

EBOOK **DESCOMPLICANDO** **PROGRAMAÇÃO** **COM PYTHON**



MÃO NA MASSA!
CÓDIGOS, DICAS E DESAFIOS





Tópicos

Resumo do eBook

O que é Python?

Mão na massa.

Criando seu Primeiro Código no Google Colab

1. Como Começar a Trabalhar no Google Colab?
2. Tipos Primitivos de Dados - Python.
3. Operadores Aritméticos - Python.
4. Operadores Relacionais Simples.
5. Operadores Relacionais Compostos.
6. Função print().
7. Função input().
8. Estruturas Condicionais.
9. Estruturas Condicionais Aninhadas e o elif;
10. Operadores Lógicos: and; or; not; Função range() e os operadores in; not in.
 - a. Operador and:
 - b. Operador or:
 - c. Operador not:
 - d. Função range():
 - e. Operador in; not in:

Exercícios e Desafios.

11. Um pouco de Álgebra Linear
 - a. Vetores
 - b. Matrizes
 - c. Filas
 - d. Pilhas
 - e. Árvore e Floresta
12. Bibliotecas básicas de análise de dados em Python
 - f. Pandas
 - g. Numpy
 - h. NLTK (Natural Language Toolkit)
 - i. Matplotlib

Bônus e Saiba mais



Resumo do eBook

Este eBook é um guia fundamental para aqueles que buscam adentrar no mundo da programação através da linguagem Python, conhecida pela sua clareza e ampla gama de aplicações. Inicialmente, somos apresentados à pergunta: **O que é Python?**, estabelecendo as bases do que é esta linguagem e porque ela se destaca como uma ferramenta poderosa e versátil tanto para iniciantes quanto para desenvolvedores experientes.

Com uma abordagem prática, o leitor é conduzido a **mão na massa** e aprende a configurar seu primeiro ambiente de desenvolvimento no Google Colab, uma plataforma que permite escrever e executar código Python diretamente no navegador.

A jornada de aprendizado prossegue com um mergulho nos **tipos primitivos de dados e operadores aritméticos** em Python, facilitando o entendimento dos blocos básicos de construção de qualquer programa. Além disso, explora-se os **operadores relacionais simples e compostos**, bem como as funções **print()** e **input()**, essenciais para a interação com o usuário.

O leitor avança para estruturas mais complexas, estudando as **estruturas condicionais e aninhadas**, e então explora os **operadores lógicos** (**_and_**, **_or_**, **_not_**). A funcionalidade da **função range()** e os operadores **in** e **not in** são expostos de forma clara, permitindo a compreensão plena de como manipular sequências de dados.

Após a teoria, vêm uma série de **exercícios e desafios** práticos para sedimentar o conhecimento adquirido. O salto para aplicações mais sofisticadas ocorre com um módulo de **álgebra linear**, abordando a manipulação de vetores e matrizes, assim como a apresentação de exemplos práticos desses conceitos em Python.

A parte final do eBook se dedica a introduzir as **bibliotecas básicas de análise de dados em Python**, como Pandas, Numpy, NLTK e Matplotlib, destacando a importância do Python no crescente campo da ciência de dados.

Como **bônus**, o material oferece recursos adicionais para quem deseja aprofundar seus estudos e expandir suas habilidades na linguagem Python e além, garantindo que o leitor se mantenha atualizado e apto a explorar todas as possibilidades que a programação em Python oferece.

O que abordaremos neste primeiro momento?

Serão abordados os conteúdos de introdução à Python, a definição de Python, e os conteúdos primordiais da linguagem.

- ❑ **Colab** (<http://colab.research.google.com>) ou **repl.it** <https://repl.it/> ou lápis e papel.
- ❑ Recomendo Python 3.7 ou superior).
- ❑ [App QPython3](#) (Disponível na Play Store).
- ❑ [Python Master](#) (Disponível na Play Store).

Neste artigo foi utilizado o **Google Colab**, todavia podem usar o [VS CODE](#), mas requer instalação e configuração na máquina local.

O que é Python?

Python é uma linguagem de programação de alto nível de abstração (comandos mais próximos da linguagem humana, do que a linguagem de máquina).



Python é uma linguagem dinâmica, interpretada, robusta, multi plataforma, multi-paradigma (orientação a objetos, funcional, reflexiva e imperativa.) Foi lançada em 1991 por Guido van Rossum, é uma linguagem livre (até para projetos comerciais), e hoje pode-se utilizá-la para programar desktops, web e mobile.

Mão na Massa

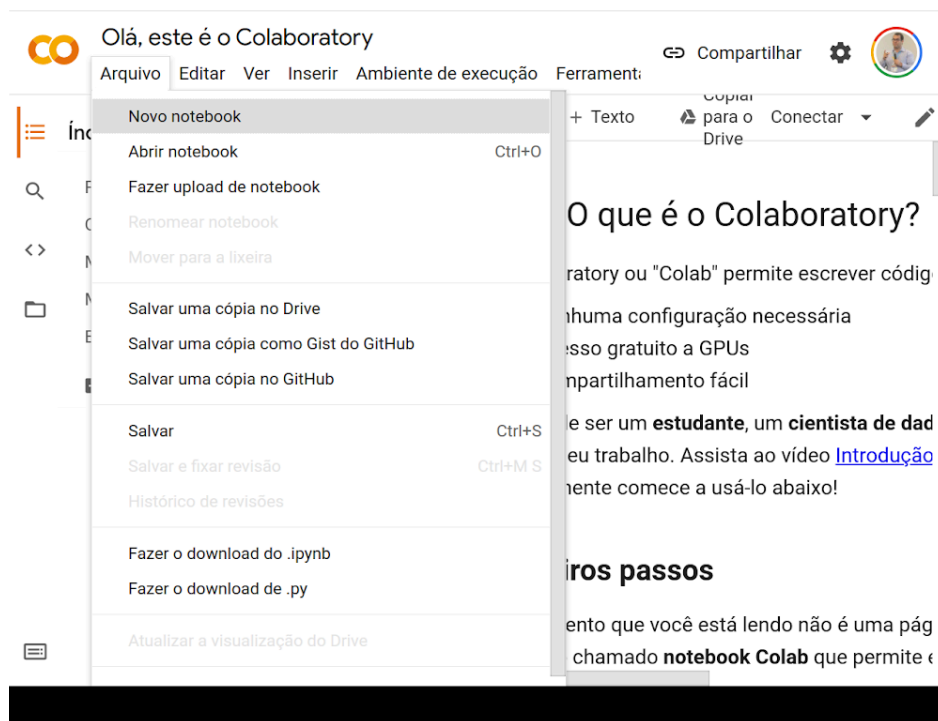
Criando seu Primeiro Código no Google Colab

