**Python Fundamentos para Anáise de Dados 3.0**

*Curso Gratuito Oferecido Pela Data Science Academy*.

Link para os Cursos: <https://www.datascienceacademy.com.br/cursosgratuitos>

& Para iniciar, dicas:

- Mantenha seu código limpo e organizado.

- Esparso é melhor que denso.

- Sempre documente (comente) seu código.

- Siga os padrões.

- Erros nunca são silenciosos, a não ser que seja de propósito.

- “ Simples é melhor que complexo e complexo é melhor que complicado “.

- Não se sinta obrigado a criar classes sem uma boa razão.

& **Identificação**: (Tab ou 4 spaces). Faz parte da sintaxe em python.

& **Comentários em Python**: Começam com o caractere # ou 3 aspas duplas “””...”””!.

# é um comentário em uma única linha.

“”” é um comentário em mais de uma linha”””.

& Python possui 2 tipos de números principais:

Int – números inteiros, positivos ou negativos. Ex: -7 e 7.

Float – números fracionários, positivos ou negativos. Ex -7.1 e 7.1

& Podemos usar a função type(), para saber qual é o tipo de um número.

& Podemos usar as funções int() ou float() para converter para números inteiros ou fracionários.

& As variáveis são usadas em nosso código Python para armazenar valores que queremos usar mais tarde. São espaços em memória que armazenam valores.

Por exemplo, nós podemos armazenar o valor 10 na variável b.

b = 10

O sinal de igual atribui o valor à direita (10) à variável do lado esquerdo (b). Você pode sobrescrever uma variável com um novo valor sempre que quiser. A variável assumirá o novo valor.

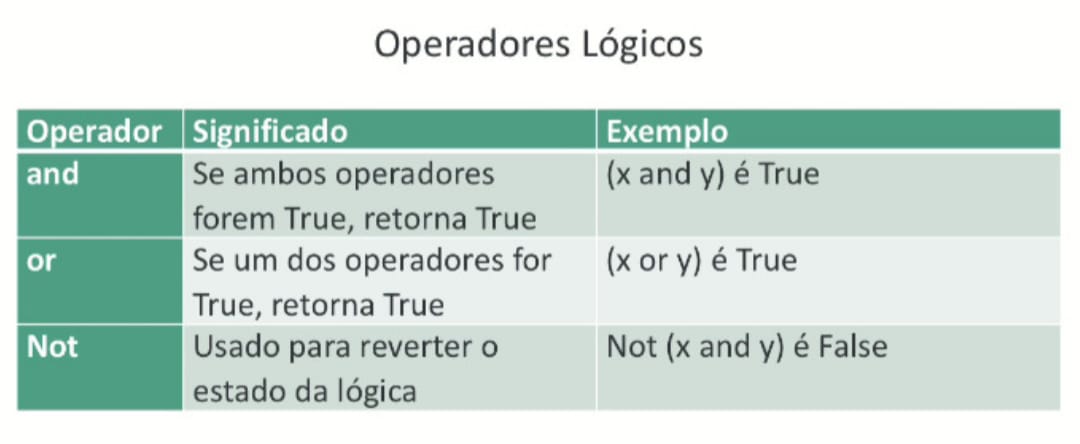
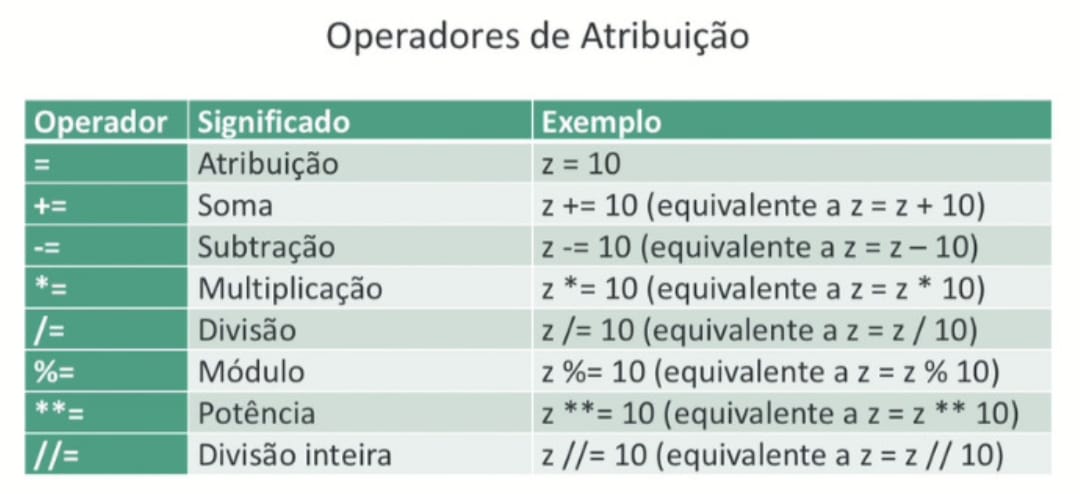
& **Lembrando**: Um sinal de igual é atribuição de valor. =

