

APOSTILA – v.1.0.0

primeiros scripts em Python

Entender → Codificar → Validar

```
nome = input('Informe seu nome: ')
```



By Pythonisk Team

primeiros scripts em Python



By Pythonisk Team

Primeiros Scripts em Python

Curso de Iniciação a Programação de Computadores com uso da linguagem Python.

> © 2022 – by iDuka TEAM

Introdução

O Python

Python é uma linguagem de programação de propósito geral, de alto nível, fácil aprendizado e bastante indicada para novatos em programação de computadores, embora também seja muito utilizada por desenvolvedores experientes, especialmente na área de ciência de dados.



[Logotipo do Python](#)

Foi criada por Guido Van Rossum em 1989, na Holanda.



Guido Van Rossum

A Linguagem Python chega a ser tão simples e direta que não nos soa incomum falar que um trecho do código criado seria praticamente uma frase digitada no idioma inglês. Por exemplo, caso se queira mostrar uma frase na tela, estando em modo texto (no prompt de comandos) e dentro do ambiente do python, basta digitar um `print("Hello, world")` que teremos este resultado, simples assim.

Características

São várias as características do Python que o tornam uma linguagem tão procurada e que atrai a cada dia mais desenvolvedores. A seguir, apresentamos algumas dessas características com uma breve descrição. A similaridade com o idioma inglês, como se estivesse praticamente escrevendo neste idioma chega a ser surpreendente.

fácil de aprender

O Python é uma linguagem fácil de ser aprendida e poderosa para trabalharmos. A mesma possui uma sintaxe limpa e clara, como também contém um conjunto de bibliotecas estáveis e bem estruturadas.

fácil leitura e compreensão

A sintaxe da linguagem é minimalista, isto é, mantém na sua escrita somente o necessário, o que torna o código em muitas vezes muito similar a um texto diretamente em Inglês.

fácil manutenção

Em decorrência da simplicidade sintática e da excelente

estrutura das bibliotecas, a manutenção de códigos, seja aquele que desenvolvemos ou mesmo de terceiros, é muito mais fácil e compreensível.

escalável

As aplicações escritas em Python podem ser facilmente escaláveis conforme a necessidade do software que estamos desenvolvendo. Pode ser usada da prototipagem até a produção.

multiplataforma

O interpretador do Python é escrito com a Linguagem C e C++, assim, o mesmo pode ser portado a todas as plataformas que possuam compiladores para a linguagem. Tendo em vista que o C++ é a linguagem mais difundida e base de praticamente toda a informática, temos compiladores nativos ou portados para quase todas as plataformas existentes. Neste curso abordaremos o uso do Python na plataforma de 64 bits com o Windows.

... e muitas outras ...

Onde aplicar

O Python pode ser usado em vários ambientes, desde a programação em script para computadores, automação de procedimentos, programação desktop, mobile e web, até em campos de estudos como Inteligência Artificial, Big Data, Internet das Coisas e Machine Learning.

O desenvolvedor que opta por usar a linguagem Python conta com uma [documentação de uso por meio de PEPs](#), uma repleta [biblioteca de códigos](#) que pode ser usada de forma livre e gratuita por se tratar de uma linguagem de programação aberta, vindo a seguir a filosofia de [software livre](#).

Quem usa

A adesão de desenvolvedores ao Python tem sido enorme, o que leva a uma vasta documentação disponível e uma comunidade bastante acessível, no intuito de facilitar a busca de respostas sobre dúvidas que surgirem ao longo do dia a dia de trabalho.

A seguir temos os nomes de algumas empresas que usam o Python:

- Google
- Netflix
- IBM
- Microsoft
- Globo
- Meta (Facebook, Instagram, WhatsApp ...)

- ... e muitas outras, de vários portes ...

Instalação

A primeira versão pública de Python foi lançada em 20 de fevereiro de 1991, apesar de estar sendo desenvolvida desde 1989. Um resumo da evolução da linguagem pode ser vista a seguir:

Versões do Python

(1989) — 0.9(1991) — 1.0(1994) — 2.0(2000) — 3.0(2008)
— Python 3.10.4(março de 2022) — Python 3.11(2022 em beta) — Python 3.12(em desenvolvimento)

Como visto, a versão do Python lançada em março de 2022 foi a 3.10.4, mas quaisquer outras versões, desde que superior a 3.0, servirá para atender ao propósito deste curso introdutório de lógica de programação de computadores com uso do Python.

A versão 2.0 está em desuso e nem todas as bibliotecas atuais continuam inteiramente compatíveis com esta versão. O uso da versão 3 atual (3.10.4) é o indicado para o aprendizado, embora ainda possa existir algum código legado da versão 2.7 no mercado para manutenção.

Instalando o Python

Geralmente no Linux e no Mac, o Python já vem instalado por padrão devido ao fato de ter um uso constante pelos aplicativos desses ambientes derivados do Unix.

Inicialmente vamos verificar se o Python está instalado na máquina que iremos usar. Para isso, devemos abrir o prompt de comandos (tela do shell) e digitar:

```
>>> python          # (no ambiente Windows)
>>> python --version # (estando no Linux ou Mac)
>>> python3 --version # (estando no Linux ou Mac)
```

Se uma mensagem de erro aparecer como *comando não reconhecido*, por exemplo, é sinal que precisaremos instalar o Python na máquina em uso no momento.

Considerando que a maioria dos usuários utiliza o Windows como principal sistema operacional, o processo de instalação será visto nesse ambiente. Dados de instalação no Linux ou Mac poderão ser vistos posteriormente em aula a parte.

Instalando o Python no Windows

Para instalar o Python no Windows, você deve entrar no [site oficial](#) e baixar a versão adequada ao seu sistema operacional, sendo que a maioria das máquinas atuais é de 64 bits, portanto baixe esta versão do instalador.

Link para download do [PYTHON!](#)



[Tela do instalador do Python](#)

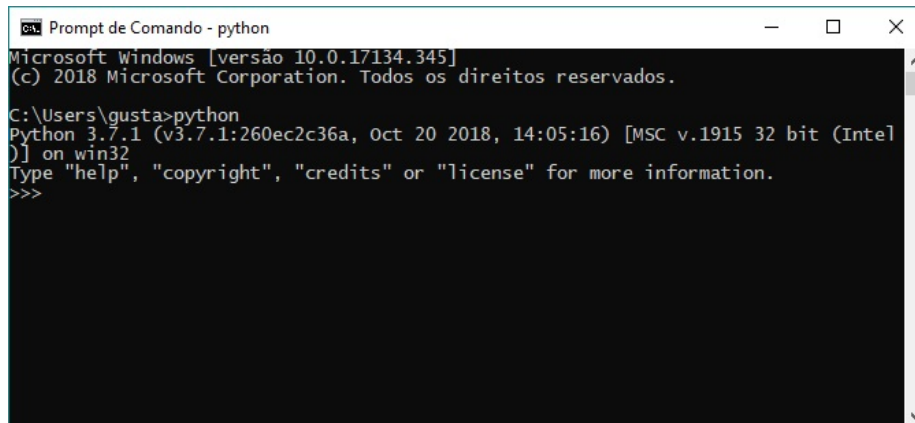
Ao concluir o download, você deve executar o instalador dando dois cliques no mesmo e continuar seguindo os passos vistos na tela. Você deve marcar a parte de “acrescentar o Python ao Path” - Add Python to PATH - o que permitirá que o comando seja reconhecido em qualquer parte do sistema e diretório que esteja acessando e rodando os scripts.

No processo de instalação do Python ocorre também a instalação da ferramenta *IDLE*, um editor de códigos onde nossos primeiros scripts serão criados e executados. Ao concluir a instalação, você pode abrir novamente o prompt de comandos (teclando simultaneamente “WIN+R” ou clicando em “iniciar > executar”) e digitar “powershell” para executar o shell no ambiente Windows, agora com o Python. Digite novamente o comando “python” para confirmar que o Python foi instalado com sucesso.

Usando o Idle

Execute o IDLE em sua máquina Windows clicando em “iniciar > powershell” e depois digite “python”. No Linux ou Mac, abra uma tela do shell, podendo ser o terminal padrão de seu

ambiente, como o `gnome-terminal`, `xfce4-terminal`, `tilda`, `terminator`, etc. E digite `"python"`. Algumas distribuições Linux você precisa digitar `"python3"`.



Tela do IDLE

O triplo `">>>"` que aparece indica que você está com o IDLE em execução e que já podemos partir para criar os primeiros códigos.

Vamos executar uma das tarefas mais primordiais na computação, a de mostrar uma mensagem na tela do computador.

```
>>> print("Hello, world!") # comando print mostra uma mensagem na tela
Hello, world!
```

Vamos aprender mais um comando, o que é usado para encerrar o Idle:

```
>>> exit()
```

Observe a presença dos parênteses no final do comando, o que indica que é uma ação a ser executada, sem receber nenhum parâmetro adicional, neste caso.

Com isto o programa é encerrado e a tela do Idle é fechada.

Ok, você fez o primeiro uso do Python em seu computador. Um novo mundo se abre para você a partir de agora. Vamos prosseguir.

Comentários

Um comentário é um texto que aparece no código mas o interpretador ou compilador Python não vai considerar. Serve para facilitar o entendimento do que se está a codificar, possibilita dicas para futuras manutenções ou simplesmente

para documentar o código, de forma que outros desenvolvedores saibam o que está sendo implementado e de que forma foi implementada a atividade do determinado trecho de código.

