direita. Ex: $5^{**2}$
%	Módulo	2	Obtém o resto da divisão dos operandos. Ex: $7 \% 2$

Fonte: O autor.

As operações com menor precedência são executadas por último, neste caso ao observar a expressão  $2 + 3 * 5$ , assim

como na matemática, a primeira operação a ser realizada é a multiplicação ( $3 * 5$ ), somente então a adição será realizada ( $2 + 15$ ). Estas precedências podem ser alteradas fazendo uso do parêntese ( ), dando precedência para a expressão que estiver dentro do parêntese, desta forma:

```
a = 2 + 3 * 5  
b = (2 + 3) * 5  
  
print("O valor de a: ", a)  
print("O valor de b: ", b)
```

Saída:

```
===== RESTART: /Users/macbookpro/Documents/exemplo1.py =====  
0 valor de a: 17  
0 valor de b: 25  
>>>
```

Nós também temos os operadores relacionais, que permitem realizar comparações entre valores:

Tabela 5 – Operadores relacionais em Python

OPERADOR	COMPARAÇÃO	DESCRIÇÃO
$==$	Igualdade	Dois sinais de $=$ compara se dois valores são idênticos. Ex: $a == b$
$!=$	Diferença	Compara se dois valores são diferentes. Ex: $a != b$

>	Maior	Compara se o primeiro valor é maior que segundo. Ex: a > b
<	Menor	Compara se o primeiro valor é menor que o segundo. Ex: a < b
>=	Maior ou Igual	Compara se o primeiro valor é maior ou igual ao segundo valor. Ex: a >= b
<=	Menor ou Igual	Compara se o primeiro valor é menor ou igual ao segundo. Ex: a <= b

Fonte: O autor.

As expressões relacionais sempre retornarão um valor lógico (**True** ou **False**), veja no exemplo a seguir:

```
a = 2
b = 3

print(a == b)
print(a != b)
print(a > b)
print(a < b)

print(a >= a)
print(a <= b)
```

Saída:

```
=====
False
True
False
True
True
True
>>>
```

Existem mais operadores que não cobriremos aqui, mas você pode conferir a lista completa na documentação do Python<sup>4</sup>.

Uma aplicação interessante em Python é a multiplicação de strings, algo que não é possível em outras linguagens, veja:

```
chaves = 'pi'  
print(chaves * 10)
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo2/chaves.py =====  
pipipipipipipipipi  
>>> |
```

Temos também os operadores lógicos que podem ser utilizados juntamente com os relacionais fazendo comparações, a seguir os operadores lógicos:

Tabela 6 – Operadores lógicos em Python

OPERADOR	OPERAÇÃO	PRIORIDADE	DESCRIÇÃO
or	Disjunção	1	A disjunção entre duas operações resultaria verdadeiro se um dos valores for verdadeiro.

<sup>4</sup> Disponível em:  
<[https://docs.python.org/3.5/reference/lexical\\_analysis.html#operators](https://docs.python.org/3.5/reference/lexical_analysis.html#operators)>

and	Conjunção	2	A conjunção resultará em verdadeiro se, e somente se, todos os valores examinados forem verdadeiros
not	Negação	3	A negação inverte o valor lógico da variável examinada. Se o valor for verdadeiro ele se tornará falso, e vice-versa.

Fonte: O autor.

Vamos examinar estas operações com o Python:

```
a = 2
b = 3
c = 4

print(a > b or a > c)
print(a < b and a < c)
print(not a == b)
```

Saída:

```
False
True
True
>>>
```

Veja que definimos as variáveis  $a = 2$ ,  $b = 3$  e  $c = 4$ .

Então fizemos um pergunta ao Python como que “Python, **a** é maior que **b** OU **a** é maior que **c**?” então na primeira linha da saída temos a resposta False, pois a não é nem maior que b e nem maior que c, a na disjunção somente obtemos uma saída verdadeira se pelo menos uma expressão resultar em verdadeiro, como nossas duas expressões resultaram falso, a disjunção resultou falso.

Então perguntamos ao Python: “**a** é menor que **b** E **a** é menor que **c**?”, o Python responde: *True*, pois na conjunção obtemos resultado verdadeiro se todas as expressões resultarem verdadeiro, como 2 é menor que 3 e também é menor que 4, nossa avaliação lógica retornou verdadeiro.

Por fim perguntamos ao Python se **a** e **b** são idênticos e informar o valor lógico inverso (*como assim?* você pergunta), temos que 2 não é igual a 5, então o resultado da avaliação seria **Falso**, mas como dissemos ao Python para nos informar o **inverso** ele respondeu **Verdadeiro** (*True*). Um detalhe interessante é que sempre que uma variável estiver inicializada com o valor 0 ela retornará Falso.

## FUNÇÕES INTRINSECAS

Funções intrínsecas são funções (instruções) pré-definidas em uma linguagem de programação. Estas funções auxiliam o programador para não ter que reinventar a roda em seus algoritmos. Por exemplo, a instrução **print( )** é uma função pré-definida (*built-in function* de acordo com BARRY, 2015) da linguagem que permite exibir uma mensagem na tela. No site oficial do Python encontramos uma tabela com as funções intrínsecas existentes no Python 3:

Tabela 7 – Funções intrínsecas no Python 3

Built-in Functions				
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>

Built-in Functions				
	)			
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

Fonte: <<https://docs.python.org/3/library/functions.html>>

Você não precisa decorar todas estas funções, mas a medida que for construindo seus algoritmos vai utilizar algumas delas, existem outras funções que não estão presentes nesta lista mas você poderá conhecê-las na documentação do Python, outras apresentaremos no decorrer deste livro. A descrição destas pode ser encontrada na documentação do Python. Algumas destas você já utilizou aqui em alguns exemplos, como por exemplo `print()`, `int()`, `input()` e `type()`. Lembra-se do `int()` para converter números em dados literais para números inteiros? Também temos as funções `str()` e `float()` para converter dados em literais e números reais respectivamente.

```
a = 2  
print(str(a))  
print(float(a))|
```

Saída:

```
2  
2.0
```

## ENTRADA DE DADOS

Como vimos, o Python possui uma função interna chamada **input( )** que recebe uma entrada do usuário. Nesta instrução podemos fornecer uma string para ajudar o usuário a saber que tipo de dado ele deve fornecer, como no exemplo a seguir:

```
nome = input("Insira seu nome ")  
print("Olá ", nome, " como está hoje?")
```

Saída:

```
===== RESTART: /Users/macbookpro/Documents/exemplo1.py =====
Insira seu nome bruno
Ola bruno como esta hoje?
>>>
```

Como vimos, a instrução **input()** recebe uma string por padrão, e caso precisemos que a entrada seja de outro tipo de dado temos que fazer a conversão com **int()** ou **float()**, desta forma:

```
nome = input("Insira seu nome ")
idade = int(input("Insira sua idade "))
altura = float(input("Insira sua altura "))
print("Seu nome e ", nome)
print("Voce tem ", idade, " anos de idade")
print("Voce medee ", altura, " de altura")
```

Perceba que ao obter a idade, primeira declaramos a variável então atribuímos a ela a instrução **int()** pois ela deve ser um número inteiro, e dentro da instrução **int()** inserimos o **input()**. Lembra quando dissemos que tudo que estiver dentro do parêntese acontece antes? É exatamente isto, primeiro ocorre a chamada da função **input()** que irá fazer uma solicitação de entrada ao usuário. Quando ele realizar esta entrada o valor fornecido será processado pela função **int()**, convertendo o valor para um dado do tipo inteiro, que então

será armazenado na variável idade. O mesmo acontece com a altura, primeiro um valor é solicitado, então convertido para o tipo de dados float e então atribuído à variável altura. Quando acessarmos estas variáveis elas já estarão com o tipo de dado que convertemos. A saída para o algoritmo acima seria parecida com esta:

```
===== RESTART: /Users/macbookpro/Documents/exemplo1.py =====
Insira seu nome Bruno
Insira sua idade 26
Insira sua altura 1.60
Seu nome é Bruno
Voce tem 26 anos de idade
Voce mede 1.6 de altura
>>>
```

## EXERCICIOS ELABORADOS

Primeiramente vamos mostrar alguns problemas e suas soluções em Python, vamos explicar a elaboração dos exercícios e alguns exercícios para que você leitor resolva sozinho utilizando os conhecimentos aprendidos até aqui.

- 1) Formular um algoritmo que leia e apresente os dados de uma pessoa: nome, idade, endereço, telefone de contato e mostre as informações obtidas na tela.

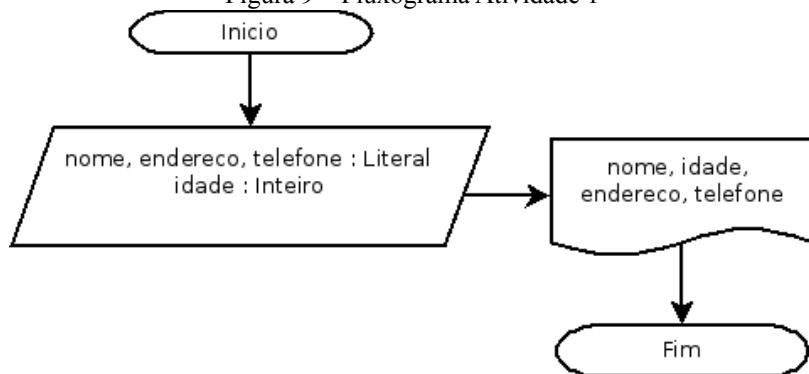
**Problema a ser resolvido:** Obter informações de uma pessoa e exibir as informações na tela;

**Dados de entrada:** nome, idade, endereço, telefone;

**Processamento:** Não há processamento, apenas entrada e saída de informações;

**Saída:** Exibir os dados da pessoa;

Figura 9 – Fluxograma Atividade 1



Fonte: O autor

---

```
nome = input("Insira seu nome ")
idade = int(input("Insira sua idade "))
endereco = input("Insira seu endereço ")
telefone = input("Insira seu telefone ")

print("Nome: ", nome)
print("Idade: ", idade)
print("Endereço: ", endereco)
print("Telefone para contato: ", telefone)
```

Primeiramente declaramos a variável nome e atribuímos a ela uma *string* informada pelo usuário, em seguida a variável

idade irá receber um valor de entrada do usuário que será convertido em um inteiro. Então atribuímos um valor de entrada para a variável endereço e mais uma para a variável telefone. Como não temos processamento as próximas instruções são comandos de saída, exibindo as informações obtidas correspondentes aos dados da pessoa.

```
===== RESTART: /Users/macbookpro/Documents/exercicio.py =====
Insira seu nome Bruno
Insira sua idade 26
Insira seu endereço Alameda Python numero 3
Insira seu telefone (98) 7654-3210
Nome: Bruno
Idade: 26
Endereço: Alameda Python numero 3
Telefone para contato: (98) 7654-3210
>>>
```

**Observações:** É muito comum obtermos o valor de um telefone como string, pois o usuário pode entrar com caracteres especiais como parêntese ( ) e o traço -. Também recomenda-se que nas variáveis que possuem acento ou cedilha este caractere seja substituído ou omitido como no caso de endereço (declaramos como endereço). Salienta-se ainda o cuidado ao declarar instruções dentro de instruções como no caso de **int(input( ))** onde abrimos dois parênteses então precisamos fechar dois parênteses, do contrário você receberá um erro.

2) Sua professora de matemática pediu que você calculasse a área e o perímetro de um quadrado e lhe passou as fórmulas para o cálculo da área sendo  $A = L \times L$  (o tamanho do lado do quadrado vezes ele mesmo) e o perímetro deve ser obtido através da soma dos quatro lados do quadrado. Vamos escrever um algoritmo que faça seu dever de casa:

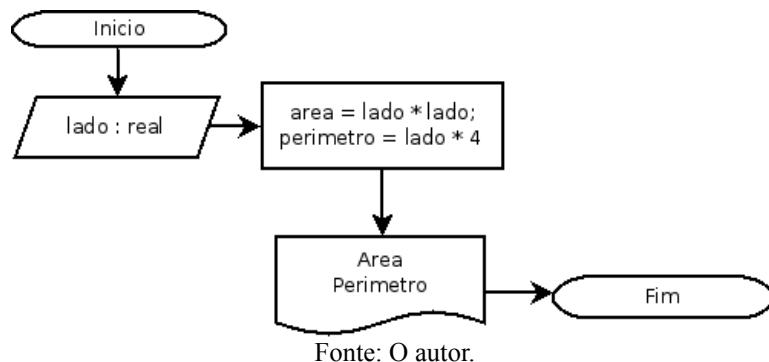
**Problema a ser resolvido:** Calcular a área e o perímetro de um quadrado;

**Dados de entrada:** area, perimetro, lado

**Processamento:** calcular a área ( $A = \text{lado} \times \text{lado}$  ou  $\text{lado}^{**2}$ ) e calcular o perímetro ( $b \times 4$  ou  $b + b + b + b$ );

**Saída:** Exibir o resultado do cálculo;

Figura10 – Fluxograma Atividade 2



```
lado = float(input("Insira o tamanho dos lados do quadrado: "))  
area = lado * lado  
perimetro = lado * 4  
print("A area do quadrado e de: ", area)  
print("O perimetro do quadrado e de: ", perimetro)
```

Primeiro precisamos do valor correspondente ao tamanho dos lados do quadrado então declaramos a variável lado e atribuímos a ela o valor que for inserido pelo usuário convertido em um número real, em seguida fazemos o processamento, multiplicando o lado por ele mesmo e atribuindo o resultado à variável area, a variável perímetro recebe o valor de lado multiplicado por 4. Em seguida temos as saídas exibindo os resultados.

```
===== RESTART: /Users/macbookpro/Documents/exercicio2.py =====
Insira o tamanho dos lados do quadrado: 3
A area do quadrado e de: 9.0
O perimetro do quadrado e de: 12.0
>>> |
```

**Observação:** Como o valor de *area* deveria ser *lado* ao quadrado poderia-se ter utilizado *lado\*\*2* que funcionaria da mesma forma, o operador **\*\*** eleva o valor da esquerda à potência do valor da direita, tente fazer isto no seu IDLE.

- 3) Sua professora gostou tanto do seu algoritmo computacional que lhe pediu para elaborar outro algoritmo, dessa vez um que realize o cálculo da área de um triângulo, sabendo que a área do triângulo é igual a base x altura, vamos elaborar este algoritmo:

2

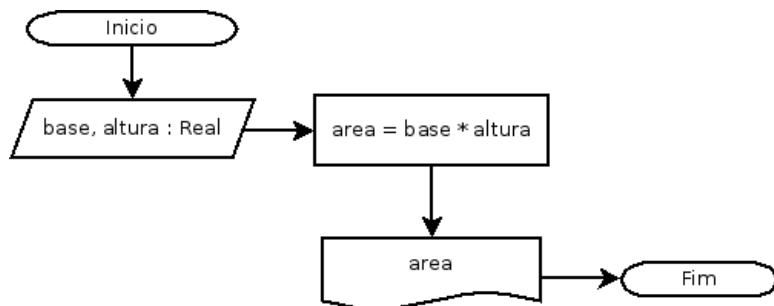
**Problema a ser resolvido:** Calcular a área de um triângulo;

**Dados de entrada:** base, altura;

**Processamento:** calcular a área  $A = (\text{base} \times \text{altura}) / 2$ ;

**Saída:** Exibir o resultado do cálculo;

Figura 11 – Fluxograma atividade 3



Fonte: O autor.

```
base = float(input("Insira o valor da base do triangulo: "))
altura = float(input("Insira o valor da altura do triangulo: "))

print("A area do triangulo e: ", (base * altura)/2)
```

Primeiro vamos receber a entrada necessária. Declaramos uma variável base e atribuímos a ela o valor inserido pelo usuário convertido em número real, depois fazemos o mesmo com a variável altura. Desta vez inserimos o processamento diretamente no comando de saída através de uma expressão, assim economizamos uma variável fazendo com que o algoritmo tenha um melhor desempenho. Em um pequeno programa como este pode não fazer nenhuma diferença, mas a medida que for construindo algoritmos mais complexos o número de linhas e comandos pode impactar no

desempenho do seu programa.

```
===== RESTART: /Users/macbookpro/Documents/exercicio3.py =====
Insira o valor da base do triangulo: 3.9
Insira o valor da altura do triangulo: 7.2
A area do triangulo e: 14.04
>>>
```

**Observação:** Se você andou brincando com o Python pode já ter percebido que em alguns cálculos com números reais o Python pode retornar um valor com várias casas decimais, isso é normal não se assuste, veremos como lidar com isso no decorrer do livro. Caso você ainda não tenha visto isso tente calcular  $2.90 \times 1.43$ .

### *Exercícios propostos*

- 1) Elabore um algoritmo em Python que leia, calcule e escreva a média aritmética entre quatro números;
- 2) Elabore um algoritmo em Python que receba um número inteiro e escreva na tela o número fornecido, o antecessor desse número e o sucessor desse número;
- 3) Elabore um algoritmo em Python que:
  - a) Primeiro exiba uma mensagem de boas vindas;
  - b) Pergunte o nome do usuário;
  - c) Exiba uma mensagem dizendo uma mensagem de olá seguida pelo nome do usuário seguida por outra mensagem fazendo um elogio.
- 4) Elabore um algoritmo em Python que calcule a área e o perímetro de um círculo, sabendo que  $A = \pi r^2$  e  $P=2\pi r$ .

## CAPITULO 3

---

...

### COMENTÁRIOS

É muito comum nas linguagens de programação a utilização de comentários em certas partes do algoritmo. Estes comentários servem para documentar o código e possibilitar o entendimento de certas sessões do algoritmo por outros programadores que venham a ler seu código posteriormente.

Em Python declaramos um comentário ao inserir o caractere `#`. Quando o Python se deparar com um comentário durante a interpretação do código, ele ignorará a linha comentada, ou seja, o Python não liga para os seus comentários.

---

```
#Este e um comentario
#Sempre que o python encontrar este # simbolo ele ira descartar o que estiver
#depois dele
print("esta instrucao sera executada")
#print("esta instrucao nao sera executada")
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo16.py =====
esta instrucao sera executada
>>>
```

Perceba que tudo que estava depois do `#` não foi

executado, pois o Python identificou como um comentário e logo passou para a próxima instrução válida. Este é o comentário de uma linha, porém também é possível inserir comentários de várias linhas desde que estejam entre aspas triplas (" comentário " ou """comentário""") como no exemplo a seguir:

```
"""
Este e um comentario de
varias linhas

"""

Nada disso
sera

interpretado pelo python
"""
```

Vamos comentar nosso código de agora em diante para facilitar o entendimento do contexto.

## ESTRUTURA CONDICIONAL

Grande parte das vezes em nossos algoritmos, teremos que optar por executar uma tarefa ou outra, dependendo de uma condição específica. As estruturas condicionais, também denominadas estruturas de seleção ou decisão (PUGA;

RISSETTI, 2010 – p.56) são dos tipos simples e composta (ASCENCIO; CAMPOS, 2010; LEAL, 2016; PUGA; RISSETTI, 2010). Vamos ver cada um deles.

## ESTRUTURA CONDICIONAL SIMPLES

Vamos exemplificar com um algoritmo bem prático dos nossos dias, ascender a luz. Geralmente para ascender uma lâmpada em um cômodo na sua casa, é bem comum que se aperte um interruptor. Se a lâmpada estiver desligada ela se ligará. Vamos fazer um algoritmo Python bem simplificado para demonstrar o funcionamento da estrutura condicional simples.

```
luz = False #luz apagada  
ascender_luz = input("Gostaria de ascender a luz? [s/n]")  
  
if ascender_luz == 's': #se a entrada for 's'  
    luz = True #a luz ascende  
  
print(luz) #no final verificamos se a luz esta acesa ou apagada
```

Veja que a instrução **if** irá analisar a variável `ascender_luz`, e se o conteúdo armazenado nesta variável for igual a string “s” ele executará os comandos abaixo do **if**. Se o conteúdo da variável for diferente da comparação o Python irá

pular para a próxima instrução que não estiver dentro do **if**.

