

---

UNIVERSIDADE FEDERAL FLUMINENSE  
CENTRO TECNOLÓGICO  
ESCOLA DE ENGENHARIA  
CURSO DE ENGENHARIA DE TELECOMUNICAÇÕES  
PROGRAMA DE EDUCAÇÃO TUTORIAL  
GRUPO PET-TELE



Tutorial de Introdução ao Python

(Versão: 2k9)

Niterói - RJ

Maio / 2009

---

# Sumário

<b>1</b>	<b>Características básicas da linguagem</b>	<b>2</b>
<b>2</b>	<b>Obtenção e instalação</b>	<b>2</b>
<b>3</b>	<b>Variáveis</b>	<b>3</b>
<b>4</b>	<b>Strings</b>	<b>3</b>
4.1	<i>Manipulando Strings</i> . . . . .	4
<b>5</b>	<b>Operações matemáticas</b>	<b>5</b>
<b>6</b>	<b>Entrada de Dados</b>	<b>6</b>
<b>7</b>	<b>Estruturas de controle</b>	<b>7</b>
7.1	<i>While</i> . . . . .	7
7.2	<i>If</i> . . . . .	7
7.3	<i>For</i> . . . . .	8
<b>8</b>	<b>Funções</b>	<b>9</b>
8.1	Variáveis em funções . . . . .	11
8.2	Recursividade . . . . .	11
8.3	Módulos . . . . .	12
<b>9</b>	<b>Listas</b>	<b>13</b>
9.1	Inserindo um novo dado a uma lista . . . . .	15
9.2	Impressão dos conteúdos da lista . . . . .	15
9.3	Determinar em que ordem um elemento aparece na lista . . . . .	16
9.4	Remover um elemeto de uma lista . . . . .	16
9.5	Descobrir o tamanho de uma lista . . . . .	16
9.6	<i>Range</i> . . . . .	17
<b>10</b>	<b>Expressões booleanas</b>	<b>17</b>

# Introdução

A idéia de um ser humano, enquanto for apenas um pensamento, é algo amorfo, que existe por si mesma, não necessitando de nada, além de si, para descrevê-la. Entretanto para compartilhá-la com outras pessoas precisamos descrever esta idéia com palavras e frases, transformando este pensamento em uma linguagem natural humana.

Computadores e outros dispositivos eletrônicos programáveis possuem sua própria forma de “pensar”, isto é, o código binário. Máquinas seguem instruções determinadas pela energização (ou não) de determinadas partes em seus componentes.

Então para transmitir uma idéia para uma máquina devo me comunicar em zeros e uns? A resposta para esta pergunta é um “sim” e um “não”. Um “sim” pois, de fato, a “idéia” que será recebida pela máquina será descrita como uma sequência de zeros e uns. O “não” se refere a dizer que embora seja possível traduzir sua idéia para uma linguagem natural da máquina (binário), este não é o processo mais comum. A programação diretamente em binário é algo demasiadamente complexo para humanos, pois a forma básica de expressão do ser humano são as palavras e não sequências numéricas.

Para contornar este problema foram criadas as linguagens de programação, que são pontes entre a linguagem natural humana e a linguagem da máquina (binária), mesclando conceitos das linguagens de máquina e natural humana. As linguagens de programação são classificadas em vários níveis de acordo com sua proximidade com a linguagem humana.

Linguagens de baixo nível tem a função de descrever uma situação mais próxima do “ponto de vista” de uma máquina, ao passo que linguagens de alto nível são concebidas com o propósito de tornar fácil a tradução de um pensamento (algoritmo), se assemelhando com fidelidade à linguagem humana que seria necessária para descrever aquela idéia.

Neste tutorial descreveremos a linguagem de programação Python cujo objetivo é se aproximar bastante da linguagem humana, sendo assim intuitiva, fácil e ao mesmo tempo sendo bastante flexível, se adequando ao uso em diversas situações.

## 1 Características básicas da linguagem

Python é uma linguagem de programação interpretada, de código-fonte aberto e disponível para vários sistemas operacionais. Ser uma linguagem interpretada significa dizer que ao se escrever um programa, este não será compilado (traduzido para uma linguagem de máquina), mas sim “lido” por um outro programa (chamado de interpretador) que traduzirá para a máquina o que seu programa quer dizer.

O interpretador para Python é interativo, ou seja, é possível executá-lo sem fornecer um script (programa) para ele. Ao invés disso, o interpretador disponibilizará uma interface interativa onde é possível inserir os comandos desejados um por um e ver o efeito de cada um deles. Neste tutorial, se encontra a representação “>>>” antes da maioria dos comandos apresentados, o que quer dizer que este comando está sendo inserido no interpretador interativo (e sempre abaixo, o resultado gerado).