Dois sinais de igual operação de igualdade. ==

& Existem algumas regras que devem ser seguidas ao definir nomes de variáveis:

1. Os nomes das variáveis não podem começar com um número.
2. Não pode haver espaços no nome; utilize \_ em vez disso.
3. Não é possível usar qualquer um desses símbolos: ‘ “,<>/|\()@#$%^&\*~-+!
4. Não se pode usar uma palavra reservada como nome de variável: False; class; finally; is; return; None; continue; for; lambda; try; expect; True; def; from; nonlocal; while; and; del; global; not; with; in; as; elif; if; Or; yield; Assert; Else; import; Pass; break; raise.





& Strings são usadas em Python para gravar informações em formato de texto, como nomes por exemplo. Strings em Python são na verdade uma sequência de caracteres, o que significa, basicamente, que Python mantém o controle de cada elemento da sequência.

Python entende a string “Olá”, como sendo uma sequência de letras em uma ordem específica. Isso significa que você será capaz de usar a indexação para obter um caracter específico (como a primeira letra ou a última letra)

Strings – sequência imutável de caracteres ou apenas 1 caracter

Ex: “Essa é uma string”

“a”

& Indexando Strings: Já sabemos que Strings são uma sequência. Isso significa que Python pode usar índices para chamar partes da sequência.

Em Python, usamos **colchetes []** para representar o índice de um objeto.

* **EM PYTHON, a indexação começa por 0**

Por exemplo:

texto = “Python e Análise de Dados”

texto[0] = P

texto[1] = y

texto[2] = t

É importante ressaltar que as strings têm uma importante propriedade conhecida como imutabilidade. Isto significa que uma vez que é criada uma string, os elementos dentro dela não podem ser substituídos ou alterados.

& Funções Build-in de Strings: Python é uma linguagem orientada a objeto, sendo assim as estruturas de dados possuem atributos (propriedades) e métodos (rotinas associadas às propriedades). Tanto os atributos quanto os métodos são acessados usando ponto (.).

Os métodos estao sob a forma:

objeto.atributo

objeto.método()

objeto.método(parâmetros)

Parâmetros são argumentos extras, que podemos passar para o método.

& As listas podem ser consideradas a versão geral de uma sequência em Python. Ao contrário de Strings, as listas são **mutáveis, ou seja, os elementos dentro de uma lista podem ser alterados.**

As listas são construídas com o uso do colchetes [] e vírgulas separando cada elemento da lista.

Ex: lista = [item1, item2, ..., itemz]

Listas não têm tamanho fixo (o que significa que não podemos espicificar quão grande uma lista será).

Listas não têm restrições de tipo fixo.

Uma grande característica de estruturas de dados em Python é que elas suportam alinhamento. Isto significa que podemos usar estruturas de dados dentro de estruturas de dados.

& Os dicionários são construídos com o uso de chaves {} e vírgulas separando cada elemento do dicionário.