Comentário de fim de linha

Comentários de fim de linha são feitos mediante o uso de um caracter # - cerquilha -. Tudo que tiver a partir desse caracter será desconsiderado pelo interpretador.

```
# Eis um comentário
print('Comentando o código') # aqui é outro comentário
# O comando print é executado
```

Comentário em múltiplas linhas

Já para comentários maiores, com duas linhas ou mais, utilizamos três aspas simples ou duplas, no começo e na última linha do comentário.

```
'''
A partir daqui
    o comentário é de
        múltiplas linhas
'''
```

Palavras reservadas

O Python possui 33 palavras reservadas, as quais não podemos usar para outra finalidade que não seja a prevista na sintaxe e semântica da linguagem.

- São usados apenas caracteres alfabéticos nas palavras reservadas.
- True, False, None são as únicas que possuem uma letra maiúscula, as demais são grafadas em letra minúscula.
- and, or, not, is, if, elif, else, while, for, break, continue, return, in, yield, try, except, finally, raise, assert, import from, as, class, def, pass, global, nonlocal, lambda, del e with são as demais.

Para ver a lista de palavras reservadas digite em seu Idle:

```
>>> import keyword
>>> keyword.kwlist
```

Comando print()

O comando `print()` é utilizado para mostrar uma informação na tela. Esta informação pode ser recebida por meio de parâmetro como texto diretamente, pode ser uma expressão aritmética ou ainda um nome de um identificador. Sempre precisa ser do tipo texto, conforme a sintaxe definida no Python.

```
>>> mensagem = 'Oi, mundo!'
>>> print('Mensagem: ')
>>> print(mensagem)
```

```
Mensagem:
Oi, mundo!
```

```
>>> print(2 + 3)
5
```

Tipos de Dados Fundamentais

Vamos falar neste momento sobre alguns tipos de dados usados no Python, de acordo com as grandezas encontradas no dia a dia do programador. Para as grandezas representadas por números "inteiros", como idade ou quantidade de filhos, há o tipo *int*. Já para o formato texto, utilizaremos o tipo "string", *str*.

Para números com parte fracionária, como salário, altura em metros, peso de uma pessoa, usaremos o tipo *float* e ainda, para números complexos, o tipo *complex*.

Há também um tipo onde a variável só pode assumir os valores booleanos (ou lógicos) verdadeiro (*True*) ou falso (*False*), o tipo *bool*.

Outros tipos, bem como a forma como os dados são organizados, serão vistos posteriormente.

Um tipo de dados *int* pode ser representado na forma decimal(0-9), binária(0 & 1), octal(0-7) ou ainda hexadecimal(0-9 & a-f | A-F)

type()

Para sabermos o tipo de dado que a variável contém armazenado no momento, usamos a função *type()*. Veja o exemplo e digite o texto a seguir no *Idle*:

```
>>> type("Patricia")
```

```
<class 'str'>
```

No Python, textos são identificados com uso de aspas simples `'` ou aspas duplas `""`. Vejamos agora como os números são identificados:

```
>>> type(3)
<class 'int'>
```

```
>>> type(3.14)
<class 'float'>
```

Observe que no Python, assim como na maioria das linguagens de programação, o separador de casas decimais é o `"."` (ponto), em vez da usual `","` (vírgula) aqui no Brasil. Ajustes são feitos na parte da representação desses números na tela, quando se fizer necessário.

Agora vamos ver a representação de um tipo booleano:

```
>>> type(True)
<class 'bool'>
```

Há outros tipos de dados (as coleções), como `list` (lista), `tuple` (tupla), `set` (conjunto) e `dict` (dicionário). Estes serão vistos posteriormente, no decorrer do curso, na parte de estruturas de dados.

A tabela a seguir apresenta os diversos tipos de dados primitivos utilizados no Python.

Tipo	Dado
texto	<code>str</code>
numérico	<code>int</code> , <code>float</code> , <code>complex</code>
sequenciais	<code>list</code> , <code>tuple</code> , <code>range</code>
mapeamento	<code>dict</code>
conjunto	<code>set</code> , <code>frozenset</code>
booleano	<code>bool</code>
binário	<code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>

Tabela 01: Tipos de Dados Fundamentais
Em **negrito** temos os tipos que vamos ver neste curso

`id()`

A função `id()` apresenta o endereço de memória onde o objeto referente a variável foi alocada.

```
>>> a = 10
>>> print(id(a))
140342382248464 # Inteiro referente ao endereço do objeto
```

Identificadores

Operador de Atribuição

Para salvar dados numa variável(identificador) usamos o operador de atribuição, com o caractere =. O código a seguir cria uma variável de nome fruta e armazena o valor de texto "uva":

```
fruta = "uva"
```

Para armazenar texto em uma variável, podemos usar aspas simples ou aspas duplas. O código abaixo teria o mesmo resultado:

```
fruta = 'uva'
```

Você pode usar três aspas simples (ou duplas) e criar a variável com o conteúdo aparecendo em diversas linhas de código.

```
fruta = '''
    Uva da serra
    de Gravatá '''
```

Com a linha de comando do PowerShell aberta, insira “python” para executar o interpretador do Python 3. (Algumas plataformas preferem usar o comando py ou python3, tente, pois isso também deverá funcionar). Você saberá que obteve êxito porque um prompt com ">>>" - três símbolos maior - será exibido. Há vários métodos internos que permitem que você faça modificações em “cadeias de caracteres”, ou sequências de letras, símbolos e números no Python.

Vamos aos exemplos:

Crie um identificador 'frase' com o comando e conteúdo a seguir:

```
>>> frase = 'Olá, mundo!'
Olá, mundo!
```

Obs.: Pressione 'Enter' para ir para uma nova linha. O conteúdo armazenado em 'frase' é apresentado na tela sempre que você executar o comando `print(frase)`.

```
>>> print(frase)
Olá, mundo!
```

Percebe-se que isso exibirá o texto contido na variável - "Olá, mundo!" - como resultado retornado no prompt de comandos.

Vamos agora descobrir o tamanho ou quantidade de caracteres usados da variável 'frase' com a função `len(frase)`.

```
>>> print(len(frase))
11
```

Isso mostrará que há 11 caracteres armazenados no identificador. Observe que o espaço em branco também é contado como "um caractere" no tamanho total.

Mostre quantas vezes a letra "l" (ele minúsculo) aparece na variável de cadeia de caracteres 'frase':

```
>>> print(frase.count("l").)
1
```

Pesquise um caractere específico na variável de cadeia de caracteres. Vamos encontrar a posição do ponto de exclamação com `frase.find("!")`.

```
>>> print(frase.find("!"))
10
```

Isso mostrará que o primeiro caractere contendo o ponto de exclamação é encontrado na 10ª posição da cadeia de caracteres 'frase'.

Agora substitua o ponto de exclamação por um ponto de interrogação:

```
>>> print(frase.replace("!", "?"))
Olá, mundo?
```

Estes são alguns dos comandos iniciais. Um estudo um pouco mais abrangente sobre as strings será visto no decorrer do

curso.

Identificadores

Um identificador, como visto no anterior 'frase', é um nome usado em um programa Python. São, como o nome diz, nomes que identificam determinados espaços de memória para nos referirmos dentro de nosso código como locais de acesso e armazenamento de dados e informações. Podem ser usados em *Classes, Funções, Métodos e Variáveis*, termos estes que serão vistos no decorrer deste curso.

Os identificadores são usados no Python segundo determinadas regras:

- Apenas caracteres são permitidos para uso nos identificadores:
 - caracteres alfabéticos → (a-z, A-Z)
 - caracteres numéricos → (0-9)
 - caracter underscore → (_)

Usando caracteres diferente destes, um erro será reportado.

- O primeiro caractere não pode ser um número:
 - nome1 é identificador permitido
 - 1nome NÃO é identificador permitido
- O Python é "case sensitive", o que indica que uso de letra maiúscula ou letra minúscula são tratados de forma distinta.
 - Nome, NOME, nome e nOME são identificadores distintos entre si.
- Um identificador escrito totalmente em letras maiúsculas é usado como constante, *por convenção*. O dado armazenado pode ser alterado no decorrer da execução do código, diferente de outras linguagens.
- Um identificador que inicie com *underscore* é privado, de uso restrito dentro de determinado trecho do código. Com uso de dois *underscores* no início, é fortemente privado. E, por fim, quando iniciadas e terminadas por dois *underscores*, o identificador é definido como um nome especial de uso definido pela linguagem ou também denominados, métodos mágicos.
 - `__init__(self, hour=0, minute=0, second=0):`
- Não se usam as palavras reservadas como identificador.

- Procure usar nomes que facilitem o entendimento e propósito do identificador. Nomes como *salario_liquido*, *salario_bruto*, *frete*, *nome_do_cliente*, *valor_total*, seriam bastante adequados para utilização.

No próximo tópico iremos iniciar o aluno no uso de textos (*strings*) no Python.

Trabalhando com String

A linguagem Python disponibiliza vários métodos para lidar com textos dentro dos programas. Esses comandos que veremos a seguir são muito importantes e facilitarão e muito a manipulação dos dados de nossos scripts.