Antes de qualquer coisa, você não precisa instalar nada na sua máquina, apenas ter um gmail pessoal válido/funcionando e vamos começar.

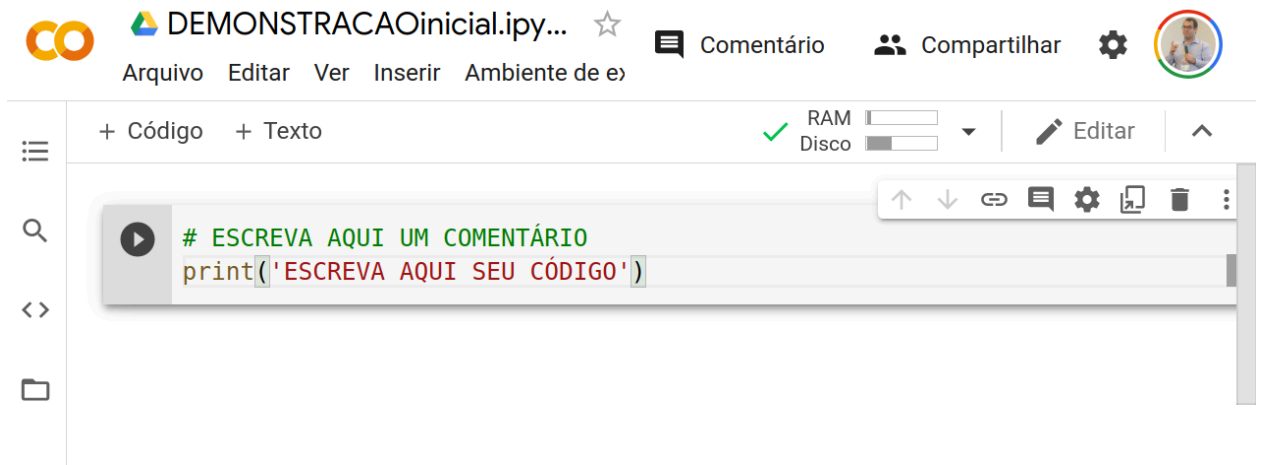
1. Como Começar a Trabalhar no Google Colab?

Usando o seu gmail pessoal, acesse o Google Colab, para criar seu ambiente virtual de desenvolvimento, siga os passos listados abaixo:

1 - No canto superior esquerdo temos o **botão Arquivo**, clique nele e logo em seguida **clique em Novo notebook**, o **Colab** irá carregar um novo arquivo/espço virtual, onde irá abrigar os seus códigos.



2 - Com o arquivo/espço virtual carregado, clique no campo / célula + Código ou + Texto caso queira escrever uma mensagem em linguagem natural.



AVISO: Não esqueça, o Colab salva seus arquivos online automaticamente no Google Drive do seu gmail cadastrado no início. Agora você já está pronto para escrever sua primeira linha de código, oficialmente, em Python =)

Observação: Para executar o seus códigos, clique no botão PLAY, no canto superior esquerdo.

2. Tipos Primitivos de Dados - Python.

No Python temos 4 tipos de dados, **Inteiros**(int), **Reais**(float), **Booleanos** - Verdadeiro/Falso (bool - True/False), e **Caracteres** (string).

| | |
|---|----------------|
| 1 | x = 2 |
| 2 | print(type(x)) |

Neste exemplo, atribuímos o valor de 2 a variável x e pedimos ao programa para que escreva na tela, usando o comando **print*** o tipo de dado da variável x.

O programa nos retorna o seguinte resultado : “<class 'int'>”, ou seja, a classe, ou o tipo de dado da variável x é inteiro (int).

- Comando *print*: **digite print* e, entre parênteses, você escreve o que o programa deve imprimir na tela, no caso do exemplo, o tipo de dado da variável x*, veremos com detalhes mais a frente.**
- Comando *type*: **digite type* e, entre parênteses, você escreve uma variável para saber o seu tipo, no caso do exemplo , está sendo requisitado o tipo de dado da variável x, e ele retorna int, ou inteiro.**

| | |
|---|------------------|
| 1 | x = 'Olá mundo!' |
| 2 | print(x) |
| 3 | print(type(x)) |

Neste segundo exemplo, nós atribuímos à variável x, a string “Olá mundo”. Para escrever uma string em python, uma **cadeia de caracteres**, a string sempre deve estar dentro de aspas, seja elas simples ou duplas, o recomendado é utilizar aspas simples.

Como resultado, o programa nos retorna o seguinte: **Olá mundo! , <class 'str'>.**

O programa imprimiu a string atribuída a x (Olá mundo!), na tela e nos retornou a classe dela, (str, ou string).

```
1 x = 2
2 y = 3
3 print(x == y)
```

Dados booleanos são um pouco mais complexos.