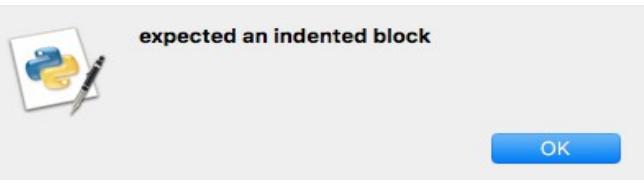


`if ascender_luz == 's':  
 luz = True #observe a indentacao`



Caso o código pertencente ao **if** não esteja tabulado à direita o Python não irá compreender e irá lhe informar algo como isto:

Figura 11 - Erro de indentação



Fonte: O autor.

Se você indentou direitinho seu algoritmo suas saídas poderão se parecer com estas:

```
===== RESTART: /Users/macbookpro/Documents/luz1.py =====
Gostaria de ascender a luz? [s/n]s
True
>>> |
```

Veja que caso a entrada seja diferente do que é esperado

o Python irá ignorar o comando `luz=True`, e a luz permanecerá apagada:

```
=====
RESTART: /Users/macbookpro/Documents/luz1.py =====
Gostaria de ascender a luz? [s/n]n
False
>>> |
```

Vamos melhorar este algoritmo para nos dizer se a luz está acesa ou apagada:

```
luz = False #luz comeca apagada

ascender_luz = input("Gostaria de ascender a luz? [s/n]")

if ascender_luz == 's':
    luz = True

if luz == True:    #se ascender a luz
    print("A luz esta acesa") #nos informe que agora a luz esta acesa|
```

Saída:

```
=====
RESTART: /Users/macbookpro/Documents/luz1.py =====
Gostaria de ascender a luz? [s/n]s
A luz esta acesa
>>>
```

Caso você não queira ascender a luz, o programa irá terminar em branco, pois não demos mais instruções ao Python, e ele somente executaria as que fornecemos se as expressões avaliadas na instrução `if` forem verdadeiras.

```
=====
RESTART: /Users/macbookpro/Documents/luz1.py =====
Gostaria de ascender a luz? [s/n]n
>>> |
```

## ESTRUTURA CONDICIONAL COMPOSTA

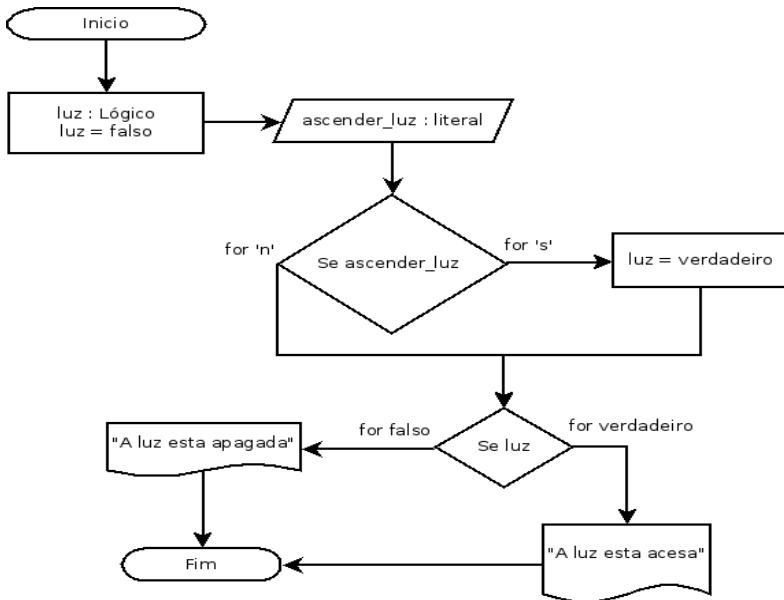
Na estrutura condicional composta nós temos mais de uma opção de desvio para uma condição a ser analisada. Vamos melhorar ainda mais o algoritmo da luz:

```
luz = False #luz começa apagada  
ascender_luz = input("Gostaria de ascender a luz? [s/n]")  
  
if ascender_luz == 's':  
    luz = True  
  
if luz == True: #se ascender a luz  
    print("A luz está acesa") #nos informe que agora a luz está acesa  
else: #se não  
    print("A luz está apagada") #nos informe que a luz está apagada
```

O **else** é um segmento do **if** que diz ao Python, “caso a condição anterior for falsa, faça isto aqui ok?”, como uma alternativa que será realizada em oposição a outra. Leal (2016b, p. 65) afirma que a estrutura de decisão composta é melhor utilizada “Quando uma condição implica a execução de um ou outro bloco. Em situações que há duas condições mutuamente exclusivas”, ou seja, só pode acontecer uma das escolhas, as outras serão ignoradas pelo interpretador.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/luz2.py ======  
Gostaria de ascender a luz? [s/n]n  
A luz está apagada  
>>> |
```

Figura 12 – Fluxograma para ascender a luz

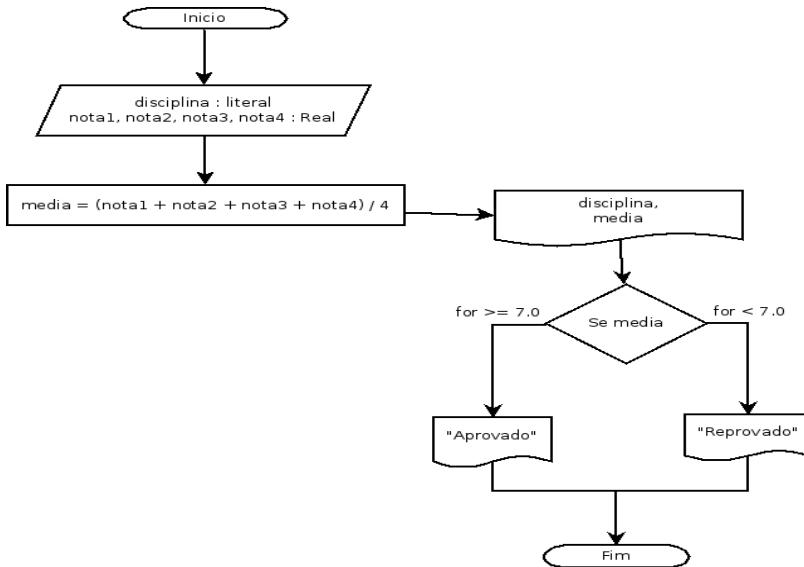


Fonte: O autor.

Seguindo para mais um exemplo, vamos imaginar que seu professor comentou na aula que para que você seja aprovado na disciplina você precisa ter uma média igual ou superior à 7.0, para descobrir a média você somaria as quatro notas de cada bimestre e dividiria por quatro e Se a média for igual ou superior a 7.0 você é aprovado, Senão, você fica de

recuperação.

Figura 13 – Fluxograma para media do aluno



Fonte: O autor.

Vamos implementar um algoritmo em Python para calcular sua média, primeiro vamos receber o nome da disciplina então as quatro notas e o programa deve informar sua média nesta disciplina.

```

#solicitamos o nome da disciplina
disciplina = input("Insira o nome da disciplina: ")

#solicitamos as notas de cada bimestre
nota1 = float(input("Insira a nota do 1 Bimestre: "))
nota2 = float(input("Insira a nota do 2 Bimestre: "))
nota3 = float(input("Insira a nota do 3 Bimestre: "))
nota4 = float(input("Insira a nota do 4 Bimestre: "))

#calculamos a media
media = (nota1 + nota2 + nota3 + nota4) / 4

#exibimos a media e a disciplina
print("Disciplina: ", disciplina)
print("Media: ", media)

if (media < 7.0): #se a media for menor que 7.0 o aluno é reprovado
    print("Reprovado")
else: # senao é aprovado
    print("Aprovado")

```

As instruções **if** e **else** em Python analisam uma condição, no caso de `if(media < 7.0)` nós passamos para o Python uma expressão relacional (`media < 7.0`), o Python vai analisar se a media é menor que 7.0 e verificar se é verdade ou mentira, caso seja verdade ele executará o comando logo abaixo do **if**, tabulado à direita, e caso a expressão se revele falsa ele irá ignorar e pular para os comandos abaixo do **else**.

As saídas:

```

===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo18.py =====
Insira o nome da disciplina: FIsica
Insira a nota do 1 Bimestre: 2.4
Insira a nota do 2 Bimestre: 1.9
Insira a nota do 3 Bimestre: 5.8
Insira a nota do 4 Bimestre: 6.3
Disciplina: FIsica
Media: 4.1
Reprovado
>>> |

```

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo18.py ====
Insira o nome da disciplina: Historia
Insira a nota do 1 Bimestre: 9.6
Insira a nota do 2 Bimestre: 6.8
Insira a nota do 3 Bimestre: 8.8
Insira a nota do 4 Bimestre: 9.2
Disciplina: Historia
Media: 8.6
Aprovado
>>> |
```

Seu professor gostou do programa, mas agora ele disse que esqueceu de lhe avisar que caso a nota esteja entre 6.0 e 6.9 o aluno ficará de recuperação;

Então é só colocarmos mais um if, certo?



Bem, na verdade o Python tem mais um comando de decisão composta para não precisarmos ficar repetindo **if's** toda hora. O comando **elif** pode ser utilizado depois do **if** e antes do **else**, desta forma:

```

#solicitamos o nome da disciplina
disciplina = input("Insira o nome da disciplina: ")

#solicitamos as notas de cada bimestre
nota1 = float(input("Insira a nota do 1 Bimestre: "))
nota2 = float(input("Insira a nota do 2 Bimestre: "))
nota3 = float(input("Insira a nota do 3 Bimestre: "))
nota4 = float(input("Insira a nota do 4 Bimestre: "))

#calculamos a media
media = (nota1 + nota2 + nota3 + nota4) / 4

#exibimos a media e a disciplina
print("Disciplina: ", disciplina)
print("Media: ", media)

if media >= 7.0: #se a media for menor que 7.0 o aluno é reprovado
    print("Aprovado")

elif media >= 6.0 and media <= 6.9: # se a media for entre 6.0 e 6.9 recuperacao
    print("Recuperacao")

else: # senao é aprovado
    print("Reprovado")

```

Tivemos que fazer algumas mudanças no nosso programa, primeiro trocamos o primeiro if que verificava se a media era menor que 7.0, se a media fosse menor que 7.0 o aluno seria reprovado, mas como temos alunos que ainda podem ficar de recuperação vamos ver somente as medias que estão acima ou igual a 7.0, do contrario se a média for entre 6.0 e 6.9 será exibido uma mensagem dizendo que o aluno está em recuperação, e se ele não for aprovado nem estiver de recuperação, então ele está reprovado.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo19.py =====
Insira o nome da disciplina: Quimica
Insira a nota do 1 Bimestre: 5.6
Insira a nota do 2 Bimestre: 6.2
Insira a nota do 3 Bimestre: 6.7
Insira a nota do 4 Bimestre: 6.0
Disciplina: Quimica
Media: 6.125
Recuperacao
>>> |
```

Mas qual a diferença entre **if** e **elif**?



A diferença está no desempenho do algoritmo. Quando temos varias instruções do tipo **if** (estrutura condicional simples), o interpretador irá checar cada uma delas. Se tivermos uma estrutura composta por **if**, **elif** e **else** o interpretador irá checar o primeiro **if** e caso seja verdadeiro ele irá ignorar o resto da composição, ou seja, se a primeira verificação for verdadeira as seguintes serão descartadas, lembrando que você pode inserir quantos **elif's** quiser. Em um algoritmo complexo, demasiadas verificações de **if** podem comprometer o desempenho do algoritmo, tornando sua execução mais lenta. Então como saber qual tipo de instrução utilizar? Simples, como dito anteriormente, se você tiver diferentes situações onde somente uma poderá ocorrer, deve-se utilizar a estrutura de decisão composta, pois as alternativas são

mutuamente excludentes. Caso você precise fazer uma única verificação utiliza-se a estrutura simples.

É preciso ficar atento à um aspecto muito importante para o funcionamento do seu código, a indentação. A indentação é o recuo à direita da margem pelo texto, no caso do Python a indentação é OBRIGATÓRIA, pois o interpretador irá considerar como pertencente à instrução if tudo o que estiver indentado, do contrário o comando poderá ser executado em uma hora inoportuna do programa ou pior gerar um erro.

Figura 13 – Indentação do código

```
if (media < 7.0): #se a media for menor que 7.0 o aluno e reprovado
    print("Reprovado")
else: # sendo e aprovado
    print("Aprovado")
```

Fonte: O autor.

Veja na figura acima que o **if** é cercado por um retângulo azul que representa um bloco de comandos, os comandos pertencentes a este bloco estão indentados quatro espaços à direita da margem (basta pressionar o botão *tab* uma vez), cercados por um retângulo vermelho. Os comando que

estiverem dentro do vermelho somente serão executadas se as verificações feitas no azul forem verdadeiras do contrário o interpretador passará para o próximo bloco de comandos.

A estrutura condicional em Python nos permite uma gama de verificações, inclusive verificar se determinadas letras ou palavras existem em uma string, veja:

```
frase = "Quem ensina aprende o que ensina e quem aprende ensina ao aprender - FREIRE"  
if 'FREIRE' in frase: #Se existir a palavra FREIRE na frase  
    print("Graaaaande Mestre Paulo Freire")  
else:  
    print("Qualquer outra coisa")
```

Saída:

```
RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemploIfinString.py  
Graaaaande Mestre Paulo Freire  
>>> |
```

Perguntamos ao Python se uma determinada palavra ou letra existe em uma string, ele irá analisar a string e se existir ele irá retornar verdadeiro, noutro caso ele retorna falso. Lembra-se da instrução `type()`? Vamos fazer um algoritmo que recebe uma variável e verifica qual o tipo da variável escrevendo na tela:

```
caixa = 72 #definimos uma variavel com um valor  
  
caixa = str(type(caixa)) #verificamos seu tipo e guardamos como string  
  
if 'str' in caixa:  
    print("Essa variavel e do tipo String (Literal)")  
  
elif 'int' in caixa:  
    print("Essa variavel e do tipo Inteiro")  
  
elif 'float' in caixa:  
    print("Essa variavel e do tipo float")  
  
elif 'bool' in caixa:  
    print("Essa variavel e do tipo bool")  
  
else:  
    print("Outro")
```

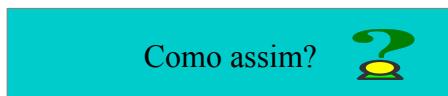
Lembra que a instrução `type( )` retorna um texto esquisito dizendo algo tipo <class 'int'>? Então, o que fizemos foi adicionar um valor a uma variável, depois nessa mesma variável adicionamos esta mensagem esquisita através da instrução `type( )` convertida para *string* através da instrução `str( )`, você também deve se recordar que as instruções que estão nos parênteses mais internos acontecem primeiro, certo? Depois verificamos se a variável contém a abreviatura correspondente ao seu tipo no nome e escrevemos uma mensagem um pouco mais elegante do que <class 'int'> para informar o tipo da variável que estamos lidando.

```
= RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/verificarTipos.py =
Essa variavel e do tipo Inteiro
>>>
```

Tente mudar o tipo de variável para testar os outros resultados. Pesquise sobre o tipo *complex* em Python e implemente o algoritmo para reconhecer este tipo de dado.

## ESTRUTURA CONDICIONAL ANINHADA

Na estrutura condicional aninhada, também conhecida por encadeamento (PUGA; RISSETTI, 2010) são estruturas condicionais dentro de outras estruturas condicionais. Desta forma ao realizar uma verificação condicional que retorne verdadeiro, o interpretador poderá encontrar com outros testes condicionais.



Vamos exemplificar melhor em um algoritmo que verifica o maior dentre 3 números escolhidos:

```
#primeiro recebemos a entrada
num1 = int(input("Insira o primeiro numero "))
num2 = int(input("Insira o segundo numero "))
num3 = int(input("Insira o terceiro numero "))

if num1 > num2: #estrutura condicional externa

    if num1 > num3: #estrutura condicional interna (aninhada)
        print("O primeiro numero foi o maior, ", num1)
    else:
        print("O terceiro numero e o maior, ", num3)
else:

    if num2 > num3:
        print("O segundo numero e o maior, ", num2)
    else:
        print("O terceiro numero e o maior, ", num3)
```

Neste exemplo primeiro recebemos três entradas que devem ser números inteiros, então verificamos se o num1 é maior que o num2, caso isso seja verdade partimos para uma outra verificação, que analisa se o num1 é maior que o num3, e se for verdade isso significa que o primeiro número foi o maior, do contrário o terceiro número é o maior. Se a primeira condição for falsa, partimos para outra verificação, que analisa se o num2 é maior que o num3.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo20.py =====
Insira o primeiro numero 1
Insira o segundo numero 2
Insira o terceiro numero 3
O terceiro numero e o maior,  3
>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo20.py =====
Insira o primeiro numero 3
Insira o segundo numero 2
Insira o terceiro numero 1
O primeiro numero foi o maior,  3|
>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo20.py =====
Insira o primeiro numero 1
Insira o segundo numero 3
Insira o terceiro numero 2
O segundo numero e o maior,  3
>>>
```

Existem sim formas diferentes e até mais eficazes de se implementar este algoritmo, porém para nível de exemplo este algoritmo é capaz de demonstrar o funcionamento da estrutura condicional aninhada. A grande diferença entre usar as condições aninhadas, de acordo com Leal (2016a, p. 70) “é que o uso destes encadeados melhora o desempenho do algoritmo, isto é, torna o algoritmo mais rápido por realizar menos testes e comparações. Ou ainda, executar um menor número de passos para chegar à solução do problema.”

## EXERCICIOS ELABORADOS

4) Elabore um algoritmo que receba o nome e a idade de uma pessoa e informe se é menor de idade, maior de idade

ou idoso.

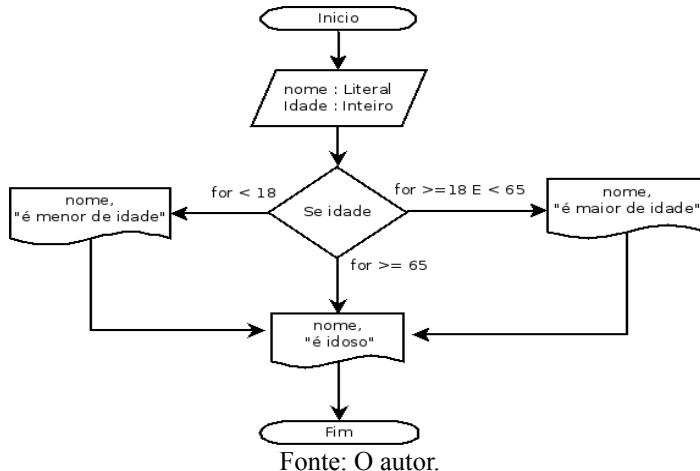
**Objetivo:** Informar se a pessoa é menor de idade, maior de idade ou idoso;

**Entrada:** nome e idade;

**Processamento:** Verificar se a idade é menor que 18 (menor de idade), maior que 18 e menor que 65 (adulto) ou maior que 65 (idoso);

**Saída:** escrever se a pessoa é menor, maior ou idoso;

Figura 14 – Fluxograma atividade 4



Fonte: O autor.

```

nome = input("Insira seu nome: ") #ler nome
idade = int(input("Insira sua idade: ")) #ler idade

if idade < 18: #verifica se a idade e menor que 18
    print(nome, " voce e menor de idade")

elif idade >= 18 and idade < 65: #senao verifica se a idade est entre 18 e 65
    print(nome, " voce e maior de idade")

else: #senao for nenhuma das acima, entao e idoso
    print(nome, " voce ja e idoso")

```

Saída:

```

>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo21.py =====
Insira seu nome: bruno
Insira sua idade: 26
bruno  voce e maior de idade
>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo21.py =====
Insira seu nome: dollynho
Insira sua idade: 14
dollynho  voce e menor de idade
>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo21.py =====
Insira seu nome: jureval
Insira sua idade: 72
jureval  voce ja e idoso
>>> |

```

5) Elabore um algoritmo que calcule o IMC (Índice de Massa Corporal) de uma pessoa de acordo com seu peso e altura. O programa deve informar se a pessoa está abaixo do peso (IMC menor que 20), normal (IMC entre 20 e 25), excesso de peso (entre 26 e 30), obesa (IMC entre 31 e 35) ou com obesidade mórbida (acima de 35). O cálculo do IMC é

dado por  $\frac{peso}{altura^2}$  .