- Para criar uma string “s” com conteúdo “Olá” usamos o comando de atribuição (=) e as aspas.

```
>>> s = "Olá"
```

- Para mostrar uma string “s”.

```
>>> print(s)
Olá
```

- Para mostrar o tamanho de uma string “s”.

```
>>> print(len(s))
3
```

- Para concatenar uma string “s” com outra “r” usamos o sinal de +.

```
s = 'Olá'
r = 'mundo!'
>>> print(s + r)
Olámundo! # Observe a necessidade de se acrescentar um espaço em branco
```

- Converta a variável de cadeia de caracteres em letras maiúsculas e depois para letras minúsculas. Use os métodos `upper()` e `lower()`.

```
>>> print(s.upper())
OLÁ
>>> print(s.lower())
olá
```

- Para capitalizar (deixar a primeira letra maiúscula) uma string “s”, o método usado é o `capitalize()`. Para deixar todas as primeiras letras de cada palavra da string em maiúsculo usamos o método `title()`

```
>>> s = "oi, eu vou ver o nascer do sol."
>>> print(s.capitalize())
Oi, eu vou ver o nascer do sol.
>>> print(s.title())
Oi, Eu Vou Ver O Nascer Do Sol.
```

- Para substituir uma substring dentro de uma string “s” salvando o resultado na string “r”.

```
>>> s = 'Olá'
>>> r = 'Oi'
>>> r = s.replace("Olá", "Oi, mundo!")
>>> print(r)
Oi, mundo!
>>> print(s)
Olá
```

Obs.: Note que o dado armazenado em 's' não foi alterado.

- Para testar se uma string “s” inicia com um texto informado.

```
>>> s = 'Oi, mundo'
>>> print(s.startswith('Oi'))
True
```

- Para testar se uma string “s” termina com um termo informado.

```
>>> print(s.endswith('mundo!'))
True
```

Execute os comandos a seguir em seu Idle e faça um relato sobre os resultados encontrados.

- Para mostrar alguns termos de uma string “s”.

```
>>> s = 'Olá, mundo!'
>>> print(s[0])
```



```
>>> print(s[0:5])
>>> print(s[0:5])
>>> print(s[0:-6])
>>> print(s[-1])
>>> print(s[:-1])
>>> print(s[5:])
>>> print(s[::2]) # Imprime os caracteres nos índices pares
>>> print(s[1::2]) # Imprime os caracteres nos índices ímpares
```

- Para mostrar uma string “s” de forma invertida

```
>>> s = '12345abc'
>>> print(s[::-1])
```

- Para verificar se uma string “s” só contém números.

```
>>> s = '12345'
>>> print(s.isdigit())
True
```

- Para verificar se uma string “s” só contém letras e números.

```
>>> s = '12345abc'
>>> print(s.isalnum())
True
```

Recebendo dados - *input()*

Recebemos dados do usuário através da leitura do teclado usando a função `input()`, como visto a seguir:

```
nome = input('Informe seu nome: ')
print(nome)
```

A função `input()` sempre retorna o dado recebido como string, `str`. Caso precisemos executar ações com números, por exemplo, devemos realizar a devida conversão, ou *casting*, para o tipo desejado.

Casting de Tipos

```
idade = int(input('Informe a sua idade: '))
print(str(idade))
```

Além da conversão para inteiro, podemos usar as seguintes conversões:

- `float()`
- `complex()`
- `bool()`
- `str()`

Operadores

Agora vamos dar entrada ao estudo dos operadores em Python. Scripts que envolvem expressões aritméticas fazem uso constante desse assunto. Considere os operadores aritméticos contidos na tabela a seguir que nos permitirão realizar as principais operações da Matemática:

Aritméticos

Os operadores aritméticos possibilitam a realização de operações matemáticas, exatamente como acontecem naquela Ciência Exata. A precedência das operações, obviamente, seguem a mesma prioridade.

Operadores Aritméticos			
Operação	Operador	Exemplo	Resultado
adição	+	14 + 2	16
subtração	-	12 - 3	9
multiplicação	*	3 * 4	12
divisão	/	13 / 3	4.3333...
exponenciação	**	2 ** 5	32
divisão inteira	//	13 // 3	4
módulo	%	13 % 3	1

Obs: O operador módulo retorna o resto da divisão inteira entre dois números.

Tabela: Operadores Aritméticos

Em seguida, vamos aos exercícios através do IDLE para aprender algumas operações aritméticas com o Python dentro do campo dos inteiros e fracionários.

Execute uma linha por vez fazendo as observações do resultado ou faça um script com todas as operações.

```

12 + 4 # resultado 16
12 - 4 # resultado 8
12 * 4 # resultado 48
12 / 4 # resultado 3
12 ** 2 # resultado 144 ( 122)
12 // 5 # resultado 2
12 % 5 # resultado 2 ( resto de 12 / 5)

```

Vamos realizar mais alguns exercícios durante as aulas síncronas utilizando estes operadores aritméticos e obter o resultado dos respectivos cálculos. A precedência entre os operadores é a mesma utilizada na Matemática e será vista na aula online.

Exercitar-se nos códigos é o que tem de mais importante no processo de aprendizagem de uma linguagem de programação. Incentivamos esse exercitar como forma de aprendizado constante. Faça bom uso das listas de exercícios.

Agora usaremos os operadores de comparação ou relacionais, que são também de uso similar ao usado na Matemática:

Operadores Relacionais

Esses operadores servem para fazer comparações entre dois valores, retornando sempre um valor lógico de "*verdadeiro*" ou "*falso*" como resposta.

Em consonância com o tipo de operador anterior, os operadores relacionais também funcionam como na Matemática, indicam a relação de comparação entre dois elementos. O retorno será sempre um entre dois dos valores possíveis, *Verdadeiro* ou *Falso*, ou seja, dos tipos True ou False, respectivamente.

Operadores Relacionais			
Operação	Operador	Exemplo	Resultado
igual a	==	3 == 3	True
diferente de	!=	3 != 3	False
maior que	>	3 > 3	False
maior que ou igual a	>=	3 >= 3	True
menor que	<	3 < 3	False
menor que ou igual a	<=	3 <= 3	True

Tabela: Operadores Relacionais

Obs: Não confundir, o operador de atribuição é um sinal de “=” e o operador de comparação tem dois sinais de igual, ficando “==”.

E vamos a mais uns exemplos para exercitar:

```
12 == 4 # resultado False
12 != 4 # resultado True
12 > 4 # resultado True
12 >= 4 # resultado True
12 < 4 # resultado False
12 <= 4 # resultado False
```

Lógicos

Para efetuar uma operação lógica entre sentenças utilizamos os operadores a seguir.

Operadores Lógicos				
Operação	Operador	Exemplo	Ação	Resultado
not (não)	not	A = False	not A	True
and (e)	and	a=2, b=3	(a < 12) and (b < 10)	True and True = True
or (ou)	or	a=2, b=13	a < 12) or (b < 10)	True or False = True

Tabela: Operadores lógicos

O operador NOT resulta no oposto da condição lógica do operador. Converte o “*verdadeiro*” em “*falso*” e vice-versa.

Tabela verdade not	
A	not A
False	True
True	False

Tabela Verdade NOT

O operador AND resulta em “*verdadeiro*” quando todas as

condições também forem de valor “verdadeiro”. Se houver uma condição com valor “falso”, o resultado será “falso”.

Tabela verdade and		
A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Tabela Verdade AND

O operador OR resulta em “verdadeiro” quando umas condições também forem de valor “verdadeiro”. Se houver todas as condições com valor “falso”, o resultado será “falso”.

Tabela verdade or		
A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Tabela Verdade OR

O comando print()

Já usamos aqui por algumas vezes o comando *print()* para mostrar informação na tela. Agora vamos ver como deixar a saída um pouco mais amigável e o código mais legível.

format()

Vamos usar três exemplos para mostrar como aplicamos o método *format()* dentro do *print()*.

```
nome = 'Alberto'
idade = 19
print('O aluno ' + nome + ' tem ' + str(idade) + ' anos.' )
O aluno Alberto tem 19 anos.
print('O aluno {} tem {} anos.'.format(nome, idade))
print(f'O aluno {nome} tem {idade} anos.')
```

Optamos em usar o formato simplificado (com f'') mas o uso é livre e opte pelo que melhor se adaptar neste começo de atuação como desenvolvedor.

Caracteres especiais ou de escape

Aprenda a usar o caracter de escape ('\') para adicionar caracteres extra que indicam uma ação ou formatação do conteúdo da *variável ou conjunto de parâmetros* do *print()*.

```
\n ==> Nova Linha
\t ==> Espaçamento Horizontal
\r ==> Retorno de Carro
\b ==> Espaço para trás
\f ==> Avança Folha
\v ==> Espaçamento Vertical
\' ==> Aspas simples
\" ==> Aspas duplas
\\ ==> Barra invertida
```

Controle de fluxo

Vamos iniciar agora o estudo de problemas onde há a necessidade de tomada de decisão, onde apenas um ou outro comando deverá ser executado, baseado na entrada informada.

Condicionais

Até agora os nossos programas (scripts) foram executados linha a linha, um comando por vez, todos os comandos até o fim do script. Agora vamos usar uma estrutura que permite a tomada de decisão.

if (se)

```
>>> if (condição):
>>>     comando1
>>>     comando2
```

Alternativa:

```
if (condição1):comando
```

Vejamos um código com exemplo mais real:

Digamos que precisamos resolver uma questão de idade, onde o indivíduo precisa ser classificado em maior de idade, baseado numa informação recebida. Neste caso, faremos uso da estrutura *if* e esta mesma se comporta como a seguir:

```
>>> idade = 18
>>> if idade > 17:
>>>     print("Maior de idade")
```

Temos que, caso a condição seja satisfeita, o retorno será verdadeiro e o comando (mostrar uma mensagem de maior de idade na tela) é executado. Em sendo falsa, o comando posterior será executado. Agora no caso de precisarmos escolher entre uma ou outra condição, a estrutura utilizada será a *if, else* como visto a seguir:

if-else (se-senão)

```
>>> if (condição):
>>>     comandoV
>>> else:
>>>     comandoF
```

Exemplo de código:

```
>>> idade = 18
>>> if idade > 17:
>>>     print("Maior de idade")
>>> else:
>>>     print("Menor de idade")
```

Nesse exemplo, se a idade for maior que 17 (e este é o caso) a primeira mensagem, "maior de idade", é exibida, caso contrário a segunda mensagem, "menor de idade", será exibida.