## 2 Obtenção e instalação

Caso o usuário esteja utilizando um sistema Linux ou OS X (Apple), o interpretador para Python já vem instalado por padrão, sendo apenas necessário escrever o comando “python”

no seu programa de terminal favorito. Para usuários do sistema operacional Windows, o interpretador para Python deve ser baixado através do site <http://www.python.org> e instalado. Neste último sistema o usuário encontra um utilitário para fazer o papel de terminal (e editor de python) no Windows, denominado IDLE.

### 3 Variáveis

Variáveis são formas de se armazenar dados para uso posterior. Estas podem ser classificadas em 3 tipos básicos (existem outros, que serão abordados mais a frente):

- *int* - Um número inteiro
- *float* - Um ponto flutuante
- *string* - Uma sequência de caracteres

Ao contrário da maioria das outras linguagens, em Python, não é necessário declarar as variáveis que serão usadas, tampouco definir seu tipo. A própria sintaxe do dado a ser armazenado identifica o tipo da variável para armazená-lo. Por exemplo, caso deseje-se atribuir o valor 3 à variável **A**, basta digitar **A=3**. Python saberá que **A** é um inteiro (tipo “*int*”). Por outro lado, se o valor a ser armazenado fosse 3,2 que é um dado do tipo “ponto flutuante”, este deveria ser expresso como **A=3.2**. Observe que, para Python, **A=3** e **B=3.0** são variáveis de tipos diferentes e isto deve ser levado em conta ao se realizar certos tipos de manipulações de dados.

### 4 Strings

String é um tipo de objeto formado por uma sequência imutável de caracteres que nos permite trabalhar com textos. Exemplo:

```
>>> a = “Bom Dia”
>>> print a
Bom Dia
```

Percebemos que elas são delimitadas por aspas, podemos utilizar tanto aspas duplas como as simples. Se utilizarmos aspas duplas, como o mostrado no exemplo acima, podemos usar as simples para aplicações dentro do texto que estamos escrevendo, o contrário também é verdadeiro.

Exemplo:

```
>>> b = 'O lema do governo JK era:\n “Cinquenta anos em cinco.”'
>>> print b
O lema do governo JK era:
“Cinquenta anos em cinco.”
```

No exemplo acima utilizamos um outro artifício para trabalharmos com strings, o `\n`. Este por sua vez, tem a função de “pular uma linha” e escrever o texto, que está depois do `\n`, nessa nova linha. Tanto isso é verdade que no mesmo exemplo, quando usamos o

comando `print` é mostrada a parte do texto que diz: “Cinquenta anos em cinco.”, impresso na linha seguinte.

Outro artifício parecido com `\n`, é o `\t` que tem o objetivo de acrescentar ao texto que vem após, um espaço de tabulação.

Há também outra aplicação para as aspas, que consiste na delimitação do texto por três aspas (duplas ou simples).

Exemplo:

```
>>> a = ''' Quatro times do Rio de Janeiro:
Botafogo
Vasco
Fluminense
Flamengo'''
>>> print a
Quatro times do Rio de Janeiro:
Botafogo
Vasco
Fluminense
Flamengo
```

Nessa aplicação para as aspas, quando usamos o comando `print`, tudo o que estiver entre as três aspas será impresso exatamente da mesma forma como foi escrito.

## 4.1 *Manipulando Strings*

Pelo fato de uma string ser uma sequência imutável, isso nos dá a possibilidade de manipularmos essa sequência, consideremos o exemplo abaixo:

```
>>> a = 'matemática'
>>> a[2]+a[-5]+a[-4:]
'tática'
```

Isso nos mostra que uma string segue uma determinada indexação onde cada caractere assume um endereço que, pode ser acessado colocando o nome da variável, que contém a string, e após, entre os colchetes, o endereço da célula que contém o caractere desejado.

Existe também um recurso conhecido como operador `%`, que serve para formatar as strings, basicamente são três os tipos de formatação que temos:

- `%s` - Serve para substituir string;
- `%d` - Serve para substituir números inteiros em uma frase destinada a um `print`;
- `%f` - Serve substituir floats (números em aritmética de ponto flutuante).

Observação:

As três formatações acima relacionadas são normalmente para aplicações em uma frase destinada a um `print`.

Exemplo:

```
>>> compra= 'maçã'
>>> tipo='verde'
```

```
>>> quilos = 1,5
>>> print 'Maria comprou %f quilos de %s %s .' %(quilos,compra,tipo)
Maria comprou 1,5 quilos de maçã verde.
```

Observação:

Com vimos acima o operador % pode ser utilizado para formatação de números também. Com ele é possível também determinar a quantidade de números, após a vírgula de um float.

