



Módulo 4 - Python

Bootcamp IGTI: Programador de Software Iniciante

Marcelo Sampaio

Python

Bootcamp IGTi: Programador de Software Iniciante

Marcelo Sampaio

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1.	Introdução ao Python	5
PEP-8		5
Capítulo 2.	Exibindo e recebendo dados na tela	9
Capítulo 3.	Variáveis	10
Escopo de variáveis.....		11
Strings.....		11
Concatenação de strings		12
Manipulação de strings		12
Capítulo 4.	Listas.....	14
Criar listas com range		14
Manipular listas		15
Capítulo 5.	Tuplas	17
Manipular tuplas.....		17
Capítulo 6.	Dicionários	19
Manipular dicionários		19
Capítulo 7.	Set.....	21
Manipular sets.....		21
Capítulo 8.	Expressões booleanas	22
Capítulo 9.	Operadores lógicos em Python	24
Operador and.....		24
Operador or.....		24
Operador not.....		25
Capítulo 10.	Estruturas condicionais: if, else, elif	26
Capítulo 11.	Loops	28
Loop for.....		28

Loop while.....	29
Capítulo 12. Funções em Python	31
Referências	32

Capítulo 1. Introdução ao Python

Python é uma linguagem de programação de código aberto criada por um holandês em 1991, com a finalidade de ser uma linguagem de fácil compreensão e intuitiva, mas ainda assim tão poderosa quanto as suas maiores competidoras.

PEP-8

PEP significa *Python Enhancement Proposal*, ou propostas de melhorias para o Python. Cada PEP é um documento que apresenta uma nova função ou novos procedimentos na linguagem, e serve como um guia para uma melhor utilização do código.

O PEP8 é um documento que nos informa convenções para o código Python com a finalidade de manter a consistência do código.

É importante que quem tem a intenção de programar com Python conheça o PEP8. Apresentaremos um resumo dele aqui. Pouco a pouco você irá aprendê-lo. Na verdade, os editores de texto modernos de Python já fazem automaticamente o PEP8.

Observação: *Essa sessão deve ser lida novamente ao final do curso. Muitos dos termos aqui não são ainda conhecidos por você, não se preocupe! Vamos conhecê-los pouco a pouco.*

Vejamos abaixo as convenções da codificação Python:

1. Identação (recuo de texto em relação à margem)

Use 4 espaços de indentação antes de escrever uma instrução. A linguagem Python é totalmente dependente da indentação para a leitura do código.

Exemplos:

✓ Certo:

if 'a' in 'caderno':

```
    print('tem')
```



Aqui foram colocados 4 espaços

✗ Errado:

if 'a' in 'caderno':

```
print('tem')
```

2. Linhas em Branco

Entre funções e classes: duas linhas em branco.

Métodos dentro de uma classe: uma única linha em branco.

✓ Certo:

```
1 if 'a' in 'caderno':
```

```
2     print('tem')
```

```
3
```

```
4
```

```
5 name = 'Maria'
```

```
6 input(name)
```

✗ Errado:

```
i1 if 'a' in 'caderno':
```

```
2     print('tem')
```

```
3 name = 'Maria'
```

```
4 input(name)
```

```
5
```

```
6
```

3. Imports

Devem ser feitos em linhas separadas.

✓ **Certo:**

```
import: cys
```

```
import: os
```

✗ **Errado:**

```
import: cys, os
```

Observação: imports devem ser colocados no início do arquivo.

4. Nomes de classes

Para nomes de classes, utilize as primeiras letras maiúsculas, sem separação (ex: CamelCase)

✓ **Certo:**

```
class PrimeiraAulaPython
```

✗ **Errado:**

```
class primeira_aula_python
```

5. Nomes de funções ou variáveis

Use letras minúsculas para palavras simples ou palavras separadas por underline (ex: snake_case).