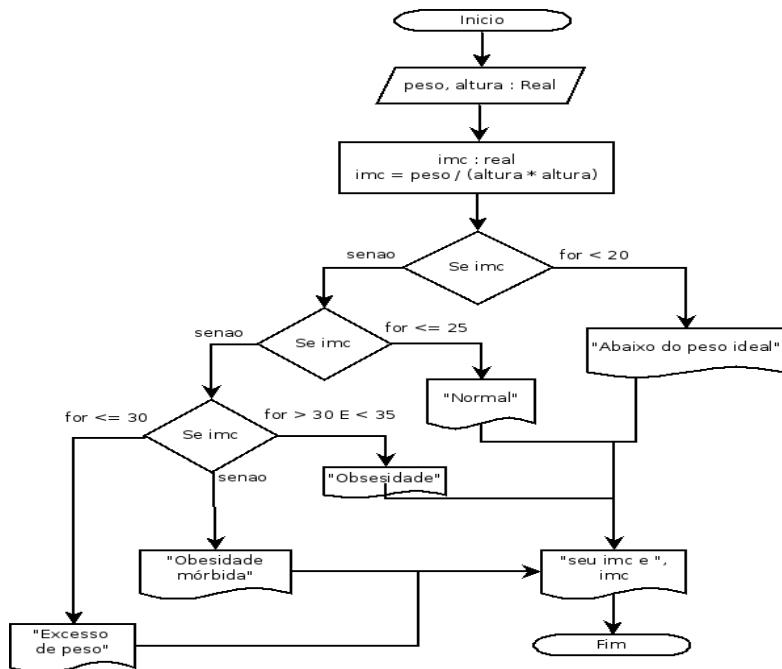
**Objetivo:** Verificar o Índice de Massa Corporal de um indivíduo;

**Entrada:** Valores de peso e altura;

**Processamento:** Calcular o IMC, verificar em que faixa o IMC se encontra;

**Saída:** Mostrar o IMC e a faixa em que a pessoa se encontra;

Figura 15 – Fluxograma atividade 5



Fonte: O autor.

```

peso = float(input("Insira seu peso: ")) #receber o valor do peso
altura = float(input("Insira sua altura: ")) #receber o valor da altura

imc = peso/(altura**2) #calcular o imc

if imc < 20: #se o imc for menor que 20
    print("Abaixo do peso ideal")

else: #senao
    if imc <= 25: #se for menor que 25
        print("Seu imc esta normal")
    else: #senao
        if imc <= 30: #se for menor que 30
            print("Excesso de peso")
        elif imc > 30 and imc <= 35: #ou se for maior que 30 e menor que 35
            print("Obesidade")
        else: #ou entao se for maior que isso
            print("Obesidade morbida")

print("Seu imc e: ", imc) #finalmente mostramos o imc

```

Saída:

```

>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo3/exemplo22.py
Insira seu peso: 54
Insira sua altura: 1.62
Seu imc esta normal
Seu imc e:  20.576131687242793
>>>

```

### *Exercícios propostos*

- 1) Elabore um algoritmo que leia o percurso em quilômetros, o tipo de automóvel e informe o consumo estimado de combustível, sabendo que um automóvel do tipo A faz 26 Km com um litro de gasolina, um automóvel do tipo B faz 20 Km e um automóvel do tipo C faz 7 Km.

- 2) Formule um algoritmo que leia cinco números e conte quantos deles são negativos.
- 3) Escreva um algoritmo que recebe uma letra e verifica se é uma vogal ou uma consoante.
- 4) Formule um algoritmo que leia o código do produto e a quantidade desse produto que o cliente está adquirindo, informe os produtos que o cliente comprou, a quantidade e o total a pagar:

Código	Produto	Valor
1	Refrigerante	R\$ 4.50
2	Água Mineral	R\$ 3.00
3	Feijão	R\$ 8.00
4	Arroz	R\$ 7.25

## CAPITULO 4

---

...

### ESTRUTURAS DE REPETIÇÃO

As estruturas de repetição fornecem meios para que possamos repetir determinada parte do algoritmo sem que tenhamos que reescrever os comandos necessários para isso. Ja pensou em um algoritmo que recebe cem números? Ou então mil números? Como você acha que faria isto? Com mil comandos de **input**? Neste capítulo vamos conhecer as estruturas de de repetição, também denominadas como laço de repetição ou loop. Para Ascencio e Campos (2010, p.93) “uma estrutura de repetição é utilizada quando um trecho do algoritmo ou até mesmo o algoritmo inteiro precisa ser repetido” e Leal (2016b, p.82) afirma que “a vantagem da estrutura de repetição é que não precisamos reescrever trechos de código idênticos, reduzindo assim o tamanho do algoritmo”, a mesma autora ainda enfatiza que as estruturas de repetição podem ser de dois tipos: Laços contados e laços condicionais, vamos conhecer cada um deles.

## ESTRUTUTURA FOR

A estrutura **for** em Python é uma instrução do tipo laço contado, que significa que vamos repetir um determinado conjunto de instruções um determinado número de vezes já preestabelecido.

De acordo com Leal (2016a, p. 90) “os laços contados são aqueles que utilizamos quando sabemos previamente quantas vezes o trecho do código precisa ser repetido. Por exemplo, realizar a leitura de 100 números, efetuar o somatório dos

```
nome = input("Insira seu nome: ") #primeiro recebemos o nome

"""queremos mostrar o nome 10 vezes
ao inves de escrever dez vezes print(nome)
podemos simplesmente fazer o seguinte:
"""

for contagem in range(10):
    print(nome)
```

números entre 500 e 700 e outros.” Vamos ver como funciona o laço **for** em Python construindo um algoritmo que escreva seu nome dez vezes na tela:

Primeiro damos a instrução **for** que irá analisar uma variável chamada contagem em um alcance (**range**) de 10 números. Como isso funciona? Bem, a variável contagem somente será utilizada para fazer a contagem do laço, essa contagem se dará após o comando **in** que avaliará quantas

vezes o laço se repetirá através do comando **range()** que recebe um número inteiro. Veja que na instrução **for** também é necessário que se faça a indentação do código que deseja-se repetir.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo23.py =====
Insira seu nome: Bruno
>>>
```

Veja que o laço repete a instrução **print(nome)** o número de vezes que pedimos que fizesse. Vamos exemplificar novamente pedindo ao Python que escreva na tela a tabuada do 2:

```
| for i in range(11): #a variavel i e nosso contador
|   print("2 x ", i, " = ", 2 * i) #perceba como o i ira mudar em cada repeticao
```

Aqui ao invés de nomear a variável contadora com o identificador de contador, demos o nome de **i**, é muito comum em linguagem de programação que as variáveis dos laços de repetição assumam um identificador como este, pois ela somente será utilizada para contar o laço, porém você poderá

inserir o identificador desejar. Nossa **range( )** agora é 11 pois o Python começa a contar do zero, para que o último número a ser calculado seja o 10 precisamos fornecer um número acima para o **range( )**, do contrário ele iria mostrar a tabuada do zero ao nove (tente fazer isso, repita esse algoritmo com **range(10)**). Dentro de nosso **print** temos uma string que diz “2 x ” (repare nos espaços em branco na string, sem eles a escrita na tela deixaria os caracteres colados uns nos outros) então passamos após a vírgula o valor de **i**. Cada vez que o laço **for** executar as instruções em seu bloco de comandos, a variável **i** será incrementada em 1, até que chegue a 11. Então a cada execução a variável **i** assumirá o valor do número de vezes em que o laço foi repetido. Então fornecemos outra string “ = ” e então uma expressão que multiplica o valor de **i** por 2. Veja a saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo24.py ====
2 x  0  =  0
2 x  1  =  2
2 x  2  =  4
2 x  3  =  6
2 x  4  =  8
2 x  5  =  10
2 x  6  =  12
2 x  7  =  14
2 x  8  =  16
2 x  9  =  18
2 x  10  =  20
>>>
```

Observe como o valor de **i** se inicia em zero e a cada

vez que a instrução **print** é executada ela é incrementada, depois do sinal de = temos a multiplicação de i por 2. Tente mudar o número dentro da instrução range para ver como o algoritmo se comporta. Dizemos que a instrução **for** é uma estrutura do tipo laço contado porque sabemos exatamente quantas vezes devemos repetir os comandos, neste caso apresentado o laço deve ser percorrido o número de vezes que for passado para o comando **range( )**. Neste comando **range( )** não precisamos passar um número diretamente, também podemos passar um valor numérico guardado em uma variável<sup>5</sup>. Veja este próximo exemplo:

```
mensagem = input("Insira uma mensagem: ")
repeticoes = int(input("Insira quantas vezes deseja repetir a mensagem: "))

for i in range(repeticoes):
    print(mensagem)
```

Primeiro recebemos uma mensagem do usuário através do comando **input( )**, esta mensagem ficará guardada na variável mensagem, depois recebemos um valor numérico do usuário que é armazenado na variável repeticoes. No laço **for** repetimos a mensagem do usuário o número de vezes que ele informou.

---

<sup>5</sup> Na instrução **range()** também é possível passar o valor mínimo e um máximo separados por vírgula assim **range(10,20)**. Desta forma o contador iniciará em 10 e contará até 20.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo25.py =====
Insira uma mensagem: Python é legal
Insira quantas vezes deseja repetir a mensagem: 5
Python é legal
>>> |
```

Em Python, a instrução **for** também pode ser utilizada para iterar pelos caracteres de uma *string*. Iterar pela *string* quer dizer que o Python vai passar verificando caracter por caracter da *string* que você quer analisar, veja no exemplo a seguir:

```
palavra = 'bola'    #definimos uma string
for letra in palavra: #para cada letra na string
    print(letra)      #escreva a letra na tela
```

A cada repetição do laço **for** o Python vai escrever a letra que ele identificou, uma de cada vez até chegar ao fim da string, quando o Python identificar que já não há mais caracteres para ele verificar o loop se encerra.

```
-- RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/iterarString.py ==
b
o
l
a
>>> |
```

Lembra quando verificamos a existência de um determinado caractere em uma string com a instrução **if**?

Vamos fazer um algoritmo que leia um número de ponto flutuante e caso este tenha sido inserido com vírgula (,) ao invés de ponto (.) o Python substitua os caracteres para nós:

```
entrada = input("Insira sua altura: ") #pedimos uma altura  
  
for caracter in entrada: #para cada caracter na entrada  
    if caracter == ',': #se houver virgula na entrada  
        entrada = float(entrada.replace(',', '.')) #substitua por um ponto  
  
print(entrada) #escreva na tela o novo valor
```

Neste exemplo recebemos uma entrada referente a altura do usuário, é muito comum que o usuário ao inserir um número fracionário digite uma vírgula ao invés de ponto, e se isso ocorrer durante o processamento de um cálculo, ou mesmo durante a conversão para *float*, o Python retornará um erro. Com este algoritmo conseguimos contornar esta situação, recebemos a entrada em forma de *string*, analisamos cada um dos caracteres da *string* e se algum deles for uma vírgula, substituímos a vírgula por um ponto com a instrução **replace()**. A instrução **replace()** recebe dois parâmetros, o primeiro deles é o caractere que queremos remover e o segundo é o caractere que vamos inserir no lugar, desta forma:

```
variavel.replace(caracter_a_substituir, caracter_a_inserir)
```

Também fazemos a conversão de *string* para *float* através da instrução **float()** e atribuímos o resultado da conversão para a entrada. Quando o loop encerrar escrevemos na tela o novo valor da variável entrada que está pronto para ser utilizado em qualquer cálculo de números fracionários:

```
RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/substituirCaractere.py
Insira sua altura: 1,78
1.78
>>>
```

## ESTRUTUTURA WHILE

A instrução **while** é uma estrutura de repetição do tipo condicional, o que significa que os comandos inseridos no bloco de instruções **while** se repetirão até que uma determinada condição se satisfaça. Lembra da instrução **if**? Ela analisa uma condição e caso seja verdadeira executa um bloco de instruções. A instrução **while** funciona de maneira similar, porém ela ficará executando os comandos enquanto a condição for verdadeira. Acompanhe no exemplo:

```
num = 0          #definimos uma variavel num com valor zero
nome = "Python" #definimos uma string qualquer

while num < 5: #enquanto num tiver valor menor que cinco
    print(nome) #escreva a string na tela
    num += 1    #incremente a variavel num
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo26.py =====
Python
Python
Python
Python|
Python
>>>
```

Na estrutura de repetição condicional é muito importante que a variável analisada seja modificada dentro do bloco **while**, nesse caso analisamos a variável num, e dissemos ao Python “Repita essas instruções enquanto num for menor que 5 ok?”, e dentro do bloco **while**, após a instrução **print**, nós temos num += 1, que é equivalente a num = num + 1. Sem esse incremento o loop ficaria repetindo infinitamente a mensagem, pois num seria sempre menor que 5.

Outro adendo importante a se ressaltar a respeito desta estrutura é a inicialização da variável avaliada. A inicialização de uma variável quer dizer que antes de avaliarmos esta variável devemos atribuir um valor a ela. Diferentemente da

estrutura for onde declaramos uma variável contadora no próprio comando, a instrução while precisa que a variável possua um valor inicial atribuído antes de chegar no loop, ou então nosso interpretador Python irá retornar um erro:

```
nome = "wubalubadubdub"  
  
while num < 5:  
    print(nome)
```

Veja no exemplo acima que dissemos ao Python para repetir o conteúdo da variável nome enquanto o valor de *num* for menor que 5. Mas a variável num não foi declarada em lugar algum, então o Python não saberá o que você quer analisar exatamente, devolvendo este erro:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo27.py =====  
Traceback (most recent call last):  
  File "/Users/macbookpro/Desktop/livro/code/capitulo4/exemplo27.py", line 4, in  
    <module>  
      while num < 5:  
NameError: name 'num' is not defined  
>>>
```

Lembra-se do *Traceback*? Sim exatamente, é a forma que o Python tem de dizer que houve um erro no seu algoritmo, nesse caso ele retornou um **NameError**, informando que o identificador num não foi definido. Se você se recordar, já vimos este erro no capítulo 2, é muito comum este tipo de erro

quando tentamos manipular uma variável que não foi declarada ou quando erramos o nome de um identificador.

A estrutura de repetição condicional é caracterizada por repetir uma determinada sequência de comandos indefinidamente (LEAL, 2016a p. 96) , ou seja, não sabemos quantas vezes o loop deve se repetir. Para exemplificar vamos supor que temos que escrever um algoritmo que receba vários números do usuário, e continuará recebendo números até que seja digitado o valor zero (0). Neste algoritmo vamos contar quantos números foram inseridos pelo usuário e mostrar na tela depois de sair do loop. Veja:

```
inseridos = 0 #variavel para contar quantos numeros sera inseridos
num = 1         #variavel para controlar o loop

while num != 0: #o loop vai repetir ate que o usuario digite zero

    num = int(input("Insira um numero ou aperte 0 para sair: ")) #pedimos um num

    if num != 0: #caso a entrada seja diferente de zero aumentamos o contador
        inseridos += 1

print("Voce inseriu ", inseridos, " numeros") #escrevemos na tela
```

Perceba que nesse loop while não temos um incremento para sair do loop, mas uma condição bem específica que é tratada dentro do loop, nesse caso, a entrada do usuário. Toda vez que o usuário inserir uma entrada diferente de zero

fazemos uma verificação **if** num != 0 (o sinal != significa diferente), e se for diferente de zero incrementamos o contador. Quando o usuário inserir 0 o loop while irá realizar novamente a análise, nesse ponto o Python irá detectar que num é igual a zero, então ele não executará o **while**. Depois do loop, temos outra instrução informando quantos números o usuário inseriu, esta instrução está fora do loop como podemos ver pela indentação. A saída deve ser parecida com esta:

```
>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo28.py =====
Insira um numero ou aperte 0 para sair: 4
Insira um numero ou aperte 0 para sair: 6
Insira um numero ou aperte 0 para sair: 4
Insira um numero ou aperte 0 para sair: 8
Insira um numero ou aperte 0 para sair: 1
Insira um numero ou aperte 0 para sair: 9079
Insira um numero ou aperte 0 para sair: 012
Insira um numero ou aperte 0 para sair: 0
Voce inseriu 7 numeros
>>>
```

Podemos criar um laço while diretamente com booleanos (**True** ou **False**) e para saímos do loop basta utilizar a instrução **break**:

```
while True: #enquanto for verdade

    nome = input("Insira seu nome ") #solicite um nome

    if nome == "fim": #se o nome for fim
        break #sairmos do loop
    else:
        print("Oi ", nome) #senao diga oi

    print("Fim do loop") #esta instrucao so acontece quando o loop encerra
```

Criamos a condição True no laço que continuará sempre a executar, dentro do loop criamos uma condição de saída que quando satisfeita freia o loop e parte para a próxima instrução depois do laço, neste caso uma instrução **print( )** informando que o laço acabou, veja a saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo28-1.py ====
Insira seu nome Bruno
Oi Bruno
Insira seu nome Joana
Oi Joana
Insira seu nome Andre
Oi Andre
Insira seu nome fim
Fim do loop
>>> |
```

## ESTRUTUTURAS DE REPETIÇÃO ANINHADAS

Assim como nas estruturas condicionais, podemos aninhar ou encadear as estruturas de repetição. De acordo com Manzano e Oliveira (1997) *apud* Leal (2016a) “Não existem regras para o encadeamento das estruturas de repetição. De modo que você precisa conhecer cada uma delas para saber quando é conveniente encadeá-las, quais devem ser utilizadas e como”, porém com o tempo e prática você saberá exatamente que tipo de instrução utilizar para elaborar seus algoritmos.

Pode-se encadear qualquer tipo de estrutura de

repetição dentro da outra. Pode-se inserir **while** dentro de **while**, **for** dentro de **for**, **while** dentro de **for**, **for** dentro de **while**, da forma como seu algoritmo exigir, mas é importante ressaltar que a indentação esteja correta para cada bloco de repetição para que você não se depare com execuções inesperadas do seu código. Vamos a um exemplo:

```
loop = 's' #inicializamos a condicao do loop while

while loop != 'n': #enquanto loop for diferente de 'n'

    soma = 0      #inicializamos a soma

    #somar quantos numeros o usuario quiser
    maximo = int(input("Quantos numeros voce deseja somar? "))

    for i in range(maximo): #loop para somar a quantidade desejada
        entrada = int(input("Insira um numero para somar: "))
        soma += entrada

    #aqui ja estamos fora do loop for, mas dentro do loop while ainda
    print("O resultado da soma e: ", soma)

    #verificamos se o usuario quer continuar no programa
    loop = input("Gostaria de realizar outra soma? s/n ")
```

Neste exemplo temos dois níveis de repetição, o mais externo é o loop while, ele será o laço principal do programa e continuará executando até que o usuário entre com o valor correspondente para encerrar o laço. Dentro deste laço principal nós temos um segundo laço interno que irá repetir de acordo com o número de somas que o usuário deseja realizar. Ao completar seus ciclos o laço interno se encerrará dando

continuidade ao programa, porém o programa continua dentro do laço principal que continuará a se repetir até que o valor necessário para interromper sua execução seja fornecido pelo usuário.

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exemplo29.py ====
Quantos numeros voce deseja somar? 5
Insira um numero para somar: 2
Insira um numero para somar: 3
Insira um numero para somar: 2
Insira um numero para somar: 5
Insira um numero para somar: 1
O resultado da soma e: 13
Gostaria de realizar outra soma? s/n n
>>> |
```

Você pode aninhar quantas estruturas de repetição forem necessárias em seu algoritmo assim como as estruturas condicionais, assim como pode inserir estruturas de repetição em estruturas condicionais e vice-versa. Veja que no exemplo anterior poderia-se ter utilizado o laço **while** ao invés do **for** e o algoritmo comportaria-se da mesma forma, experimente modificar o algoritmo anterior e utilizar o loop **while** internamente no lugar do loop **for**.

