

INTRODUÇÃO AO *PYTHON*

Objetivos

-  Apresentar as origens da linguagem *Python*.
-  Apresentar as aplicações da linguagem *Python*.

Conteúdos

-  Origem do *Python*
-  Por quê programar em *Python*

Prof. Marivaldo Pereira dos Santos

INTRODUÇÃO AO PYTHON

Em 1989, Guido Van Rossum publicou a primeira versão do *Python*, através do Instituto de Pesquisa Nacional para Matemática e Ciência da Computação. Derivada da linguagem C, *Python* surgiu para ser uma alternativa mais simples e produtiva que a própria linguagem C.



INTRODUÇÃO AO PYTHON

Em 1991, *Python* ganha sua primeira versão estável, começando a gerar uma comunidade de desenvolvedores, empenhada em aprimorá-la, porém, só no ano de 1994 é que foi lançada a versão 1.0, ou seja, a primeira versão oficial e não mais de testes. Desde seu lançamento oficial, *Python* já passou por diversos aperfeiçoamentos em sua estrutura e bibliotecas.

INTRODUÇÃO AO *PYTHON*

Atualmente o *Python* está integrado em praticamente todas as novas tecnologias, assim, como é fácil implementá-la em sistemas obsoletos. Grande parte das distribuições *Linux* possuem *Python* nativamente e seu reconhecimento desde 2009 fez que virasse a linguagem padrão do Curso de Ciências da Computação do MIT.

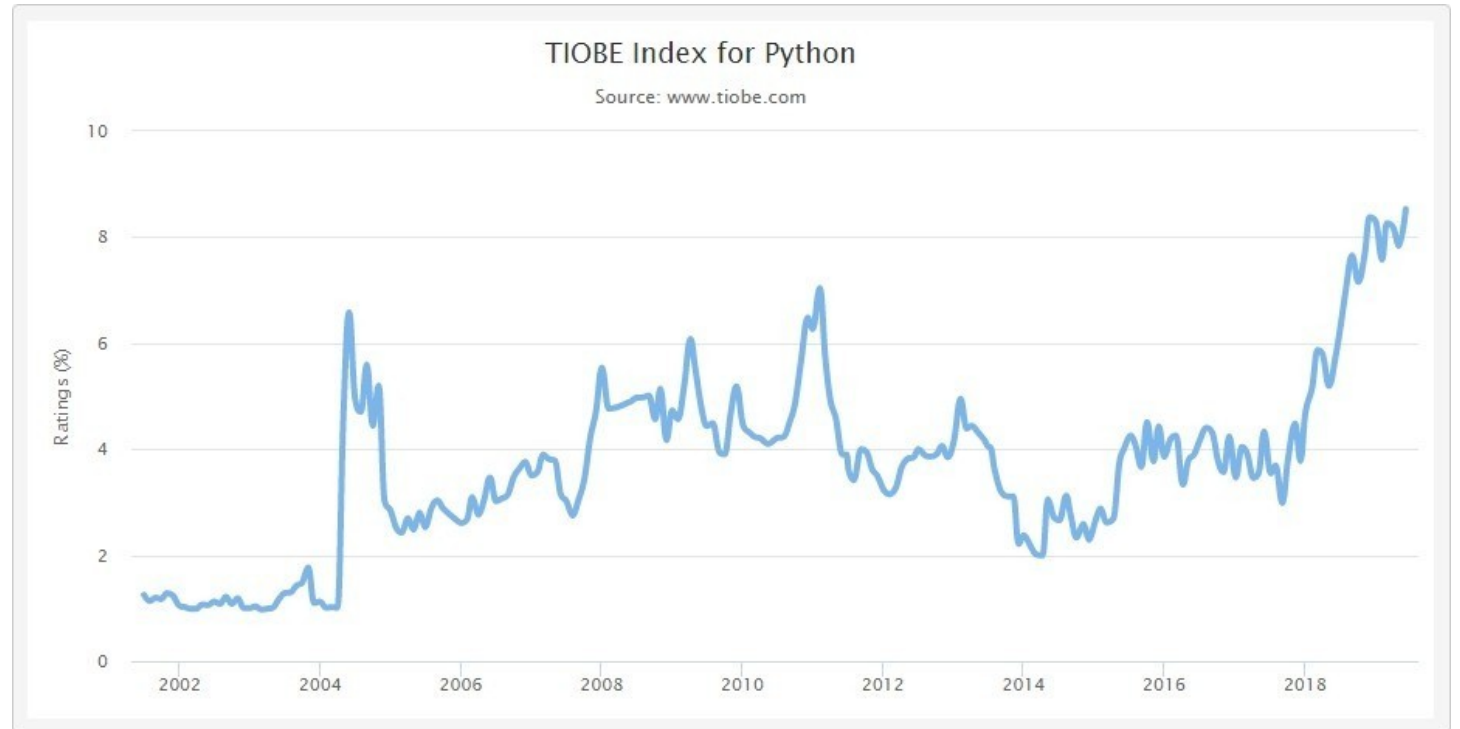
INTRODUÇÃO AO PYTHON

Python é uma linguagem de programação moderna, com ela é possível criar qualquer tipo de sistema para qualquer propósito e plataforma. Com *Python* é possível desenvolver para qualquer sistema operacional, *web*, *mobile*, *data science*, *machine learning*, *blockchain* etc. Exemplos de aplicações que usam parcial ou totalmente *Python*:

- *Youtube*
- *Google*
- *Instagram*
- *DropBox*
- *SpoGy*
- *Reddit*

INTRODUÇÃO AO PYTHON

De acordo com as estatísticas de sites de linguagens, *Python* é uma das linguagens com maior crescimento em relação as demais no mesmo período. O *Tiobe Index* foi a linguagem de programação de 2018.