✓ **Certo:**

```
numero = 5
```

```
numero_par = 2
```

❌ Errado:

Numero = 5

numeropar = 2

6. Espaço em branco em declarações e expressões

Evite espaços em branco entre parênteses e colchetes, imediatamente antes de uma vírgula, de ponto e vírgula ou de dois pontos, imediatamente antes do parêntese de abertura que inicia a lista de argumentos de uma chamada de função, e espaços em branco à direita de qualquer lugar.

✅ Certo:

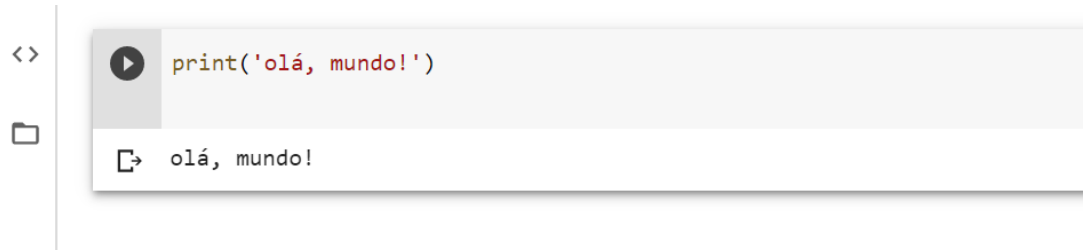
```
funcao(primeiro[23], {segundo:4})
```

❌ Errado:

```
funcao ( primeiro [23] , { segundo:4 } )
```

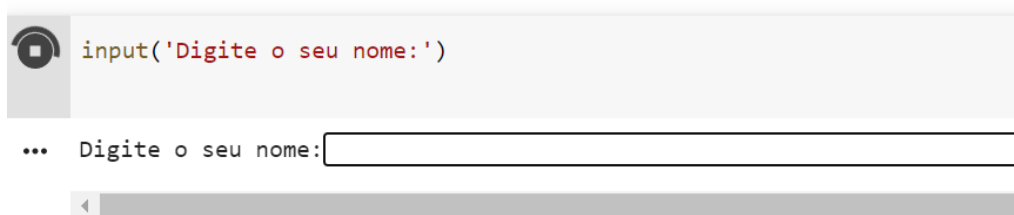

Capítulo 2. Exibindo e recebendo dados na tela

O comando para exibir dados na tela é o **print()**. A mensagem escrita dentro dos parênteses será exibida na tela. Veja o exemplo abaixo:



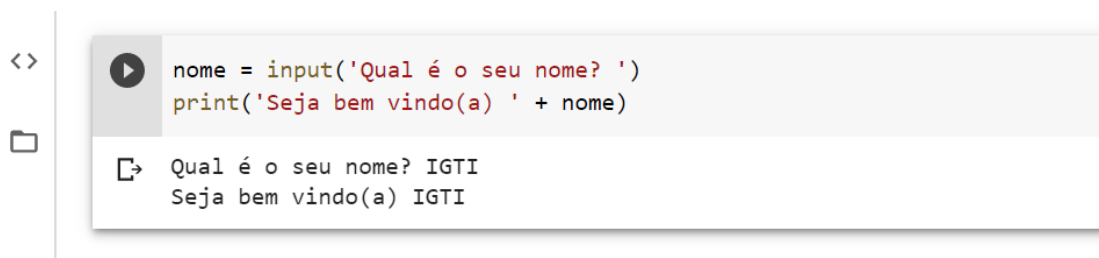
```
<>
print('olá, mundo!')
olá, mundo!
```

O comando utilizado para receber uma entrada de dados, ou seja, para que o usuário digite algo no programa, é o **input()**. Veja o exemplo abaixo:



```
input('Digite o seu nome:')
... Digite o seu nome: 
```

Vamos criar um programa utilizando a entrada e saída de dados:



```
<>
nome = input('Qual é o seu nome? ')
print('Seja bem vindo(a) ' + nome)
Qual é o seu nome? IGTI
Seja bem vindo(a) IGTI
```

Capítulo 3. Variáveis

Na programação, a variável é uma posição frequentemente localizada na memória, previamente identificada e capaz de reter e representar um dado.