Entendendo o código acima:

Atribuímos à variável x o valor 2. atribuímos a y o valor de 3, e pedimos pra imprimir na tela “x == y”, ou seja, x igual a y. se x for igual a y ele imprime True, se x for diferente de y ele imprime False, como 2 é diferente de 3, ele imprime False. (dois sinais de igual juntos simbolizam igualdade, um único sinal de igual é utilizado somente em atribuição de valores, vamos entender isso melhor mais pra frente)

Dados booleanos são extremamente úteis e serão usados ao longo de toda sua vida como programador(a).

Dados do tipo float são feitos para trabalhar com números quebrados, com casas decimais, e com números negativos, a lógica é a mesma da dos números inteiros.

3. Operadores Aritméticos - Python

O Python pode ser utilizado como uma calculadora matemática avançada.

Praticamente, todos os operadores aritméticos funcionam da mesma forma como os conhecemos da matemática elementar. Por exemplo, para trabalharmos com as **4 principais funções matemáticas**, soma, subtração, multiplicação e divisão, temos os operadores conforme tabela a seguir.

| Operação | Operador |
|----------|----------|
| adição | + |

| | |
|---------------|---|
| subtração | - |
| multiplicação | * |
| divisão | / |

Temos também, operadores para exponenciação, divisão inteira, e resto de divisão.

| Operação | Operador |
|------------------------|-----------|
| exponenciação | ** |
| divisão inteira | // |
| resto da divisão | % |

| | |
|---|-----------------------|
| 1 | <code>x = 2**3</code> |
| 2 | <code>print(x)</code> |

A exponenciação é feita da seguinte forma : `base**expoente`, no exemplo, temos `2**3`, ou seja, 2 elevado a 3ª potência; o programa retorna o valor 8.

| | |
|---|------------------------|
| 1 | <code>x = 10//3</code> |
| 2 | <code>print(x)</code> |

A divisão inteira é feita pelo operador (//) e nos retorna a parte inteira de uma divisão, por exemplo, 10 dividido por 3 ? O resultado é 3,333333.... porém o programa só nos retorna a parte inteira da divisão, ou seja, 3.

| | |
|---|-----------|
| 1 | x = 10//3 |
| 2 | print(x) |

Para a operação de resto da divisão, podemos utilizar o mesmo código do exemplo anterior, abordamos a operação 10//3 == 3, essa operação tem o resto 1, na operação de resto de divisão nós pedimos ao programa que nos retorne justamente esse valor restante, sendo, 10 % 3 igual a 1.

4. Operadores Relacionais Simples.

Existem 3 relações possíveis entre 2 operandos, são elas:

| Descrição | Operador |
|-----------|----------|
| Maior que | > |
| Menor que | < |
| Igual a | == |

Para obtermos a relação entre 2 membros, temos que utilizar a seguinte estrutura:

<membro à esquerda> OPERADOR <membro à direita>

É importante observar que a inversão dos membros ocasiona na inversão do resultado da expressão, isto é, se o membro que estiver à esquerda for para a direita e vice-e-versa, a relação entre eles será o contrário.

5. Operadores Relacionais Compostos.

| Descrição | Operador |
|------------------|----------|
| Maior ou igual a | >= |
| Menor ou igual a | <= |
| Diferente de | != |

Maior ou igual a: Verifica se o valor A é maior ou igual ao valor B.

Menor ou igual a: Verifica se o valor A é menor ou igual ao valor B.

Diferente de: Verifica se o valor A é diferente do valor B.

Testem esse padrão, testando todos os operadores relacionais, prestando atenção no resultado, True ou False.

| | |
|---|-------------------------------|
| 1 | <code>a = 2</code> |
| 2 | <code>b = 3</code> |
| 3 | <code>print(a >= b)</code> |

6. Função print().

A função primordial, utilizada para simplesmente imprimir na tela, a função print() imprime variáveis, valores, strings, e todos juntos também, na sua tela. vamos para o exemplo prático para entender melhor.

| | |
|---|---|
| 1 | <code>print('Olá mundo!, 2, 2.5, True, False')</code> |
|---|---|

A função print, no exemplo, está imprimindo a string “Olá mundo!” strings sempre entre aspas, o número 2, o número 2.5, e os valores True e False.

Para imprimir diferentes tipos de dados dentro de um único print, eles têm sempre que estar separados por vírgula.

Podemos também formatar uma string para receber valores dela, para isso, simplesmente coloque a letra “f”, antes de escrevê-la, e digite o valor que deverá ser incluído na função entre chaves.

| | |
|---|---|
| 1 | <code>numero = 2</code> |
| 2 | <code>print(f'O número {numero} é par.')</code> |

Entendendo este código:

Atribuímos o valor 2 a variável numero, na linha seguinte, chamamos a função print, formatada (letra f antes da string), Lembre-se que, para incluir valores externos dentro de uma string, ela precisa estar entre chaves, como no exemplo, a variável número está entre chaves. O programa nos retorna o seguinte: **O número 2 é par.** , o programa SEMPRE vai “colocar” o valor que está dentro da variável no espaço formatado.

7. Função input().

Nem sempre vamos trabalhar somente com valores já determinados, às vezes, precisamos pedir ao usuário que digite um valor, para aí sim, podemos trabalhar em cima dele, A função input serve justamente para isso, ela pede ao usuário que digite um valor, e o guarda dentro de uma variável, como no exemplo abaixo.



| | |
|---|-----------------------------------|
| 1 | nome = input('Digite seu nome: ') |
| 2 | print(f'Seu nome é {nome}.') |

***OBS: Sempre digite os exemplos no seu Colab, para praticar e se habituar com a linguagem.**

Bom, no exemplo acima, **nós pedimos ao usuário que digite seu nome**, e em seguida nós o imprimimos, Seu nome é {nome digitado pelo usuário}.

OBS*: A função input pode ser modificada de acordo com suas necessidades, podendo ser convertida para inteiro e para float(número real).