INTRODUÇÃO AO PYTHON

Em países mais desenvolvidos tecnologicamente e até mesmo as escolas de ensino fundamental estão adotando o ensino de programação em sua grade de disciplinas, boa parte delas ensinando linguagem *Python*. Espera-se que a linguagem *Python* cresça exponencialmente, uma vez que novas áreas como *data Science* e *machine learning* se popularizem ainda mais.

Estudos indicam que para os próximos 10 anos, cerca de um milhão de novas vagas surgirão, demandando profissionais de programação. Certamente uma parcela dessa demanda será para programadores com conhecimentos em *Python*.

REFERÊNCIAS



PYTHON – Tiobe Index. Disponível em: <<https://www.tiobe.com/tiobe-index/python/>> Acesso em 26 mai. 2019

FELTRIN, Fernando. Python do zero à programação orientada a objetos. Amazon Kindle. Posição 272-367 Site

Oficial Python. Disponível em: <<https://www.python.org>> Acesso em 15 jun 2019

TIPOS DE DADOS

Objetivos

-  Apresentar os tipos de dados da linguagem *Python*.
-  Apresentar exemplos de código com os tipos de dados.

Conteúdos

-  Tipos de dados
-  Estrutura básica de um programa

TIPOS DE DADOS

Antes de iniciarmos com as definições dos tipos de dados em *Python*, vamos conhecer um pouco da estrutura de um programa escrito em *Python*. Enquanto que em algumas linguagens de programação precisamos definir uma estrutura básica para iniciar a programação, *Python* já nos oferece praticamente tudo o que é necessário pré-carregado, de forma que ao abrirmos uma ferramenta de desenvolvimento, a nossa preocupação inicial é realmente programar.

TIPOS DE DADOS

Python é uma linguagem “*bateries included*”, termo em inglês para pilhas incluídas, ou seja, ele já vem com o necessário para seu funcionamento pronto para uso. O clássico exemplo “Olá mundo” para ser apresentado na tela é escrito da seguinte forma:

```
print('Olá Mundo!')
```

Por mais simples que pareça, isto é tudo que precisamos para que seja exibida a mensagem na tela do computador. Mais adiante veremos com mais detalhes a sintaxe da linguagem *Python*.

TIPOS DE DADOS

Já quando vamos realizar operações aritméticas, precisamos tratar os números conforme seu tipo, por exemplo:

8 é um número inteiro

8.2 é um número real

O número 8 é identificado em *Python* como `int` (número inteiro) e 8.2 como *float* (número com casa decimal ou ponto flutuante).

TIPOS DE DADOS

Durante a escrita dos códigos temos que ter o cuidado para não misturar os tipos de dados, pois o interpretador não irá distinguir que tipo de operação está sendo realizada.

Veja o exemplo:

Podemos dizer que Maria tem 9 anos, e neste contexto para o interpretador o 8 é uma *string*, como qualquer outra palavra da sentença. Agora se desejamos somar dois números, o interpretador espera que estes números sejam *int* ou *float*, mas nunca *string*.

TIPOS DE DADOS

Tipos de dados mais comuns em Python:

Tipo	Descrição	Exemplo
int	Número inteiro, sem casas decimais	15
float	Número real, com casas decimais	27.8
bool	Booleano / Binário (0 ou 1)	0
string	Texto com qualquer caractere alfanumérico	'Disciplina Linguagem de Programação I'
list	Listas	[4, 'João', 45.8]
dict	Dicionários	{'nome':'Fulano de Tal','idade':23}

TIPOS DE DADOS

Note, que para cada tipo de dado, há uma forma de representação (sintaxe própria), para que o interpretador os reconheça como tal. Vejamos alguns exemplos de criação de variáveis:

```
var_string = 'Conjunto de caracteres alfanuméricos' var_int = 45
```

```
var_float = 99.87
```

```
var_bool = 1
```

```
var_lista = [1, 6, 'Maria', 'Joana'] var_dicionário = {'nome': 'Jonas',  
'idade': 35}
```




REFERÊNCIAS

FELTRIN, Fernando. Python do zero à programação orientada a objetos. Amazon Kindle. Posição 272-367

Site Oficial Python. Disponível em: <<https://www.python.org>> Acesso em 15 jun 2019

DECLARAÇÃO DE VARIÁVEIS

Objetivos

-  Apresentar, declarar e utilizar variáveis em linguagem *Python*.
-  Apresentar exemplos de código com declarações de variáveis.
-  Apresentar exemplos de código com entrada de dados.