Exemplo:

```
>>> num=245.47876749
>>> print '%.2f' %(num)
245.48
```

Percebemos que Python fez uma aproximação do número real, possibilitando que o resultado de uma futura operação seja o mais preciso possível.

## 5 Operações matemáticas

Além de ser uma poderosa linguagem de programação, Python sabe lidar bem com matemática. Suas capacidades matemáticas se estendem desde operações básicas até operações com números complexos.

Abaixo, vamos começar aprendendo as 4 operações matemáticas básicas. Lembre-se que “>>>” significa a digitação de um comando no interpretador.

```
Soma:
>>>2+3
5
Subtração:
>>>2-3
-1
Multiplicação:
>>>2*3
6
Divisão:
>>>2/3
0
```

Na divisão, pode-se optar por resultados como números inteiros (ou o inteiro mais próximo) ou resultado exato (um ponto flutuante). Para isso, deve-se fornecer um inteiro (para se obter um inteiro como resultado) ou um ponto flutuante (para se obter o resultado exato).

```
Divisão:
>>>2.0/3.0
0.6666666666666667
```

A operação aritmética de potenciação também pode ser feita, assim como a sua inversa, a radiciação.

```
Potenciação:
>>> 2**3
```

Não existe um comando específico para a radiciação em Python (exceto para a raiz quadrada). Entretanto, para obter a raiz  $n$ -ésima de um número basta elevá-lo por  $1/n$  onde “ $n$ ” é o índice da raiz. As regras para a obtenção de números inteiros ou pontos flutuante também se aplicam a este caso.

```
Radiciação:
>>>8**(1.0/3.0)
2.0
8
```

## 6 Entrada de Dados

Através do comando *raw\_input* podemos receber do usuário uma *string*. O tamanho da *string* pode ser conhecido pelo comando *len()*.

```
>>> nome = raw_input('Digite seu nome: ')
Digite seu nome: Pedro
>>> nome
'Pedro'
>>> len(nome)
5
>>> sobrenome = raw_input('Agora digite o sobrenome: ')
Agora digite o sobrenome: Albuquerque
```

Operações matemáticas não podem ser feitas com *strings*, apenas com *floats* e inteiros, porém se somarmos *strings*, Python as juntará, num processo chamado concatenação e se multiplicarmos uma *string* ela será repetida.

```
>>> nome + sobrenome
'PedroAlbuquerque'
>>> nome*3
'PedroPedroPedro'
```

Se a priori sabemos que o dado digitado pelo usuário não será uma *string*, podemos utilizar como entrada de dados o comando **input()**:

```
>>> idade = input('Digite sua idade: ')
Digite sua idade: 27
>>> altura = input('Qual a sua altura: ')
Qual a sua altura: 1.75
>>> type(idade)
<type 'int'>
>>> type(altura)
<type 'float'>
```

Observação: O dado inserido pelo usuário já é dinamicamente tipado. Para saber qual o tipo da variável atribuída, basta digitar o comando `type()` que ele devolverá a variável referente ao objeto entre parênteses.

## 7 Estruturas de controle

Os comandos de Python são executados pelo computador, linha por linha e as estruturas de controle permitem ao programador modificar a ordem em que cada comando será executado bem como se ele será ou não executado.

### 7.1 *While*

Começaremos por uma estrutura de repetição. O *loop while* tem como função repetir determinado bloco de comando, até que uma determinada condição pare de ser atendida. Para que o bloco seja executado no momento correto, devemos manter uma rígida indentação, ou seja, tudo que pertencer ao bloco do **while**, deve estar um espaço a frente, isto ajuda a deixar o código legível e organizado.

```
>>> while #condição for verdadeira :  
...     #bloco de comandos pertencentes ao while  
#continuação do programa
```

Vejamos, por exemplo, como calcular um número fatorial:

```
>>> resultado = 1  
>>> num_org = input('Entre com um número inteiro: ')  
>>> Entre com um número inteiro: 6  
>>> num = num_org  
>>> while num > 1:  
...     resultado = resultado * num  
...     num = num - 1  
...  
>>> print 'O número ', num_org, '! é: ',resultado  
>>> O número 6! é: 720
```

Se a condição estabelecida no **while** for sempre verdadeira, como  $2 < 1$ , o seu *loop* será infinito, pois a condição será sempre atendida. Caso esse seja seu desejo, pode-se usar também o comando **while 1** em vez de estabelecer uma condição qualquer.