Devemos atentar para a indentação, o que indica que o comando está dentro da estrutura correspondente. E observar também que os ">>>" não fazem parte do comando e sim do IDLE, nesse caso.

Uma terceira forma de usar o *if* é a composta por várias opções

de escolha, como veremos adiante:

if-elif- ... -else (se-senão se-senão)

```
if (condição1):
    comando
elif (condicao2):
    comandos
elif (condicao3):
    comandos
else:
    comandos
```

Exemplo de código:

```
>>> idade = 18
>>> if idade < 12:
>>>     print("Infantil")
>>> elif idade < 18:
>>>     print("Juvenil")
>>> else:
>>>     print("Adulto")
```

A saída mostrada na tela no exemplo acima será *“Adulto”*. Os testes são realizados sequencialmente até que a condição resulte em *“verdadeira”*. O laço é abandonado assim que isso acontece e os demais comandos, caso existam, não serão executados.

```
>>> idade = 16
>>> if idade < 12:
>>>     print("Infantil")
>>> elif idade < 18:
>>>     print("Juvenil")
>>> else:
>>>     print("Adulto")
Juvenil
```

Laços de Repetição

Agora vamos conhecer as estruturas referente aos **laços ou loops**, onde determinados comandos são executados várias vezes, até determinada condição ser satisfeita.

A estrutura de repetição **for** é usada quando a interação e a ação são feitas no mesmo elemento do código.

for (para)

```
for x in [1,2,3,4,5,6]:  
    print(x)
```

Saída:

```
1  
2  
3  
4  
5  
6
```

Pode ser usado da mesma forma em strings:

```
for letra in 'Estudante':  
    print(letra)
```

Saída:

```
E  
s  
t  
u  
d  
a  
n  
t  
e
```

...ou com uma mensagem final, como em:

```
for x in [ 1, 3, 5, 7, 9, 11]:  
    print(x)  
else:  
    print('Fim')
```

Saída:

```
1  
3  
5  
7  
9  
11  
Fim
```

```
lista = ['a', 'e', 'i', 'o', 'u']
for i in lista:
    print(i)
```

Saída:

```
a
e
i
o
u
```

pass

Dentro de scripts, podemos usar uma instrução que nos ajuda a ter acesso a determinada parte do código mas não executamos efetivamente nada. A instrução `pass` é muito utilizada quando queremos um código vazio, uma instrução vazia, sem fazer nada de específico. Para trabalharmos em outras áreas do código e em seguida criamos a parte referente ao *pass*

```
while 1:
    pass
```

O código acima só é interrompido após pressionarmos o *control+c* no teclado. Trata-se de um laço infinito.

range(n)

O interpretador Python possui várias funções e tipos integrados que estão sempre disponíveis. Uma dessas ferramentas é o `range*`. A forma de uso é a seguinte:

```
range(fim)
range(início, fim,[passo])
```

Uma sequência de iteradores, passos ou itens numéricos consecutivos é gerada.

Com *início*, *fim* e *passo* sendo inteiros. Informando só um inteiro, é considerado como o fim da contagem. *Passo* é opcional. Caso não seja informado, a sequência é de forma que cada elemento vai avançando de um em um.

Criaremos a seguir uma lista com o *range()*:

```
a = list(range(5))
print(a)
```

Saída:

```
[0, 1, 2, 3, 4]
```

for com range

O `range()` é o responsável por fornecer a sequência de números inteiros e estes são inseridos na variável `i` criada e depois mostrada na execução do script. Observe o código a seguir:

```
for i in range(5):  
    print(i)
```

Saída:

```
0  
1  
2  
3  
4
```

for com enumerate

Similar ao anterior, com o acréscimo de um número sequencial, informando a posição de cada item mostrado.

```
lista = ['a', 'e', 'i', 'o', 'u']  
for i, vogal in enumerate(lista):  
    print(i, vogal)
```

Saída:

```
0 a  
1 e  
2 i  
3 o  
4 u
```

while

Usado para executar tarefas repetidamente mediante o controle de uma variável de controle.

```
contador = 0
```

```
while contador <= 5:  
    print(contador)  
    contador += 1
```

Saída:

0
1
2
3
4
5

while (com break)

O *break*, quando executado, interrompe o laço por completo, vai para a instrução posterior que se encontrar fora do laço.

```
contador = 0
while contador <= 5:
    print(contador)
    contador += 1
    if contador > 3: break
print('saiu')
```

Saída:

0
1
2
3
saiu

while (com continue)

```
contador = 0

while contador <= 5:
    print(contador)
    if contador == 2:
        contador += 2
        print('dois!')
        continue
    contador += 1

print('saiu')
```

Saída: 0

1
2
dois!
4

5
saiu

while (com True)

E para executar um laço até determinada condição ser satisfeita, usa-se o *while(True)*. Também pode ser usado *while 1:*

O exemplo a seguir apresenta um código onde a contagem é iniciada e o laço é interrompido quando o valor da variável *i* ultrapassar o valor 10.

```
i = 0
while (True):
    print(i)
    i += 1
    if(i > 10): break

print("Terminou")
```

Saída:

```
0
1
2
3
4
5
6
7
8
9
10
Terminou
```

Coleções de Dados

Vimos os tipos de dados primitivos, onde uma unidade de cada item é armazenada. No dia a dia do desenvolvimento de sistemas nos deparamos com uma realidade mais abrangente, onde dados que precisam ser considerados como multiplicidade, como coleções.

O tipo Lista

Ao propormos soluções por meio um programa ou script, há uma necessidade de agruparmos os dados recebidos ou tratados de forma organizada. Uma das maneiras mais versáteis é por meio

do uso de listas, o tipo `list`.

Exemplo:

```
a = []
print(type(a))
>>> <class 'list'>
```

Usando o tipo lista - *list*

Uma lista pode ser criada com elementos de tipos diferentes, como *int*, *string*, *float*, separados por vírgulas e entre colchetes, embora tipicamente seja usada com elementos de mesmo tipo, o que é mais lógico para a maioria dos propósitos, por assim dizer.

As variáveis do tipo `list` são criadas com o seguinte formato:

```
nome_da_variável = [elemento1, elemento2, elemento3]
```

Exemplo:

```
frutas = []
itens = ['arroz', 'feijão', 'café', 'açúcar', 'óleo', 'macarrão']
```

O código acima cria uma lista vazia denominada "*frutas*" e outra contendo alguns "*itens de compra*". Cada elemento pode ser acessado através de seu *índice*, como visto nas strings, lembrando que o primeiro índice é 0 (zero).

Exemplo:

```
frutas = ['pera', 'uva', 'maçã']
print(frutas[0])
>>> pera
print(frutas)
>>> ['pera', 'uva', 'maçã']
```

Principais Métodos da *list*

As listas são tipos mutáveis, o que quer dizer que podemos alterar o seu conteúdo em tempo de execução.

`.append(item)`

O método *append* adiciona um item ao final da lista. Um elemento por vez é adicionado a cada operação. Para adicionar

vários itens, utilizamos a estrutura `for` e assim vamos executando em loop até finalizar.

Exemplo:

```
frutas = []
frutas.append('pera')
frutas.append('uva')
frutas.append('maçã')
print(frutas)
>>> ['pera', 'uva', 'maçã']
```

O método *append* também permite que uma lista seja adicionada ao final de outra, como visto a seguir:

Exemplo:

```
frutas = ['pera', 'uva']
outras_frutas = ['maçã', 'banana']
frutas.append(outras_frutas)
print(frutas)
>>> ['pera', 'uva', 'maçã', 'banana']
```

.sort()

O método *sort* pode ser utilizado sempre que se quiser ordenar uma lista. O padrão é a lista ser ordenada de forma crescente. Para ordem decrescente, o parâmetro *reverse=True* deve ser usado. O formato é visto a seguir:

Exemplo:

```
list.sort(key=..., reverse=...)
```

.pop()

O método *pop* retira um elemento de uma lista e retorna o valor deste. Um inteiro positivo pode ser passado como parâmetro, indicando a posição do elemento a ser retirado da lista. Caso esse inteiro não seja informado, o último elemento da lista é retirado.

Exemplo:

```
frutas = ['pera', 'uva', 'maçã', 'banana']
frutas.pop()
>>> banana
print(frutas)
```

```
>>> ['pera', 'uva', 'maçã']
```

E neste exemplo, informando o índice...

Exemplo:

```
frutas = ['pera', 'uva', 'maçã', 'banana']
frutas.pop(2)
print(frutas)
>>> ['pera', 'uva', 'banana']
```

.remove(item)

Este método é usado quando precisamos eliminar a primeira ocorrência do elemento de valor "item" na lista.

Exemplo:

```
frutas = ['pera', 'abacate', 'uva', 'maçã', 'abacate', 'banana']
frutas.remove('abacate')
print(frutas)
>>> ['pera', 'uva', 'maçã', 'abacate', 'banana']
```

Se não existir valor igual, uma exceção de tipo *ValueError* é levantada.