Em Python, os tipos de dados básicos são:

- Tipo inteiro: armazena números inteiros.
- Tipo string: armazena um conjunto de caracteres.
- Tipo float: armazena números em formato decimal.
- Tipo Booleano: armazena sempre True (Verdadeiro) ou False (Falso).

Cada variável pode armazenar apenas um tipo de dado.

Diferente de outras linguagens de programação, em Python não é preciso declarar de que tipo será a variável. Quando se faz uma atribuição de valor, a variável automaticamente se torna a do tipo de valor apresentado (duck tape).

Os nomes das variáveis devem ser iniciados com uma letra, mas pode possuir outros tipos de caracteres, como números e símbolos. O símbolo underline (_) também é aceito no início do nome de variáveis.

Veja alguns exemplos de variáveis:

```
numero = 123
print(numero)
123
```

A variável `numero` automaticamente é considerada do tipo **inteiro**.

```
numero = 'hello world'
print(a)
hello word
```

A variável “numero” automaticamente é considerada do tipo **string**.

```
numero = 4.56
```

```
print(a)
```

```
4.56
```

A variável `numero` automaticamente é considerada do tipo ***float***.

Cuidado: Colocamos de propósito o nome da variável de `numero`. O nome que você escolhe para uma variável não significa nada a não ser para você. Escolha bons nomes!

Observações:

- A string deve ser apresentada entre aspas simples, aspas simples triplas, aspas duplas ou aspas duplas triplas.
- Em linguagens de programação utilizamos o ponto (`.`) para fazer a separação decimal dos números

Escopo de variáveis

As variáveis podem ser classificadas como globais ou locais, conforme o local onde elas foram declaradas dentro do documento.

- **Variáveis globais** são aquelas declaradas no início ou ao longo de um documento, e que podem ser utilizadas em qualquer parte dele.
- **Variáveis locais** são declaradas dentro de alguma função, e será válida somente para este bloco.

Strings

String é uma sequência de caracteres. Na linguagem Python, um dado é considerado do tipo string sempre que:

- Estiver entre aspas simples: `'12345'`, `'IGTI'`, `'aula'`

- Estiver entre aspas simples triplas: ““123””, ““IGTI””, ““aula””
- Estiver entre aspas duplas: “12345”, “IGTI”, “aula”
- Estiver entre aspas duplas triplas: “““12345”””, “““IGTI”””, “““aula”””

Observação: A forma mais comum de utilização na linguagem Python é **aspas simples**.

Concatenação de strings

Concatenação é um termo utilizado em computação para designar a operação de unir o conteúdo de duas strings.

Para concatenar string, utiliza-se o operador **+**.

Veja os exemplos abaixo:

```
print('boot' + 'camp')
print('bootcamp' + 'IGTI')
print('bootcamp ' + 'IGTI')
```

```
bootcamp
bootcampIGTI
bootcamp IGTI
```

Manipulação de strings

Manipular as strings é realizar operações úteis nas strings com métodos embutidos, como encontrar o comprimento de uma string, substituir um caractere de uma string por outro, etc.

Vejam os principais métodos para manipular as strings:

<pre>teste = 'Todo azul do mar.' len(teste)</pre>	len- encontra o tamanho de uma string
<pre>17</pre>	
<pre>[] teste = teste.replace('azul', 'verde') teste</pre>	replace- substitui na string o primeiro trecho indicado pelo segundo
<pre>'Todo verde do mar.'</pre>	
<pre>[] teste.count('o')</pre>	count- informa quantas vezes um caractere aparece na string
<pre>3</pre>	
<pre>[] teste.find('r')</pre>	find- retorna o primeiro índice de um determinado caractere na string
<pre>7</pre>	
<pre>[] teste.split()</pre>	split- transforma a string em uma lista
<pre>['Todo', 'verde', 'do', 'mar.']</pre>	
<pre>[] teste.split('Todo')</pre>	split(trecho da string)- separa da string somente o trecho selecionado
<pre>['', ' verde do mar.']</pre>	
<pre>[] 'aquele'.join(teste.split('Todo'))</pre>	join- junta os itens da string com um item delimitado
<pre>'aquele verde do mar.'</pre>	
<pre>[] teste.upper()</pre>	upper- retorna todos os itens em caixa alta.
<pre>'TODO VERDE DO MAR.'</pre>	
<pre>[] teste.lower()</pre>	lower- retorna todos os itens em caixa baixa
<pre>'todo verde do mar.'</pre>	