### 7.2 *If*

Outra estrutura que estudaremos é a condicional. O comando **if** direciona o computador a tomar uma decisão, baseado nas condições determinadas. Se a condição for atendida, um bloco de comandos será executado, caso contrário, o computador executa outros comandos.

```

>>> if #condição1 for verdadeira:
...     #bloco de comandos 1
... elif #condição2 for verdadeira:
...     #bloco de comandos 2
... else:
...     #bloco de comandos 3

```

Nessa estrutura podemos utilizar quantas condições forem necessárias, para isso, basta repertimos o *elif* seguido das condições desejadas. Tendo isso em vista, vejamos um programa de adivinhação de um número:

```

>>> num = 23
>>> adv = 0
>>> while adv != num:
...     adv = input('Insira um número: ')
...     if adv < num:
...         print 'É maior!'
...     elif adv > num:
...         print 'É menor!'
...     else:
...         print 'Você acertou!'
...
>>> Insira um número:

```

Como vimos, em Python não é necessário que utilizemos a *tag end* para terminarmos uma estrutura. Isto é devido a indentação do programa. Em Python, uma estrutura é terminada automaticamente quando a *tag* da próxima linha começa no início da linha.

### 7.3 For

O comando **for**, em Python, difere do que normalmente se vê em outras linguagens de programação, onde esse comando tem a finalidade de realizar uma iteração baseada numa progressão aritmética, percorrendo os números definido pelo usuário, enquanto em Python a iteração é feita percorrendo os itens de uma sequência, seja ela uma lista ou até mesmo uma *string*. Vamos analisar o código abaixo:

```

>>> for contador in range(1, 11):
...     print contador

```

A estrutura acima utiliza uma variável criada, no caso **contador**, para percorrer cada elemento da lista criada com o comando **range(1,11)**, com isso, cada repetição feita pelo *loop for* fará com que a variável contador aponte para um diferente valor dentro da lista formada pela função **range** e logo em seguida imprima esse valor.

A saída que teríamos neste caso seria:

```
1
```

2  
3  
4  
5  
6  
7  
8  
9  
10

Ou seja, um “print” de todos os elementos da lista.

A função **for** também pode ser aplicada com o uso de *strings*, observemos o caso abaixo:

```
>>> lista = ['vida', 42, 'o universo', 6, 'e', 7, 'tudo']  
>>> for item in lista:  
...     print 'O item atual é:',print
```

E obteríamos como saída:

```
O item atual é: vida  
O item atual é: 42  
O item atual é: o universo  
O item atual é: 6  
O item atual é: e  
O item atual é: 7  
O item atual é: tudo
```

## 8 Funções

As linguagens de programação em geral têm o intuito de automatizar ações tornando-as mais rápidas.

Se houver alguma ação que seja grande e utilizada com frequência, temos a opção de criar uma função que cumpra o seu objetivo, reduzindo o espaço ocupado pelo nosso programa final, além de deixá-lo com uma aparência mais limpa, visto que o tamanho do código irá diminuir. Essas funções também são muito úteis na tarefa de debuggar o seu código, visto que você não precisará vasculhar o código atrás do erro, basta entrar na função e modificá-la. Um exemplo de como podemos diminuir um código está descrito abaixo.

Se em um determinado problema, precisarmos descobrir se dois valores absolutos são iguais podemos utilizar o código 1, descrito abaixo, todas as vezes que precisarmos realizar essa descoberta, ou podemos simplesmente usar o código 2 criando uma função que cumpra esse objetivo exigindo apenas que apresentemos os valores que devemos analisar.

Dado que temos os valores:

$a = 23$   $b = -23$

Função 1:

```
>>> if a < 0:
...     a = -a
>>> if b < 0:
...     b = -b
>>> if a == b:
...     print 'Os valores absolutos de ', a, ' e ', b, ' são iguais'
... else:
...     print 'Os valores absolutos de ', a, ' e ', b, ' são iguais'
```

Função 2:

```
>>>def compara_absolutos(a,b):
...     """ Essa função retorna se os valores absolutos das variáveis requeridas são iguais"""
...     if a < 0:
...         a = -a
...     if b < 0:
...         b = -b
...     if a == b:
...         print 'Os valores absolutos de ', a, ' e ', b, ' são iguais'
...     else:
...         print 'Os valores absolutos de ', a, ' e ', b, ' são iguais'
... 
```

Abaixo apresentamos a sintaxe necessária para criação de funções utilizando a linguagem Python.

```
>>>def funcao(variavel1,variavel2,...,variavelN):
...     bloco de comandos
...     return
```

É que ao chamar uma função, podemos passar a esta alguns parâmetros (valores ou *strings*): **funcao(1234,"pet-tele","UFF-1234")**, porém em algumas funções ao colocarmos nosso código na seção bloco de comandos, a função automaticamente definirá os parâmetros necessários. Por exemplo, se houver uma função que faça cálculos matemáticos, devolvendo um resultado numérico, será necessário que todas as variáveis chamadas pela função sejam inteiros ou *floats*.