Ex: dict = {k1:v1, k2:v2, ..., kn:vn}

Mapeamentos são uma coleção de objetos que são armazenados por uma chave, ao contrário de uma sequência de objetos armazenados por sua posição relativa.

Um dicionário Python consiste de uma chave e, em seguida, um valor associado. Esse valor pode ser quase qualquer objeto Python.

Dicionários são mapeamento de chaves e valores.

Ex: {chave1: valor1, chave2: valor2}

& Tuplas são muito semelhantes às listas, no entanto, ao contrário de listas, tuplas são imutáveis, o que significa que não podem ser alteradas. Tuplas são usadas para apresentar dados que não devem ser alterados, como os dias da semana ou datas em um calendário.

Tuplas não são usadas com frequência, como listas por exemplo, mas são usadas quando é necessário imutabilidade. Se em um programa é necessário certeza de que os dados não sofrerão mudança, então tupla pode ser a solução. Ela fornece uma fonte conveniente de integridade de dados.

São construídas com o uso de **parênteses ()** e vírgulas separando cada elemento da tupla.

Ex: tupla = (item1, item2, ..., itemz)

& O condicional **if (se)** nos permite dizer ao computador para executar ações com base em um determinado conjunto de resultados.

Verbalmente, podemos imaginar que estamos dizendo ao computador: “Ei, caso isso aconteça, execute esta ação.”

O Elif substitui a necessidade de criar várias estruturas de if...else aninhadas.

Ex: if(expressão 1):

print(“comando executado caso a expressão 1 seja Verdadeira”)

elif(expressão 2):

print(“comando executado caso a expressão 1 seja Falsa e expressão 2 seja Verdadeira”)

else:

print(“comando executado caso as expressões 1 e 2 sejam Falsas”)

**OBS: Importante ter uma boa compreensão de como funciona o recuo em Python (indentação) para manter a estrutura em ordem no seu código.**

& **For**: Valida cada item em uma série de valores.

Ex: **for** ITEM **in** SÉRIE-DE-ITEMS:

If ITEM > 0:

Executar comandos

**SUPER LEMBRETE:** Tuplas () ; Lista [] ; Dicionário {}

& O **loop while** em Python é uma das formas mais comuns para executar iteração. A instrução **while** será executada repetidamente, seja uma única instrução ou grupo de instruções, desde que uma condição seja verdadeira.

Valida cada item em uma série de valores.

Ex: while (expressão1)

print (“comando executado caso a expressão1 seja Verdadeira”)

**Importante ressaltar** que dentro do código é preciso fazer com que a expressão1 deixe de ser verdadeira em algum momento, caso contrário o loop vai ficar executando infinitamente e irá travar o computador.

& A função **range()** nos permite criar uma lista de números em um intervalo específico.

A função **range()** tem o seguinte formato:

range([start],[stop],[step])

[start] – número que inicia a sequência

[stop] – número que encerra a sequência (não é incluído na sequência)

[step] – diferença entre cada número da sequência

Ex: range(50, 101, 2)

& **Métodos** permitem executar ações específicas no objeto e podem tambem ter argumentos, exatamente como uma função.

Os métodos são executados sob a forma:

Ex: objeto.método (arg1, arg2, etc...)

Com o Jupyter Notebook podemos ver rapidamente todos os métodos possíveis para um objeto, usando a tecla TAB.

Por exemplo, os **métodos para o objeto lista** são:

append ; count ; extend ; insert ; pop ; remove ; reverse ; sort

& **Função** é um dispositivo que agrupa um conjunto de instruções para que elas possam ser executadas mais de uma vez. Funções também permitem especificar os parâmetros que podem servir como entrada para as funções.

Em um nível mais fundamental, a construção de funções nos permite reutilizar código, sem ter que escrevê-lo novamente. Nas aulas de Strings, utilizamos a função **len()** para obter o comprimento de uma String. Com funções, escrevemos o código uma única vez e repetimos a mesma instrução, fazendo a chamada à função, quantas vezes forem necessárias.