## EXERCICIOS ELABORADOS

- 6) Escreva um algoritmo que apresente todos os números divisíveis por 5 que sejam menores que 200.

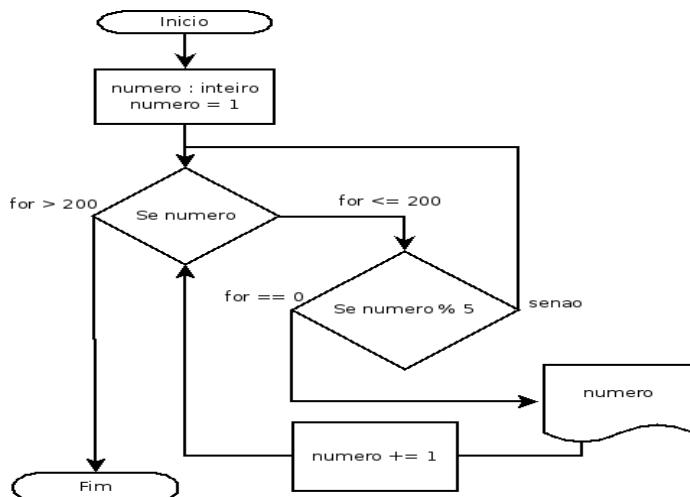
**Problema a ser resolvido:** Calcular todos os números divisíveis por 5 que sejam menores ou igual a 200;

**Dados de entrada:** nenhum;

**Processamento:** calcular de 1 a 200 dividindo por 5, os resultados iguais a 0 devem ser mostrados na tela;

**Saída:** Exibir os divisíveis por 5;

Figura 16 – Fluxograma atividade 6



Fonte: O autor.

Código-fonte:

```
for numero in range(1, 201): #para cada numero de 1 a 200
    if numero % 5 == 0:      #se o numero for divisivel por 5
        print(numero)       #escreva esse numero na tela
    print('sao divisiveis por 5')#escreva uma mensagem amigavel no final
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exercicio6.py ====
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
125
130
135
140
145
150
155
160
165
170
175
180
185
190
195
200
sao divisiveis por 5
>>> |
```

---

Perceba que utilizamos 201 como alcance máximo para nosso range( ), pois se passássemos 200 como parâmetro o Python ia contar do 0 ao 199 e não dividiríamos o 200 também. Tente fazer isso para ver a diferença.

Observe também que no fluxograma não temos exatamente um símbolo para representar o laço de repetição, o

loop se baseia nas avaliações condicionais, estas orientam o fluxo de dados do algoritmo até que sejam satisfeitas.

- 7) Escreva um algoritmo que receba a idade e o estado civil de várias pessoas e imprima a quantidade de pessoas casadas, solteiras, separadas e viúvas. O algoritmo finaliza quando for informado o valor zero para idade.

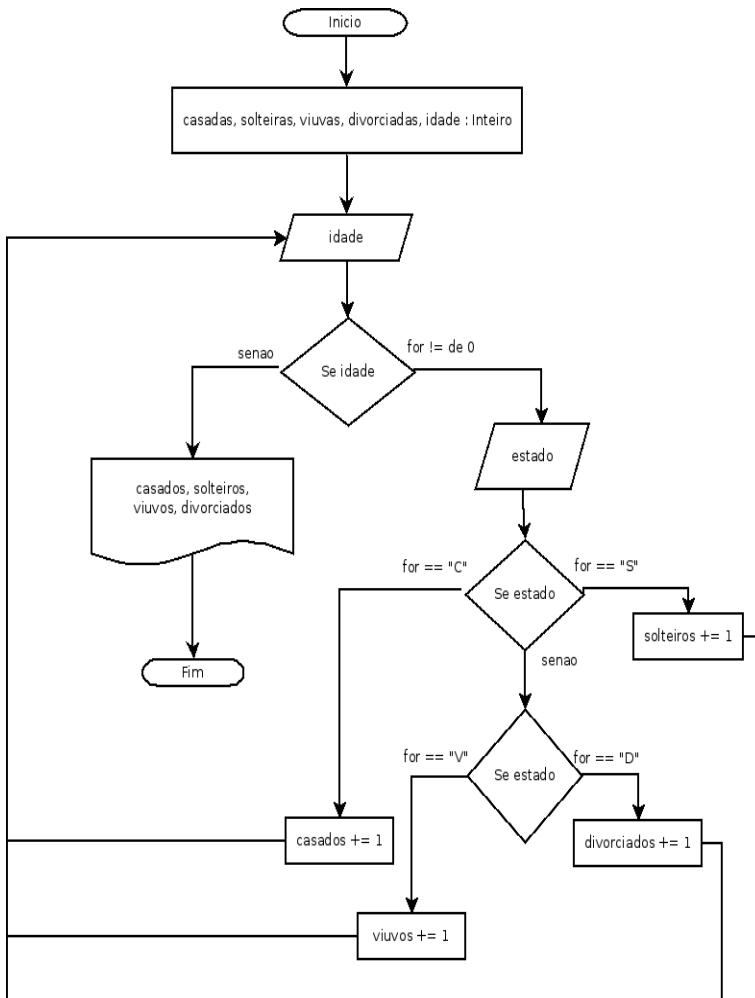
**Problema a ser resolvido:** Verificar e contabilizar o estado civil de várias pessoas;

**Dados de entrada:** idade, estado civil;

**Processamento:** calcular o número de pessoas solteiras, casadas, separadas e viúvas;

**Saída:** Informar a quantidade de pessoas solteiras, casadas, separadas e viúvas;

Figura 17- Fluxograma atividade 7



Fonte: O autor.

## Código-fonte:

```
casadas = 0      #primeiro inicializamos as variaveis
solteiras = 0
viuvas = 0
divorciadas = 0

idade = int(input("Insira sua idade ou 0 para sair: "))

#o programa só funciona se a idade for maior que zero

while idade != 0: #loop externo
    print("Informe o estado civil") #informe o usuário para que insira o estado civil

    estado = "" #inicializamos a variável do loop interno
    while estado != 'C' and estado != 'S' and estado != 'D' and estado != 'V':
        #veja na string abaixo o uso de \n isso diz ao Python para pular uma linha
        estado = input("C: Casado \nS: Solteiro \nD: Divorciado \nV: Viuva \n")

        #.upper() transforma os caracteres da string em CAIXA ALTA
        estado = estado.upper()
    #Aqui acaba o loop interno

    if estado == 'C':    #Ainda dentro do loop interno fazemos a verificação
        casadas += 1
    elif estado == 'S':
        solteiras += 1
    elif estado == 'D':
        divorciadas += 1
    elif estado == 'V':
        viuvas += 1
    else:
        print("Algo errado")

    idade = int(input("Insira sua idade ou 0 para sair: ")) #pedimos a idade mais uma vez

#fim do loop externo
print("O numero de solteiros é: ", solteiras) #informamos os resultados
print("O numero de casados é: ", casadas)
print("O numero de divorciados é: ", divorciadas)
print("O numero de viúvos é: ", viuvas)
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo4/exercicio7.py ====
Insira sua idade: 26
Informe o estado civil
C: Casado
S: Solteiro
D: Divorciado
V: Viuwo
s
Insira sua idade: 45
Informe o estado civil
C: Casado
S: Solteiro
D: Divorciado
V: Viuwo
d
Insira sua idade: 32
Informe o estado civil
C: Casado
S: Solteiro
D: Divorciado
V: Viuwo
c
Insira sua idade: 89
Informe o estado civil
C: Casado
S: Solteiro
D: Divorciado
V: Viuwo
v
Insira sua idade: 12
Informe o estado civil
C: Casado
S: Solteiro
D: Divorciado
V: Viuwo
s
Insira sua idade: 0
O numero de solteiros e: 2
O numero de casados e: 1
O numero de divorciados e: 1
O numero de viuvos e: 1
>>>
```

Perceba que neste algoritmo utilizamos alguns macetes. Primeiro perceba a utilização dos caracteres \n dentro da string no **input()**. Estes caracteres servem para que o interpretador pule uma linha, toda vez que o Python se deparar com \n

dentro de uma *string* ele irá fazer a quebra de uma linha. Outro apetrecho interessante é o método **upper()**. Ao trabalhar com *strings* podemos chamar este método adicionando **.upper()** ao final da variável contendo a *string* para que as letras fiquem maiúsculas, desta forma podemos garantir que, se caso o usuário insira um 's' minúsculo ele seja logo convertido para maiúsculo, pois como você deve se lembrar o Python é *case sensitive* o que o faz diferenciar letras maiúsculas e minúsculas. O Python também conhece os métodos **.lower()** para deixar as letras minúsculas e o método **.capitalize()** para deixar somente a primeira letra maiúscula e as outras minúsculas. Vamos conhecer mais coisas legais sobre *strings* no decorrer do livro.

### *Exercícios propostos:*

- 1) Construa um algoritmo que leia números inteiros até que seja inserido um número negativo. Ao final, informe a média dos números, o maior e o menor valor inserido.
- 2) Elabore um algoritmo que imprima todas as tabuadas do 1 ao 10.
- 3) Construa um algoritmo que receba o nome de uma pessoa e sua idade, o algoritmo deve escrever na tela o nome da pessoa em letras maiúsculas o número de vezes equivalente

à sua idade.

- 4) Escreva um algoritmo que simule um cardápio, cada item do cardápio tem um código que vai de 1 a 6. Mostre os produtos do cardápio na tela, o código do item e o valor, o usuário poderá fazer pedidos pelo código do item até que seja inserido o valor zero. No final escreva na tela os itens que o usuário escolheu e o valor a pagar.

## CAPITULO 5

---

...

### ESTRUTURAS DE DADOS

As estruturas de dados em lógica de programação são responsáveis por permitir que agrupemos diversos tipos de dados em uma única variável, em outras linguagens é comum que estas estruturas sejam divididas em homogêneas e heterogêneas, pois há estruturas específicas para agrupar um único tipo de dado (como uma lista enorme de números inteiros), porém no Python é possível misturar diferentes tipos em uma única lista, desta forma pode-se dizer que as estruturas de dados em Python são praticamente todas heterogêneas, mas nada lhe impede de criar listas somente de inteiros ou somente de literais.

Até agora estávamos armazenando os valores em variáveis separadas, mas existem momentos em que precisamos armazenar muitos dados, o que nos exigiria a declaração de inúmeras variáveis. “Suponha, por exemplo, o caso de um treino de classificação de uma corrida de Fórmula

1, em que é necessário verificar os tempos obtidos por todos os pilotos para avaliar qual será o primeiro no grid de largada. Para fazer essa ordenação, é necessário armazenar o tempo de todos os pilotos e, depois, realizar a ordenação desses tempos.”(PUGA; RISSETTI, 2010 p. 83), já pensou quantas variáveis precisaríamos criar? Uma para cada piloto, mais uma para cada tempo, isso exigiria muito consumo de memória do computador, e seria desnecessário já que possuímos as estruturas de dados para agrupar todos estes valores. Vamos conhecer as estruturas de dados em Python.

## ESTRUTURAS DE DADOS UNIDIMENSIONAIS

Em lógica de programação, as estruturas unidimensionais são popularmente conhecidas como vetores, vetor é uma estrutura de dados unidimensional (ASCENCIO; CAMPOS, 2010), unidimensional porque ela possui apenas uma dimensão linear onde os dados serão listados um após o outro em sequência.

Em Python estas estruturas são denominadas listas, e assim como uma lista comum os itens serão incluídos e identificados por um índice que corresponde à sua localização

na lista.

Uma lista é uma coleção de itens em uma ordem em particular. Podemos criar uma lista que inclua as letras do alfabeto, os dígitos de 0 a 9 ou os nomes de todas as pessoas da sua família. Você pode colocar qualquer informação que quiser em uma lista, e os itens de sua lista não precisam estar relacionados de nenhum modo em particular. (MATTHES, 2016 p. 70)

Vamos exemplificar isto com uma situação bem comum. Suponha que sua mãe pediu para que você fosse ao mercado e comprasse bananas, queijo, trigo e uma lata de azeite. Logo você pega um pedaço de papel e anota:

- 1 – Banana;
- 2 – Queijo;
- 3 – Trigo;
- 4 – Azeite;

Vamos fazer a mesma coisa lista em Python agora:

```
lista = ["banana", "queijo", "trigo", "Azeite"] #criamos nossa lista  
print(lista) #escrevemos o conteúdo da lista
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo30.py =====  
['banana', 'queijo', 'trigo', 'Azeite']  
>>> |
```

A sintaxe para criar uma lista em Python é:

$$\text{identificador} = [\text{conteúdo}]$$

Em Python as listas são definidas por um par de [ ], e os

itens que estiverem dentro dos colchetes pertencem a lista, mais de um item deve ser separado por vírgula. Veja no exemplo anterior que guardamos quatro valores literais dentro de uma única variável e quando chamamos esta variável na instrução **print( )** o Python nos diz todos os valores que existem nessa variável. Nós podemos verificar quantos itens existem em uma lista com a instrução **len( )**.

```
lista = ["banana", "queijo", "trigo", "Azeite"] #criamos nossa lista  
print(len(lista)) #escrevemos o numero de itens na lista
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo30.py ====  
4  
=>>
```

Desta forma ao trabalharmos com listas muito extensas em que precisamos verificar a quantidade de itens nesta lista basta passarmos a lista como parâmetro para a instrução **len( )**.

Agora vamos supor que antes de você sair para o mercado sua mãe lhe diz que esqueceu de dizer que é pra você trazer cenouras também, então você precisa adicionar mais um item em sua lista, para fazer isto no Python basta utilizar o comando **.append( )** com o nome do item a ser adicionado

entre os parênteses, desta forma:

```
lista = ["banana", "queijo", "trigo", "Azeite"] #lista inicial
print(len(lista)) #verifique o tamanho da lista
print(lista)      #verifique os itens na lista

lista.append("cenoura") #adicone a cenoura no fim da lista

print(len(lista)) #verifique novamente o tamanho da lista
print(lista)      #verifique novamente os itens na lista
```

Utilizamos as instruções `len()` para verificar que a lista aumentou de tamanho, escrevemos antes e depois de adicionar o item para verificar que as alterações foram feitas com sucesso. Agora você já pode ir ao mercado comprar as coisas para sua mãe.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo31.py =====
4
['banana', 'queijo', 'trigo', 'Azeite']
5
['banana', 'queijo', 'trigo', 'Azeite', 'cenoura']
>>>
```

Ao chegar no mercado você colocou em sua cestinha a banana e o azeite, e resolveu riscar da sua lista os itens que você já pegou para não acabar se perdendo. Para remover um item conhecido de sua lista basta escrever o item a ser removido como parâmetro na instrução `.remove()`, desta forma:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #lista  
lista.remove("banana") #removemos a banana  
lista.remove("azeite") #removemos o azeite  
  
print(lista) #verificamos quais itens ainda estao na lista
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo32.py =====  
['queijo', 'trigo', 'cenoura']  
>>> |
```

No começo deste capítulo foi dito que as estruturas de dados em Python são heterogêneas, o que significa que podemos misturar tipos de dados em uma única lista. Nos exemplos anteriores fizemos uma lista de literais, mas é possível inserir dados de tipos distintos em uma mesma lista Python. Veja neste exemplo:

```
meus_dados = [] #criamos uma lista vazia  
  
nome = input("Insira seu nome: ") #pedimos que o usuário escreva o nome  
meus_dados.append(nome) #inserimos o nome na lista  
  
idade = int(input("Insira sua idade: ")) #pedimos a idade  
meus_dados.append(idade) #inserimos a idade na lista  
  
altura = float(input("Insira sua altura: ")) #pedimos a altura  
meus_dados.append(altura) #inserimos a altura na lista  
  
print(meus_dados) #escrevemos o conteúdo da lista
```

Primeiro criamos uma lista vazia com o identificador meus\_dados, para criar uma lista vazia basta atribuir os colchetes vazios [], então fazemos as solicitações de entrada de três tipos de dados diferentes um literal (nome) um inteiro (idade) e um real (altura), adicionando cada um deles em nossa lista, no final do algoritmo escrevemos o conteúdo da lista.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo33.py =====
Insira seu nome: Bruno
Insira sua idade: 26
Insira sua altura: 1.63
['Bruno', 26, 1.63]
>>> |
```

Veja como as listas em Python suportam diferentes tipos de dados, por este motivo são caracterizadas como heterogêneas, diferentemente dos vetores em algumas linguagens de programação que somente aceitam dados de um único tipo. Dissemos também no início deste capítulo que os itens da lista são identificados por um índice, assim podemos pedir para o Python um determinado elemento da lista pelo índice. Se você se lembrar bem, o Python começa a contar do zero, isso também se aplica as listas, logo o primeiro índice da lista é 0, e todos os elementos na sequência seguem a numeração do índice. A nossa lista de compras dos primeiros exemplos possui no índice 0 o item “banana”, no índice 1 o

item “queijo”, no índice 2 o item “trigo” e assim por diante. Para que o Python identifique o item pelo índice basta escrever o nome da lista seguido pelo índice entre colchetes, desta forma:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista  
print(lista[0]) #escreva o item do indice 0  
print(lista[4]) #escreva o item do indice 4
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo34.py =====  
banana  
cenoura  
>>> |
```

Perceba que apesar de nossa lista possuir 5 elementos, o último elemento da lista é representado pelo índice 4, é bem comum o Python retornar um erro ao passarmos um índice inexistente para ele, necessitando um pouco de cuidado ao solicitar os elementos pelo índice. No exemplo anterior pedimos que o Python escrevesse o item da posição 0 e da posição 4, e caso não existisse uma posição 4, o Python retornaria um erro:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista  
print(lista[0]) #escreva o item do indice 0  
lista.remove("cenoura")  
print(lista[4]) #escreva o item do indice 4
```

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo35.py =====
banana
Traceback (most recent call last):
  File "/Users/macbookpro/Desktop/livro/code/capitulo5/exemplo35.py", line 7, in
    <module>
    print(lista[4]) #escreva o item do indice 4
IndexError: list index out of range
>>> |
```

Neste exemplo, ao pedirmos que o Python escreva a posição 0 ocorre tudo bem, ele escreve “banana”, mas quando o algoritmo chegar a instrução que solicita ao Python para escrever a posição 4 ele retornará um *Traceback* do tipo **IndexError**, pois antes desta instrução havíamos removido o item da posição 4, e como não haviam mais itens na sequência, a posição do índice 4 ficou inexistente. Ao tentar analisar isto o Python tenta nos dizer “ei programador esse índice não existe”.