## 8.1 Variáveis em funções

Em geral, quando estamos eliminando código repetitivo por meio de funções também temos algumas variáveis repetidas neste código. Em Python, as variáveis podem ter tratamentos diferentes em função de onde se encontram. Todas as variáveis que vimos até agora são chamadas de variáveis globais, ou seja, em qualquer momento ou em qualquer parte do código do seu programa, você poderá utilizá-las (seja para ler seu conteúdo ou atribuir valores).

Funções tem um tipo especial de variáveis que são chamadas de variáveis locais. Estas variáveis existem apenas dentro da função, de forma que caso o programador determine uma função e armazene um certo dado em uma variável local, ao término desta função a variável será destruída, não sendo possível recuperar seu valor.

Mas como então, pode-se recuperar um valor de uma função? Para isto usamos o comando **return** seguido de algum valor ou uma variável. Neste caso, a variável local “esconderá” a variável global, enquanto a função estiver rodando. Um pouco confuso? Então confira o exemplo abaixo:

```
>>>a = 4
>>>def print_func():
...     a = 17
...     print 'in print_func a = ', a
... 
```

Agora chame a função **print\_func()** e peça o valor **a** utilizando o comando **print**, seu resultado deve ser este:

```
>>>print_func()
in print_func a = 17
>>>print 'a = ', a
a = 4
```

Com isto, podemos concluir que variáveis criadas dentro de uma função (variáveis locais), não afetam as variáveis que estão fora dela (variáveis globais). As variáveis locais existem apenas dentro do espaço limitado pela função, não podendo assim nem ser usada, nem afetar nada fora deste espaço.

## 8.2 Recursividade

A recursividade é um tipo de iteração (repetição) na qual uma função chama a si mesma repetidamente até que uma condição de saída seja satisfeita. Abaixo temos um exemplo de uma função responsável por calcular o fatorial de números positivos inteiros e demonstra como uma função pode chamar a ela mesma utilizando a propriedade recursiva.

```
>>>def fatorial(n):
...     if n <= 1:
...         return 1
...     return n * fatorial(n - 1)
```

```

...
>>>print '2! = ',fatorial(2)
2! = 2
>>>print '3! = ',fatorial(3)
3! = 6
>>>print '4! = ',fatorial(4)
4! = 24::
>>>print '5! = ',fatorial(5)
5! = 120

```

Um passo-a-passo de como a função é executada, utilizando o comando **fatorial(n)**, para  $n = 3$ :

1. Quando chamamos a função **fatorial(3)**, fornecemos à função o valor 3 para o parâmetro de entrada  $n$ .
2. O comando **if** testa se  $n$  é menor ou igual a 1, como este não é, a função continua.
3. Nesta parte a função pretende retornar o valor  **$n$ \*fatorial( $n-1$ )**, ou seja,  **$3$ \*fatorial( $2$ )**. Entretanto, ela ainda não possui o valor de **fatorial( $2$ )**. Para isso, a função **fatorial** é novamente chamada com  $n = 2$ , retornando o valor  **$2$ \*fatorial( $1$ )**.
4. Como ainda não temos **fatorial( $1$ )**, a função chama a si mesma mais uma vez para calculá-lo.
5. Com isto, atingimos a condição de saída do **if** que está na função **fatorial** e então o valor retornado é 1.

Vamos fazer uma retrospectiva do que aconteceu ao se executar esta função:

```

fatorial(3)
3 * fatorial(2)
3 * 2 * fatorial(1)
3 * 2 * 1 = 6

```

### 8.3 Módulos

Pensando na reutilização de código, a linguagem Python já possui um conjunto de funções prontas para serem usadas ou agregadas em seus programas. Essas funções estão agrupadas em estruturas denominadas módulos. Para a utilização desses módulos é preciso utilizar o comando **import nome\_do\_módulo**.

Após ter importado o módulo, qualquer função pertencente a ele pode ser utilizada através do comando **nome\_do\_módulo.função(argumento)**.

É possível importar do módulo apenas a função desejada. Para isso, utilizamos o comando **from nome\_do\_módulo import função**, e a função estará disponível para

utilização.