O formato geral de uma função é:

def nome da função(arg1, arg2):

‘’’ Aqui vão os comentários, documentando sua função ‘’’

<Aqui vai seu código>

<Retorno desejado pela função>

**“ Funções** em Python são uma forma de escrever a sua lógica em um único pacote e utilizá-la em diferentes lugares no seu código e quantas vezes quiser. ”

& **Expressões lambda** nos permitem criar funções “anômimas”. Isto significa que podemos fazer rapidamente funções ad-hoc (criadas em tempo de execução) sem a necessidade de definir uma função usando a palavra reservada **def.** Objetos de função desenvolvidos executando expressões lambda funcionam exatamente da mesma forma como aqueles criados e atribuídos pela palavra reservada **def**. Mas há algumas diferenças fundamentais que fazem lambda útil em funções especializadas.

* O corpo do lambda é uma única expressão, não um bloco de instruções.
* O corpo do lambda é semelhante a uma instrução de retorno do corpo def.

Expressões lambda realmente são uteis, quando usadas em conjunto com as funções map(), filter() e reduce().

Expressões lambda são usadas para criar funções simples. São também chamadas funções in-line ou apenas funções anônimas.

Ex: lambda x: x\*\*2 lambda (palara reservada) ; x: (parâmetro de entrada); x\*\*2 (retorno da função, que é o x elevado a segunda potência.)

Diferença entre def e lambda para criar funções:

Def - > cria um objeto e atribui um nome a ele (nome da função).

Lambda - > cria um objeto, mas o retorna como um resultado em tempo de execução.

& **Arquivos –** Métodos e suas Utilizações

open() Usada para abrir o arquivo.

read() Leitura do arquivo.

write() Gravação no arquivo.

seek() Retorna para o início do arquivo.

readlines() Retorna a lista de linhas do arquivo.

close() Fecha o arquivo.

& **Módulos** em Pythons são simplesmente arquivos Pyton com a extensão **.py**, que implementam um conjunto de funções. Importando o módulo em nosso script Python, usando o comando import: **import math**

Também é possível importar funções específicas de um módulo. **from math import sqrt**

& **Pacotes** são uma forma de estruturar os módulos Python. **import modulo Import pacote.modulo**

Um pacote é um conjunto de módulos Python. Enquanto um módulo é um único arquivo Python, um pacote é um diretório de módulos Python contendo um arquivo **\_\_init\_\_.py**

Para fazer a importação: **import pacote.modulo**

& O repositório PyPi é onde ficam armazenadas os pacotes Python. **pypi.python.org/pypi**

& **Programação Funcional** é uma programação orientada à expressão(Diferentemente de uma Programação Orientada a Objetos).

Algumas funções orientadas à expressão: map(Função, Sequência) ; reduce(Função, Sequência) ; filter(Função, Sequência) ; lambda ; list comprehension

& **Map** é uma função que recebe 2 argumentos: uma função ; uma sequência. ( map(Função, Sequência) )

O primeiro argumento é o nome de uma função e o segundo uma seuquência (por exemplo uma lista).

map() aplica a função a todos os elementos da sequência.

Uma nova lista com os elementos alterados pela função é retornado.

& **Reduce** é uma função que recebe 2 argumentos (assim como a função map): uma função ; uma sequência.

reduce(Função, Sequência)

Ao contrário da função map que aplica a função a cada elemento da sequência e retorna uma outra sequência de elementos, a função reduce aplica a função passada como parâmetro aos elementos da sequência, até que reste apenas um elemento.

& **Filter** também recebe 2 argumentos, uma função e uma sequência. Ex: **filter(função, sequência).**

A função **filter()** oference uma maneira conveniente para filtrar todos os elementos de uma sequência, para os quais a função retorne True. A função passada como parâmetro para **filter()**, deve retornar um valor booleano, True ou False. A função será aplicada a todos os valores de uma sequência e os valores serão retornados, apenas se retornarem True para a função.