Observe que, se, por exemplo você precisar saber a idade do usuário e escrever desse mesmo jeito:

| | |
|---|-------------------------------------|
| 1 | idade = input('Digite sua idade: ') |
| 2 | print(f'Você tem {idade} anos.') |

Teste esse código, funcionou normalmente não é ?

Não, não funcionou corretamente, e eu vou te dizer o porquê no exemplo abaixo.

| | |
|---|-------------------------------------|
| 1 | idade = input('Digite sua idade: ') |
| 2 | print(f'Você tem {idade} anos.') |
| 3 | print(type(idade)) |

Ao pedir a idade do usuário, um número inteiro, ele imprime tudo normalmente, mas no momento que você pede o tipo de dado da variável idade, ele nos retorna : **<class 'str'>**, “ué, mas como, se o numero 23 é um número inteiro, por que ele nos retornou tipo string ? “

Veja bem, a função input(), ela vem pré setada para receber strings, lembre-se que, strings são cadeias de caracteres, sejam eles números, ou símbolos ou qualquer outra coisa.

Para receber números e tê-los como do tipo inteiro, precisamos **converter o input para inteiro**.

Para converter a função input() para inteiro, apenas digite:

`int(input('Mensagem dizendo o que deve ser informado pelo usuário'))`

```
1 idade = int(input('Digite sua idade: '))
2 print(f'Você tem {idade} anos.')
3 print(type(idade))
```

Fazendo essa alteração, pedindo o type, ou o tipo de dado da variável idade, o programa nos retorna: **<class 'int'>**, e assim que fizermos essa alteração, ele receberá somente números inteiros, qualquer outra coisa será tratada como erro.

Mesma coisa para valores float (reais), conforme exemplo:

```
1 peso = float(input('Digite seu peso: '))
2 print(f'Você pesa {peso}KGs')
3 print(type(peso))
```

O programa retorna: **<class 'float'>**.

8. Estruturas Condicionais.

Muitas vezes, precisamos estabelecer certas “condições” dentro do código, se a condição for cumprida, executamos x funcionalidade, e caso não, executamos outras estruturas condicionais que servem para isso.

Se condição == verdadeira:

bloco de código para caso condição == verdadeiro

se não:

bloco de código para caso condição == falso

No python, o se e o senão são representadas, pelas funções:

if e else;

As funções, faça e então, do visualg, não existem em python, eles são substituídos pelos dois pontos ":" e pela indentação. Em Python, a indentação possui função bastante especial, até porque, os blocos de instrução são delimitados pela profundidade da indentação, isto é, os códigos que estiverem rente a margem esquerda, farão parte do primeiro nível hierárquico. Já, os códigos que estiverem a 4 espaços da margem esquerda, estarão no segundo nível hierárquico e aqueles que estiverem a 8 espaços, estarão no terceiro nível e assim por diante.

***PARA CRIAR UMA INDENTAÇÃO EM PYTHON, UTILIZE A TECLA "TAB" DO SEU TECLADO, E ELE CRIARÁ A INDENTAÇÃO CORRETA (DE QUATRO ESPAÇOS), AUTOMATICAMENTE.**

Todos os blocos são delimitados pela profundidade da indentação e por isso, a sua importância, é vital para um programa em Python. O mau uso, isto é, utilizar 4 espaçamentos enquanto deveríamos estar utilizando 8, acarretará na não execução, ou então, no mal funcionamento em geral.

Para entendermos melhor vamos utilizar um exemplo juntando todo o conhecimento adquirido até agora.

```
1 num = int(input('Digite um número: '))
2 if num % 2 == 0:
3     print(f'O número {num} é par')
4 else:
5     print(f'O número {num} é ímpar')
```

Perceba que os códigos `print(f'o número {num} é par')` e, `print(f'o número {num} é ímpar')` estão deslocados mais a direita. eles estão no chamado **segundo nível hierárquico**, eles só serão executados caso a condição imposta seja cumprida.

Os códigos colados na margem à esquerda são os que estão no **primeiro nível hierárquico**, e serão executados sempre.

Entendendo o código:

```
1 num = int(input('Digite um número: '))
2 if num % 2 == 0:
3     print(f'O número {num} é par')
4 else:
5     print(f'O número {num} é ímpar')
```

Primeiro, pedimos ao usuário que digite um número, e depois a seguinte lógica é feita:

Se o resto da divisão do número por 2 for igual a zero:

imprime(o número {num} é par)

se não:

imprime(o número {num} é ímpar)

Todos concordamos que um número é PAR, quando o resto da sua divisão por 2 é 0, ou seja, é uma divisão exata. Caso tenha algum resto ele é considerado ímpar.

9. Estruturas Condicionais Aninhadas e o elif;

Para estabelecer mais de uma condição, precisamos utilizar o comando **elif**, que pode ser utilizado múltiplas vezes para estabelecer múltiplas condições diferentes:

```
1 num = float(input('Digite um número: '))
2
3 if num > 0:
4     print('Este número é positivo')
5
6 elif num == 0:
7     print('Este número é neutro')
8
9
10 else:
11     print('Este número é negativo')
```


Perceba que foi adicionada mais uma condição, para caso o número seja igual a zero, isso pode ser feito diversas vezes, as estruturas condicionais sempre seguem esse padrão **if - else, ou if - elif - else**; nunca se deve começar com else, ou com elif, sempre siga esse padrão.

Ao final de cada condição, use os dois pontos “:”, *enter* para avançar para a próxima linha, e automaticamente, criar a indentação.

10. Operadores Lógicos: and; or; not; Função range() e os operadores in; not in.

a. Operador and:

O “e” é o operador and, em Python. Seja o seguinte teste:

teste condição1 and condição2:

O programa só retornará **True** se as DUAS condições forem cumpridas, caso somente uma seja cumprida, ele retornará **False**

| | |
|---|---|
| 1 | resposta = int(input('Qual sua idade: ')) |
| 2 | if resposta>=18 and resposta <=65: |
| 3 | print('Você é obrigado a votar!') |
| 4 | else: |
| 5 | print('Você não é obrigado a votar!') |

Veja que para o IF ter resultado verdadeiro, ambas as condições devem ser verdadeiras: tanto deve ter 18 anos ou mais E deve ter 65 anos ou menos, Essa é a característica do operador and.