Conteúdos

-  Declaração de variáveis
-  Atribuição de valores para variáveis
-  Leitura de dados, via teclado

DECLARAÇÃO DE VARIÁVEIS

A linguagem *Python* é um tipo de como
linguagem de programação conhecida dinamicamente
tipada, ou seja, *Python* não obriga o
desenvolvedor a declarar o tipo

DECLARAÇÃO DE VARIÁVEIS

Note, que a tipagem dinâmica só funciona quando usado o sinal de atribuição, se apenas colocarmos uma linha sem o sinal de atribuição o interpretador *Python* mostrará uma mensagem de erro. A linguagem Python, também permite criarmos nossas variáveis sem atribuição de valor. E isso é útil, quando precisamos definir variáveis que receberão conteúdo em um momento posterior à sua declaração. Neste caso, o tipo de dado da variável deve ficar explícito. Veja o exemplo de declaração de uma variável *float* vazia:

```
minhaVariavel: float
```

DECLARAÇÃO DE VARIÁVEIS

Note, que a tipagem dinâmica só funciona quando usado o sinal de atribuição, se apenas colocarmos uma linha sem o sinal de atribuição, o interpretador *Python* mostrará uma mensagem de erro. A linguagem *Python*, também permite criarmos nossas variáveis sem atribuição de valor. E isso é útil, quando precisamos definir variáveis que receberão conteúdo em um momento posterior à sua declaração. Neste caso, o tipo de dado da variável deve ficar explícito com uso de dois pontos. Veja o exemplo de declaração de uma variável *float* vazia:

```
minhaVariavel: float
```

DECLARAÇÃO DE VARIÁVEIS

Na declaração de variáveis temos que tomar cuidado com a forma de escrita dos nomes, pois *Python* é '*case sensitive*', ele faz diferenciação entre letras maiúsculas e minúsculas.

```
Numero01 = 10
```

```
numero01 = 10
```

As variáveis acima são diferentes, apesar de conter o mesmo valor, a sugestão é que se escreva o nome das variáveis começando com letras minúsculas; e evite o uso de caracteres especiais para nomear variáveis.

DECLARAÇÃO DE VARIÁVEIS

Leitura de dados pelo teclado: para facilitar a interação do usuário com um programa em *Python*, podemos ler conteúdo digitado pelo teclado e armazenar em variáveis. Para isso, usamos o comando `'input'`. Veja o exemplo:

```
teclado = input('Digite um número inteiro : ')
numero01 = int(teclado)
teclado = input('Digite outro número inteiro : ')
numero02 = int(teclado)
print('A soma dos dois números é: ', numero01 + numero02)
```

DECLARAÇÃO DE VARIÁVEIS

Declaração de variáveis, explicitando seu tipo: conforme visto, anteriormente, é possível declarar uma variável determinando seu tipo e depois atribuir um valor. Veja o exemplo:

```
>>> var01: int
>>> var02: int
>>> var01 = 20
>>> var02 = 55
>>> print(var01 + var02)
75
```



REFERÊNCIAS

FELTRIN, Fernando. Python do zero à programação orientada a objetos. Amazon Kindle. Posição 272-367

Site Oficial Python. Disponível em: <<https://www.python.org>> Acesso em 28 jun 2019

OPERADORES

Objetivos **ARITMÉTICOS**

-  Apresentar a sintaxe dos operadores aritméticos.
-  Apresentar como utilizar os operadores aritméticos.

Conteúdos

-  Operadores Relacionais, linguagem *Python*
-  Exemplos de Código

OPERADORES ARITMÉTICOS

Operadores Aritméticos: Os operadores aritméticos, como o nome sugere, serão usados para realizar operações matemáticas nos blocos de código a serem desenvolvidos. O *Python* por padrão já vem com bibliotecas pré-alocadas, que nos permitem a qualquer momento fazer operações matemáticas simples como soma, subtração, multiplicação e divisão. Para operações mais complexas, também, é possível importar bibliotecas externas para executar tais funções.

OPERADORES ARITMÉTICOS

Operadores Aritméticos	
Operador	Função
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto inteiro da divisão
**	Exponenciação
//	Divisão que retorna um número inteiro arredondado

OPERADORES ARITMÉTICOS

Exemplo adição:

```
print(5 + 6)
```

O resultado será 11

```
>>> print(5 + 6)  
11
```

OPERADORES RELACIONAIS, LÓGICOS E ARITMÉTICOS

Exemplo subtração:

```
print(5- 6)
```

O resultado será -1

```
>>> print(5 - 6)  
-1
```

OPERADORES ARITMÉTICOS

Exemplo multiplicação:

```
print(5 * 6)
```

O resultado será 30

```
>>> print(5 * 6)  
30
```

OPERADORES ARITMÉTICOS

Exemplo divisão:

```
print(12 / 4)
```

O resultado será 3.0

Obs: o resultado da divisão

sempre será um valor

float

```
>>> print(12 / 4)  
3.0
```

OPERADORES ARITMÉTICOS

Exemplo resto inteiro da divisão:

```
print(15 % 2)
```

O resultado será 1

```
>>> print(15 % 2)  
1
```


OPERADORES ARITMÉTICOS

Exemplo exponenciação:

```
print(3 ** 2)
```

O resultado será 9

```
print(3 ** 4)
```

O resultado será

81

```
>>> print(3 ** 2)  
9
```

```
>>> print(3 ** 4)  
81
```

OPERADORES ARITMÉTICOS

Exemplo divisão com resto inteiro:

`print(15 // 2)`

O resultado será 7 `print(15.2 //`

`6.2)`

O resultado será 6.0

```
>>> print(15 // 2)
```

```
7
```

```
>>> print(15.2 // 2.2)
```

```
6.0
```



REFERÊNCIAS

FELTRIN, Fernando. Python do zero à programação orientada a objetos. Amazon Kindle. Posição 272-367

Site Oficial Python. Disponível em: <<https://www.python.org>> Acesso em 15 jun 2019

OPERADORES RELACIONAIS

Objetivos

-  Apresentar a sintaxe dos operadores relacionais.
-  Apresentar o modo como utilizar operadores relacionais em linguagem *Python*.

Conteúdos

-  Operadores Relacionais Exemplos de Código
- 

OPERADORES RELACIONAIS

Operadores relacionais são, em essência, aqueles que fazem a comparação entre dois ou mais operandos. Estes operadores possuem uma sintaxe própria, que deve ser respeitada para que não haja conflito com o interpretador de código *Python*. O resultado de uma comparação relacional será dada como um valor booleano, *True* ou *False*.

OPERADORES RELACIONAIS

Operadores Relacionais	
Operador	Função
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou menor que
==	Igual a
!=	Diferente de

OPERADORES RELACIONAIS

Exemplo operadores relacionais :

```
>>> var01 = 10
>>> var02 = 5
>>> print(var01 > var02)
True
>>> print(var02 > var01)
False
```

OPERADORES RELACIONAIS

Exemplo operadores relacionais :

```
>>> numero01 = -20
>>> numero02 = 5
>>> print(numero01 == numero02)
False
>>> print(numero01 != numero02)
True
```


OPERADORES RELACIONAIS

Exemplo operadores relacionais :

```
>>> numA = 3
>>> numB = 4
>>> print(numB >= numA)
True
>>> print(numA <= numB)
True
```

REFERÊNCIAS

FELTRIN, Fernando. Python do zero à programação orientada a objetos. Amazon Kindle. Posição 272-367

Site Oficial Python. Disponível em: <<https://www.python.org>> Acesso em 15 jun 2019

OPERADORES LÓGICOS

Objetivos

-  Apresentar a sintaxe dos operadores lógicos. Apresentar
-  como utilizar operadores lógicos em *Python*.

Conteúdos

-  Operadores Lógicos Exemplos de Código
- 

OPERADORES LÓGICOS

Os operadores lógicos, possuem a mesma base lógica dos operadores relacionais, retornar como valores das comparações valores lógicos *True* ou *False*. A principal diferença é que com os operadores lógicos é possível montar expressões lógicas de maior complexidade, podendo, inclusive, incluir operadores matemáticos e relacionais na mesma expressão.

OPERADORES LÓGICOS

Operadores Lógicos	
Operador	Função
<i>and</i>	Operador Lógico E
<i>or</i>	Operador Lógico OU
<i>not</i>	Operador de negação

OPERADORES LÓGICOS

Por exemplo, a expressão $7 \neq 3$ tem como valor *True* (pois: 7 diferente de 3, verdade), mas se prepararmos a expressão $7 \neq 3 \text{ and } 3 > 5$ o resultado será *False*.

$7 \neq 3 \rightarrow \text{True}$ $3 > 5 \rightarrow \text{False}$

$\text{True and False} \rightarrow \text{False}$

OPERADORES LÓGICOS

Tabela verdade operador lógico <i>and</i>				
V	and	V	=	<i>True</i>
V	and	F	=	<i>False</i>
F	and	V	=	<i>False</i>
F	and	F	=	<i>False</i>

OPERADORES LÓGICOS

Na tabela verdade do operador *and*, basta qualquer proposição ser falsa, para que invalide todas as outras verdadeiras. Veja o exemplo:

V	<i>and</i>	V	<i>and</i>	V	<i>and</i>	V	=	<i>True</i>
V	<i>and</i>	V	<i>and</i>	V	<i>and</i>	F	=	<i>False</i>

OPERADORES LÓGICOS

Tabela verdade operador lógico or				
V	<i>or</i>	V	=	True
V	<i>or</i>	F	=	True
F	<i>or</i>	V	=	True
F	<i>or</i>	F	=	False

OPERADORES LÓGICOS

Na tabela verdade do operador or, basta qualquer proposição ser verdadeira, para que invalide todas as outras falsas, veja o exemplo:

V	or	V	or	V	or	V	=	True
V	or	F	or	F	or	F	=	False



REFERÊNCIAS

FELTRIN, Fernando. Python do zero à programação orientada a objetos. Amazon Kindle. Posição 272-367

Site Oficial Python. Disponível em: <<https://www.python.org>> Acesso em 15 jun 2019

ESTRUTURAS CONDICIONAIS

Objetivos

-  Apresentar como implementar ponto de decisão em um programa *Python*.
-  Desenvolver a sintaxe de programação para estruturas condicionais.