Capítulo 4. Listas

Lista, em Python, é um conjunto ordenado de elementos ou itens. Podemos colocar qualquer tipo de dado em uma lista, e ela é mutável, sendo possível manipulá-la.

Diferentemente de outras linguagens de programação, nas quais cada lista possui tamanho e o tipo de dado fixo, em Python podemos criar uma lista e adicionar qualquer tipo de elemento.

O primeiro valor de uma lista tem índice 0. Os seus elementos devem ser delimitados por colchetes.

Veja abaixo representações:

```
lista1 = [1, 2, 3, 4, 5]
```

```
lista2 = ['Bootcamp', 'IGTI']
```

```
lista3 = ['B', 'o', 'o', 't', 'c', 'a', 'm', 'p', 'l', 'G', 'T', 'I']
```

```
lista4 = [1, 'b', True, [1, 2, 3], 45.11]
```



Veja que nesta lista temos elementos inteiros, string, booleano, lista, float.

Criar listas com range

Range é uma função utilizada para gerar sequência de números de forma ordenada. Possui 3 formas de serem geradas e apresentadas:

1. `range(n)`, que gera uma sequência de **0** a **n-1**.

Exemplo: `range(7)` → (0, 1, 2, 3, 4, 5, 6)

2. `range(a, n)`, que gera uma sequência entre **a** e **n-1**.

Exemplo: `range(5, 10)` → (5, 6, 7, 8, 9)

3. `range(a, n, s)`, que gera uma sequência entre **a** e **n-1**, sendo **s** o valor do passo entre os números.

Exemplo: `range(2, 10, 3)` → (2, 5, 8)

Exemplos de criação de listas utilizando o `range`:

```
lista5 = list(range(10))
```

```
print(lista5)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
lista6 = list(range(2, 8))
```

```
print(lista6)
```

```
[2, 3, 4, 5, 6, 7]
```

Manipular listas

Como a lista é uma estrutura mutável, existem várias funções que podem ser utilizadas para manipulá-las. Vejamos algumas delas a seguir:

```
[2] lista1 = [1, 7, 9, 2, 65, 4, 31, 55]
(lista1.sort())
print(lista1)
```

sort- ordenar uma lista em ordem crescente

```
[1, 2, 4, 7, 9, 31, 55, 65]
```

```
[10] lista2 = ['abacaxi', 'laranja', 'banana', 'caju']
lista2.append('uva')
print(lista2)
```

append- acrescentar um item ao final da lista

```
['abacaxi', 'laranja', 'banana', 'caju', 'uva']
```

```
[12] lista2.append(['batata', 'cenura'])
print(lista2)
```

append- com esta função também é possível acrescentar uma lista dentro da lista

```
['abacaxi', 'laranja', 'banana', 'caju', 'uva', ['batata', 'cenura']]
```

```
[17] lista3 = [1, 12, 43, 73, 24, 92]
lista3.extend([23, 41, 66])
print(lista3)
```

extend- acrescentar uma lista em formato de itens ao final de uma lista

```
[1, 12, 43, 73, 24, 92, 23, 41, 66]
```

```
[18] lista4 = [1, 2, 3, 4, 5]
lista4.insert(3, 'novo elemento')
print(lista4)
```