Todos os testes devem ser true, para o resultado ser true.

Se um deles for *false*, o resultado é falso.

b. Operador or:

O operador or é parecido, mas completamente diferente do operador and, ele usa a mesma forma de digitação ao ser aplicado.

O or trabalha com diversas possibilidades, por exemplo, para executar um programa ele precisa que as seguintes condições sejam cumpridas(usaremos letras do alfabeto para representar as condições): **a OU b OU c OU d**: no caso do **and** ele precisa que todos sejam verdade, no operador **or** ele precisa que somente uma das condições seja verdade para que ele retorne verdade. Ficou meio confuso ? Veja o exemplo:

```
1 print('1. Idoso')
2 print('2. Gestante')
3 print('3. Cadeirante')
4 print('4. Nenhum destes')
5 resposta=int( input('Você é: ') )
6
7 if (resposta==1) or (resposta==2) or (resposta==3) :
8     print('Você tem direito a fila prioritária')
9 else:
10    print('Você não tem direito a nada. Vá pra fila e fique quieto')
```

Se qualquer um deles for verdade, esse **if** vai ser verdade, pois usamos o operador lógico **or** (OU). Pra ser prioridade, OU você tem que ser idoso, OU tem que ser gestante OU tem que ser cadeirante.

Não precisa ser os três, basta um deles ser verdade, que o teste retorna verdade.

***OBS: Pratique, a prática é primordial para entender os conceitos, apenas ler sobre não fará você entender todos os assuntos de primeira, se for o caso, digite o código exemplo no seu PyCharm apenas para ver como ele funciona **NÃO COPIE E COLE**.**

c. Operador not:

O operador **not** é o mais simples de todos, ele pega uma expressão e a inverte, caso ela seja verdade, ela será falsa e vice versa

```
1 a = 4
2 b = 2
3 print(not a > b)
```

Como sabemos, 4 é maior que 2, então a expressão deveria retornar **True**, Porém como usamos o **not** antes da expressão ele inverteu o seu resultado. teremos **False** como resultado.

um exemplo mais complexo utilizando **not**;

```
1  banda = input('Qual melhor banda do mundo? ')
2
3  if not banda=='DT':
4      print('Errado!')
5  else:
6      print('Correto, é DT')
```

d. Função range():

Como funciona a função range do Python? A função "range()" retorna uma lista no intervalo definido. Assim, temos uma forma rápida e fácil para gerarmos listas de valores numéricos que estejam num determinado intervalo, por exemplo,

range(início, fim, passo)

por exemplo, **range(1, 100, 2)**, ele cria uma **lista** todos os números num intervalo de 1 até 100 pulando de 2 em 2, (1, 3, 5, 7...) Caso queira um passo normal, escreva somente o início e o passo

range(1, 7): representa todos os números num intervalo de 1 até 6, “ué, não é entre 1 e 7 ? tem o 7 ali ó”, **não**, é uma regrinha da sintaxe do Python, ele ignora o último número, e só representa até o penúltimo.

o range é extremamente útil naqueles momentos, você tem que ter todos os números de 1 até 1000 na tela, você vai fazer, imprimir um por um?, logicamente não, você utilizará a função range, ela é muito utilizada em conjunto com o loop for, que veremos em outro momento.

e. Operador in; not in:

Lembra da função range() ? ela será também importante para esses dois aqui, os operadores “in” e “not in” representam exatamente “dentro de” “não está dentro de”, são usados com a função range(), por exemplo:

```
1  a = int(input('Digite um número: '))
2  if a in range(1, 300):
3      print(f'{a} está entre 1 e 300.')
4  else:
```

5

```
print(f'{a} não está entre 1 e 300.')
```

“a” recebe 3, se a estiver num intervalo entre 1 e 300: escreva : “a está entre 1 e 300”, se não escreva “Não está entre 1 e 300”, como a é 3, e 3 está entre 1 e 300 ele retorna True, e imprime o bloco dentro do if.

O not in é exatamente o Inverso, teste o código acima, substituindo o **in** por **not in** e veja o resultado.

A lógica, substituindo o **in** por **not in** é:

“a” recebe 3, se a **não estiver** num intervalo entre 1 e 300: escreva : “a está entre 1 e 300”, se não escreva “false”, **como a está num intervalo entre 1 e 300**, ele retornará *false* e escreverá o bloco dentro do *else*.



Exercícios e Desafios

1. Faça um CÓDIGO que mostre a mensagem "Olá Mundo!" na tela.
2. Faça um CÓDIGO que peça um número e então mostre a mensagem O número informado foi [número].
3. Faça um CÓDIGO que peça dois números e imprima a soma.
4. Faça um CÓDIGO que peça as 4 notas bimestrais e mostre a média final.
5. Faça um CÓDIGO que converta metros para centímetros.
6. Faça um CÓDIGO que peça o raio de um círculo, calcule e mostre sua área.
7. Faça um CÓDIGO que calcule a área de um quadrado, em seguida mostre o dobro desta área para o usuário.
8. Faça um CÓDIGO que pergunte quanto você ganha por hora e o número de horas trabalhadas no mês. Calcule e mostre o total do seu salário no referido mês.
9. Faça um CÓDIGO que peça um valor e mostre na tela se o valor é positivo ou negativo.
10. Faça um CÓDIGO que verifique se uma letra digitada é "F" ou "M". Conforme a letra escrever: F - Feminino, M - Masculino, Sexo Inválido.