Conteúdos

 Estrutura *if* Estrutura *else* Estrututura *elif*



ESTRUTURAS CONDICIONAIS

Quando estudamos sobre lógica de programação e algoritmos é importante entendermos que toda ação tem uma reação, dessa forma, quando transcrevemos ideias para código, uma coisa que muito ocorre é nos depararmos com tomadas de decisão, as quais irão influenciar os rumos de execução do nosso programa.

ESTRUTURAS CONDICIONAIS

Vários tipos de programa se baseiam em metodologias de estruturas condicionais, onde são programadas todas as possíveis tomadas de decisão que o usuário pode ter e o programa executa e retorna informações, conforme o usuário vai escolhendo dentre as opções apresentadas. Lembremo-nos das aulas de algoritmos. onde preparávamos algoritmos com as estruturas SE e SENÃO para direcionar a execução de um determinado programa, de acordo com a escolha do usuário.

ESTRUTURAS CONDICIONAIS

A sintaxe de programação do *Python* para trabalhar com condicionais é bastante simples em comparação com outras linguagens. A seguir veremos as estruturas condicionais da linguagem *Python*:

- *if*
- *else*
- *elif*

ESTRUTURAS CONDICIONAIS

if (se):

Conforme o interpretador executa o código, linha por linha, ao encontrar a palavra reservada *if*, o mesmo identifica que ali existe um ponto de decisão. De acordo com a decisão que o usuário toma ou de acordo com a validação de algum parâmetro, o código executará uma instrução ou não executará nada, ignorando esta condição e saltando para o bloco seguinte.

Vejamos o exemplo prático

ESTRUTURAS CONDICIONAIS

```
a = 34
b = 67
c = 4
if b > a:
    print('b é maior que a')
```

Um comando *if* sempre será seguido de uma instrução, que sendo verdadeira, irá executar um bloco de código indentado a ela.

ESTRUTURAS CONDICIONAIS

```
a = 34
b = 34
c = 4
if b > a:
    print('b é maior que a')
elif b == a:
    print('a e b são iguais')
```

Perceba que agora temos uma nova condicional

elif. Seguindo a

se a condição do *if* é verdadeira, como não é, o código pula para a segunda condicional, colocando que se b e a são iguais, executará o segundo comando *print*.

ESTRUTURAS CONDICIONAIS

```
a = 33
b = 1
c = 608
if b > a:
    print('b é maior que a')
elif b == a:
    print('a e b são iguais')
else:
    print('b é menor que a')
```

Agora temos a condicional *else* que é acionado quando nenhuma das instruções anteriores forem verdadeiras. O *else* pode ter um bloco de código próprio, porém, perceba que ele não precisa de nenhuma instrução, já que sua finalidade é justamente mostrar que nenhuma condicional anterior foi válida.

ESTRUTURAS CONDICIONAIS

```
var1 = 18
var2 = 2
var3 = 'Maria'
var4 = 4

if var2 > var1:
    print('var1 é maior que var2')
elif var2 == 500:
    print('var2 vale 500')
elif var3 == var2:
    print('var3 e var2 são iguais')
elif var4 is str('4'):
    print('var4 não é do tipo string')
else:
    print('Nenhuma condição é verdadeira')
```

REFERÊNCIAS

FELTRIN, Fernando. Python do zero à programação orientada a objetos. Amazon Kindle. Posição 1749-1857

Site Oficial Python. Disponível em: <<https://www.python.org>> Acesso em 15 jun 2019

ESTRUTURAS DE REPETIÇÃO

Objetivos

- ✎ Apresentar como implementar estruturas de repetição em um programa *Python*.
- ✎ Desenvolver a sintaxe de programação para estruturas de repetição.

Conteúdos

- ✎ Estrutura de repetição *For*
- ✎ Estrutura de repetição *While*

ESTRUTURAS DE REPETIÇÃO

A linguagem *Python* possui dois comandos em *loop*, ou seja, em estrutura de repetição. Usaremos as estruturas de repetição para executar repetidas vezes uma instrução. Os comandos que veremos na sequência são:

for while

ESTRUTURAS DE REPETIÇÃO

FOR

O comando *for* será utilizado em situações que precisamos trabalhar com uma estrutura de repetição, onde conhecemos seus limites, ou seja, quando sabemos o tamanho de um determinado intervalo, o número de elementos contidos em uma lista. Exemplo:

```
for x in range(0, 6):  
    print(x)
```


ESTRUTURAS DE REPETIÇÃO

Note, que no início da linha existe o comando *for* seguido de uma variável temporária *x*, logo em seguida o comando *in range*, que basicamente define um intervalo a percorrer (de 0 até 6); e finalizando o comando *print* para exibir o conteúdo de *x*.

```
for x in range(0, 6):  
    print(x)
```




0
1
2
3
4
5

A contagem dos elementos do intervalo foi de 1 a 5, o número 6 está fora do range, servindo apenas como orientação, para o interpretador reconhecer que ali é o fim do intervalo.

ESTRUTURAS DE REPETIÇÃO

```
nomes = ['Maria', 'João', 'Pedro', 'Letícia']  
for n in nomes:  
    print(n)
```



Maria
João
Pedro
Letícia

Outra situação comum é quando já temos uma lista de elementos e queremos percorrê-la para exibir seu conteúdo. Neste exemplo, como já temos uma lista com 4 nomes e quando o *for* é executado. Ele, internamente, percorrerá todos os valores contidos na lista e inclui os nomes na variável *n*, para no final exibir pelo comando *print*.