Você também pode definir o seu próprio módulo. Defini-se as funções desejadas e ao final, você salva o seu módulo com a extensão `.py`. Exemplo:

Digite em um editor de texto simples:

```
#IMC.py

def indice(altura,peso):
    imc = peso/(altura**2)
    return imc

def estado(imc):
    if imc < 24.9:
        print 'NORMAL'
    elif 24.9 < imc < 29.9:
        print 'PESO A MAIS'
    elif 29.9 < imc < 40:
        print 'LIGEIRA OBESIDADE'
    elif imc > 40:
        print 'OBESIDADE'
    else:
        print 'MAGRO DEMAIS'

def pesoideal(peso,altura):
    a = 20*(altura**2)
    b = 24.9*(altura**2)
    print 'Seu peso ideal se encontra entre %f e %f' %(a,b)
```

Agora, salve o seu arquivo como `IMC.py`. De agora em diante, o módulo já pode ser utilizado por qualquer programa em Python.

## 9 Listas

Listas são sequências de variáveis. Após definidas, podem ser modificadas de várias maneiras, pois são mutáveis.

Para definir uma lista basta digitar:

```
lista = [9,8,7]
```

O endereçamento dos componentes começa no 0. Então, se você deseja usar um determinado elemento da lista, basta chamá-lo: `lista[x]`. Com isso, você está se referindo ao elemento de posição `x` na lista.

Alguns comandos referentes a listas:

`len(lista)` informa o tamanho da lista

**lista.append(x)** adiciona o elemento x no final da sua lista

**lista.extend([6,5,4])** adiciona uma lista inteira no final da sua lista

**lista[y]= x** insere o valor y na posição x da lista

Nós já vimos anteriormente que variáveis comuns armazenam um único valor. Entretanto, existem outros tipos de variáveis capazes de armazenar mais de um valor. Em Python, chamamos essas variáveis com capacidade de armazenamento de listas ou vetores. Vamos explicar a sintaxe de uma lista através das linhas de código exibidas abaixo:

```
>>> meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho', 'Julho', 'Agosto', 'Setembro', 'Outubro', 'Novembro', 'Dezembro']
>>> while 1:
...     qual_mes = input("Escolha um mês (1-12)? ")
...     if 1 <= which_one <= 12:
...     print "O mês é ", meses[qual_mes - 1]
```

Ao rodar este programa obteremos a seguinte saída como exemplo:

```
>>> Escolha um mês (1-12)? 5
O mês é Maio
>>> Escolha um mês (1-12)? 12
O mês é Dezembro
>>> Escolha um mês (1-12)?
```

Primeiramente criamos a nossa lista, a definição de uma lista é análoga a de uma variável qualquer, porém isolando o conteúdo com colchetes. O uso das aspas (duplas ou simples) apenas é necessário caso estejamos inserindo na lista uma *string*, se armazenássemos apenas um número, não seria necessário.

Exemplo:

```
teste = ['vida', 42, 'universo', 6, 'e', 7]
```

Depois inserimos o comando “*while 1:*” que faz com que o nosso programa entre em *loop*. O programa vai rodar indefinidamente até ser dado o comando Ctrl+D, ou até que o programa seja fechado. Uma vez nesse *loop*, é definida pelo usuário a variável “qual\_mes”, e depois de um tratamento de erros, feito com a utilização de um comando *if*, é nos devolvido o nome do mês selecionado, Na última linha do código utilizamos uma propriedade da lista, que é buscar um dado da lista, que se dá escrevendo o nome da lista e entre colchetes o número referente ao local da lista onde está o dado requerido (lista[número]). Observe que na nossa linha de código *print* é dado pelo mês escolhido menos um, ou seja: o Python indexa suas listas partindo do zero.

Além de selecionar um elemento de uma lista, temos algumas outras propriedades que apresentaremos em seguida.

## 9.1 Inserindo um novo dado a uma lista

Para inserir um novo dado a uma lista qualquer, chamamos uma método chamado *.append*:

```
>>>teste = []
>>>teste.append('zero')
>>>teste.append('um')
>>>teste
['zero','um']
```

Infelizmente o comando *.append* só consegue adicionar um dado na lista por vez, mas se quisermos adicionar mais dados podemos simplesmente somar listas, multiplicá-las, ou utilizar o comando *.extend*:

```
>>>teste.extend(['dois','três'])
>>>soma = ['quatro','cinco']
>>>teste += soma
>>>teste + ['seis']
['zero','um','dois','três','quatro','cinco','seis']
>>>teste*2
['zero','um','dois','três','quatro','cinco','seis','zero','um','dois','três','quatro','cinco','seis']
```

Observação: o código `teste += soma` é o mesmo que escrever `teste = teste + soma`.