.clear()

Esse método remove todos os elementos da lista. É usado quando precisamos garantir que a lista seja inicializada vazia, a partir de determinado trecho do código.

Exemplo:

```
frutas = ['pera', 'uva', 'maçã', 'banana']
frutas.clear()
print(frutas)
>>> []
```

.insert(pos, item)

Este método insere um item em determinada posição da lista. O primeiro argumento refere-se a posição e o segundo é o elemento a ser inserido:

```
motos = ['titan', 'ninja', 'kawasaki']
motos.insert(2, 'harley')
```



```
print(motos)
>>> ['titan', 'ninja', 'harley', 'kawasaki']
```

Outra forma de criar uma lista

E finalizando esse tópico, vamos criar uma lista dos dez primeiros quadrados de números inteiros, com a atribuição feita por meio de uma expressão na primeira atribuição:

```
quadrados = [x**2 for x in range(10)]
print(quadrados)
>>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

O tipo Tupla - *tuple*

Tupla funciona de forma similar a Lista porém, são imutáveis. Resumidamente, uma tupla seria uma lista em modo de apenas leitura. Essas estruturas se tornam interessantes quando lidamos com itens que não precisam ser modificados ao longo da execução do programa ou script. Dias da semana, meses, escolaridade, faixa etária são alguns dos exemplos de uso.

Os parênteses podem ser omitidos, ao se criar uma tupla e os elementos são separados por vírgula. Assim como nas listas, podemos ter elementos repetidos na tupla.

```
t = 10, 20, 30, 40
print(t)
print(type(t))
```

Saída:

```
(10, 20, 30, 40)
<class 'tuple'>
```

É interessante observar que durante a definição da tupla, com ou sem uso de parênteses e tendo apenas um elemento, a vírgula torna-se obrigatória, como visto nos exemplos a seguir:

Sem parênteses

```
t = 10
u = 10,
print(type(t))
print(type(u))
```

Com parênteses

```
t = (10)
u = (10,)
print(type(t))
print(type(u))
```

Ambos geram a seguinte saída:

Saída:

```
<class 'int'>
<class 'tuple'>
```

Para finalizar, mais alguns exemplos de código:

- Criação de Tupla vazia:

```
t = ()
```

- Criação de Tupla com um elemento:

```
t = 10,
t = (10,)
```

- Criação de Tupla com vários elementos:

```
t = 10,20, 30, 40
t = (10, 20, 30, 40)
```

- Criação de Tupla com a função *tuple()*:

```
valores = [10, 20, 30, 40]
t = tuple(valores)
print(type(t))
```

Saída:

```
(10, 20, 30, 40)
<class 'tuple'>
```

Ou então:

```
t = tuple(range(10, 20, 2))
```

```
print(t)
print(type(t))
```

Saída:

```
(10, 12, 14, 16, 18)
<class 'tuple'>
```

Acessando elementos da Tupla

Podemos acessar os elementos da Tupla através de seu índice (posição) ou por meio dos operadores *slices*, similar ao anteriormente feito para as *strings* e nas *listas*:

```
t = (10, 20, 30, 40, 50)
print(t[0]) # resultado: 10
print(t[-1]) # resultado: 50
print(t[10]) # resultado: IndexError: tuple index out of range

print(t[2:5]) # resultado: (30, 40, 50)
print(t[2:10]) # resultado: (30, 40, 50)
print(t[:2]) # resultado: (10, 20, 30)
```

Métodos importantes da Tupla

`len()`

Retorna o tamanho da Tupla.

```
t = (10, 20, 30, 40, 50)
print(len(t)) # resultado: 5
```

`count()`

Retorna o número de ocorrências de determinado elemento na tupla:

```
t = (10, 20, 10, 10, 50)
print(t.count(10)) # resultado: 3
```

`index()`

Retorna a posição da primeira ocorrência de determinado elemento na tupla. Se o elemento não existir na tupla, um erro tipo *ValueError* é gerado:

```
t = (10, 20, 30, 10, 30)
print(t.index(30)) # resultado: 2
```

0 tipo Set(conjuntos)

As estruturas que vimos anteriormente permitem que elementos se repitam dentro da entidade. Caso que iramos trabalhar com valores únicos dentro da estrutura, utilizamos o *set*.

No *set*, valores duplicados não são permitidos, assim como a ordem de inserção de elementos não é mantida. Há a função de ordenar de forma crescente ou decrescente, consequentemente, percebemos que a indexação não é permitida. Os elementos são mutáveis e as mesmas operações dos conjuntos (intercessão, união, diferença...) na matemática são implementadas.

Para criar um *set* procedemos como a seguir, com o uso de chaves "{" e "}".

```
s = {10, 20, 30, 40, 50}
print(s)
print(type(s))
```

Saída:

```
{40, 10, 50, 20, 30}
<class 'set'>
```

Ou ainda usando a função *set()*. Note que a criação de um conjunto vazio não dar-se-á por meio do uso de *s = {}* pois isso é um dicionário (*dict*), estrutura que será vista posteriormente.

```
s = set()
print(s)
print(type(s))
```

Saída:

```
set()
<class 'set'>
```

Funções do set

```
add(x)
```

Adicionar o item 'x' ao conjunto:

```
s = {10, 20, 40}
s.add(30)
print(s)
```

Saída:

```
{40, 10, 20, 30}
```

`update(x, y, z)`

Usado para adicionar múltiplos itens ao conjunto:

```
s = {10, 20, 40}
l = [10, 23, 33, 43]
s.update(l, range(5))
print(s)
```

Saída:*

```
{0, 33, 1, 2, 3, 4, 40, 10, 43, 20, 23}
```

`copy()`

Retorna uma cópia do conjunto, um *clone* do objeto *set*:

```
s={50, 40, 10, 30, 20}
s1 = s.copy()
print(s)
print(s1)
```

Saída:

```
{40, 10, 50, 20, 30}
{50, 20, 40, 10, 30}
```

`pop()`

Remove e retorna um elemento aleatório do *set*:

```
s={50, 40, 10, 30, 20}
print(s)
print(s.pop())
print(s)
```

Saída:

```
{40, 10, 20, 30}
40
{10, 20, 30}
```

`remove(x)`

Remove um específico elemnto do conjunto *set*. Se o elemento não existir no conjunto, um erro `KeyError` é retornado.

```
s={50, 40, 10, 30, 20}
print(s)
s.remove(50)
print(s)
```

Saída:

```
{40, 10, 50, 20, 30}
{40, 10, 20, 30}
```

`discard(x)`

Remove um específico elemnto do conjunto *set*. Se o elemento não existir no conjunto, não há erro gerado, diferente do `remove()`, que gera um erro `KeyError`.

```
s={50, 40, 10, 30, 20}
print(s)
s.discard(50)
print(s)
```

Saída:

```
{40, 10, 50, 20, 30}
{40, 10, 20, 30}
```

`clear()`

Essa operação remove todos os elementos do *set*.

```
s={10,20,30}
print(s)
s.clear()
```

```
print(s)
```

Saída:

```
{10, 20, 30}  
set()
```

Operações Matemáticas em set

Similar às operações em conjuntos na matemática, vamos ver as operações dentro dos *sets*.

`union()`

Retorna todos os elementos contidos em dois sets. Equivalente ao *'or'*.

```
x = {10, 20, 30, 40}  
y = {30, 40, 50, 60}  
print(x.union(y))  
print(x|y)
```

Saída:

```
{10, 20, 30, 40, 50, 60}  
{10, 20, 30, 40, 50, 60}
```

`intersection()`

Retorna todos os elementos comuns entre dois sets. Equivalente ao *'and'*.

```
x = {10, 20, 30, 40}  
y = {30, 40, 50, 60}  
print(x.intersection(y))  
print(x & y)
```

Saída:

```
{30, 40}  
{30, 40}
```

`difference()`

Retorna todos os elementos presentes em um set mas não em

outro. Equivalente ao '- '.

```
x = {10, 20, 30, 40}
y = {30, 40, 50, 60}
print(x.difference(y))
print(x - y)
print(y - x)
```

Saída:

```
{10, 20}
{10, 20}
{50, 60}
```

`symmetric_difference()`

Retorna todos os elementos presentes em um set OU em outro mas não em ambos. Equivalente ao '^'.

```
x = {10, 20, 30, 40}
y = {30, 40, 50, 60}
print(x.symmetric_difference(y))
print(x ^ y)
```

Saída:

```
{10, 50, 20, 60}
{10, 50, 20, 60}
```

Outras operações

in

```
s = set('curso')
print(s)
print('r' in s)
print('R' in s)
```

Saída:

```
{'u', 'c', 's', 'o', 'r'}
True
False
```


O equivalente para testar a não existência de elemento é o `'not in'`

E, por fim, a criação de um *set* pode ser feita por meio de expressões, como visto no exemplo a seguir:

```
s = {x * x for x in range(10)}  
print(s)
```

Saída:

```
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

E agora vamos estudar uma estrutura de muita aplicabilidade dentro dos sistemas, o dicionário (`dict`).

O tipo `dict`(Dicionário)

Os objetos estudados acima são agrupamentos que comportam elementos simples. Para algumas aplicações as entidades precisam de pares de atributos no formato *chave-valor*. Esse formato é disponibilizado pelos dicionários:

Os dicionários são muito utilizados nos algoritmos de *Inteligência Artificial*, por ser muito prático nessa abordagem.

```
d = {chave1: valor1, chave2: valor2, chave3: valor3}
```

O acesso a cada elemento não é feito por índice e a ordem de inserção de elementos não é preservada. Por consequência, não é permitido também o uso de *slices*, ou fatiamento. A duplicidade de elementos também não é permitida e são mutáveis e dinâmicos.