ESTRUTURAS DE REPETIÇÃO

WHILE

O comando *while* será executado enquanto em sua instrução houver uma condição verdadeira. Exemplo:

```
x = 1
while x < 6:
    print(x)
    x += 1
```

ESTRUTURAS DE REPETIÇÃO



```
x = 1
while x < 6:
    print(x)
    x += 1
```

1
2
3
4
5

Declarada a variável `x` com valor inicial 1 é, em seguida, colocada a condição de que enquanto o valor de `x` for menor que 6, imprime o valor de `x` e acrescenta 1 repetidamente. Perceba que isto é um *loop*, ou seja, a cada ação o bloco de código salva seu último estado e repete a instrução, até atingir a condição deixar de ser verdadeira.

ESTRUTURAS DE REPETIÇÃO

Outra possibilidade é que durante a execução do *while*, é possível programar um *break*, que é um comando que interrompe a execução de um determinado bloco de código ou instrução. Normalmente, o uso de *break* se dá quando é utilizado por mais de uma condição de parada, que, se a instrução de código atingir qualquer uma dessas condições, ele para sua execução. Por exemplo:



ESTRUTURAS DE REPETIÇÃO

```
x = 1
while x < 10:
    print(x)
    x += 1
    if x == 4:
        break
```

Enquanto o valor da variável x for menor que 10, continue imprimindo seu conteúdo e adicionando 1 ao seu valor, mas se em algum momento x for igual a 4, pare a repetição.

STRINGS

Objetivos

-  Apresentar as funcionalidades do tipo *String* em *Python 3*.
-  Desenvolver a sintaxe de programação com tipos de dados *String*.

Conteúdos

-  Definição de *String* em *Python 3* Formatação de *Strings*
-  Principais comandos com *Strings*
- 

STRINGS

Como visto anteriormente, o tipo de dado *string* é utilizado para armazenar informações alfanuméricas ou qualquer tipo de texto. Quando atribuímos um conjunto de caracteres, representando palavras ou textos, devemos obrigatoriamente seguir a sintaxe correta, para que o interpretador leia os dados como tal.

A sintaxe para qualquer tipo de texto é basicamente colocar o conteúdo dessa variável entre aspas ' ', uma vez atribuindo dados do tipo *string* para uma variável. Exemplo: nome = 'João'.

STRINGS

Na versão 3 *Python* (versão mais recente) ficou condicionado por padrão o uso de aspas ‘ ’, para determinar que o conteúdo, assim, identificado seja do tipo *string*.

STRINGS

Formatação de *Strings*: Quando desenvolvemos códigos com *strings*, é bastante comum que se deseje formatá-las em algum momento ou aplicar outras operações, e isto é facilmente realizado com alguns comandos que veremos a seguir.

- Converter uma *string* para maiúsculo
- Converter uma *string* para minúsculo
- Busca dentro de *string*
- Desmembrando uma *string*

STRINGS

Convertendo o conteúdo para maiúsculo: para converter o conteúdo de uma *string* em letras maiúsculas, usamos o comando *upper()*.

```
>>> mensagem = 'Linguagem de Programação I - Python'
>>> print(mensagem.upper())
LINGUAGEM DE PROGRAMAÇÃO I - PYTHON
```

STRINGS

Convertendo o conteúdo para minúsculo: para converter o conteúdo de uma *string* em letras minúsculas, usamos o comando *lower()*.

```
>>> mensagem = 'LINGUAGEM DE PROGRAMAÇÃO I - PYTHON'  
>>> print(mensagem.lower())  
linguagem de programação i - python
```

STRINGS

Buscando dados dentro de *strings*: podemos realizar buscas dentro de *strings* e identificar se este conteúdo está presente ou não. Para isso, usamos o comando *in*.

```
>>> mensagem = 'Ontem choveu e hoje está fazendo sol'
>>> print('hoje' in mensagem)
True
```

O resultado da consulta com o comando *in* será *True* ou *False*, sempre lembrando que a busca é pelo termo exato, ou seja, é *case sensitive*.

Para o exemplo acima, a busca por Hoje retornaria a *False*.

STRINGS

Desmembrando uma *string*: um comando bastante útil e interessante para uso com *strings* é o *split()*. Com este comando é possível desmembrar uma *string* em palavras separadas, a fim de realizarmos formatações ou operações por palavras de forma isolada. Vejamos o exemplo:

```
>>> mensagem = 'Ontem choveu e hoje está fazendo sol'
>>> print(mensagem.split())
['Ontem', 'choveu', 'e', 'hoje', 'está', 'fazendo', 'sol']
```



STRINGS

Também podemos usar o comando *split* para desmembrar uma *string*, usando um caracter como referência. Imagine dados salvos em um arquivo CSV, onde os mesmos são separados por vírgulas. Podemos usar o comando *split* para desmembrar o conteúdo, informando que o separador dos dados é a vírgula. Exemplo:

```
>>> mensagem = '1,4,5,67,78,9'
>>> print(mensagem.split(','))
['1', '4', '5', '67', '78', '9']
```

FUNÇÕES

Objetivos

-  Desenvolver a sintaxe de programação para preparação de funções.
-  Apresentar as diferenças e aplicações de funções com e sem parâmetros.