Quando você não tiver certeza de quantos elementos possui a lista, mas mesmo assim quiser verificar o último elemento da lista, basta utilizar [-1] como índice, e o Python irá buscar o último item da lista, independente de quantos elementos ela possua:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista
lista.remove("cenoura")
print(lista[-1]) #escreva o ultimo item da lista
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo36.py ====
azeite
>>> |
```

Através dos índices também é possível alterar o conteúdo da lista fazendo uma nova atribuição ao índice, isto irá sobrescrever o elemento armazenado naquela posição, veja:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista
lista[0] = 'sorvete'
lista[1] = 'chocolate'
lista[2] = 'pizza'
lista[3] = 'coxinha'
lista[4] = 'pao de queijo'

print(lista)
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo36-1.py ===
['sorvete', 'chocolate', 'pizza', 'coxinha', 'pao de queijo']
>>> |
```

Modificamos todas as posições da lista e atribuímos novos valores a cada posição do índice, essa lista parece ter ficado bem mais gostosa, mas infelizmente não é o que sua mãe pediu pra você comprar.

Uma das formas mais comum de se trabalhar com listas é iterar através do laço **for**, assim podemos analisar item por item da lista, veja:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista  
for item in lista: #para cada item na lista  
    print(item)    #escreva o item na tela
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo37.py =====  
banana  
queijo  
trigo  
azeite  
cenoura  
>>> |
```

Com o loop percorremos cada um dos elementos e vamos escrevendo-o na tela. O **for** é o mais comum de se utilizar porém é perfeitamente possível fazer o mesmo com o loop **while**, você saberá quando utilizar um ou outro no decorrer de seus algoritmos. Da mesma forma que podemos percorrer a lista com um laço, também é possível verificar se um determinado item existe na lista com uma estrutura condicional, como no exemplo a seguir:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista  
  
if 'queijo' in lista: #se tiver 'queijo' na lista  
    print("Queijo esta na lista") #diga que ha queijo na lista
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo38.py =====  
Queijo esta na lista  
>>> |
```

Viu como é fácil trabalhar com listas em Python? Vamos ver mais uma função bacana para se utilizar nas listas, o método **sort( )**. Com este comando podemos ordenar nossa lista de forma organizada, no exemplo a seguir vamos organizar a nossa lista de compras em ordem alfabética:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista
lista.sort() #organiza a lista em ordem alfabetica
print(lista) #escreva a lista
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo39.py =====
['azeite', 'banana', 'cenoura', 'queijo', 'trigo']
>>> |
```

É possível fazer o inverso e organizar os itens da lista de trás para frente com o método **.reverse( )**, assim a lista será invertida ao contrário, como neste exemplo:

```
lista = ["banana", "queijo", "trigo", "azeite", "cenoura"] #nossa lista
lista.reverse() #inverte a lista ao contrario
print(lista) #escreva a lista
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo40.py =====
['cenoura', 'azeite', 'trigo', 'queijo', 'banana']
>>>
```

Tanto o método **sort( )** como o **reverse( )** também funcionam com listas numéricas, se o conteúdo da sua lista for

composto de diversos números aleatórios desordenados, basta chamar o método **sort( )** para organiza-los de forma crescente, e caso queira ordena-los em forma decrescente basta chamar o método **sort( )** depois o método **reverse( )**. Tente fazer isto!

Existe uma outra forma de criar uma lista, através da instrução **list( )** passando como parâmetro a instrução **range( )** contendo o tamanho desejado para sua lista, desta forma:

```
minha_lista = list(range(10)) #crie uma lista com 10 posicoes  
print(minha_lista) #mostre o conteudo da lista
```

A instrução **list()** combinada com a instrução **range( )** criam uma lista de números inteiros automaticamente, lembrando que o Python começa a contar do zero, sabendo que nosso range é 10 a lista deverá conter uma sequência de 0 a 9:

```
== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo40-1.py ==  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> |
```

Uma outra forma peculiar, e bem eficiente de se criar uma lista é utilizando um loop **for** dentro da lista. Isso em Python é chamado *list comprehension*, muito em breve você estará familiarizado com estes conceitos, veja um exemplo:

```
minha_lista = [numeros for numeros in range(10)] #crie uma lista com 10 posicoes  
print(minha_lista) #mostre o conteudo da lista
```

O resultado seria o mesmo:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo40-2.py ====
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> |
```

Você se recorda do algoritmo de soma? Bom o Python tem uma função embutida chamada **sum()**, que faz exatamente a mesma coisa, soma os números entre parênteses, mas uma característica interessante desta instrução é que ela pode receber uma lista de números e retorna como resultado a soma total de todos os números da lista, veja:

```
lista = [2, 3, 5] #lista de numeros
total = sum(lista) #passamos a lista para a instrucao sum()
print(total)
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo40-3.py ====
10
>>>
```

## ESTRUTURAS DE DADOS MULTIDIMENSIONAIS

Uma matriz é uma estrutura de dados multidimensional, o que quer dizer que ela tem várias dimensões, ou seja uma lista que contém outras listas. Até

agora vimos vetores unidimensionais, ou seja, listas com elementos singulares em sequência, já as matrizes podem conter outras listas dando uma característica multidimensional. Para exemplificar melhor, vamos dizer que você estabeleceu uma lista de afazeres do seu dia:

1 – Ir na padaria:

- 1.1 – Comprar pão;
- 1.2 - Comprar manteiga;
- 1.3 Comprar queijo;

2 – Passar na autopeças:

- 2.1 – Calibrar os pneus do carro;
- 2.2 Trocar o óleo;

3 – Passar na farmácia:

- 3.1 – Comprar remédio;
- 3.2 - Pesar-se na balança;

4 – Voltar pra casa:

- 4.1 – Fazer café;
- 4.2 - Tomar o remédio;
- 4.3 - Limpar a casa;
- 4.4 - Ler um livro

Muitas vezes nossas tarefas são compostas de outras listas de tarefas, sendo assim nossa lista de atividades do dia é multidimensional. Em Python poderíamos definir essa lista desta forma:

```
afazeres = [ #abrimos uma lista
    ['comprar pao', 'comprar manteiga', 'comprar queijo'], #na padaria
    ['calibrar pneu', 'trocar oleo'], #no mecanico
    ['comprar remedio', 'pesar na balanca'], #na farmacia
    ['fazer cafe', 'tomar remedio', 'limpar a casa', 'ler um livro'] #em casa
] #fechamos a lista

for lista in afazeres:
    print(lista)
```

Utilizamos um laço for para percorrer a primeira dimensão da nossa lista, para cada elemento da lista afazeres será exibido na tela, como o conteúdo de afazeres são quatro listas:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo41.py =====
['comprar pao', 'comprar manteiga', 'comprar queijo']
['calibrar pneu', 'trocar oleo']
['comprar remedio', 'pesar na balanca']
['fazer cafe', 'tomar remedio', 'limpar a casa', 'ler um livro']
>>> |
```

---

Para referenciar uma um item da lista interna pelo índice utilizamos dois pares de colchetes passando o índice da lista a ser verificada no primeiro e o índice do elemento no segundo:

```
afazeres = [ #abrimos uma lista
    ['comprar pao', 'comprar manteiga', 'comprar queijo'], #na padaria
    ['calibrar pneu', 'trocar oleo'], #no mecanico
    ['comprar remedio', 'pesar na balanca'], #na farmacia
    ['fazer cafe', 'tomar remedio', 'limpar a casa', 'ler um livro'] #em casa
] #fechamos a lista

print(afazeres[0][0]) #mostra o primeiro item da primeira lista
print(afazeres[-1][2]) # mostra o terceiro item da ultima lista
```

Saída:

```
=====
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo42.py =====
comprar pao
limpar a casa
>>> |
```

Você pode aninhar quantas listas quiser desde que tenha conhecimento da complexidade que estará lidando. Cada linha aninhada será uma dimensão na sua estrutura de dados. Vamos dizer que você tem uma lista que contém uma lista que contém uma lista que contém uma lista...

Até buguei com tantas listas dentro de listas...

```

matriz = ['1 dimensao da matriz', #abrimos uma lista
          ['2 dimensao da matriz',#abrimos uma lista dentro da lista
           ['3 dimensao da matriz',#abrimos uma lista na lista da lista
            ['4 dimensao da matriz']]#abrimos uma lista na lista da lista na lista
          ]]] #feche todas as listas

#vamos acessar as listas pelo indice

print(matriz[0]) #acesso a primeira dimensao da matriz
print(matriz[1][0]) #acesso a segunda dimensao da matriz
print(matriz[1][1][0]) #acesso a terceira dimensao da matriz
print(matriz[1][1][1][0]) #acesso a quarta dimensao da matriz

```

Observe atentamente este código para acessar as camadas mais internas da matriz multidimensional. Para cada camada mais profunda um novo índice deve ser utilizado entre colchetes para acessar o elemento existente. Neste exemplo criamos listas dentro de listas onde o primeiro item de cada lista é uma string dizendo em qual dimensão da lista estamos, este elemento pode ser verificado pelo índice 0 desta lista. O segundo elemento de cada lista é uma nova lista. Veja a saída:

```

===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo43.py =====
1 dimensao da matriz
2 dimensao da matriz
3 dimensao da matriz
4 dimensao da matriz
>>>

```

Recorda-se de como criar uma lista com *list comprehension*? Podemos fazer isto para criar uma matriz bidimensional, veja:

```

#vamos criar uma lista com 3 listas de cinco numeros de 0 a 4
matriz = [[numeros for numeros in range(5)] for numeros in range(3)]

print("A matriz:")
print(matriz) #podemos acessar a matriz toda

print("As listas:")
for listas in matriz: #ou acessar cada lista
    print(listas)

print("Os numeros:") #ou acessar cada dado de cada lista
for lista in matriz:
    for numero in lista:
        print(numero)

#ou acessar pelo indice com ja vimos anteriormente

```

Saída:

```

== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exercicio43-1.py ==
A matriz:
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
As listas:
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
Os numeros:
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
>>> |

```

Você pode pensar em uma matriz multidimensional como se fosse uma tabela com linhas e colunas, considerando a

matriz anterior com 3 listas de 5 inteiros, poderíamos considerar as listas como colunas e as posições dos itens nas listas as linhas:

Figura 18 – Tabela de matriz bidimensional

0	0	0	
1	1	1	
2	2	2	
3	3	3	
4	4	4	

↓                    ↓                    ↓

matriz[3][0]      matriz[1][2]      matriz[2][0]

Fonte: O autor.

Para exemplificar vamos criar uma matriz representando o mês de Junho de 2017, nesta matriz haverá sete listas representando os dias da semana, e nas listas internas os dias do mês que cairão naquele dia da semana:

```
junho = [
    [4, 11, 18, 25],      #domingos
    [5, 12, 19, 26],      #segundas
    [6, 13, 20, 27],      #tercas
    [7, 14, 21, 28],      #quartas
    [1, 8, 15, 22, 29],   #quintas
    [2, 9, 16, 23, 30],   #sextas
    [3, 10, 17, 24]       #sabados
]

print('Quinta dia', junho[4][2])
```

Saída:

```
== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo43-2.py ==
Quinta dia 15
>>> |
```

Figura 19 – Matriz representativa do mês de Junho de 2017

junho de 2017							< > Hoje	
dom	seg	ter	qua	qui	sex	sáb		
28	29	30	31	1 de jun	2	3		
4	5	6	7	8	9	10		
11	12	13	14	15	16	17		
18	19	20	21	22	23	24		
25	26	27	28	29	30	1 de jul		

Arrows point to specific cells:

- An arrow points down from the date "28" to the label "junho[0][0]".
- An arrow points down from the date "11" to the label "junho[2][0]".
- An arrow points down from the date "1" (labeled "1 de jun") to the label "junho[4][2]".
- An arrow points down from the date "30" to the label "junho[5][4]".

Fonte: O autor

Agora pense um pouco e tente responder, quais dias seriam os dias representados por:

junho[3][3], junho[1][2], junho[6][3], junho[0][2]

Agora que você conhece sobre as listas e os índices, vamos fazer uma rápida abordagem sobre os índices em strings. Os dados do tipo literal (strings) também podem ser analisados por meio de índices da mesma forma como nas listas, veja:

```
nome = input("Insira seu nome: ") #solicita um nome  
  
print(nome[0]) #escreve o primeiro caractere do nome  
print(nome[2]) #escreve o terceiro caractere do nome  
print(nome[-1]) #escreve o ultimo caractere do nome  
print(nome[-2]) #escreve o penultimo caractere do nome
```

Saída:

```
== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo43-3.py ==  
Insira seu nome: bruno  
b  
u  
o  
n  
>>> |
```

O *slicing* (fatiamento) é uma técnica muito prática e comumente utilizada em Python para trabalhar com strings, podemos fatiar uma palavra pelo seu índice e depois concatená-las novamente para formar outras palavras, veja neste próximo exemplo, vamos pedir uma palavra par ao usuário e substituir a primeira letra pela última letra.

```
palavra = input("Insira uma palavra: ") #pedimos uma palavra  
palavra = palavra[-1] + palavra[1:-1] + palavra[0] #slicing  
print(palavra) #escreva a palavra
```

Começamos pedindo uma palavra ao usuário que ficara guardada na variável palavra. Então realizamos o *slicing* para substituir as letras. Nós queremos a última letra no início, então

vamos pegá-la pelo índice palavra[-1] vamos concatenar (somar as strings) com a parte da palavra que não contém nem a primeira nem a última letra, então vamos utilizar o índice [1:-1]. Quando utilizamos o sinal de : estamos dizendo ao Python “Ei queremos somente os caracteres que estão entre a posição 1 e -1 desta string ok?”, então concatenamos no final o primeiro caractere identificado pelo índice [0], como resultado obtemos a palavra com primeira letra substituída pela última:

```
--- RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo43-4.py ---
Insira uma palavra: Bola
aolB
>>> |
```

## TUPLAS

As tuplas são como as listas, são estruturas de dados onde podemos agrupar diversos tipos de dados, porém a tupla se caracteriza por ser imutável, o que quer dizer que, uma vez definida, os valores da tupla não poderão ser alterados. Depois de declarada, não podemos adicionar nem remover dados à tupla. Para criar uma tupla em Python basta definirmos um identificador e atribuir os valores entre parênteses e separados por vírgula:

```
tupla = ('valor literal', 45, -12, 5.67) #declaracao da tupla  
print(tupla) #informe o conteudo da tupla
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo44.py ====  
('valor literal', 45, -12, 5.67)  
>>> |
```

Podemos acessar os valores da tupla pelo índice da mesma forma como acessamos nas listas. A utilização das tuplas está intimamente relacionada a definição de variáveis constantes, ou seja, valores que não poderão ser modificados no decorrer do algoritmo, como por exemplo a definição de uma altura e largura fixa para uma janela em um programa de computador.

```
TAMANHO = (1280, 720)  
  
print("Altura: ", TAMANHO[0])  
print("Largura: ", TAMANHO[1])
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo44.py ====  
Altura: 1280  
Largura: 720  
>>> |
```

Não podemos modificar os valores da tupla, mas é possível sobrescrever a tupla com novos valores:

```
TAMANHO = (1280, 720)

print("Dimensoes originais")
for dimensao in TAMANHO: #para cada valor na tupla
    print(dimensao)          #escreva o valor

TAMANHO = (800, 550) #sobreescrivemos a tupla

print("Novas dimensoes")
for dimensao in TAMANHO:
    print(dimensao)
```

Saída:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo45.py ====
Dimensoes originais
1280
720
Novas dimensoes
800
550
>>>
```

## DICIONÁRIOS

Dicionários são semelhantes às listas, são estruturas que nos permitem guardar diferentes tipos de dados, porém seu índice não será um número e sim um dado definido pelo próprio programador, este dado será a chave de acesso para um valor qualquer.

Entender os dicionários permite modelar uma diversidade de objetos do mundo real de modo mais preciso. Você será capaz de criar um dicionário que representa uma pessoa e armazenar quantas informações quiser sobre ela. Poderá armazenar o nome, a idade, a localização a profissão e qualquer outro

aspecto de uma pessoa que possa ser descrito. (MATTHES, 2016 p. 139)

Para declarar um dicionário utilizamos a seguinte sintaxe:

*identificador = {chave : valor}*

Onde identificador, como sabemos é o nome da variável que conterá o dicionário, o dicionário é caracterizado por um par de {}, entre as chaves temos um par de dados determinados “par *chave*-*valor*” (MATTHES, 2016). Sempre que precisarmos acessar um valor de um dicionário iremos acessar através da chave, da mesma forma que acessávamos um item da lista pelo índice. Vamos começar com este exemplo mesmo dado por Matthes, e elaborar um dicionário que guarda o nome, a idade e localização:

```
#Definimos um dicionario
pessoa = {'nome' : 'Bruno', 'idade' : 26, 'local' : 'Curitiba/PR'}

#acessamos o valor atraves da chave
print('Nome: ', pessoa['nome'])
print('Idade: ', pessoa['idade'])
print('Localizacao: ', pessoa['local'])
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo46.py =====
Nome: Bruno
Idade: 26
Localizacao: Curitiba/PR
>>>
```

Os dicionários assim como as listas, podem ser modificados, basta atribuir um novo valor para a chave:

```
#dicionario
pessoa = {'nome' : 'Bruno', 'idade' : 26, 'local' : 'Curitiba/PR'}

#escrevemos o nome
print('Nome original: ', pessoa['nome'])

#atribuimos novo valor para a chave
pessoa['nome'] = 'Kleber'

#escrevemos o novo nome
print('Novo nome: ', pessoa['nome'])
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo47.py =====
Nome original: Bruno
Novo nome: Kleber
>>> |
```

Os dicionários podem ser percorridos por um laço for fornecendo a chave o valor para o laço e assim podemos verificar tanto chave quanto valor em um dicionário escrevendo-os na tela, veja um exemplo:

```
frutas = {'chave1': 'melancia', #definimos um dicionario de frutas
          'chave2': 'laranja',
          'chave3': 'morango'}
}

for chave, valor in frutas.items(): #para cada chave e valor no dicionario de frutas
    print(chave + " = " + valor) #escreva a chave e o valor

#SIIIM PODEMOS FAZER ISSO EM PYTHON <3
```

Através do método `.items()` podemos fornecer duas variáveis de busca para o laço `for` e varrer todos os valores e

chaves de um dicionário.