insert- acrescentar um novo item à lista em um local específico, informando a posição do índice

```
[1, 2, 3, 'novo elemento', 4, 5]
```

```
[20] lista5 = lista3 + lista4
print(lista5)
```

somar duas listas- Podemos criar uma lista a partir de 2 outras listas, somando-as.

```
[1, 12, 43, 73, 24, 92, 23, 41, 66, 1, 2, 3, 'novo elemento', 4, 5]
```

```
[22] lista6 = ['agua', 'terra', 'fogo', 'ar']
lista6.reverse()
print(lista6)
```

reverse- reverter a ordem da lista, do último ao primeiro

```
['ar', 'fogo', 'terra', 'agua']
```

```
[24] lista7 = lista6.copy()
print(lista7)
```

copy- fazer uma cópia de uma lista

```
['ar', 'fogo', 'terra', 'agua']
```

```
lista8 = [1, 2, 3, 4, 5, 6]
lista8 = list(enumerate(lista8))
print(lista8)
```

enumerate- gerar uma lista enumerada

```
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]
```

```
[39] lista9 = [10, 11, 12, 13]
print(lista9 * 3)
```

multiplicação- repetir os elementos de uma lista multiplicando-os.

```
[10, 11, 12, 13, 10, 11, 12, 13, 10, 11, 12, 13]
```


Capítulo 5. Tuplas

Assim como a lista, a tupla é um conjunto sequencial de itens. A grande diferença entre elas é que, enquanto podemos adicionar elementos à lista em qualquer momento, a tupla é **imutável**. Toda operação em uma tupla gera uma nova tupla.

Diferentemente das listas, a tupla não é delimitada através de colchetes. Ela pode ser representada das seguintes formas:

1. Tupla delimitada por parênteses.

Exemplo: `tupla1 = (1, 2, 3, 4, 5)`

2. Tupla declarada sem a utilização de parênteses.

Exemplo: `tupla2 = 1, 2, 3, 4, 5`

3. Tupla contendo um único elemento.

Exemplo: `tupla3 = 1,`



Observação: o que caracteriza a tupla em uma declaração contendo apenas um elemento é a utilização da vírgula. Um único número dentro de parênteses não é uma tupla.

Manipular tuplas

Ainda que a tupla seja uma estrutura imutável e seus elementos não possam ser alterados livremente, existem algumas funções através das quais é possível manipulá-la. Vejamos a seguir:

```
[1] tupla1 = tuple(range(11))  
print(tupla1)
```

Podemos criar uma tupla a partir do range

```
↳ (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
[5] tupla3 = (1, 2, 3, 4)  
print(sum(tupla3))  
print(max(tupla3))  
print(min(tupla3))
```

Somente para tuplas formadas exclusivamente de valores inteiros:
sum- soma dos valores da tupla
max- maior valor da tupla
min- menor valor da tupla

```
↳ 10  
4  
1
```

```
[6] print(len(tupla3))
```

Len- conta o tamanho de uma tupla

```
↳ 4
```

```
[4] tupla1 = (1, 2, 3)  
tupla2 = (4, 5, 6)  
print(tupla1 + tupla2)  
print(tupla1)  
print(tupla2)
```

Podemos somar duas tuplas, mas podemos observar que seus valores não foram alterados.

```
↳ (1, 2, 3, 4, 5, 6)  
(1, 2, 3)  
(4, 5, 6)
```

```
[9] tupla1 = (1, 2, 3)  
tupla2 = (4, 5, 6)  
tupla3 = tupla1 + tupla2  
print(tupla3)
```

Podemos criar uma tupla a partir de tuplas existentes, somando-as.

```
↳ (1, 2, 3, 4, 5, 6)
```

```
[10] tupla1 = (1, 2, 3)  
print(1 in tupla1)  
print(7 in tupla1)
```

Podemos verificar de determinado elemento está contido na tupla.

```
↳ True  
False
```

Capítulo 6. Dicionários

Dicionário é uma coleção de itens na qual cada elemento que a compõe possui uma chave associada a ele.