11. Um pouco de Álgebra Linear

“Existe algo mais inútil ou menos útil que Álgebra?” by Billy Connolly

A Álgebra Linear é um ramo da matemática que lida com **espaços vetoriais**. Apesar de achar que neste tópico não vou conseguir ensinar álgebra linear de forma plena, ela sustenta um grande número de conceitos e técnicas de data science, o que significa que eu devo a você, ao menos, uma tentativa. O que aprenderemos neste tópico, usaremos excessivamente no decorrer do ano.

a. Vetores

Abstratamente, os vetores são objetos que podem ser somados juntos (para formar vetores novos) e que podem ser multiplicados pelos *escalares* (por exemplo, números), também para formar vetores novos.

Concretamente (para nós), os vetores são pontos em algum espaço de dimensão finita. Apesar de você não pensar em seus dados como vetores, eles são uma ótima maneira de representar dados numéricos.

Por exemplo, se você tiver as alturas, pesos e idades de uma grande quantidade de pessoas, pode tratar seus dados como vetores tridimensionais (altura, peso, idade). Se você estiver ensinando uma turma com quatro testes, pode tratar as notas dos estudantes como vetores quadridimensionais (teste1, teste2, teste3, teste4).

A abordagem inicial mais simples é representar vetores como listas de números. Uma lista de três números corresponde a um vetor em um espaço tridimensional, e vice-versa, observe:

| | |
|---|-----------------------------------|
| 1 | altura_peso_idade = [181, #altura |
| 2 | 100, #peso |
| 3 | 40] #idade |
| 4 | print(altura_peso_idade) |
| 1 | notas = [10, #teste1 |
| 2 | 9, #teste2 |

```
3     8, #teste3
4     7 ] #teste4
5 print(notas)
```

Um problema com essa abordagem é que queremos realizar aritmética nos vetores. Como as listas de Python não são vetores (e, portanto, não facilita a aritmética com o vetor), precisaremos construir essas ferramentas aritméticas nós mesmos. Então, vamos começar por aí.

Para iniciar, frequentemente precisamos de dois vetores. Os vetores se adicionam componente a componente. Isso significa que, se dois vetores *vetA* e *vetB* possuem o mesmo tamanho, sua soma é somente o vetor cujo primeiro elemento seja *vetA*[0] + *vetB*[0], cujo o segundo elemento seja *vetA*[1] + *vetB*[1], e assim por diante.

ATENÇÃO:

"Se eles não possuírem o mesmo tamanho, então não poderemos somá-los."

by Joel Grus em Data Science do Zero.

Por exemplo, somar os vetores [1, 2] e [2, 1] resulta em [1 + 2, 2 + 1] ou [3, 3].

Podemos facilmente implementar isso com vetores *zip* juntos e usar uma compreensão de lista para adicionar os elementos correspondentes:

```
1 def vet_add(vet1, vet2):
2     #soma elementos correspondentes
3     return [vet1_i + vet2_i
4             for vet1_i, vet2_i in zip(vet1, vet2)]
```

Da mesma forma, para subtrair dois vetores, apenas subtraia os elementos correspondentes:

```
1 def vet_sub(vet1, vet2):
```

```

2      #subtrai elementos correspondentes
3      return [vet1_i - vet2_i
4              for vet1_i, vet2_i in zip(vet1, vet2)]

```

Às vezes queremos somar uma lista de vetores, ou seja, criar um vetor novo cujo primeiro elemento seja a soma de todos os primeiros elementos, cujo segundo elemento seja a soma de todos os segundos elementos, e assim sucessivamente. A maneira mais fácil de fazer isso é adicionar um vetor de cada vez:

```

1      def vet_sum(vets):
2          # soma todos os elementos correspondentes
3          result = vets[0] # começa com o primeiro vetor
4          for vet in vets[1:]: # depois passa por todos os outros
5              result = vet_add(result, vet) # e os adiciona ao resultado
6          return result

```

Disponível em: <https://github.com/clovesrocha/ETEPDLPC/blob/master/sumvetselemcorresp.ipynb>

Se você pensar a respeito, estamos apenas reduzindo (reducing) a lista de vetores usando `vet_add`, o que significa que podemos reescrever de forma reduzida usando funções de alta ordem:

```

1      def vet_sum(vets):
2          return reduz(vet_add, vets)

```

ou até mesmo:

```

1      vet_sum = part(reduz, vet_add)

```


embora esse último seja mais esperto do que útil.

Também precisamos ser capazes de multiplicar um vetor por um escalar, que simplesmente fazemos ao multiplicar cada elemento do vetor por aquele número:

```
1 def scalar_mult(c, v):  
2     #c é um número, v é um vetor  
3     return [c * v_i for v_i in v]
```

b. Matrizes

Primeiro, você precisará instalar o NumPy se ainda não o tiver instalado. Você pode instalá-lo usando o pip:

Em desenvolvimento... no repo.:

<https://github.com/clovesrocha/MiniCursoIP/blob/master/matrizes.ipynb>

```
import numpy as np  
  
# Criando uma matriz  
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
  
# Acessando elementos da matriz  
elemento = matriz[0, 1] # Obtém o elemento na linha 0 e coluna 1 (valor 2)  
  
# Obtendo o número de linhas e colunas  
num_linhas = matriz.shape[0] # Retorna o número de linhas (3)  
num_colunas = matriz.shape[1] # Retorna o número de colunas (3)  
  
# Iterando sobre elementos da matriz  
for linha in matriz:  
    for elemento in linha:
```

| | |
|--|---|
| | <pre> print(elemento) # Operações matriciais matriz1 = np.array([[1, 2], [3, 4]]) matriz2 = np.array([[5, 6], [7, 8]]) soma = matriz1 + matriz2 # Soma de matrizes produto = np.dot(matriz1, matriz2) # Produto de matrizes # Transposição de matriz matriz_transposta = matriz.T # Indexação booleana matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) condicao = matriz > 5 # Cria uma matriz de booleanos baseada na condição elementos_maiores_que_5 = matriz[condicao] # Obtém elementos maiores que 5 # Outras operações úteis incluem: média, mínimo, máximo, etc. media = np.mean(matriz) valor_minimo = np.min(matriz) valor_maximo = np.max(matriz) print(matriz) </pre> |
|--|---|

c. Filas

| | |
|---|---|
| 1 | <code>fila = [100, 101, 102, 103, 104]</code> |
| 2 | <code>fila.append(105) # insere um elemento no final da fila</code> |
| 3 | <code>print(fila)</code> |

```

1 # Parte 2 com class
2 class Fila(object):
3     def __init__(self):
4         self.dados = []
5
6     def insere(self, elemento):
7         self.dados.append(elemento)
8
9     def retira(self):
10        return self.dados.pop(0)
11
12    def vazia(self):
13        return len(self.dados) == 0