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo47-1.py ====
chave1 = melancia
chave2 = laranja
chave3 = morango
>>>
```

Recorda-se quando criamos aquele pequeno algoritmo que recebe as notas de um aluno de uma disciplina e calcula sua média? Para esse algoritmo precisávamos definir quatro variáveis para as notas do aluno, através do dicionário basta criarmos as chaves para os valores que desejamos obter (no caso as notas do aluno), veja como seria obter e guardar as notas e medias de um aluno em um dicionário:

```
aluno = {           #abrimos um dicionario com identificador de aluno
    'nome' : '',   #definimos uma chave nome com valor em branco
    'nota1' : 0,   #definimos uma chave nota1 com valor zero
    'nota2' : 0,   #definimos uma chave nota2 com valor zero
    'nota3' : 0,   #definimos uma chave nota3 com valor zero
    'nota4' : 0,   #definimos uma chave nota4 com valor zero
    'media' : 0    #definimos uma chave media com valor zero
}

aluno['nome'] = input("Insira o nome do aluno: ") #solicita o nome do aluno
aluno['nota1'] = float(input("Insira a primeira nota: ")) #solicita as notas
aluno['nota2'] = float(input("Insira a segunda nota: "))
aluno['nota3'] = float(input("Insira a terceira nota: "))
aluno['nota4'] = float(input("Insira a quarta nota: "))

#calcula a media
aluno['media'] = (aluno['nota1'] + aluno['nota2'] + aluno['nota3'] + aluno['nota4']) / 4

#informe os dados
print("Aluno: ", aluno['nome'])
print("Media: %.1f" % aluno['media'])
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo48.py =====
Insira o nome do aluno: Bruno
Insira a primeira nota: 5.7
Insira a segunda nota: 6.3
Insira a terceira nota: 8.3
Insira a quarta nota: 9.2
Aluno: Bruno
Media: 7.4
>>>
```

Se você prestou bem atenção deve ter reparado que na última instrução **print** deste algoritmo utilizamos uma *string* formatada, isso significa que utilizamos um caracter especial (%) para substituir o valor da variável designada para dentro da *string*. Neste caso escrevemos “**Media: %.1f**” , onde % irá ser substituído pelo valor de **aluno['media']**, e .1f significa que precisamos mostrar apenas uma casa após o ponto flutuante do número fracionário. Após fechar a *string* utilizamos novamente o % para referenciar a variável que contém o valor a ser mostrado na *string*.

Agora já pensou que seu professor precisa guardar as notas de muitos alunos, tantos que nem sabemos quantos? Como implementar um algoritmo para ajudar nosso professor? Vamos fazer uma pergunta ao usuário perguntando quantos alunos possui a classe, e utilizamos esse valor para criar uma

lista com este tamanho. Para cada elemento desta lista vamos atribuir um dicionário contendo o nome do aluno, uma lista com suas quatro notas, sua media e sua situação final (se aprovado, recuperação ou reprovado), para preencher a lista completa de alunos da turma vamos utilizar um laço de repetições, veja:

```

#entrar com a disciplina
disciplina = input("Insira sua disciplina: ")
#Entrar com a quantidade de alunos da turma
quantidade = int(input("Insira o numero de alunos da turma: "))

#a turma e uma lista com tamanho igual a quantidade informada
turma = [alunos for alunos in range(quantidade)]

#para cada aluno na turma
for aluno in turma:

    #informe os dados do aluno e guarde em um dicionario
    turma[aluno] = {'nome':input("Insira o nome do aluno: "),
                    'notas':[float(input("Insira a nota: ")) for notas in range(4)],
                    'media':0,
                    'sit': ''}

    #calcule a media referenciaando-a pelo indice
    turma[aluno]['media'] = (
        turma[aluno]['notas'][0] + turma[aluno]['notas'][1] +
        turma[aluno]['notas'][2] + turma[aluno]['notas'][3]) / 4
    #as notas estao dentro de uma lista no dicionario que esta dentro da lista da turma

    #agora vamos verificar se o aluno esta aprovado
    if turma[aluno]['media'] < 6.0:
        turma[aluno]['sit'] = 'Reprovado'

    elif turma[aluno]['media'] > 6.0 and turma[aluno]['media'] < 7.0:
        turma[aluno]['sit'] = 'Recuperacao'

    else:
        turma[aluno]['sit'] = 'Aprovado'

#depois que seu loop receber todos os alunos vamos informar os resultados na tela
print("Disciplina: %s" % disciplina) #escreva a disciplina
for aluno in turma: #para cada aluno na turma
    print("Nome: %s" % aluno['nome'].capitalize()) #escreva o nome

    bi = 1 #vamos usar este bi para indicar o bimestre

    for notas in aluno['notas']: #aninhamos um for para contar as notas na lista do dicionario

        print("Nota %i : %.1f" % (bi, notas)) #Escreva as notas da lista de notas
        bi += 1 #incremenete o bimestre

    print("Media Final: %.1f" % aluno['media']) #mostre a media final
    print("Resultado: %s" % aluno['sit'])      #mostre o resultado

```

Este código parece um pouco complicado a uma primeira vista mas vamos analisá-lo por partes. Primeiro

requisitamos que o professor insira sua disciplina e a quantidade de alunos que compõem a turma. Até aqui tudo bem já estamos familiarizados com isso, então criamos uma variável turma, vamos atribuir a ela uma lista de tamanho igual ao número de alunos na turma. Através do laço de repetição vamos iterar pela turma, e para cada posição da lista vamos atribuir um dicionário contendo as informações do aluno. Ao criar a chave no dicionário atribuímos com valor um `input()`, desta forma as informações inseridas pelo usuário vão estar armazenadas em nosso dicionário. Para receber as notas utilizamos uma *list comprehension* para receber 4 entradas de números inteiros. Veja que aqui nós temos uma estrutura multidimensional, para acessar os dados das notas vamos ter que fornecer três índices ao Python, o primeiro é o da primeira dimensão da lista, a lista externa que contém os dicionários `turma[indice]`. A segunda dimensão é o dicionário que contém as informações do aluno, para acessar uma informação do dicionário devemos fornecer a chave após fornecer o índice da lista acima `turma[indice][chave]`. A terceira é a lista dentro do dicionário, para acessarmos os dados precisamos fornecer o índice do elemento da lista após ter fornecido a chave do dicionário E o índice da lista externa `turma[indice_externo]`

*[chave][índice]*. Através destes índices podemos somar as notas do aluno e atribuir à média e verificar se o aluno está reprovado, aprovado ou em recuperação. Para informar a saída utilizamos um laço de repetição encadeado, o primeiro para mostrar o nome de cada aluno, a media e o resultado final, o segundo somente para iterar pela lista interna do dicionário. Utilizamos strings formatadas onde substituímos o valor de % por um valor indicado após a string, o **%s** substitui um valor literal, o **%i** substitui um valor inteiro, o **%f** substitui um valor real. A saída é exemplificada para três alunos, mas pode ser utilizada para uma turma com trinta, cem mil alunos se você tiver paciência...

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exemplo49.py =====
Insira sua disciplina: Ed. Fisica
Insira o numero de alunos da turma: 3
Insira o nome do aluno: joaozinho
Insira a nota: 1.5
Insira a nota: 2.4
Insira a nota: 4.2
Insira a nota: 3.9
Insira o nome do aluno: pedrinho
Insira a nota: 6.0
Insira a nota: 5.5
Insira a nota: 6.4
Insira a nota: 7.0
Insira o nome do aluno: luizinho
Insira a nota: 9.0
Insira a nota: 10.0
Insira a nota: 9.8
Insira a nota: 8.9
Disciplina: Ed. Fisica
Nome: Joaozinho
Nota 1 : 1.5
Nota 2 : 2.4
Nota 3 : 4.2
Nota 4 : 3.9
Media Final: 3.0
Resultado: Reprovado
Nome: Pedrinho
Nota 1 : 6.0
Nota 2 : 5.5
Nota 3 : 6.4
Nota 4 : 7.0
Media Final: 6.2
Resultado: Recuperacao
Nome: Luizinho
Nota 1 : 9.0
Nota 2 : 10.0
Nota 3 : 9.8
Nota 4 : 8.9
Media Final: 9.4
Resultado: Aprovado
>>>
```

## EXERCICIOS ELABORADOS

- 8) Escreva um algoritmo que leia um vetor com 30 elementos inteiros e escreva-os em ordem contrária da leitura.

**Problema a ser resolvido:** ler um vetor de 30 elementos e imprimir seu inverso;

**Dados de entrada:** trinta inúmeros inteiros;

**Processamento:** inverter o vetor ao mostrar o resultado;

**Saída:** Informar os elementos do vetor em ordem contrária;

```
vetor = [] #criamos um vetor vazio

for i in range(30): #vamos contar ate 30
    #a cada contagem pedimos um numero e o colocamos no final da lista
    vetor.append(int(input("Insira um numero inteiro: ")))

vetor.reverse() #invertemos o vetor

for elemento in vetor: #para cada elemento no vetor
    print(elemento) #informe na tela o elemento
```

Saída:

```
Insira um numero inteiro: 27
Insira um numero inteiro: 28
Insira um numero inteiro: 29
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
>>>
```

- 9) Faça um algoritmo que possua uma lista com dez posições, e leia do usuário números em ordem aleatória até que a lista seja preenchida completamente pelos números inseridos. Sem utilizar instruções para ordenar automaticamente a faça com que a saída mostre todos os números da lista em forma ordenada e crescente.

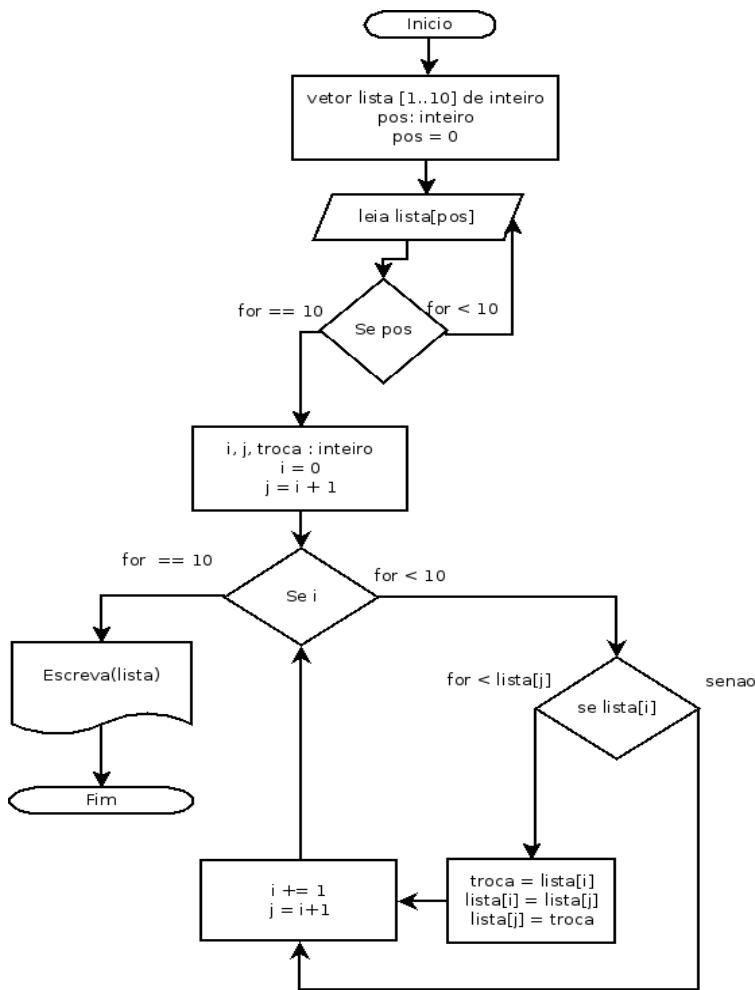
**Problema a ser resolvido:** ler um vetor de 10 elementos e imprimir de forma ordenada;

**Dados de entrada:** dez inúmeros inteiros;

**Processamento:** ordenar o vetor manualmente e mostrar o resultado;

**Saída:** Informar os elementos do vetor em ordem crescente;

Figura 20 – Fluxograma bubble sort



Fonte: O autor.

```

lista = list(range(10)) #criar uma lista com 10 posicoes

for pos in lista: #para cada posicao na lista
    lista[pos] = int(input("Insira um numero: ")) #entre com um inteiro e atribua a posicao

for i in range(10): # vamos contar de um a 10, i sera o contador
    j = i+1 # j sera o contador para o numero vizinho
    for j in range(10):
        if lista[i] < lista[j]: #se o indice de posicao i for menor que o indice de posicao j
            troca = lista[i] #fazemos uma troca, guardamos o numero na posicao i
            lista[i] = lista[j] #colocamos o numero de posicao j no lugar da posicao i
            lista[j] = troca #colocamos o numero da posicao i no lugar da posicao j

for numero in lista: #escreva os itens da lista ordenada
    print(numero)

```

Saída:

```

== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo5/exercicio9.py ==
Insira um numero: 56
Insira um numero: 21
Insira um numero: 34
Insira um numero: 99
Insira um numero: 86
Insira um numero: 0
Insira um numero: 1
Insira um numero: 45
Insira um numero: 88
Insira um numero: 9
0
1
9
21
34
45
56
86
88
99
>>>

```

**Obs:** Este é um algoritmo bem comum em computação denominado *Bubble Sort*, existem muitas formas diferentes de se ordenar os elementos em um vetor e este é um exemplo muito prático.

*Exercícios Propostos:*

- 1) Armazene o nome de alguns de seus amigos em uma lista. Exiba o nome de cada pessoa acessando cada elemento pelo índice da lista;
- 2) Faça uma lista de convidados para uma festa, exiba uma mensagem para cada convidado informando que ele está convidado para a festa.
- 3) Crie uma lista com 500 posições, através de um laço mostre quais números na lista são pares e quais são ímpares.
- 4) Escreva uma matriz para o mês em que você aniversario, mostre o dia do seu aniversário.
- 5) Formule um algoritmo que cadastra em um dicionário o registro de produtos de uma loja de alimentos, guarde os registros em uma lista. Os produtos devem conter nome, quantidade em estoque e valor de compra.

## CAPÍTULO 6

---

...

### SUB-ROTINAS E PROGRAMAÇÃO COM ARQUIVOS

Vamos conhecer neste capítulo os conceitos de sub-rotinas que nada mais são do que partes do nosso algoritmo que podem ser reaproveitadas sem termos que reescreve-las diversas vezes sempre que necessário, facilitando a depuração, manutenção e aumentando a legibilidade do nosso código. Para entender estes conceitos também será necessário compreender o escopo das variáveis, como variáveis globais e locais, tema que também será abordado neste capítulo.

Outro ponto a ser estudado neste capítulo será a manipulação de arquivos com Python, abordando elementos de inserção, leitura e remoção de dados oriundos de um arquivo guardado no disco rígido do seu computador, permitindo assim que os dados que trabalhamos fiquem armazenados de forma permanente.

### SUB-ROTINAS

Uma sub-rotina é uma parte do algoritmo que pode ser

utilizada diversas vezes sem que tenhamos que reescrever o código cada vez que seja necessário executar uma determinada tarefa. Ascencio e Campos (2010, p. 230) define as sub-rotinas como “blocos de instruções que realizam tarefas específicas”. Leal (2016a, p. 166) comenta que as sub-rotinas são responsáveis por diminuir a complexidade de determinados problemas, desta forma reduzimos um grande problema em pequenas partes, ou subproblemas, facilitando sua resolução. Desta forma utiliza-se as sub-rotinas para executar os subproblemas tratando cada parte complexa individualmente.

Uma sub-rotina é carregada apenas uma vez e pode ser executada quantas vezes for necessário, podendo ser utilizada para economizar espaço e tempo de programação. Em cada sub-rotina, além de ter acesso às variáveis do programa que o chamou (variáveis globais), pode ter suas próprias variáveis (variáveis locais), que existem apenas durante sua chamada. (LEAL, 2016a)

Na literatura encontramos as sub-rotinas em duas formas: Procedimentos e Funções, vamos analisar ambas as formas destacando suas características e como utilizá-las em Python.

## PROCEDIMENTOS

Um procedimento é uma sub-rotina que contém um bloco de instruções a serem executadas a qualquer momento

desejado. Assim como as variáveis, os procedimentos devem possuir um identificador para ser usado na hora de sua chamada. Ao ser chamado durante a execução de um algoritmo, o procedimento executa seus comandos e retorna para o fluxo normal do algoritmo logo após sua chamada, dando continuidade ao programa em execução. O procedimento caracteriza-se por não retornar nenhum valor após sua execução, ele simplesmente executa suas instruções e segue para o próximo comando do algoritmo após ter sido chamado. A sintaxe para declarar um procedimento em Python é a seguinte:

```
def identificador( ):  
    #bloco de instruções
```

A palavra-chave **def** é utilizada para definir o procedimento, no identificador atribuímos um nome para o procedimento, seguindo as regras para nomenclatura de identificadores já abordado no início deste livro, em seguida abrimos e fechamos parênteses para caracterizar o procedimento e finalizamos com dois pontos (:). Após os dois pontos serão declaradas as instruções do procedimento, estas deverão ser indentadas para que o Python reconheça que as

instruções pertencem ao bloco do procedimento. Veja um exemplo em Python para um procedimento que exibe a mensagem “Ola mundo”:

```
def olamundo():      #definimos um procedimento
    print("Ola mundo") #instrucao do procedimento

olamundo()          #chamada ao procedimento
```

Definimos um procedimento chamado olamundo( ), no seu bloco de comandos temos a instrução **print( )**. O procedimento executa a instrução **print( )** declarada em seu bloco de comandos quando houver a chamada para sua execução. Para chamar um procedimento basta escrevermos seu nome no programa, como no exemplo anterior. Ao chamarmos o procedimento olamundo( ) obtemos a execução dos seus comandos internos, neste caso uma mensagem:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo50.py ====
Ola mundo
>>> |
```

É possível no momento da definição do nosso procedimento, especificar parâmetros para execução do procedimento, estes parâmetros são fornecidos dentro dos parênteses em sua definição, e devem obrigatoriamente ser utilizados no bloco de comandos. Ao chamar o procedimento

fornecemos um argumento para o procedimento, os argumentos serão substituídos pelos parâmetros listados na definição e as instruções utilizarão os argumentos na execução do procedimento.

```
def cumprimento(nome): #definimos o parametro nome no procedimento
    print("Ola %s, prazer em conhecê-lo!" % nome) #utilizamos o parametro na mensagem
cumprimento('Bruno') #fornecemos a string 'Bruno' na chamada do procedimento
```

Veja que ao definirmos o procedimento, dentro dos parênteses declaramos um parâmetro chamado nome, este parâmetro está sendo utilizado na mensagem dentro do procedimento. Quando chamamos o procedimento fornecemos um argumento que substituirá o parâmetro nome para ser utilizado na mensagem.

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo51.py ====
Ola Bruno, prazer em conhecê-lo!
>>>
```

Não entendi. Qual a diferença entre parâmetro e argumento?



De acordo com Matthes (2016, p. 187),

parâmetro é um valor que deve ser fornecido para a sub-rotina na hora de sua definição, já o argumento é um valor passado para a chamada da sub-rotina durante a execução do algoritmo. Neste caso, nome é o parâmetro, pois foi declarado no procedimento durante sua definição, e a string 'Bruno' é um argumento que foi passado para o procedimento durante sua chamada na execução do algoritmo.

Leal(2016a) coloca uma distinção entre parâmetros que apresenta os mesmos conceitos acima, como sendo os parâmetros formais aqueles declarados juntamente à sub-rotina e os parâmetros reais aqueles que substituem os parâmetros formais ao chamar a sub-rotina no programa principal (os argumentos). Na literatura estes conceitos caracterizam a distinção entre parâmetro e argumento, ou parâmetro formal e parâmetro real. Vamos exemplificar com mais um procedimento que realiza a soma ou a subtração de dois números:

```

def soma(a, b): #definimos um procedimento para soma com dois parametros
    print("O resultado e: ", a + b) #o procedimento informa a soma dos parametros

def subt(a, b): #tambem definimos um procedimento para subtracao
    print("O resultado e: ", a - b) #que informa a subtracao entre os parametros

num1 = int(input("Insira o primeiro numero ")) #leia o primeiro numero
num2 = int(input("Insira o segundo numero ")) #leia o segundo numero

print("Qual operacao deseja realizar?") #informe as opcoes de operacoes
print("[1] Soma \n[2]Subtracao")
escolha = int(input()) #leia a opcao desejada

if escolha == 1: #se escolher soma
    soma(num1, num2) #chame o procedimento de soma e passe os numeros como argumento

elif escolha == 2: #se escolher subtracao
    subt(num1, num2) #chame o procedimento de subtracao e passe os numeros como argumento

else: #senao
    print("Escolha invalida") #informe uma mensagem de erro

```

Definimos inicialmente os procedimentos do algoritmo, ambos recebem dois parâmetros **a** e **b**, e quando chamados exibem uma mensagem informando o resultado da operação entre os parâmetros. As primeiras instruções do algoritmo solicitam dois números ao usuário, então pedimos para que faça uma escolha entre soma e subtração, e caso a escolha seja correspondente a soma então chamamos o procedimento de soma e fornecemos os números de entrada como argumentos, ou se a escolha corresponder à subtração então chamamos o

procedimento de subtração fornecendo as entradas como argumentos.