& **List Comprehension** aplica uma expressão arbitrária (ao invés de aplicar apenas uma função) a uma sequência de elementos. List Comprehension permite desenvolver listas usando uma notação diferente. Seria essencialmente uma linha de loop for, construída dentro de []. Ex: **lst = [x for x in “sequência”]**.

Normalmente, a função **List Comprehension** é mais veloz que a função **map()** ou **filter()**.

& **Zip()** agrega os valores de duas sequências e retorna uma tupla. Ex: zip(sequência, sequência). Pode ser usado quando o número de elementos for diferente em cada sequência, mas o objeto resultante terá o mesmo número de elementos da sequência menor. Ex: **zip([1,2,3,4],[1,2,3]) = (1,1) (2,2) (3,3)**.

& **Enumerate** permite retornar o índice de cada valor em uma sequência, à medida que você percorre toda a sequência. Enumerate retorna uma tupla no formato tupla(índice, valor). Recebe apenas um parâmetro. Ex: enumerate(sequência).

& **Tratamento de Erros e Exceções**:

Mesmo quando uma expressão estiver sintaticamente correta, ainda poderão ocorrer erros e neste caso, chamamos de Exceções.

Podemos tratar exceções em Python da seguinte forma:

**try:**

**Aqui vão as operações**

**except Exceção 1:**

**Se houver a Exceção 1, execute esse bloco**

**except Exceção 2:**

**Se houver a Exceção 2, execute esse bloco**

**Else:**

**Se não houver exceção, execute esse bloco**

OBS: Ainda existe a palavra reservada **finally** que nos permite executar o código mesmo que exceções ocorram. **Finally** sempre é executado.

***Capítulo de PROGRAMAÇÃO ORIENTADA A OBJETOS ABAIXO.***

& A orientação a objetos é um modelo de **análise**, **projeto** e **programação** de sistemas de **software** baseado na composição e interação entre diversas unidades de software chamadas de objetos.

A **Programação Orientada a Objetos (POO)**, foi criada para tentar aproximar o mundo real e o mundo virtual. A ideia fundamental é tentar simular o mundo real dentro do computador.

Na Programação Orientada a Objetos, o programador é responsável por moldar o mundo dos objetos, e definir como os objetos devem interagir entre si.

Os objetos “conversam” uns com os outros através do envio de mensagens e o papel principal do programador é definir quais serão as mensagens que cada objeto pode receber e também qual a ação que o objeto deve realizar ao receber cada mensagem.

Exemplos de linguagens orientadas a objetos utilizadas: **Java, C#, C++, Ruby, Lisp, Python**, **etc**.

& **Programação Estruturada** - Sequência, Decisão, Iteração(Repetição).

**Programação Orientada a Objetos** - Aplicação voltada a objetos. Aplicação > Objeto > Atributo > Método.

*Principais conceitos da Programação Orientada a Objetos:*

& **Classes e Objetos** > A classe é a estrutura básica do paradigma de orientação a objetos, que representa o tipo do objeto, um modelo a partir do qual os objetos serão criados.

Uma classe é apenas um molde. Uma especificação que define o que um objeto desse tipo deverá ter como atributo e como ele deve se comportar.

Por exemplo: Podemos criar a classe **Livro.**

A classe é uma espécie de template que define a natureza de um futuro objeto.

A partir de classes, nós construímos instâncias. Cada instância é um objeto.

Uma instância, é um objeto específico, criado a partir de uma classe.

Objetos representam entidades, com suas qualidades(atributos) e ações(métodos) que estas podem realizar.

Em Python, tudo é um **Objeto**.

*Objetos definidos pelo usuário em Python são criados a partir de instâncias de classes criadas usando a palavra reservada* *class*.

O nome de uma classe começa com letra maiúscula.

*Exemplo*: **class** > Car

**objects** > Mercedes, Bmw, Audi

Em Python, novos objetos são criados a partir das classes. O objeto é uma instância da classe, que possui características próprias.