Para criar um dicionário vazio devemos fazer uso de um dos seguintes formatos:

```
d = {}  
d = dict()
```

Após criar um dicionário vazio, como visto acima, a inserção de elementos é feita por meio da atribuição do valor a respectiva chave, como a seguir:

```
d[100] = 'Patricia'  
d[200] = 'Moana'
```

```
d[300] = 'Marcio'
print(d)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Marcio'}
```

Para criar um dicionário com alguns valores, os mesmos valores do exemplo anterior, vamos usar o formato abaixo:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Marcio'}
print(d)
```

Obviamente, a saída será a mesma anterior.

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Marcio'}
```

.get()

Para acessarmos a informação contida em determinada chave, usamos o método *get()*, como a seguir:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Marcio'}
print(d.get(200))
```

Saída:

Moana

Caso a chave passada não exista no dicionário, um erro *KeyError* é gerado. O ideal é fazer uma validação de chave antes de tentar exibí-la:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Marcio'}
chave = int(input('Informe a chave: '))
if chave in d:
    print(d.get(chave))
else:
    print('Chave não existe no dicionário')
```

Saída:

Informe a chave: 500

Chave não existe no dicionário

Observe que usamos um casting de *string* recebida do input para *int*.

Uma opção mais elegante é informarmos um valor padrão, caso não exista a chave pesquisada, como apresentado em:

```
d = {100:'Patricia', 200:'Moana', 300:'Marcio'}
chave = int(input('Informe a chave: '))
print(d.get(chave, 'Não encontrado')) # Caso não encontre a chave
```

Saída:

```
Informe a chave: 400
Não encontrado
```

.del()

Usamos o método *del* para remover elementos do dicionário. A chave-valor associado é permanentemente apagado do dicionário. Vejamos:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}
print(d)
del d[300]
print(d)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}
{100: 'Patricia', 200: 'Moana', 400: 'Marcio'}
```

.pop()

Mesma função do anterior, remover um elemento, porém o *pop* retorna o valor do elemento retirado do dicionário:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}
print(d)
saiu=d.pop(300)
print(saiu)
print(d)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
Ozzy  
{100: 'Patricia', 200: 'Moana', 400: 'Marcio'}
```

Caso o elemento não exista no dicionário, um erro *KeyError* é retornado.

.popitem()

Mesma função do anterior, remover um elemento. A diferença é que esse elemento é aleatório, pode ser qualquer um do dicionário. O *popitem* retorna o valor do elemento retirado do dicionário:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
print(d)  
saiu=d.popitem()  
print('Retirado: ',saiu)  
print(d)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
Retirado: (400, 'Marcio')  
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy'}
```

Caso o comando seja dado em um dicionário vazio, um erro *KeyError* é retornado.

.clear()

Usado para apagar todos os elementos do dicionário, deixando no formato vazio. Caso queiramos apagar por completo o dicionário, tornando-o inacessível, o método a ser usado é o *del*:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
print(d)  
d.clear()  
print(d)  
del d  
print(d)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
{}  
NameError: name 'd' is not defined
```

O erro gerado acima demonstra que o dicionário 'd' ficou indisponível, não foi encontrado.

keys()

O método *keys* retorna todas as chaves contidas no dicionário em uma lista.

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
print(d)  
  
print(d.keys())  
  
for k in d.keys():  
    print(k)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
dict_keys([100, 200, 300, 400])  
100  
200  
300  
400
```

values()

O método *values* retorna todos os valores dos elementos contidos no dicionário em uma lista.

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
print(d)  
  
print(d.values())  
  
for k in d.values():  
    print(k)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
dict_keys(['Patricia', 'Moana', 'Ozzy', 'Marcio'])  
Patricia  
Moana  
Ozzy  
Marcio
```

items()

Para complementar, o método *items* retorna uma lista de tuplas contendo o par "chave-valor" de cada elemento do dicionário.

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
print(d)  
  
for k,v in d.items():  
  
    print(k,"-->",v)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
100 --> Patricia  
200 --> Moana  
300 --> Ozzy  
400 --> Marcio
```

copy()

Cria uma cópia exata do dicionário. O formato de uso é mostrado no exemplo a seguir:

```
d = {100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}  
d1 = d.copy()  
print(d1)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 300: 'Ozzy', 400: 'Marcio'}
```

setdefault(chave, valor)

Com o método *setdefault* é passado um par chave-valor de um elemento a ser pesquisado em um dicionário. Caso essa chave pesquisada seja encontrada no dicionário, o valor correspondente é mostrad, senão, a chave-valor é adicionada ao dicionário e este valor é retornado.

```
d = {100: 'Patricia', 200: 'Moana', 400: 'Marcio'}
print(d.setdefault(100, 'Ozzy'))
print(d.setdefault(300, 'Ozzy'))
print(d)
```

Saída:

Patricia

Ozzy

```
{100: 'Patricia', 200: 'Moana', 400: 'Marcio', 300: 'Ozzy'}
```

update(x)

Com este método *update* podemos adicionar todos os elementos de um dicionário passado como parâmetro a outro dicionário.

```
d = {100: 'Patricia', 200: 'Moana', 400: 'Marcio', 300: 'Ozzy'}
x = {500: 'Tonny', 600: 'Arya'}
d.update(x)
print(d)
```

Saída:

```
{100: 'Patricia', 200: 'Moana', 400: 'Marcio', 300: 'Ozzy', 500: 'Tonny'}
```

Os *dicionários* também estão preparados para serem criados por meio de expressões, assim como acontece nos *sets* e em outras estruturas. Vejamos como isto funciona:

```
quadrados={x:x*x for x in range(1,10,2)}
triplos={x:3*x for x in range(1,6)}

print(quadrados)
print(triplos)
```

Saída:

```
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

```
{1: 3, 2: 6, 3: 9, 4: 12, 5: 15}
```

Bem simples e direto, não é?

E finalizamos aqui o estudo de mais uma importante estrutura de dados utilizada no Python. Embora a intenção seja voltada ao nível iniciante, procuramos ser bem amplos no aprendizado. Vamos seguindo nos estudos!!!

Funções

Em determinados exemplos de scripts, percebemos que há trechos do código que são executados diversas vezes. Não é boa prática essa repetição de códigos ao longo dos scripts porque dificultam a manutenção, fica pouco legível, enfim, uma série de motivos para evitar a repetição.

Podemos usar o que chamamos de *função* e isolar esse trecho de código em um local único e fazendo chamadas sempre que preciso, e o melhor, sem quebrar o desenrolar da atividade.

Ainda podemos coonsiderar a *reusabilidade* do código até em outros scripts, o que torna uma grande vantagem aplicar essa solução. As funções que acompanham o software Python automaticamente são chamadas de *funções incorporadas ou funções pré-definidas*, inclusive já usamos algumas dessas neste curso, como a *type()*, *input()*, *print()* etc.

E o mais importante também nesse assunto é que podemos criar as nossas próprias funções, adequando a linguagem ao nosso dia a dia acadêmico e até profissional. As funções que são desenvolvidas pelo programador explicitamente de acordo com os requisitos de negócios são chamadas de *funções definidas pelo usuário*.

Sintaxe

Para criar uma função, devemos considerar o formato a seguir:

```
def nome_da_função(parâmetros) :  
    """ doc string """  
    comandos  
    -----  
    return valor
```

Considere que:

- *def* é obrigatório e trata-se de uma palavra reservada que

- indica uma função
- `return` é opcional e indica o retorno da função
- `docstring` é um trecho opcional onde informações da função podem ser inseridas e formatadas para uso em sistemas de ajuda

Vamos ver alguns exemplos de uso das funções:

Função sem parâmetros

Exemplo de uma função que, quando acionada, exibe uma saudação na tela. Observe que primeiramente a função é definida e, em seguida é chamada duas vezes no decorrer do código.

```
def diga_oi():  
    print('Oi, tenha um bom dia!')
```

```
diga_oi()  
diga_oi()
```

Saída:

```
Oi, tenha um bom dia!  
Oi, tenha um bom dia!
```

Função com parâmetros

Dessa vez a saudação se torna mais dinâmica, com a função alterando a mensagem e deixando-a mais personalizada. Fizemos uso de uma *string* formatada, para exemplificar.

```
def diga_oi(nome):  
    print(f'Oi, {nome}. Tenha um bom dia!')
```

```
diga_oi('Marcio')  
diga_oi('Patricia')
```

Saída:

```
Oi, Marcio. Tenha um bom dia!  
Oi, Patricia. Tenha um bom dia!
```

Return

Quando usamos o *return*, a função retornará um valor e este valor pode ser armazenado em uma variável do trecho "chamador"

para ser usado posteriormente. Esse exemplo nos mostra como isso acontece. Vamos criar uma função que calcula e retorna a soma de dois números recebidos:

```
def somar(num1, num2):  
    return num1 + num2  
  
resultado = somar(12,36)  
print(resultado)
```

Saída:

48

Obviamente esse exemplo é bem simples para o nível dos alunos, mas o intuito é apresentar o processo de retorno de valores das funções.

Alguns conceitos importantes nesse momento:

1. Um grupo de linhas de código com algum nome é chamado de **Função**
2. Um grupo de funções salvas em um arquivo, é chamado de **Módulo**
3. Um grupo de Módulos é o que chamamos de **Biblioteca**

Manipulando Arquivos - *file*

Nos sistemas computacionais, há basicamente dois formatos de arquivos utilizados, os arquivos de texto e os arquivos binários.