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo52.py ====
Insira o primeiro numero 5
Insira o segundo numero 3
Qual operacao deseja realizar?
[1] Soma
[2]Subtracao
1
O resultado e: 8
>>>
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo52.py ====
Insira o primeiro numero 5
Insira o segundo numero 3
Qual operacao deseja realizar?
[1] Soma
[2]Subtracao
2
O resultado e: 2
>>>
```

Você já conheceu aqui algumas formas interessantes de se construir algoritmos, implemente este último exemplo adicionando dois novos procedimentos para divisão e multiplicação. Implemente também um laço que pergunta ao usuário se gostaria de realizar outra operação ao final do algoritmo. Implemente também um laço que verifica se a entrada do usuário é válida (um valor numérico) e caso seja inválido fique repetindo a solicitação até que um valor válido seja inserido.

## FUNÇÕES

As funções são sub-rotinas idênticas aos procedimentos,

a declaração das funções é exatamente igual e sua estrutura é a mesma dos procedimentos assim como também podem possuir parâmetros. A única diferença entre uma função e um procedimento é que as funções sempre retornam valores.

Uma função nem sempre precisa exibir sua saída diretamente. Em vez disso, ela pode processar alguns dados e então devolver um valor ou um conjunto de valores. O valor devolvido pela função é chamado de *valor de retorno*. A instrução **return** toma um valor que está em uma função e o envia de volta à linha que a chamou. Valores de retorno permitem passar boa parte do trabalho pesado de um programa para funções, o que pode simplificar o corpo de seu programa. (MATTHES, 2016)

Um valor de retorno pode ser armazenado em um variável e ser processado ou manipulado por outras partes do algoritmo como operações aritméticas, lógicas, condicionais e até mesmo ser utilizado como argumento em outra função. Veja um exemplo que retorna o quadrado de um número:

```
def quadrado(numero): #a definicao de uma funcao e igual a de um procedimento
    return numero**2 #usamos a instrucao return para retornar um valor

entrada = int(input("Insira um numero: ")) #pedimos um numero

quad = quadrado(entrada) #chamamos a funcao e guardamos o retorno em quad

print("O quadrado de %i e %i" % (entrada, quad)) #escrevemos o resultado

print("O quadrado do quadrado e ", quadrado(quad)) #o retorno tambem pode ser impresso
```

Primeiramente definimos nossa função, da mesma forma como o procedimento, porém damos a instrução **return** para retornar o valor na hora de sua chamada. Então pedimos um valor inteiro ao usuário que será passado como argumento

para a nossa função. Veja que ao chamar a função também atribuímos a mesma em uma variável chamada quad. Perceba que neste momento não estamos armazenando uma função em uma variável e sim o valor retornado por esta função. O resultado armazenado na variável quad é exibido na primeira instrução **print( )**, e também é utilizado como argumento em uma nova chamada à função quadrado( ) que definimos, retornando um novo valor resultante da operação interna da nossa função. Quando retornamos um valor não precisamos utilizar uma função **print( )** dentro da função para exibir o valor, ele poderá ser exibido se chamarmos a função dentro da instrução **print( )**.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo53.py =====
Insira um numero: 2
O quadrado de 2 é 4
O quadrado do quadrado é 16
>>> |
```

Vamos para outro exemplo, adaptado de Matthes(2016). Vamos escrever uma função que possui dois parâmetros que devem ser literais correspondentes ao nome e sobrenome de uma pessoa, essa função deve retornar o nome completo e formatado.

```
def formatar(nome, sobrenome):      #definimos a funcao
    formatado = nome + " " + sobrenome #processamento da funcao
    return formatado.title()         #retorno da funcao

meu_nome = "bruno" #string com nome
meu_sobrenome = "LUVIZOTTO CARLI" #string com sobrenome

nome_completo = formatar(meu_nome, meu_sobrenome) #chamada da funcao

print("Meu nome e ", nome_completo) #saída
```

Definimos a função para receber dois parâmetros, no processamento fazemos a soma das strings, este processo é de unir strings é denominado concatenação. Retornamos as strings concatenadas e com a primeira letra de cada palavra com letra maiúscula. Definimos um nome e um sobrenome para ser fornecido à função, e seu retorno é atribuído à variável nome\_completo que é em seguida exibida.

```
...
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo54.py =====
Meu nome e Bruno Luvizotto Carli
>>> |
```

## PARÂMETROS

Os parâmetros como vimos, são valores abstratos declarados em uma sub-rotina, no momento da chamada à sub-rotina fornecemos um valor real para os parâmetros

denominados argumentos. Quando declaramos parâmetros em uma sub-rotina, obrigatoriamente devemos fornecer os argumentos em sua chamada, do contrário teríamos um erro:

```
def multiplica(a, b): #definimos uma função que retorna a multiplicação de dois números
    return a * b

x = 2 #declaramos dois números
y = 5

print(multiplica(x)) #mas fornecemos somente um na chamada da função
```

Saída:

```
=====
RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo55.py ====
Traceback (most recent call last):
  File "/Users/macbookpro/Desktop/livro/code/capitulo6/exemplo55.py", line 7, in
    <module>
      print(multiplica(x)) #mas fornecemos somente um na chamada da função
TypeError: multiplica() missing 1 required positional argument: 'b'
>>> |
```

Se não fornecermos os argumentos necessários obteremos um *Traceback* informando que na chamada à função `multiplica()` está faltando um argumento. Isto é porque os parâmetros definidos na função são obrigatórios, porém é possível definir parâmetros optativos no Python, basta inicializar o parâmetro na definição da sub-rotina. Veja:

```
def multiplica(a, b=1): #inicializamos o parâmetro b com o valor 1
    return a * b

x = 2 #declaramos dois números
y = 5

print(multiplica(x)) #mas fornecemos somente um na chamada da função
```

Declaramos a mesma função do exemplo anterior, mas

desta vez inicializamos o segundo parâmetro com o valor 1, desta forma o parâmetro se torna optativo, e se não fornecermos o segundo valor a função utilizará o valor inicializado, retornando a multiplicação pelo valor 1:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo56.py =====
2
>>> |
```

E no caso de fornecermos um argumento para um parâmetro inicializado, a sub-rotina irá utilizar o valor fornecido como parâmetro no lugar do valor inicializado:

```
def multiplica(a, b=1): #inicializamos o parametro b com o valor 1
    return a * b

x = 2 #declararamos dois numeros
y = 5

print(multiplica(x, y))
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo57.py =====
10
>>>
```

## ESCOPO DAS VARIÁVEIS

Quando trabalhamos com sub-rotinas devemos ficar atentos ao escopo das variáveis, se você declarar uma variável dentro de uma sub-rotina não poderá acessá-la no programa principal, mas poderá acessar as variáveis do programa principal dentro da sub-rotina, porém não poderá modificar seu valor a não ser que declare-a como global. As variáveis declaradas dentro das sub-rotinas são denominadas variáveis locais e as variáveis declaradas no programa principal são denominadas variáveis globais. Veja, vamos demonstrar com um exemplo:

```
def sub(): #essa subrotina mostra o valor de x
    x = 2
    print(x)

x = 12 #definimos um valor para vaeiavel x

sub() #chamamos a subrotina

print(x) #mostramos o valor real de x
```

Definimos uma função chamada sub( ), nela possuímos uma variável x com valor 2, mas no programa principal também temos uma variável x com valor 12, ao chamar a sub-rotina é impresso o valor da variável local da sub-rotina, porém o valor de x permanece 12:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo59.py =====
2
12
>>>
```

Isto se dá porque não informamos ao Python na definição da sub-rotina que x deveria ser referenciado como uma variável global. Veja:

```
def sub(): #essa subrotina mostra o valor de x
    global x #chamamos a variavel global x
    x = 2     #redefinimos seu valor para 2
    print(x)

x = 12 #definimos um valor para variavel x

sub() #chamamos a subrotina

print(x) #mostramos o valor real de x
```

Desta vez informamos que a variável x é uma variável global, assim podemos modificar seu valor dentro da sub-rotina:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo60.py =====
2
2
>>> |
```

Se tentarmos modificar uma variável declarada dentro da sub-rotina vamos levantar um erro, pois esta variável não pode ser utilizada no decorrer do algoritmo principal:

```
def sub(): #definimos uma sub-rotina
    x = 5 #variavel local da sub-rotina sub()
    print("x = ", x)

sub() #chamamos a subrotina

if x < 10: #se x for menor que 10
    print("x menor que 10") #escreva que x é menor que 10
```

A variável x só existe dentro da sub-rotina:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo61.py =====
x = 5
Traceback (most recent call last):
  File "/Users/macbookpro/Desktop/livro/code/capitulo6/exemplo61.py", line 7, in
    <module>
      if x < 10:
NameError: name 'x' is not defined
>>>
```

Lembra-se deste erro? A variável de nome x não foi definida. Isto porque quando declaramos uma variável local em uma sub-rotina, esta somente existirá durante o processamento da sub-rotina. Ou seja, quando chamamos uma sub-rotina no algoritmo esta será executada e as variáveis ali declaradas serão inicializadas na memória, quando todos os comandos da sub-rotina tiverem sido executados e ela chegar ao fim de sua execução, suas variáveis locais serão destruídas. Isto é um método eficiente para poupar o uso de memória. As variáveis

globais permanecerão ativas na memória enquanto seu algoritmo estiver executando, sempre que possível processe diversas variáveis em uma função e de um retorno para o programa principal, isto irá aumentar a eficiência do seu código.

## **RECURSIVIDADE**

A recursividade, em linguagem de programação é a capacidade de uma sub-rotina de chamar-se a si própria, fundamentando este conceito, Leal(2016a) afirma que “a recursividade é um mecanismo que permite uma função chamar a si mesma direta ou indiretamente”. Chamar a si mesma remete a função à execução de um loop, por isto precisamos definir em uma função recursiva:

- Uma chamada recursiva;
- Uma parada;

Um exemplo clássico de chamada recursiva é o cálculo do fatorial de um número. O fatorial é calculado pelo número multiplicado por todos seus antecessores. Por exemplo o

fatorial de 5! é  $5 \times 4 \times 3 \times 2 \times 1$  que resulta em 120. Veja neste exemplo de algoritmo Python para o cálculo fatorial de um número:

```
def factorial(numero): #define a função
    if numero == 0: #aqui temos a condição de parada
        return 1
    else:
        return numero * factorial(numero - 1) #e aqui a chamada recursiva

x = factorial(5) #chamamos a função com argumento 5
print(x) #escrevemos o resultado
```

Definimos dentro da função uma condição de parada, neste caso quando a variável local número chegar à 0. Até que chegue em zero continuaremos retornando a própria função subtraindo 1 do número em seu argumento e multiplicando pelo próprio número.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo62.py =====
120
>>> |
```

Modifique este algoritmo para que ele receba uma entrada de usuário que seja um número e calcule o fatorial deste número, exibindo uma mensagem elegante na tela informando o resultado do fatorial.

## ARQUIVOS

Arquivos são formas de gravar dados de forma

permanente em seu computador. Ao criarmos um arquivo podemos inserir dados processados através de um algoritmo, ou podemos utilizar um algoritmo para buscar dados em um arquivo escrito, também podemos utilizar algoritmos para escrever novos dados em arquivos escritos, entre outras possibilidades.

Em Python utilizamos a instrução `open()` para abrir um arquivo, devemos fornecer um argumento literal com o nome e extensão do arquivo. Se o arquivo estiver salvo no mesmo diretório do programa.py que você estiver executando basta fornecer o nome e extensão do arquivo como por exemplo 'lista\_de\_compra.txt', caso esteja em outro diretório você precisará fornecer o caminho completo par ao arquivo como por exemplo 'c:windows/desktop/pasta/lista\_de\_compras.txt'.

Sempre que abrirmos um arquivo com `open()`, devemos fechar-lo com `close()`.

```
arquivo = open('lista.txt')  
arquivo.close()
```

É importante que o arquivo exista e que você tenha informado corretamente o nome ou caminho ao utilizar o exemplo anterior, ou você irá se deparar com um erro

informando que o arquivo não existe:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo63.py =====
Traceback (most recent call last):
  File "/Users/macbookpro/Desktop/livro/code/capitulo6/exemplo63.py", line 1, in
    <module>
      arquivo = open('lista.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'lista.txt'
```

A instrução **open( )** aceita alguns parâmetros adicionais que especificam o modo de abertura do arquivo, assim poderemos informar se o arquivo deve ser aberto em modo de escrita, leitura, leitura e escrita, etc. A tabela a seguir mostra os parâmetros aceitos pela instrução **open( )**:

Figura X – Modos de abertura de arquivos em Python

Parâmetro	Significado
'r'	Abre o arquivo em modo leitura(default)
'w'	Abre o arquivo em modo de escrita, sobrescreve todos os dados do arquivo anterior
'x'	Abre em modo exclusivo, falha se o arquivo já existir
'a'	Abre para escrita <i>append</i> , adicionando os dados ao final do arquivo existente
'b'	Abre em modo binário
't'	Abre em modo texto (default)

Parâmetro	Significado
'+'	Abre um arquivo de disco para atualização (leitura e escrita)
'U'	Abre em modo <u>universal newlines</u> (Descontinuado)

Fonte: <<https://docs.python.org/3/library/functions.html#open>>

Ao fornecer o modo de abertura devemos utilizar a seguinte sintaxe:

```
variavel = open('nome_do_arquivo', 'modo_de_abertura')
```

Onde variável é a variável que irá armazenar o conteúdo do arquivo, nome\_do\_arquivo deve ser o valor literal contendo o nome do arquivo a ser aberto e modo\_de\_abertura deve ser um dos valores da tabela anterior. Veja no exemplo a seguir vamos criar uma lista de números e salvá-la em um arquivo com a instrução **write()**:

```
numeros = list(range(100)) #criamos uma lista com 100 numeros
arquivo = open('numeros.txt', 'w') #criamos um arquivo 0km em modo de escrita
for numero in numeros: #para cada numero na lista de numeros
    arquivo.write(str(numero)) #escreva cada numero em forma de string no arquivo
arquivo.close() #feche o arquivo depois
```

Primeiro criamos uma lista com cem números para testar, então abrimos o arquivo 'numeros.txt', este arquivo na

verdade não existe no computador mas como fornecemos o modo de abertura 'w' (modo de escrita) o Python irá criar automaticamente este arquivo no diretório local, caso um arquivo com este nome existisse ele seria apagado e sobreescrito. Com um laço iteramos pela nossa lista escrevendo cada número dela no arquivo, note que não podemos escrever números em um arquivo, somente dados do tipo literal podem ser escritos em arquivos, então convertemos cada número para literal com **str( )** no momento de escreve-lo no arquivo. É importante fechar o arquivo no final de seu algoritmo para que este não permaneça aberto na memória do computador, isto poderá corromper os dados do arquivo.

Ótimo, agora vamos escrever um algoritmo para ler os dados no arquivo que acabamos de criar:

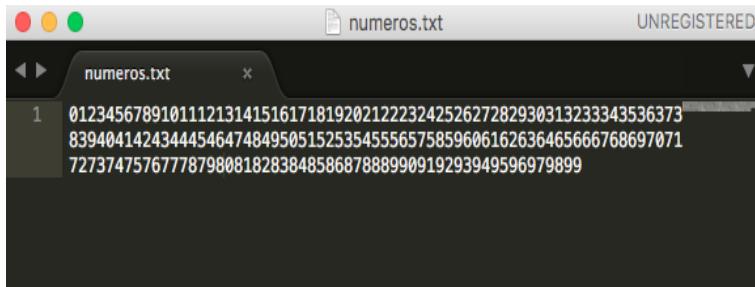
```
arquivo = open('numeros.txt', 'r') #abrimos nosso arquivo em modo de leitura
for linha in arquivo: #para cada linha no arquivo
    print(linha) #escreva o conteúdo da linha
arquivo.close() #feche o arquivo
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo65.py =====
01234567891011121314151617181920212223242526272829303132333435363738394041424344
4546474849505152535455565758596016263646566676869707172737475767778798081828384
858687888990919293949596979899
>>> |
```

Perceba que se você abrir o arquivo texto em um editor de texto padrão os dados estarão praticamente da mesma forma:

Figura 21- Arquivo gerado



Fonte: O autor.

Se tentarmos abrir novamente o mesmo arquivo em modo de escrita para tentar escrever novos dados vamos apagar os dados existentes no arquivo atual, veja:

```
arquivo = open('numeros.txt', 'w')

numeros = [999, 998, 997] #nivis numeros a serem adicionados

for numero in numeros:
    arquivo.write(str(numero))

arquivo.close() #feche o modo de escrita

arquivo = open('numeros.txt', 'r') #abra o modo de leitura
for linha in arquivo: #leia o conteudo
    print(linha)

arquivo.close()
```

Saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo66.py =====
999998997
>>>
```

Veja que no arquivo no diretório os dados também são modificados:

Figura 22 – Arquivo sobreescrito



Fonte: O autor.

Isto ocorre quando abrimos um arquivo existente em modo de escrita 'w', este modo cria um novo arquivo em branco para escrita, e se algum arquivo já existir com o nome declarado ele será apagado e sobreescrito. Para adicionarmos dados em um arquivo sem apagar o arquivo anterior precisamos abrir em modo *append*, um modo de escrita que permite adicionar dados à um arquivo já existente, o parâmetro para este modo é o 'a', veja:

```

arquivo = open('numeros.txt', 'a') #abre o arquivo em modo append
letras = ['a', 'b', 'c'] #novos itens a serem adicionados
for letra in letras:
    arquivo.write('\n'+letra)

arquivo.close() #feche o modo de escrita

arquivo = open('numeros.txt', 'r') #abra o modo de leitura
for linha in arquivo: #leia o conteudo
    print(linha)

arquivo.close()

```

Saída:

```

===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo67.py =====
999998997
a
b
c
>>>

```

Desta vez adicionamos algumas letras pulando linha, sim podemos utilizar o \n para pular linhas em arquivos, perceba que o arquivo original não foi apagado, mas novos dados foram adicionados a ele:

Figura 23 – Arquivo com dados adicionados ao final



Fonte: O autor.

Existe ainda um modo seguro de abrir um arquivo, utilizando as palavras reservadas **with** e **as**. Com **with** abrimos o arquivo sem que seja necessário fechá-lo com **open( )**, garantindo a segurança do arquivo, a palavra reservada **as** define o identificador da variável que conterá o arquivo, veja:

```
with open('numeros.txt', 'r') as arquivo:  
    for linha in arquivo:  
        print(linha)
```

Este é um modo prático e eficiente de se trabalhar com arquivos em Python, veja a saída:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo68.py =====  
999998997  
  
a  
  
b  
  
c  
=>>
```

Legal, agora que sabemos ler e escrever em arquivos permanentes vamos criar um pequeno algoritmo comercial, ele deve possuir uma sub-rotina a ser chamada para cadastrar produtos com seu devido valor e código do produto em um arquivo. Também vamos ter uma sub-rotina para mostrar os produtos disponíveis, uma para buscar um produto específico e

uma que calcule a compra.