Conteúdos

-  Funções predefinidas
-  Funções sem parâmetros
-  Funções com parâmetros

FUNÇÕES

Uma função, independente da linguagem de programação é um bloco de código, que pode ser executado e reutilizado quantas vezes forem necessárias dentro de um programa. Uma função será utilizada para agrupar um conjunto de instruções, sem a necessidade de escrever várias vezes o mesmo código.

FUNÇÕES

Imagine, que em nosso programa em *Python* são feitas várias vezes a operação de somar dois números e atribuir o resultado a uma variável, é possível criar uma vez este cálculo em forma de função e chamar sua execução sempre que necessário, por meio de um identificador que será o nome da função.

FUNÇÕES

Funções predefinidas: a linguagem *Python* já conta com uma série de funções prontas para uso, como por exemplo a função *print()*, vista nos exemplos anteriormente, que é uma função dedicada para exibir mensagens na tela do computador. Outros exemplos de funções predefinidas:

- *len()*: mostra o tamanho de um determinado dado
- *input()*: utilizada para leitura de dados via teclado

FUNÇÕES

Como visto anteriormente, funções são blocos de código que podem ser reutilizados em diferentes locais em um programa. Funções podem ou não receber parâmetros de execução, que nada mais são que informações inseridas na função para que a mesma interaja sobre estes dados. Pela sintaxe da linguagem *Python*, executamos ou “chamamos” uma função pelo seu identificador e, logo em seguida parênteses.

print()-> *print('Linguagem Python')*

input()-> *input('Digite seu nome')*

FUNÇÕES

Como visto anteriormente, funções são blocos de código que podem ser reutilizados em diferentes locais em um programa. Funções podem ou não receber parâmetros de execução, que nada mais são que informações inseridas na função para que a mesma interaja sobre estes dados. Pela sintaxe da linguagem Python, executamos ou “chamamos” uma função pelo seu identificador e logo em seguida parênteses.

```
print()-> print('Linguagem Python)  input()-> input('Digite seu nome')
```

FUNÇÕES

Funções simples, sem parâmetros: para a criação de funções em nosso código, devemos usar a palavra reservada `def` seguida do nome desejada para a função e terminando com `()`: (parênteses e dois pontos)

Exemplo:

```
def hello_world():  
    print('Meu exemplo em linguagem Python')  
    print('Hello World')  
hello_world()
```

FUNÇÕES

Funções compostas, com parâmetros: para a criação de funções com parâmetros, seguimos a mesma regra de funções simples, porém, dentro dos parênteses devemos informar um nome de parâmetro, que será uma informação adicionada à função para que a mesma interaja sobre este dado.

Exemplo:

```
def eleva_ao_quadrado(num):  
    resultado = num * num  
    print(resultado)  
eleva_ao_quadrado(5)
```

FUNÇÕES

O exemplo anterior, também, pode ser preparado com uma instrução de retorno. A diferença é que nas funções com instrução de retorno, podemos executar a função e receber o resultado em uma variável. Veja o exemplo:

```
def eleva_ao_quadrado(num):  
    resultado = num * num  
    return resultado  
calculo = eleva_ao_quadrado(5)  
print(calculo)
```


LISTAS

Objetivos

- ✎ Apresentar conceitos de estruturas de dados em linguagem *Python*.
- ✎ Apresentar a organização de dados em Listas no *Python*.

Conteúdos

- ✎ Formato de listas
- ✎ Inserção e remoção de elementos em listas
- ✎ Leitura de iteração com dados contidos em listas

LISTA

Listas em *Python* são o equivalente a *Arrays* em outras linguagens de programação. Listas são tipos de dados usados com frequência. Uma lista é um objeto que permite armazenar organizada e indexada, diversos dados. É similar a obter várias variáveis e colocar em apenas um espaço de memória no computador. Com os dados armazenados em listas, o interpretador *Python* consegue buscar e ler estes dados de maneira muito mais rápida do que trabalhar, individualmente, com eles.

LISTA

Podemos facilmente criar uma lista em *Python*, iniciando ou não com dados.

Exemplo de criação de uma lista vazia:

```
lista = []
```

Exemplo de criação de uma lista, contendo dados:

```
lista2 = [2, 6, 'Maria', 'Pedro', 2.78]
```

Uma lista é um tipo de variável que pode armazenar diferentes tipos de dados sem causar conflitos.

LISTAS

Adicionando dados manualmente em uma lista: através do comando *append()*, podemos facilmente inserir um dado a lista. Exemplo:

```
>>> lista2 = [1, 5, 'Maria', 'João']  
>>> lista2.append(4)  
>>> print(lista2)  
[1, 5, 'Maria', 'João', 4]
```

O comando *append()* é sequencial, ou seja, o primeiro *append* irá colocar um dado armazenado na posição zero, o próximo *append* na posição 1 por diante. Podemos dizer que a chamada da função *append*, o elemento será inserido no final da lista.