## 9.2 Impressão dos conteúdos da lista

Ao chamarmos a lista `teste` sem referenciarmos nenhum elemento específico estamos lidando com todos os elementos da lista. Podemos também imprimir somente alguns valores de dentro da lista, necessitando para isso apenas indicar o local:

```
>>>print 'As variáveis na lista teste são ',teste
As variáveis na lista teste são ['zero','um','dois','três','quatro']
>>>print 'As variáveis 0 e 3 da lista teste são ',teste[0],' e ',teste[3]
As variáveis 0 e 3 da lista teste são zero e três
```

Podemos também utilizar o comando *for*.

```
>>>for valor in teste:
... print valor
...
zero
um
dois
três
```

quatro

### 9.3 Determinar em que ordem um elemento aparece na lista

A função *.index* retorna o índice da primeira ocorrência do valor especificado, ou seja, para imprimirmos na tela a posição em que a *string* “quatro” aparece utilizaríamos o código:

```
>>>print lista.index("quatro")
4
```

Observação: É importante notar que a lista inicia sua contagem a partir do 0 (zero) o que explica o número 2 obtido acima.

### 9.4 Remover um elemento de uma lista

Para remover um elemento de uma lista utilizamos o comando *del*, referenciando o *index*, ou posição da lista, onde haverá a remoção.

```
>>>print 'Antes a lista 'teste' era: ',teste
Antes a lista 'teste' era: ['zero','um','dois','três','quatro']
>>>del teste[3]
>>>print 'Agora a lista 'teste' é: ',teste
Agora a lista 'teste' é: ['zero','um','dois','quatro']
```

Podemos obter o mesmo efeito utilizando o método *.remove* que procuraria a primeira ocorrência na lista para um dado valor ou *string*:

```
>>>lista += ['um']
>>>print 'Antes a lista 'teste' era: ',teste
Antes a lista 'teste' era: ['zero','um','dois','quatro','um']
>>>teste.remove("um")
>>>print 'Agora a lista 'teste' é: ',teste
Agora a lista 'teste' é: ['zero','dois','quatro','um']
```

### 9.5 Descobrir o tamanho de uma lista

Para descobrir o tamanho de uma lista, usamos o comando *len()*, como exemplificado no código abaixo:

```
>>>print 'O tamanho da lista teste é: ', len(teste)
4
```

Nos dizendo que a lista “teste” possui 4 variáveis

## 9.6 *Range*

A função *range* gera um vetor contendo números (inteiros) sequenciais, obedecendo a regra de escrita:

```
range(início,fim)
```

É importante observar que o número finalizador descrito acima não é incluído no vetor, por exemplo:

```
>>>vetor = range(1,11)
>>>print vetor
```

O código acima nos dará como saída o vetor: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] . Em outras palavras o último número é excluído do vetor, portanto o 11 não entra na composição deste.

A função *range* aceitará quaisquer números inteiros desde que o número inicial seja maior que o número final, bem como quando apenas o número final é passado para a função, portanto são válidas as construções:

```
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(-32, -20)
[-32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21]
>>> range(5,21)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(21, 5)
[]
```

No último exemplo foi retornado um vetor vazio, isso se deu porque o elemento inicial era maior que o final.

Outra característica importante desse comando é a de adicionar uma terceira variável no *range*, indicando o incremento entre os números delimitados.

```
>>>range(0,24,4)
[0, 4, 8, 12, 16, 20, 24]
```

## 10 Expressões booleanas

Ao leitor que não estiver acostumado com o título acima, Expressões Booleanas são sentenças lógicas que seguem as leis da Álgebra de Boole. A Álgebra Booleana trabalha com valores lógicos, interpretando esse valores através de números binários, ao oposto do dec-

imal, utilizando o valor 0 para definir uma operação FALSA e o valor 1 para definir uma operação VERDADEIRA. As operações entre valores lógicos são feitas de forma análoga às operações feitas entre conjuntos de elementos, sendo que cada um desses elementos é tratado como uma situação, podendo ser verdadeiros ou falsos.

Em python os seguintes valores são interpretados como falso:

*False None 0 () []*

Ou seja, os valores *False* e *None*, o valor numérico 0 e as sequências vazias são denominadas falsas enquanto todos os outros valores possíveis são interpretados como verdadeiro.