Primeiro criamos um procedimento para cadastrar produtos em um arquivo:

```
def cadastro():
    with open('produtos.txt', 'r+') as produtos: #abre o arquivo em modo de leitura e escrita (r+)
        produto = input("Insira o nome do produto: ") #receba um novo produto
        preco = input("Insira o valor de venda do produto: ") #receba o valor do novo produto
        codigo = 0 #codigo para o produto

        for i in produtos: #o codigo é definido automaticamente de acordo com o numero de linhas do arquivo
            codigo += 1

        produtos.write(str(codigo) + " - " + produto + " - R$: " + preco + "\n") #escreva o produto no arquivo

cadastro() #chamando a subrotina para testar
```

Primeiro definimos o procedimento `cadastro()` e nele abrimos o arquivo 'produtos.txt' em modo de leitura e escrita 'r+', desta forma podemos ler e escrever no arquivo, como este ainda não existia no diretório atual, o Python nos fez a gentileza de criá-lo para nós. Então solicitamos a entrada de um produto e seu valor, também precisamos criar um código para cada produto, mas como isto pode vir a ser um valor variável que não pode se repetir vamos inicializar um código de valor zero e incrementá-lo em 1 para cada linha do arquivo de produtos, desta forma cada vez que inserir-mos um novo produto o arquivo ganhará uma nova linha, e consequentemente o código será incrementado, assim temos que o código é gerado automaticamente para cada produto,

legal não? Para testar adicionei dois produtos aleatórios em minha lojinha:

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo69.py ====
Insira o nome do produto: shampoo
Insira o valor de venda do produto: 8.00
>>>
==== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo69.py ====
Insira o nome do produto: sabonete
Insira o valor de venda do produto: 3.00
>>>
```

Veja como o código do produto fica no arquivo gerado, podemos ver que ele inicia em 0 e na segunda linha o código é 1, e para cada produto que inserirmos o código irá ser incrementado. Você pode alterar o código para iniciar em números maiores como 100 e ser incrementado de 10 em 10 por exemplo, tente fazer isto.

Figura 24 – Arquivo de produtos



Fonte: O autor.

No final escrevemos o código seguido do produto e seu valor de uma forma padrão, e sempre que precisarmos incluir um novo produto basta chamarmos o procedimento `cadastro()`.

Agora vamos elaborar uma sub-rotina que mostre todos os arquivos cadastrados:

```
def verProdutos(): #define o procedimento
    with open('produtos.txt', 'r') as produtos: #abre o arquivo em modo leitura
        for linha in produtos: #para cada linha no arquivo
            print(linha) #escreva a linha com o produto

verProdutos() #chamada da sub-rotina
```

Este é simples, basta abrirmos o arquivo em modo leitura e através de um laço de repetição mostrar cada linha do arquivo, como cada produto está em uma linha do arquivo, vamos mostrar todos os produtos da lista.

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo70.py =====
0 - shampoo - R$: 8.00
1 - sabonete - R$: 3.00
>>>
```

Mas e se a lista de produtos tiver cem mil produtos cadastrados? Ja pensou ter que imprimir todos estes? Vamos fazer uma sub-rotina que pesquise no arquivo um determinado produto pelo nome:

```

def buscaProduto():
    with open('produtos.txt', 'r') as produtos:
        achou = False #o produto precisa ser encontrado

    produto = input("Insira o nome do produto: ") #solicita um produto

    for linha in produtos: #busca em cada linha do arquivo
        if produto in linha: #se o produto estiver em uma destas linhas
            print(linha) #escreva esta linha
            achou = True #achamos o produto

    if not achou: #se nao achou o produto
        print("Produto nao cadastrado") #informe que o produto nao existe

buscaProduto()

```

Aqui precisamos de uma variável que nomeamos de achou e inicializamos ela com False pois ainda não encontramos o produto a ser buscado. Solicitamos o produto a ser buscado e verificamos cada linha do arquivo, se uma delas conter o produto atribuímos True para a variável achou e escrevemos a linha que contém o produto na tela. Caso o produto não seja encontrado no arquivo devemos informar que o produto não se encontra cadastrado no arquivo. Veja no exemplo buscamos pelo produto sabonete que foi cadastrado sob código 1, mas ao buscar a caneta o programa nos diz que esta não foi cadastrada:

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo71.py =====
Insira o nome do produto: sabonete
1 - sabonete - R$: 3.00

>>>
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo71.py =====
Insira o nome do produto: caneta
Produto nao cadastrado
>>> |
```

Vamos para agora escrever a sub-rotina que calcula o valor da compra:

```
def compra():
    total = 0 #variavel com valor a pagar
    lista = [] #lista de preco dos produtos
    compras = [] #lista de compras

    while True: #loop para receber varios produtos

        item = input("Insira o codigo ou nome do produto: ") #le o produto ou codigo do produto

        if item == 'fim': #o loop funciona ate que seja inserido fim
            break
        else:
            with open('produtos.txt', 'r') as produtos: #busca o produto comprado no arquivo
                for linha in produtos:
                    if item in linha:
                        compras.append(linha) #adiciona o produto a lista de compras
                        lista.append(float(linha[-5:-1])) #adiciona somente o valor do produto a lista de precos

            total = sum(lista) #somamos o total a pagar
            print("Lista de compras: ")
            for produto in compras: #escrevemos os produtos comprados na tela
                print(produto)

            print("TOTAL A PAGAR: R$ %.2f" % total) #escrevemos o total a pagar na tela

compra()
```

Definimos um procedimento compra( ) e inicializamos três variáveis, uma para o valor total a pagar, uma para guardar a lista de preços dos produtos adquiridos e uma lista que guardará os dados do produto vindas do arquivo (código, nome e valor). Iniciamos um loop que inicia solicitando um produto

cadastrado, o programa pode aceitar o nome ou o código do item, caso o valor inserido seja 'fim' o loop encerra, do contrário ficará recebendo produtos. Quando um produto for inserido procuramos no arquivo o produto e adicionamos seus dados a lista de compras e somente os últimos caracteres contidos no arquivo (correspondentes ao preço) serão convertidos para real e adicionados à lista de valores. Quando o loop se encerrar somamos todos os valores da lista de valores e atribuímos à variável total, fornecemos a saída final que consiste na iteração pela lista de compras exibindo os dados das compras na tela e a exibição do total a pagar em uma string formatada. Veja na saída que o programa aceita tanto o nome do produto (shampoo) como o código (1 para o sabonete):

```
===== RESTART: /Users/macbookpro/Desktop/livro/code/capitulo6/exemplo72.py =====
Insira o codigo ou nome do produto: shampoo
Insira o codigo ou nome do produto: 1
Insira o codigo ou nome do produto: fim
Lista de compras:
0 - shampoo - R$: 8.00
1 - sabonete - R$: 3.00
TOTAL A PAGAR: R$ 11.00
>>> |
```

---

Legal agora que temos todas as sub-rotinas vamos juntá-las em um sistema útil! Faça o seguinte, crie uma nova pasta em seu computador chamada sistemaPython, abra um

novo arquivo no *Idle* e copie cada uma das sub-rotinas criadas anteriormente:

```

#sub-rotina para cadastrar um produto
def cadastro():
    with open('produtos.txt', 'r+') as produtos:
        produto = input("Insira o nome do produto: ")
        preco = input("Insira o valor de venda do produto: ")
        codigo = 0
        for i in produtos:
            codigo += 1
        produtos.write(str(codigo) + " - " + produto + " - R$: " + preco + "\n")

#sub-rotina para ver a lista de produtos
def verProdutos():
    with open('produtos.txt', 'r') as produtos:
        for linha in produtos:
            print(linha)

#sub-rotina de busca
def buscaProduto():
    with open('produtos.txt', 'r') as produtos:
        achou = False
        produto = input("Insira o nome do produto: ")
        for linha in produtos:
            if produto in linha:
                print(linha)
                achou = True
        if not achou:
            print("Produto nao cadastrado")

#sub-rotina para compra
def compra():
    total = 0
    lista = []
    compras = []
    while True:
        item = input("Insira o codigo ou nome do produto: ")
        if item == 'fim':
            break
        else:
            with open('produtos.txt', 'r') as produtos:
                for linha in produtos:
                    if item in linha:
                        compras.append(linha)
                        lista.append(float(linha[-5:-1]))
    total = sum(lista)
    print("Lista de compras: ")
    for produto in compras:
        print(produto)
    print("TOTAL A PAGAR: R$ %.2F" % total)

```

Salve este arquivo na pasta sistemaPython com um nome que desejar, aqui será salvo como funcionalidades.py. Neste mesmo diretório copie o arquivo 'produtos.txt', de modo que seu diretório fique parecido com este:

Figura 25 – Diretório sistemaPython

		Hoje 01:27	--	Pasta
	funcionalidades.py	Hoje 01:26	2 KB	Python Source
	produtos.txt	Ontem 19:10	47 bytes	Texto Simples

Fonte: O autor.

Muito bem agora crie um novo arquivo em branco no *Idle* e salve-o na mesma pasta com o nome main.py (ou outro que lhe agrade), vamos utilizar o nome main por ser o programa principal, muitas linguagens de programação possuem um programa principal onde as outras funcionalidades criadas são chamadas para um propósito principal, isto não é obrigatório em Python, mas torna seu código mais fácil de se compreender, seu diretório irá ficar parecido com este:

Figura 26 - Diretório sistemaPython final

		Hoje 01:31	--	Pasta
	funcionalidades.py	Hoje 01:26	2 KB	Python Source
	main.py	Hoje 01:31	Zero bytes	Python Source
	produtos.txt	Ontem 19:10	47 bytes	Texto Simples

Fonte: O autor.

Agora podemos escrever o programa principal que une

todas as funcionalidades que acabamos de criar.

## BIBLIOTECAS E MÓDULOS

A maioria das linguagens de programação possuem bibliotecas contendo códigos pré-existentes que realizam uma ampla variedade de cálculos e tarefas. Estas bibliotecas quando fazem parte da linguagem são chamadas de Biblioteca Padrão, também temos as Bibliotecas Externas que são partes de código que baixamos de alguma fonte. No Python é comum se referir às bibliotecas como módulos, podemos escrever nossos próprios módulos e importa-los para nosso algoritmo (como vamos fazer com nosso sistemaPython) ou podemos utilizar as pré-existentes no Python (que são muitos, Python é conhecido por ter baterias inclusas, o que quer dizer que ele já vem com várias bibliotecas com funcionalidades para uma ampla variedade de tarefas).

Vamos fazer o seguinte, abra seu arquivo main.py e escreva o seguinte:

```
import funcionalidades      #importa o modulo de funcionalidades que criamos  
funcionalidades.cadastro() #chamamos a funcao de cadastro do modulo funcionalidades
```

A instrução **import** diz ao Python para usar neste

algoritmo o que estiver no modulo funcionalidades. O modulo funcionalidades é nada mais nada menos que nosso arquivo funcionalidades.py (ou o nome que você salvou o arquivo com as sub-rotinas na pasta sistemaPython). Para utilizar uma das funcionalidades do modulo importado use a sintaxe:

*modulo.funcionalidade()*

Utilize o operador . para usar a funcionalidade desejada do modulo desejado, assim a funcionalidade será executada. Há outras formas de importar um módulo, veja o abaixo:

Tabela 8 – importação de módulos

Declaração	Tradução	Descrição
<code>import modulo</code>	Importe o modulo	Importa um modulo todo para o escopo atual. Você deve referenciar a funcionalidade pela sintaxe <code>modulo.funcionalidad e()</code>
<code>from modulo import funcionalidade</code>	Do modulo importe essa funcionalidade	Importa uma funcionalidade específica de um módulo. Desta forma

		você pode chamar a funcionalidade diretamente pelo nome funcionalidade( )
<b>from modulo import funcionalidade as nome</b>	Do modulo importe essa funcionalidade como nome	Cria um alias (apelido) para a funcionalidade importada. Desta forma você pode chamar a funcionalidade por um nome desejado.
<b>from modulo import *</b>	Do modulo importe tudo	Importa todas as funcionalidades do modulo.

Fonte: O autor.

Você pode utilizar a forma como for mais conveniente, muitas vezes precisamos somente de uma ou algumas funcionalidades de um módulo, então não precisamos importar todas, isto aumenta a eficiência de seu algoritmo.

Agora que sabemos sobre a importação de módulos vamos finalizar a implementação do nosso sistemaPython:

```

import funcionalidades      #importa o modulo de funcionalidades que criamos

while True:                  #criamos um loop para o programa

    print('#'*34)           #fornecemos um menu de opcoes
    print(">*11,BEM-VINDO,<*11)
    print("Escolha a operacao desejada")
    print("[1] Cadastrar produto")
    print("[2] Verificar produtos cadastrados")
    print("[3] Buscar produto")
    print("[4] Compra")
    print("[5] Sair do sistema")
    print("#'*34)

    entrada = input()       #recebe a entrada

    if entrada == '1':      #fornece a funcionalidade de acordo com a opcao
        funcionalidades.cadastro()
    elif entrada == '2':
        funcionalidades.verProdutos()
    elif entrada == '3':
        funcionalidades.buscaProdutos()
    elif entrada == '4':
        funcionalidades.compra()
    elif entrada == '5':
        break
    else:                   #e uma mensagem avisando quando a entrada for invalida
        print("#'*34)
        print("Operacao invalida")
        print("#'*34)

```

Colocamos tudo dentro de um laço while para que o programa fique rodando até que seja solicitado para encerrar, fornecemos as opções disponíveis, uma para cada funcionalidade do sistema e caso o usuário insira uma opção inválida o alertaremos disto.

```
==== RESTART: /Users/macbookpro/Desktop/livro/code/sistemaPython/main.py ====
#####
>>>>>> BEM-VINDO <<<<<<
Escolha a operacao desejada
[1] Cadastrar produto
[2] Verificar produtos cadastrados
[3] Buscar produto
[4] Compra
[5] Sair do sistema
#####
1
Insira o nome do produto: caneta
Insira o valor de venda do produto: 0.50
#####
>>>>>> BEM-VINDO <<<<<<
Escolha a operacao desejada
[1] Cadastrar produto
[2] Verificar produtos cadastrados
[3] Buscar produto
[4] Compra
[5] Sair do sistema
#####
2
0 - shampoo - R$: 8.00
1 - sabonete - R$: 3.00
2 - caneta - R$: 0.50
```

```
#####
>>>>>> BEM-VINDO <<<<<<
Escolha a operacao desejada
[1] Cadastrar produto
[2] Verificar produtos cadastrados
[3] Buscar produto
[4] Compra
[5] Sair do sistema
#####
4
Insira o codigo ou nome do produto: 2
Insira o codigo ou nome do produto: 2
Insira o codigo ou nome do produto: fim
Lista de compras:
2 - caneta - R$: 0.50

2 - caneta - R$: 0.50

TOTAL A PAGAR: R$ 1.00
#####
>>>>>> BEM-VINDO <<<<<<
Escolha a operacao desejada
[1] Cadastrar produto
[2] Verificar produtos cadastrados
[3] Buscar produto
[4] Compra
[5] Sair do sistema
#####
5
>>> |
```

Começamos cadastrando um novo item no sistema, e ao ver a lista de produtos é possível perceber que o novo item já encontra-se cadastrado e com um código válido. Realizamos a compra deste novo item e o programa nos revela o total da compra, excelente não?

Python possui muitas outras bibliotecas que não foram abordadas aqui, como por exemplo o modulo sqlite3 que permite a manipulação de bancos de dados, este por exemplo seria uma forma muito mais eficiente e segura de guardar

dados como em um sistema comercial. Procure em outras fontes sobre este modulo e tente implementar este sistemaPython com um banco de dados.

*Exercícios Propostos:*

- 1) Um supermercado popular próximo à sua casa está trabalhando em um novo sistema de cadastro de produtos. Desenvolva um cadastro de produtos que contenha código, descrição, unidade e preço para 20 produtos para o supermercado. Defina um menu com as seguintes opções:
  - Cadastrar os 20 registros;
  - Pesquisar um produto pelo código;
  - Classificar alfabeticamente os registros;
  - apresentar todos os registros;
  - sair do programa de cadastro;
- 2) Uma balada top da sua cidade vai dar entrada livre e double drink para os 10 primeiros clientes que inserirem o nome em uma lista eletrônica contendo seu o algoritmo com o seu programa. Elabore um algoritmo para o cadastro de 10 pessoas contendo as seguintes informações: nome, idade,

RG, telefone, endereço. Caso a idade do cliente seja menor que 18 anos ele não pode entrar na festa e deve ser excluído da lista. O menu deve conter opções de cadastrar, excluir e mostrar a lista completa de VIPs.

3) Você já foi em algum desses restaurantes *fast food* onde você pode montar seu próprio lanche? Você escolhe o tipo de pão(integral, italiano, com gergelim) o tamanho do *sandúba* (30cm , 50cm), o tipo de recheio (queijo e calabresa, carne bovina, almôndegas, vegetariano), o tipo de queijo(provolone, parmesão, cheddar), o tipo de salada(alface, tomates, cebola, ervilhas), o tipo de molho (ketchup, mostarda, barbecue), se quer quente ou frio e então *voilà*, você tem um sanduíche a seu gosto. Monte um algoritmo que simule a montagem de um sanduíche, Os sabores de queijo e calabresas e vegetaria custam R\$ 9.00, os outros sabores custam R\$ 12.00, o tamanho de 50cm custa o valor do sabor mais 75% do seu valor insira uma função que pergunta se o usuário aceita o dobro de queijo por um adicional de R\$ 2.00. No final mostre na tela o sanduíche completo e o valor a pagar pelo sanduíche.

## **REFERÊNCIAS**

- ABBAGNANO, N. **Dicionário de filosofia**. 5° ed. São Paulo: Martins Fontes, 2007;
- ASCENCIO, A. F; CAMPOS, E. V. A. **Fundamentos da programação de computadores**. 2° ed. 4° reimpressão. São Paulo: Pearson Prentice Hall, 2010;
- BARRY, P. **Use a cabeça: Python**. 2° reimpressão. Rio de Janeiro: Alta Books, 2015;
- LEAL, G. C. L. **Algoritmos e lógica de programação I**. Centro Universitário de Maringá NEAD, Maringá – PR: Unicesumar, 2016 (a);
- LEAL, G. C. L. **Algoritmos e lógica de programação II**. Centro Universitário de Maringá NEAD, Maringá – PR: Unicesumar, 2016 (b);
- MATTHES, E. **Curso intensivo de python**. São Paulo: Novatec, 2016;
- PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Pearson Makron Books, 2010;

PUGA, S; RISSETTI, G. **Lógica de programação e estrutura de dados, com aplicações em Java.** 2° ed. São Paulo: Pearson Prentice Hall, 2010;

TANENBAUM, A. S. **Organização estruturada de computadores.** 5° ed. 5° reimpressão. São Paulo: Pearson Prentice Hall, 2010;

PYTHON ORG, disponível em: <<https://www.python.org/>>  
Acesso em: 18/05/2017;

## **Apelo do Autor**

Olá caro leitor, se você concluiu a leitura deste livro, deixo aqui meus parabéns pois espero ter agregado um mínimo e qualquer conhecimento a ti por meio desta obra. Se o conhecimento repassado aqui lhe foi de alguma ajuda insisto que repasse este conhecimento a frente pois nenhuma moeda no mundo vale mais do que a sabedoria conquistada por meio do esforço mental a qual nos designamos.

Faço nesta parte final, um encarecido apelo para que, se este livro lhe foi útil, e não tivestes de pagar qualquer centavo por ele mas sente-se disposto a retribuir, que possa fazer uma doação de qualquer quantia para ajudar o autor a publicar esta obra em material físico e tangível e desta forma atingir mais e mais pessoas de forma a propagar o conhecimento. Este livro não possui um ISBN, pois para registro do mesmo é necessário que se arque com alguns fundos burocráticos para fins de oficializar esta material como Obra Literária real. Como autor de origem humilde não posso realizar tamanha façanha sem o incentivo monetário para tal, desta forma se você, caro leitor puder contribuir para que esta obra seja oficialmente regulamentada perante as demais obras da Biblioteca Nacional e atribuir um ISBN para este material, ficarei, como autor

demasiado grato pela sua contribuição e espero um dia, publicar oficialmente este manuscrito. Para doar qualquer quantia você pode acessar e doar através do PayPal por meio deste link:

[https://www.paypal.com/cgi-bin/webscr?cmd=\\_donations&business=brunolcarli%40gmail%2ecom&lc=BR&item\\_name=Bruno%20Luvizotto%20Carli&currency\\_code=BRL&bn=PP%2dDonationsBF%3abtn\\_donateCC\\_LG%2egif%3aNonHosted](https://www.paypal.com/cgi-bin/webscr?cmd=_donations&business=brunolcarli%40gmail%2ecom&lc=BR&item_name=Bruno%20Luvizotto%20Carli&currency_code=BRL&bn=PP%2dDonationsBF%3abtn_donateCC_LG%2egif%3aNonHosted)

Doe bitcoins por aqui:



16srApTUX3McNhwxtxYLveGmhLff7wrUAA

Agradeço sua compreensão e pela leitura deste material.  
Atenciosamente, o autor.