Os dicionários são delimitados por chaves { }. A chave de acesso e o valor são separados por dois pontos, sendo que ambos podem ser de qualquer tipo de dado. Diferentemente da lista e da tupla, os dicionários não são indexados.

Veja abaixo a representação de um dicionário:

```
semana = {'seg': 'segunda', 'ter': 'terça-feira', 'qua': 'quarta-feira'}
```

```
dados = {'conta': 123.45678, 'agencia': 4589, 'senha': 'a457qw'}
```



Observação: Não podemos ter chaves repetidas em dicionários.

Manipular dicionários

Vejamos abaixo algumas funções para manipulação de dicionários:

```
[22] estacoes = {'ver': 'verão', 'out': 'outono', 'inv': 'inverno', 'pri': 'primavera'}
      print(estacoes)

      estacoes = dict(ver = 'verão', out = 'outono', inv = 'inverno', pri = 'primavera')
      print(estacoes)
```

Criando dicionários:
- escrevendo-o
- utilizando a função **dict**

```
{'ver': 'verão', 'out': 'outono', 'inv': 'inverno', 'pri': 'primavera'}
{'ver': 'verão', 'out': 'outono', 'inv': 'inverno', 'pri': 'primavera'}
```

```
[16] estacoes = {'ver': 'verão', 'out': 'outono', 'inv': 'inverno', 'pri': 'primavera'}
      print(estacoes.get('out'))
      print(estacoes.get('frio'))
```

get: acessar elementos de um dicionário. Retorna a resposta None quando não localiza o elemento procurado.

```
{'ver': 'verão', 'out': 'outono', 'inv': 'inverno', 'pri': 'primavera'}
```

```
estacoes = {'ver': 'verão', 'out': 'outono', 'inv': 'inverno', 'pri': 'primavera'}
print('ver' in estacoes)
print('verão' in estacoes)
print('inv' in estacoes)
```

Podemos verificar se determinado elemento se encontra em um dicionário. Atenção! A busca sempre é feita através da chave.

```
True
False
True
```

```
[20] inventario = {'caneta': 20, 'caderno': 55, 'livro': 40}
      print(inventario)

      inventario['lapis'] = 30
      print(inventario)

      inventario['caneta'] = 15
      print(inventario)
```

Podemos adicionar elementos, adicionando uma chave e um valor.
Podemos atualizar algum elemento, escrevendo a chave e o novo valor.

```
{'caneta': 20, 'caderno': 55, 'livro': 40}
{'caneta': 20, 'caderno': 55, 'livro': 40, 'lapis': 30}
{'caneta': 15, 'caderno': 55, 'livro': 40, 'lapis': 30}
```

```
[21] inventario = {'caneta': 20, 'caderno': 55, 'livro': 40}
      inventario.pop('caneta')
      print(inventario)
```

pop: podemos deletar elementos dos dicionários.

```
{'caderno': 55, 'livro': 40}
```

Capítulo 7. Set

Na linguagem Python, conjuntos são chamados de *set*, e fazem referência à teoria de conjuntos da matemática. Suas principais características são:

- Os elementos não são armazenados em uma ordem específica.
- Os sets não tem elementos repetidos.
- Os sets são delimitados em Python por chaves { }, assim como nos dicionários. Porém, aqui não haverá a presença da chave de acesso/valor, somente o valor. Qualquer tipo de dado pode ser colocado em um set, inclusive misturados.