LISTAS

Removendo dados manualmente: através do `remove()`, podemos
comando `remove` o conteúdo de algum índice da
lista, exemplo:

```
>>> lista2 = [1, 5, 'Maria', 'Joana']  
>>> lista2.append(23.8)  
>>> print(lista2)  
[1, 5, 'Maria', 'Joana', 23.8]  
>>> lista2.remove('Maria')  
>>> print(lista2)  
[1, 5, 'Joana', 23.8]
```

LISTA



Vale lembrar que com o comando `remove`, estamos dizendo qual o conteúdo a ser removido. Por exemplo, se temos a lista de nomes: `['Paulo', 'Ana', 'Maria']` e rodarmos o comando `remove('Maria')`, ele irá pesquisar dentro da lista o dado específico e o remover, independentemente da sua posição, reordenando a lista, pois não existe espaço vazio em uma lista.

LISTAS

Removendo dados por índice: para excluirmos um elemento da lista, lançando mão de um índice específico, usamos o comando del. Veja sintaxe:

```
>>> lista_numeros = [2, 5, 56, -12, 56]
>>> print(lista_numeros)
[2, 5, 56, -12, 56]
>>> del lista_numeros[2]
>>> print(lista_numeros)
[2, 5, -12, 56]
```

LISTAS

Verificando a posição de um elemento: para verificarmos em que posição da lista um determinado elemento se encontra, usamos o comando *index()*. Veja o exemplo:

```
>>> lista_numeros = [2, 5, 56, -12, 56]
>>> print(lista_numeros.index(56))
2
```

Se o comando *index* for utilizado com um item que não existe na lista, o interpretador retornará uma mensagem de erro, portanto, o ideal é realizar uma validação antes da pesquisa para evitar erros inesperados.

LISTAS

Verificando se um elemento consta na lista: também podemos trabalhar com operadores em listas para consultar se um determinado elemento consta ou não na lista. Para isto, usaremos o comando *in*, como no exemplo:

```
lista_de_nomes = ['Maria', 'Ana', 'João', 'Pedro']
if 'Maria' in lista_de_nomes:
    print('Maria está na lista')
else:
    print('Maria não está na lista')
```

Por meio do comando *in*, pesquisamos se determinado elemento encontra-se na lista, o resultado da consulta será *True* ou *False*.

OPERAÇÕES COM ARQUIVOS

Objetivos

- ✎ Apresentar conceitos de arquivos em *Python*.
- ✎ Apresentar formas de leitura e escrita em arquivos.

Conteúdos

- ✎ Formato de arquivos compatíveis
- ✎ Comandos de criação de arquivos Comandos de
- ✎ leitura e escrita

OPERAÇÕES COM ARQUIVOS

A linguagem *Python* fornece funções próprias para manipulação de dados armazenados em arquivos em disco,

- podendo ser: Discos removíveis
 - CD's
 - Disquetes
 - Discos Rígidos
 - Flash Drives (Pen drive)

OPERAÇÕES COM ARQUIVOS

Dados armazenados em arquivos são persistentes, isso significa dizer que sua duração se estende além da execução do programa e permanecem salvos, mesmo com o desligamento e o religamento do computador, sendo assim, a manipulação de arquivos é apropriada para formar bancos de dados.

Existem dois formatos de entrada / saída para arquivos:

- 1º Modo texto
- 2º Modo binário

OPERAÇÕES COM ARQUIVOS

Modo texto(modo padrão): neste modo, os arquivos contém caracteres legíveis, como documentos (“.txt”, “.xml”, “.html”), código-fonte (“.py”, “.c”, “.java”) e *scripts* (“.bat”, “.sh”).

Modo binário: neste modo, os arquivos contêm códigos legíveis por programas, como imagens, programas executáveis, arquivos criptografados, comprimidos e mídia.

OPERAÇÕES COM ARQUIVOS

Manipulando arquivos de texto: acessar arquivos, seja em modo texto ou binário, é feito em três etapas:

- 1ª Abertura
- 2ª Manipulação (leitura ou escrita)
- 3ª Fechamento

OPERAÇÕES COM ARQUIVOS

Na abertura do arquivo, é informado o nome do arquivo, modo de entrada / saída a ser realizado e se o acesso será leitura ou escrita;

Na manipulação, são enviados os dados para o disco ou é feita a leitura de dados vindos do disco.

No fechamento do arquivo, é o momento que se informa ao sistema operacional, que o uso do arquivo foi concluído, liberando todos os recursos alocados na manipulação do arquivo.

OPERAÇÕES COM ARQUIVOS

Exemplo de código para abrir um arquivo, ler seu conteúdo e exibir na tela:

```
>>> arquivo = open("C:\\Linguagens I\\arquivo.txt", "r")
>>> linhas = arquivo.read()
>>> print(linhas)
Linha1
Linha2
Linha3

>>> arquivo.close()
```


OPERAÇÕES COM ARQUIVOS

Exemplo de código para criar um arquivo e escrever conteúdo:

```
>>> arquivo = open("C:\\Linguagens I\\meu_arquivo.txt", "w")
>>> arquivo.write('Hello World!')
12
>>> arquivo.close()
>>> leitura = open("C:\\Linguagens I\\meu_arquivo.txt", "r")
>>> linhas = leitura.read()
>>> print(linhas)
Hello World!
>>> leitura.close()
```