Arquivos de Texto são usados para armazenar dados em forma de caracteres, como por exemplo, "*arquivo.txt*".

Normalmente podemos usar Arquivos Binários para armazenar dados em formato binário como imagens, vídeos, áudios...

.open()

Abrimos os arquivos através do método *open()* que tem o seguinte formato, com dois parâmetros:

```
f = open(nome_do_arquivo, modo)
```

O nome do Arquivo deve estar entre aspas (simples ou duplas) ou pode ser também passado através de uma variável, como nos exemplos a seguir:

```
f = open('arquivo.txt', modo)
```

Ou usando variáveis:

```
arquivo = 'arquivo.txt'
modo = 'r'
f = open(arquivo, modo)
f.close()
```

O modo também é passado como parâmetro em formato caracter, conforme os exemplos posteriores:

```
f = open('arquivo.txt', 'r')
f.close()
```

Os modos permitidos em Python são:

- **r** → abre um arquivo existente para operação de leitura. O ponteiro do arquivo está posicionado no início do arquivo. Se o arquivo especificado não existir, uma exceção *FileNotFoundError* é gerada. Este é o modo padrão.
- **w** → abre um arquivo existente para operação de escrita. Se o arquivo já contém alguns dados então ele será substituído. Se o arquivo especificado ainda não estiver disponível, este modo criará esse arquivo.
- **a** → abra um arquivo existente para operação de acréscimo. Ele não substituirá os dados existentes. Se o arquivo especificado ainda não está disponível, então este modo criará um novo arquivo.
- **r+** → Para ler e escrever dados no arquivo. Os dados anteriores no arquivo não serão apagados. O ponteiro do arquivo é colocado no início do arquivo.
- **w+** → Para escrever e ler dados. Ele substituirá os dados existentes.
- **a+** → Para anexar e ler dados do arquivo. Não substituirá os dados existentes, escrevendo os dados a partir do final do arquivo.
- **x** → Abrir um arquivo em modo de criação exclusivo para operação de escrita. Se o arquivo já existe, a exceção *FileExistsError* é gerada.

Nota: Todos os modos acima são aplicáveis a arquivos de texto. Se os modos acima forem sufixados com *'b'* então estes representam o uso em arquivos binários.

Exemplo: rb,wb,ab,r+b,w+b,a+b,xb

.close()

Sempre que abrimos um arquivo e completarmos a operação correspondente, o comando de fechar deve ser usado. O formato é visto a seguir:

```
arq = "arquivo.txt"
f = open(arq, "r")
f.close()
```

Lendo dados de um arquivo

como já vimos, para ler os dados de um arquivo devemos utilizar o modo 'r'. Podemos ler dados de caracteres do arquivo de texto usando os seguintes métodos de leitura:

- `read()` → Para ler todos os dados do arquivo;
- `read(n)` → Para ler "n" caracteres do arquivo, sendo "n" um inteiro positivo;
- `readline()` → Para ler apenas uma linha do arquivo, até encontrar o símbolo de quebra de linha (`\n`).
- `readlines()` → Para ler todas as linhas do arquivo devolvendo o resultado como uma lista.

Exemplos: *Lendo todo o conteúdo do arquivo de nome texto.txt*

```
f=open("texto.txt",'r')
data=f.read()
print(data)
f.close()
```

Lendo 10 caracteres do arquivo de nome texto.txt

```
f=open("texto.txt",'r')
data=f.read(10)
print(data)
f.close()
```

Lendo o conteúdo (3 linhas, nesse caso) do arquivo de nome texto.txt, e mostrando o conteúdo, com uma linha por vez

```
f=open("texto.txt",'r')
linha1=f.readline()
print(linha1,end='')
linha2=f.readline()
```

```
print(linha2,end='')
linha3=f.readline()
print(linha3,end='')
f.close()
```

Lendo todo o conteúdo e mostrando uma linha por vez:

```
f=open("texto.txt",'r')
linhas=f.readlines()
for linha in linhas:
    print(linha, end='')
f.close()
```

Escrevendo dados em arquivo

Podemos escrever caracteres em arquivos usando um dos seguintes métodos:

1. `write(str)`
2. `writelines(lista de linhas)`

Usando o *write*:

```
f = open('arquivo.txt', 'w')
f.write('Pera\n')
f.write('Uva\n')
f.write('Goiaba\n')
print('Dados escritos no arquivo com sucesso.')
f.close()
```

Usando o *writelines*:

```
frutas = ['pera\n', 'uva\n', 'goiaba\n', 'caju\n']
f = open('arquivo.txt', 'w')
f.writelines(frutas)
print('Dados escritos no arquivo com sucesso.')
f.close()
```

Obs.: Ao escrever dados usando os métodos `write()` ou `writelines`, é obrigatório fornecer um separador de linha (`\n`), caso contrário, os dados serão gravados em uma única linha.

Lendo as propriedades do arquivo

O "objeto File" em python possui várias propriedades que podem

ser acessadas para termos mais informações, como nos códigos a seguir:

Uma vez que abrimos um arquivo e obtivermos o *objeto de arquivo*, podemos consultar vários detalhes relacionados a esse arquivo por meio de suas propriedades. Veremos algumas:

- `name` → Nome do arquivo aberto
- `mode` → Modo em que o arquivo foi aberto
- `closed` → Retorna um valor booleano indicando se o arquivo está fechado ou não.
- `readable()` → Retorna um valor booleano indicando se o arquivo está aberto em modo *leitura* ou não.
- `writable()` → Retorna um valor booleano indicando se o arquivo está aberto em modo *escrita* ou não.

Exemplos:

```
1. f=open("abc.txt",'w')
2. print("Nome: ", f.name)
3. print("Modo: ", f.mode)
4. print("Disponível para leitura: ", f.readable())
5. print("Disponível para escrita: ", f.writable())
6. print("Arquivo está fechado : ", f.closed)
7. f.close()
8. print("Arquivo está fechado : ", f.closed)
```

A estrutura *with*

Nos tópicos anteriores percebemos que sempre precisamos usar o método `close()` para fechar os arquivos após a execução das tarefas. Há uma estrutura apropriada para controlar este fechamento de forma dinâmica, denominada *with*. O *with* fecha automaticamente o arquivo no momento da saída dessa estrutura. Veremos como ele se comporta no exemplo a seguir:

Exemplo:

```
with open('texto.txt', 'w'):
    f.write('pera\n')
    f.write('uva\n')
    f.write('maçã\n')
    print('Arquivo aberto: ', f.closed)
print('Arquivo aberto: ', f.closed)
```

Saída:

Arquivo aberto: False

Arquivo aberto: True

Percorrendo o arquivo

Para direcionarmos o cursor no arquivo, ou seja, para gerenciarmos o ponto de leitura dentro do arquivo, usamos os métodos `.seek()` e `.tell()`.

`.tell()`

Podemos usar o método `.tell()` para perguntar a posição atual do cursor dentro do arquivo.

Veremos o trecho de código:

```
f=open("texto.txt","r")
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(3))
print(f.tell())
```

Arquivo

texto.txt

manga limão banana melão

Saída: 0

ma

2

nga

5

`.seek()`

O método `*.seek()` é responsável por mover o cursor (um ponteiro para o arquivo) para uma localização específica. O exemplo a seguir tenta elucidar como o método é usado:

Exemplo

```
data = 'Numa folha qualquer eu mostro'
```

```
texto = open('texto.txt', 'w')
texto.write(data)
texto.close()
```

```
with open('texto.txt', 'r+') as f:
```

```

frase = f.read()
print(frase)
print('Posição atual do cursor: ', f.tell())
f.seek(21)
print('Posição atual do cursor: ', f.tell())
f.write('u desenho um sol amarelo\n')
f.seek(0)
frase=f.read()
print('Arquivo após modificação:')
print(frase)

```

O arquivo inicialmente é criado vazio e depois o primeiro conteúdo é salvo neste.

Arquivo

texto.txt

Numa folha qualquer eu mostro

Saída: Numa folha qualquer eu mostro

Posição atual do cursor: 29

Posição atual do cursor: 21

Arquivo após modificação:

Numa folha qualquer eu desenho um sol amarelo

Extraindo informações do arquivo

Após aprendermos como trabalhar com os arquivos, vamos agora entrar um pouco mais nesse assunto entendendo como funciona a manipulação de diretórios e pegar mais informações desses arquivos. Para isso, as bibliotecas *os* e *sys*, padrão no Python, precisam ser utilizadas.

Explanaremos o conceito de módulos, submódulos e funções através dos exercícios seguintes.

Para verificarmos se determinado arquivo existe no diretório que estamos acessando, um script como este a seguir pode ser usado:

Exemplo:

```
import os, sys
```

```
arquivo = input('Informe o nome do arquivo: ')
```

```
if os.path.isfile(arquivo):
    print('Arquivo', arquivo, 'existe no diretório.')
```



```

        f = open(arquivo, 'r')
    else:
        print('Arquivo', arquivo, 'NÃO existe no diretório.')
        sys.exit(0)
    print('Conteúdo do arquivo: ')
    data = f.read()
    print(data)

```

Saída para arquivo não existente

```

Informe o nome do arquivo: text.txt
Arquivo text.txt NÃO existe no diretório.