Como na maioria das linguagens de programação, temos a possibilidade de utilizar a Álgebra Booleana para realizar testes lógicos usados em estruturas de controle, com esses testes podemos conferir a veracidade de operações, além de podermos montar funções condicionais *if*, laços de repetição *while* e *for*, entre outros, que podem ser inseridos em qualquer lugar de seu programa. Observemos o exemplo abaixo:

```
>>> a = 6
>>> b = 7
>>> c = 42
>>> print 1, a == 6
>>> print 2, a == 7
>>> print 3, a == 6 and b == 7
>>> print 4, a == 7 and b == 7
>>> print 5, not a == 7 and b == 7
>>> print 6, a == 7 or b == 7
>>> print 7, a == 7 or b == 6
>>> print 8, not (a == 7 and b == 6)
>>> print 9, not a == 7 and b == 6
```

Teríamos como saída o seguinte resultado:

```
1 True
2 False
3 True
4 False
5 True
6 True
7 False
8 True
9 False
```

Para entender a lógica do que foi feito acima, devemos relembrar alguns conceitos de álgebra booleana. Abaixo temos uma breve descrição sobre os operadores lógicos:

- Operador “*and*”: (em português “e”) significa que ambas as afirmações devem ser verdadeiras, caso contrário a expressão é falsa.
- Operador “*or*”: (em português “ou”) significa que se ao menos uma das afirmações for verdadeira, então toda a expressão também o será.
- Operador “*not*”: (em português “não”) significa uma inversão lógica em uma expressão, o que era verdadeiro, se torna falso, e viceversa.

As tabelas abaixo podem ajudar a clarificar alguns desses conceitos:

Expressão	Resultado
true and true	true
true and false	false
false and true	false
false and false	false
not true	false
not false	true
true or true	true
true or false	true
false or true	true
false or false	false

O programa abaixo mostra um exemplo de aplicação de expressões booleanas. Vamos dizer que precisamos de um programa que indique a situação de um aluno ao final do ano, sendo que elas são assim:

- Se o aluno obtiver nota maior ou igual e seis e não tiver mais de 10 faltas, ele passa direto;
- Se o aluno obtiver nota maior ou igual e seis e tiver mais de 10 faltas, ele fica de recuperação por falta;
- Se o aluno obtiver nota maior que quatro e menor que seis e não tiver mais de 10 faltas, ele fica de recuperação por nota;
- Se o aluno obtiver nota menor que quatro, ele repete direto;
- Se o aluno obtiver nota maior que quatro e menor que seis e tiver mais de 10 faltas, ele repete por não obter nota e por excesso de faltas;

Para montar esse programa precisaremos utilizar as Expressões Booleanas, já que precisamos fazer a verificação de duas situações, temos que verificar se o aluno tem nota e se ele tem presença.

```
>>> alunos = ['Fred', 'Suzana', 'Claudio', 'Puga', 'Robson', 'Gustavo']
>>> nota = [5.4, 6.2, 2.9, 9.9, 7.8, 4.9]
```

```

>>> faltas = [9, 5, 15, 2, 11, 12]
>>> contador = 0
>>> for aluno in alunos:
...     if nota[contador] >= 6.0 and faltas[contador] <= 10:
...         print 'Aluno: ',aluno
...         print 'Nota final: ',nota[contador]
...         print 'Faltas: ',faltas[contador]
...         print 'Resultado: Passou de ano'
...     elif nota[contador] >= 6.0 and faltas[contador] > 10:
...         print 'Aluno: ',aluno
...         print 'Nota final: ',nota[contador]
...         print 'Faltas: ',faltas[contador]
...         print 'Resultado: RecuperaÃ§Ã£o por falta'
...     elif nota[contador] >= 4.0 and nota[contador] < 6.0 and faltas[contador] <= 10:
...         print 'Aluno: ',aluno
...         print 'Nota final: ',nota[contador]
...         print 'Faltas: ',faltas[contador]
...         print 'Resultado: RecuperaÃ§Ã£o por nota'
...     elif nota[contador] >= 4.0 and nota[contador] < 6.0 and faltas[contador] > 10:
...         print 'Aluno: ',aluno
...         print 'Nota final: ',nota[contador]
...         print 'Faltas: ',faltas[contador]
...         print 'Resultado: Repetiu direto por nÃ£o obter nota e por excesso de faltas'
...     elif nota[contador] < 4.0:
...         print 'Aluno: ',aluno
...         print 'Nota final: ',nota[contador]
...         print 'Faltas: ',faltas[contador]
...         print 'Resultado: Repetiu direto por nota'
...     ""\n
...     contador+=1

```

## Referências

- [1] Labaki , Josué , *Introdução ao Python - Módulo A*, Grupo Python, UNESP-Ilha Solteira.
- [2] *The Python Tutorial 2.6*, <http://python.org/> , 2009.
- [3] Hetland , Magnus Lie, *Beginning Python: from novice to professional*, Apress, 2005.