```

d. Pilhas

- Exemplo de Pilhas no Repositório:
<https://github.com/clovesrocha/MiniCursoIP/blob/master/pilha.ipynb>
- Aproveite e conheça outros conteúdos lá.

e. Árvore e Floresta

```

1 ``python by Prof. Cloves Rocha - https://github.com/clovesrocha
2 class Node:
3     """
4     A Node class to represent the individual elements of the tree.
5     """
6     def __init__(self, key):
7         self.key = key          # The value or 'key' of the node
8         self.children = []      # A list of children nodes
9
10    class Tree:
11        """
12

```

13

```

The Tree class with a root node.
"""

def __init__(self, root_key):
    self.root = Node(root_key) # The root node of the tree

def add_child(self, parent_key, child_key):
    """
    A method to add a child node to the tree under the parent
node with
    the given parent_key.
    """
    # Recursive function to find the parent node and add the
child
    def _add_child_recursive(node, parent_key, child_key):
        if node.key == parent_key:
            # Parent found, add new child
            node.children.append(Node(child_key))
            return True
        for child in node.children:
            # Search in the children
            if _add_child_recursive(child, parent_key,
child_key):
                return True
        return False

    # Start the search from the root node
    _add_child_recursive(self.root, parent_key, child_key)

class Forest:
    """
    The Forest class containing multiple trees.
    """
    def __init__(self):

```

```

        self.trees = [] # A list to store multiple trees

    def add_tree(self, tree_root_key):
        """
        A method to add a new tree to the forest with the specified
        root key.
        """
        new_tree = Tree(tree_root_key)
        self.trees.append(new_tree)

    def add_child(self, tree_root_key, parent_key, child_key):
        """
        A method to add a child node to a tree identified by the
        tree_root_key.
        """
        # Find the tree with the given root and add a child
        for tree in self.trees:
            if tree.root.key == tree_root_key:
                tree.add_child(parent_key, child_key)
                return

# Example usage of Tree and Forest classes:
if __name__ == '__main__':
    # Create a Tree
    tree = Tree('root')
    tree.add_child('root', 'child1')
    tree.add_child('root', 'child2')
    tree.add_child('child1', 'child1_1')
    tree.add_child('child1', 'child1_2')

    # Create a Forest and add trees to it
    forest = Forest()
    forest.add_tree('tree1_root')

```

| | |
|--|--|
| | <pre> forest.add_tree('tree2_root') forest.add_child('tree1_root', 'tree1_root', 'tree1_child1') forest.add_child('tree2_root', 'tree2_root', 'tree2_child1') ... </pre> <p>Mantenha a calma e tome café...</p> <p>A classe `Node` representa nós individuais em uma árvore, cada um com uma chave e uma lista de filhos.</p> <p>A classe `Tree` representa uma única árvore com um método para adicionar nós filhos.</p> <p>A classe `Forest` representa uma coleção de árvores e permite adicionar novas árvores e nós dentro dessas árvores.</p> <p>O exemplo no final mostra como usar essas classes para criar árvores e uma floresta.</p> <p>Um abraço by Prof. Cloves Rocha - https://github.com/clovesrocha</p> |
| | |

12. Bibliotecas básicas python

a. PANDAS

Em programação de computadores, pandas é uma biblioteca de software criada para a **linguagem Python** para **manipulação e análise de dados**. Em particular, oferece estruturas e operações para manipular tabelas numéricas e séries temporais.

Importando o Pandas por convenção

Aqui vai uma informação que talvez você já conheça: sempre que importar o Pandas, utilize a regra de convenção. Isso faz com que pessoas que lerem seu código no futuro — incluindo

você mesmo — possam identificar a biblioteca mais facilmente. Por regra, Pandas deve ser importado sob o nome de pd, assim:

```
import pandas as pd
```

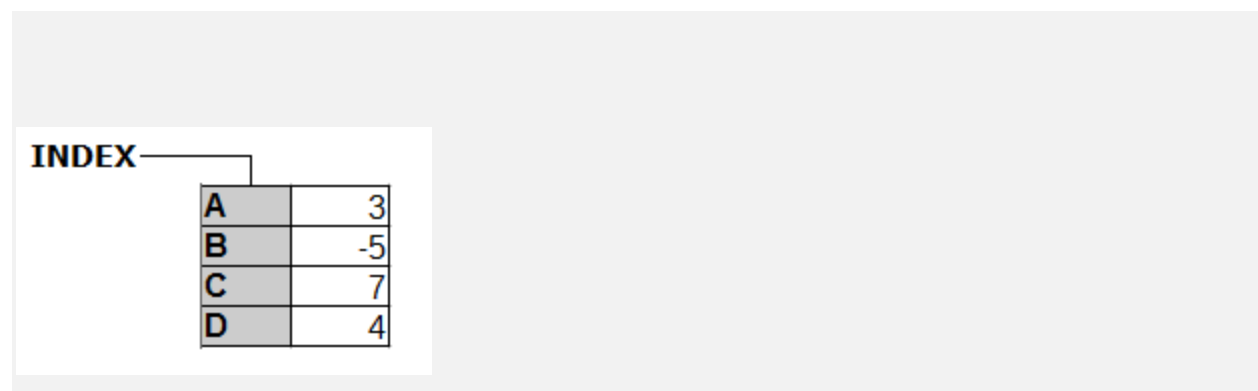
Series e DataFrame

Posso estar sendo meio óbvio falando sobre Series e DataFrame para alguém que já está acostumado a usar o Pandas, mas quero deixar claro para aqueles que estão começando a principal diferença entre esses dois tipos de Estrutura de Dados.