```

Saída para arquivo existente

```

Informe o nome do arquivo: texto.txt
Arquivo texto.txt existe no diretório.
Conteúdo do arquivo:
Numa folha qualquer eu desenho um sol amarelo

```

Observe que o comando `sys.exit(0)` interrompe a execução e não executa os demais comandos contido no resto do programa.

Contando linhas, letras e palavras

Vamos criar um exemplo de código onde pegamos um arquivo de texto modelo, contendo a letra da música 'Aquarela", do Toquinho e contaremos a quantidade de caracteres, linhas e palavras deste arquivo:

Baixar o arquivo no endereço do [github](#).

O script seguinte deve ser salvo no mesmo diretório do arquivo da letra da música que foi baixado.

Exemplo:

```

import os,sys
arquivo=input("Informe o nome do arquivo: ")
if os.path.isfile(arquivo):
    print('Arquivo', arquivo, 'existe no diretório.')
    f = open(arquivo, 'r')
else:
    print('Arquivo', arquivo, 'NÃO existe no diretório.')
    sys.exit(0)

```

```

linhas=palavras=letras=0

```

```
for linha in f:
    linhas += 1
    letras += len(linha)
    palavra=linha.split()
    palavras += len(palavra)

print("Quantidade de Linhas  :",linhas)
print("Quantidade de Palavras:",palavras)
print("Quantidade de Letras  :",letras)
```

Saída

```
Informe o nome do arquivo: **aquarela.txt**
Arquivo aquarela.txt existe no diretório.
Quantidade de Linhas   : 60
Quantidade de Palavras: 295
Quantidade de Letras   : 1550
```

Como desafio, você pode acrescentar ao código a contagem de vogais e consoantes, por exemplo...

Manipulando dados binários

Vamos ver como podemos manipular arquivos com dados binários. Neste exemplo, trata-se de um script para criar uma cópia de um arquivo binário contendo uma imagem em formato *png*.

```
f1 = open('arq1.png', 'rb')
f2 = open('arq2.png', 'wb')
dados = f1.read()
f2.write(dados)
print('Cópia do arquivo realizada com sucesso.')
```

Execute o script tendo uma imagem inicial com o nome *'arq1.jpg'* no mesmo diretório do script.

Manipulando arquivo csv

Um formato de arquivo muito utilizado é o csv - character separated values - valores separados por um delimitador. Comumente uma vírgula, pode também ser usado o ponto e vírgula, o tabulador(TAB). Arquivos nos dois primeiros formatos recebem a extensão *.csv*. Já os com *tab* recebem a extensão *.tsv*, embora menos usual. Há alguns aplicativos que trabalham bem, mesmo com extensão *.txt*. consideraremos apenas

a `.csv` no curso.

Utilizaremos mais uma biblioteca padrão (*builddin*) do Python, a `csv`.

O exemplo a seguir coleta os dados de um grupo de empregados de uma determinada empresa e depois salva em um arquivo `.csv`.

```
import csv

with open("empresa.csv","w",newline='') as f:

    w=csv.writer(f)
    w.writerow(["MAT","NOME","SALARIO","EMAIL"])
    n=int(input("Quantidade de funcionários:"))

    for i in range(n):
        mat=input("Matrícula:")
        nome=input("Nome:")
        salario=input("Salário:")
        email=input("Email:")

        w.writerow([mat,nome,salario,email])

print("Dados inseridos com sucesso!")
```

Saída:

```
Quantidade de funcionários:3
Matrícula:100
Nome:Marcio
Salário:1200.00
Email:marcio@mail.com
Matrícula:200
Nome:Patty
Salário:2200.00
Email:patty@mail.com
Matrícula:300
Nome:Moana
Salário:3200.00
Email:moana@mail.com
Dados inseridos com sucesso!
```

O atributo `newline=''` no começo do exemplo acima deve ser

usado por conta da não necessidade de se criar uma linha em branco entre as linhas de dados, ficando melhor para posterior manipulação e exibição do arquivo.

Após criado o arquivo, vamos agora preparar um script para abrir o mesmo como leitura e exibir os dados cadastrados.

```
import csv

f=open("empresa.csv",'r')
r=csv.reader(f)
dados=list(r)

for linha in dados:
    for item in linha:
        print(item,"\t",end='')
    print()
```

Saída:

MAT	NOME	SALARIO	EMAIL
100	Marcio	1200.00	marcio@mail.com
200	Patty	2200.00	patty@mail.com
300	Moana	3200.00	moana@mail.com

E assim findamos esse estudo básico de manipulação de arquivos por meio do Python. No tópico a seguir vamos aprender a manipular os diretórios (pastas) em nosso computador. Aguarde.

Manipulando Diretórios

Ao lidarmos com as atividades no dia a dia de uso dos computadores, é comum precisarmos executar tarefas diversas sobre os diretórios que estamos trabalhando, tais como:

- Saber qual o diretório que estamos trabalhando;
- Criar um diretório;
- Remover um diretório;
- Renomear um diretório;
- Listar o conteúdo de um diretório;
- ... dentre outras ...

O Python nos fornece uma forma para lidar com diretórios, através do módulo *os*. Vamos ver como funciona:

Para ver o corrente diretório

```
import os
pasta = os.getcwd()
print('Diretório corrente: ', pasta)
```

Para criar um subdiretório

```
import os
os.mkdir('exemplo')
print('Subdiretório criado.')
```

Para criar dois níveis de diretório

```
import os
os.mkdir('exemplo/outrapasta')
print('Subdiretórios criados.')
```

No exemplo acima, considere que o diretório *'exemplo'* deve existir.

Para criar vários níveis de diretório

```
import os
os.makedirs('niv1/niv2/niv3')
print('Subdiretórios criados.')
```

Para remover um subdiretório

```
import os
os.rmdir('niv1/niv2/niv3')
print('Subdiretório foi removido.')
```

Neste caso, o último subdiretório informado (*niv3*) será removido.

Para remover vários níveis de diretório

```
import os
os.removedirs('niv1/niv2/niv3')
print('Subdiretórios foram removidos.')
```

Neste caso, os subdiretório informados (*niv1*, *niv2* e *niv3*) serão removidos.

Para renomear um diretório

```
import os
os.rename('niv1', nivel1')
print('Subdiretório foi renomeado.')
```

Para exibir o conteúdo do diretório

O script a seguir exibe o conteúdo do diretório em uma lista

```
import os
print(os.listdir('.'))
```

O método `.listdir` exibe todo o conteúdo do diretório atual mas não o conteúdo de subdiretório. Para exibir todo o conteúdo inclusive de subdiretórios, devemos usar a função `walk()`, visto a seguir:

```
os.walk(path, topdown = True, onerror = None, followlinks = False)
```

Onde:

- `path` = Caminho do diretório. Retorno do `cwd`.
- `topdown = True` indica que o objeto será percorrido do início ao fim
- `onerror = None` define, em caso de erro detectado, qual função deve ser executada.
- `followlinks = True` indica que deve mostrar diretórios apontados por links simbólicos

Um objeto iterável é retornado e o mesmo pode ser percorrido com um laço `for`. Vamos ao exemplo:

```
import os
for diratual,diretorios,arquivos in os.walk('.'):
    print("Diretório corrente:",diratual)
    print("Diretórios:",diretorios)
    print("Arquivos:",arquivos)
    print()

print('Dados exibidos com sucesso.')
```

Teste o script em sua máquina, em diversos diretórios, e comente com os amigos durante a aula.

Findamos aqui essa importante parte do curso. Agora vamos apresentar alguns exemplos de uso de scripts de forma mais aproximada para as tarefas do dia a dia.

Exemplo - Script buscador

Para efeito de exemplo do poderio da linha de comandos, o script a seguir faz uma busca na internet através do buscador no site Google e retorna os cinco primeiros links sobre determinado termo informado.

O script faz uso de uma biblioteca que não vem como padrão na instalação do Python. Você precisará instalar a biblioteca "google" através do seguinte comando no powershell ou prompt de comandos:

```
pip install google
```

E em seguida criar um script conforme mostrado a seguir.

Obs.: O caracter de 'cerquilha' # no começo da linha é um comentário no código e não será considerado pelo interpretador Python.

Código 01

```
# arquivo buscador.py
# você deve instalar a biblioteca google
# pip install google

# Importando a biblioteca
from googlesearch import search

# Título apenas para melhor apresentação
print("-----Programa Buscador-----")

# Recebendo o termo a pesquisar
query = input("Informe o termo a ser pesquisado: ")

# realizando a busca e apresentando os cinco primeiros resultados
for i in search(query, start=0, stop=6):
    print(i)

# observar o tabulador dentro do "for", o que é obrigatório no Python.
# os links serão apresentados
```

As linhas referente ao comentário de código, não será efetivamente executado.



[logotipo](#)

Tela do Mu Editor

Saída em destaque:



[logotipo](#)

Destaque para o termo pesquisado: carro elétrico projeto

Obrigado

Finalizamos aqui esse breve curso introdutório a programação de computadores com uso da linguagem Python. Esperamos que seus scripts sejam exemplos de ótimo código, que você se torne um profissional de sucesso e que contribua para a busca por soluções para os diversos problemas da sociedade moderna, tornando a sua região de atuação um local melhor e muito mais humano.

Sabemos que muito há ainda a aprender sobre esse vasto campo, mas esperamos ter cumprido com a nossa atividade de, primeiramente, levar uma visão um pouco mais acessível sobre a programação de computadores e, em segundo momento, que esse curso te forneça meios de criar soluções para as tarefas do dia a dia através do melhor uso dessas maravilhosas ferramentas, os computadores.

A semente foi plantada. Bons estudos! Bons frutos!