Veja abaixo a representação de um set:

– numero = {1, 2, 3, 4, 5}

Manipular sets

Vejamos, no quadro abaixo, algumas funções para manipulação de sets:

<pre>numero = {1, 2, 3, 4, 1, 1, 2, 2, 4} print(numero)</pre>	<p>Como definir um set.</p> <p>Como não há repetição de valores, não é possível adicionar algum já existente, ele será ignorado.</p>
<pre>{1, 2, 3, 4}</pre>	
<pre>[5] numero = {1, 2, 3, 4} numero.add(5) numero.add(1) numero.add(6) print(numero)</pre>	<p>add- adiciona um item ao set. Caso seja repetido, ele será ignorado e não será adicionado ao set.</p>
<pre>{1, 2, 3, 4, 5, 6}</pre>	
<pre>[7] meses = {'janeiro', 'fevereiro', 'março'} meses.discard('janeiro') meses.discard('abril') print(meses)</pre>	<p>discard- remove elementos do set. Se solicitarmos que seja removido algum elemento que não exista no set, nenhum erro é gerado.</p> <p>ATENÇÃO! Sets não são indexados, portanto se neste exemplo solicitarmos discard(3), nada será executado.</p>
<pre>{'fevereiro', 'março'}</pre>	
<pre>[16] esportes1= {'futebol', 'basquete', 'volei', 'natacao', 'danca'} esportes2= {'natacao', 'corrida', 'boxe', 'futebol', 'tenis'} esportestotal = esportes1.union(esportes2) print(esportestotal) esportes_ambos = esportes1.intersection(esportes2) print(esportes_ambos) exclusivos_esportes1 = esportes1.difference(esportes2) exclusivos_esportes2 = esportes2.difference(esportes1) print(exclusivos_esportes1) print(exclusivos_esportes2)</pre>	<p>No exemplo ao lado, temos dois conjuntos formados por esportes.</p> <p>union- gerar um único set com todos os esportes</p> <p>intersection- gerar um conjunto com os esportes que são semelhantes em ambos os sets.</p> <p>difference- gerar um set com os esportes que são exclusivos de um set.</p>
<pre>{'tenis', 'volei', 'basquete', 'natacao', 'boxe', 'futebol', 'danca', 'corrida'} {'futebol', 'natacao'} {'basquete', 'danca', 'volei'} {'boxe', 'tenis', 'corrida'}</pre>	

Capítulo 8. Expressões booleanas

A álgebra booleana foi criada por George Boole. Expressão booleana é toda expressão onde o resultado é **True** (*verdadeiro*) ou **False** (*falso*).

Na programação, muitas vezes precisamos saber se uma expressão é **TRUE** ou **FALSE**, ou seja, se ela é verdadeira ou falsa.

Exemplo:

```
print(1 > 2)
```

False

```
print (1 < 2)
```

True

```
print (1 == 2)
```

False

```
print (1 <= 2)
```

True



Observação: os valores booleanos devem ser expressos com as iniciais maiúsculas.

Quando executamos uma condição de instrução **if**, que veremos mais à frente, o Python retorna **True** ou **False**

Exemplo:

```
a = 5
```

```
b = 10
```

```
if a > b:
```

```
    print('A é maior que B')
```

```
else:
```

```
print('B é maior que A')
```

B é maior que A

Capítulo 9. Operadores lógicos em Python

Expressões lógicas e booleanas são formadas por valores e os operadores lógicos:

- `and` (*e*)
- `or` (*ou*)
- `not` (*não*)

Operador `and`

O operador **`and`** retorna **`True`** somente quando as duas declarações forem verdadeiras.

Exemplo:

```
a = 2
```

```
print(a < 8 and a < 15)
```

`True`

```
a = 10
```

```
print(a < 8 and a < 15)
```

`False`

Operador `or`

O operador **`or`** retorna **`True`** se, pelo menos, uma das declarações for verdadeira.

Exemplo:

```
a = 2
```

```
print(a < 1 or a < 15)
```

`True`


```
a = 10
```

```
print(a < 1 or a > 15)
```

False

Operador not

O operador **not** retorna o resultado de forma invertida, ou seja, é uma negação. Se o resultado da operação for verdadeiro, retorna como falso.

Exemplo:

```
a = 2
```

```
print (not(a > 1 and a < 15))
```

False

```
a = 10
```

```
print(not(a < 1 or a > 15))
```

True

Capítulo 10. Estruturas condicionais: if, else, elif

O Python suporta as condições lógicas usuais da matemática:

- igual a : `x == y`
- diferente de: `x != 8`
- menor que: `x < y`
- maior que: `x > y`
- menor ou igual a : `x <= y`
- maior ou igual a: `x >= y`

Muitas vezes, porém, os programas precisam tomar decisões sobre qual comando deve executar, selecionando entre opções possíveis. Para isso precisamos estabelecer as **condições** para que a decisão seja tomada.

Os comandos utilizados para realizar as operações condicionais são:

- `if` (*se*)
- `else` (*senão*)
- `elif` (*junção de if e else*)

Exemplo:

```
idade = 18
```

```
if idade < 18:
```

```
    print('Menor de idade')
```

```
elif idade == 18:
```

```
    print('Tem 18 anos')
```

```
else:
```

```
print('Maior de idade')
```

Tem 18 anos

Capítulo 11. Loops

O **loop** (*estrutura de repetição*) é uma instrução que deve ser executadas várias vezes em sequência.

Existem dois tipos de loops em python: **for** (*para*) e **while** (*enquanto*).

Loop for

Utilizamos loops para iterar sobre sequências ou sobre valores iteráveis.

Com o loop **for**, podemos executar um conjunto de instruções uma vez para cada item em uma lista, tupla, conjunto, etc.

Ele é executado quando se quer iterar sobre um bloco de códigos um número determinado de vezes.

Exemplos:

```
for numero in range(1, 10):
```

```
    print(numero)
```

1

2

3

4

5

6

7

8

9

```
nome = 'Bootcamp'
```

```
for letra in nome:
```

```
    print(letra)
```

B
o
o
t
c
a
m
p

Observação: *Iteração é a execução repetida de uma sequência de instruções*

Loop while

O loop **while** executa um conjunto de instruções enquanto uma expressão booleana for verdadeira.

Exemplos:

```
numero = 0
```

```
while numero < 5:
```

```
    print(numero)
```

```
    numero = numero +1
```

0

1

2

3

4

```
senha = "
```

```
while senha != 'IGTI':
```

```
    senha = input('Digite a sua senha:')
```

```
print('Parabéns! Senha correta!!!')
```

Digite a sua senha:123

Digite a sua senha:1452

Digite a sua senha:425

Digite a sua senha:IGTI

Parabéns! Senha correta!!!

Capítulo 12. Funções em Python

As funções em Python são pequenas partes do código criadas para executar alguma função específica. Elas são reutilizáveis, ou seja, a partir do momento em que são criadas, é possível executá-las em qualquer parte do programa.

Algumas funções já são integradas ao Python. São chamadas de **Built in**, e já estamos as utilizando. Entre elas, podemos citar:

`print()`

`input()`

`sort()`

`len()`

Em python, a palavra que define o início da função é **def** e tem a seguinte estrutura:

def nome_da_função(parâmetros de entrada):

 bloco da função

Exemplo de função:



```
def estudo_funcao():  
    print('Aprendendo funcoes')  
  
estudo_funcao()
```

Aprendendo funcoes

Referências

Documentação do Python. Disponível em <https://www.python.org/>. Acesso em: 10 ago. 2020.

PEP 8. Disponível em: <https://www.python.org/dev/peps/pep-0008/>. Acesso em: 10 ago. 2020.

Python organization. Disponível em <https://python.org.br/>. Acesso em: 10 ago. 2020.

WIKIPEDIA. *Python*. Disponível em: <https://pt.wikipedia.org/wiki/Python> Acesso em: 10 ago. 2020.

WIKIPEDIA. *Variável (programação)*. Disponível em [https://pt.wikipedia.org/wiki/Vari%C3%A1vel_\(programa%C3%A7%C3%A3o\)](https://pt.wikipedia.org/wiki/Vari%C3%A1vel_(programa%C3%A7%C3%A3o)) Acesso em: 10 ago. 2020.