- **Series nada mais é que um array (JS)/vetor(Python) de 1 dimensão.** Você pode

considerar um Series também como uma coluna de uma tabela. Exemplo:

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```



The diagram illustrates a Pandas Series as a single column within a table structure. On the left, the word "INDEX" is written in bold, with a line pointing to the first row of a table. The table has two columns: the first column contains categorical labels 'A', 'B', 'C', and 'D', and the second column contains numerical values 3, -5, 7, and 4. The first column is shaded gray, and the second column is white.

| INDEX | |
|-------|----|
| A | 3 |
| B | -5 |
| C | 7 |
| D | 4 |

Saída do código acima: um array de valores indexados

- Um DataFrame é simplesmente um conjunto de Series. **Trata-se de uma estrutura**

de dados de 2 dimensões — colunas e linhas — que transforma os dados em uma bela tabela. Exemplo:

#Criando um dicionário onde cada chave será uma coluna do DataFrame >>>

```
data = {  
    'País': ['Bélgica', 'Índia', 'Brasil'],  
    'Capital': ['Bruxelas', 'Nova Delhi', 'Brasília'],  
    'População': [123465, 456789, 987654]  
}
```


#Criando o DataFrame

```
>>> df = pd.DataFrame(data, columns=['País','Capital','População'])
```



A diagram showing the relationship between the INDEX and the DataFrame table. A line connects the word "INDEX" to the first column of the table, which contains the row numbers 1, 2, and 3.

| | País | Capital | População |
|---|---------|------------|-----------|
| 1 | Bélgica | Bruxelas | 123465 |
| 2 | Índia | Nova Delhi | 456789 |
| 3 | Brasil | Brasília | 987654 |



Saída do código: um DataFrame

- Abrindo e escrevendo arquivos CSV:

#Para ler arquivos CSV codificados em ISO

```
>>> pd.read_csv('nome_do_arquivo.csv', encoding='ISO-8859-1')
```

#Para escrever arquivos CSV

```
>>> pd.to_csv('nome_do_arquivo_para_salvar.csv')
```

- Abrindo arquivos de Excel:

```
>>> xlsx = pd.ExcelFile('seu_arquivo_excel.xlsx')
```

```
>>> df = pd.read_excel(xlsx, 'Planilha 1')
```

- Removendo linhas e colunas:

#Removendo linhas pelo index

```
s.drop([0, 1])
```

#Removendo colunas utilizando o argumento axis=1

```
df.drop('País', axis=1)
```

- Coletando informações básicas sobre o DataFrame:

#Quantidade de linhas e colunas do DataFrame

```
>>> df.shape
```

#Descrição do Index


```
>>> df.index
```

#Colunas presentes no DataFrame

```
>>> df.columns
```

#Contagem de dados não-nulos

```
>>> df.count()
```

- 
- Criando uma nova coluna em um DataFrame:

```
>>> df['Nova Coluna'] = 0
```

- Renomeando colunas de um DataFrame:

#Se seu DataFrame possui 3 colunas, passe 3 novos valores em uma lista

```
df.columns = ['Coluna 1', 'Coluna 2', 'Coluna 3']
```

- Resumo dos dados:

#Soma dos valores de um DataFrame

```
>>> df.sum()
```

#Menor valor de um DataFrame

```
>>> df.min()
```

#Maior valor



```
>>> df.max()
```

```
#Index do menor valor
```

```
>>> df.idmin()
```

```
#Index do maior valor
```

```
>>> df.idmax()
```

```
#Resumo estatístico do DataFrame, com quartis, mediana, etc.
```

```
>>> df.describe()
```

```
#Média dos valores
```

```
>>> df.mean()
```

```
#Mediana dos valores
```

```
>>> df.median()
```

- Aplicando funções:

```
#Aplicando uma função que substitui a por b
```

```
df.apply(lambda x: x.replace('a', 'b'))
```

- Ordenando valores:

#Ordenando em ordem crescente

```
df.sort_values()
```

#Ordenando em ordem decrescente

```
df.sort_values(ascending=False)
```

- Operações aritméticas em Series:

```
>>> s = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
```

#Somando todos os valores presentes na Series por 2

```
>>> s.add(2)
```

#Subtraindo 2 de todos os valores

```
>>> s.sub(2)
```

#Multiplicando todos os valores por 2

```
>>> s.mul(2)
```

#Dividindo valores por 2

```
>>> s.div(2)
```

- Indexação por Boolean:

#Filtrando o DataFrame para mostrar apenas valores pares

```
df[df['População'] % 2 == 0]
```

- Selecionando valores:

#Selecionando a primeira linha da coluna país


```
df.loc[0, 'País']
```

b. NUMPY

O NumPy é uma poderosa biblioteca Python que é usada principalmente para realizar cálculos em Arrays/Vetores Multidimensionais. O NumPy fornece um grande conjunto de funções e operações de biblioteca que ajudam os programadores a executar facilmente cálculos numéricos.

Bônus

Instalação da biblioteca NumPy



O caminho mais rápido e mais fácil de instalar a biblioteca NumPy em sua máquina é usar o seguinte comando:

pip install numpy

Isso instalará a versão mais recente e mais estável do NumPy em sua máquina. Instalar através do pip é a maneira mais simples de instalar qualquer pacote Python.

NumPy é:

- um pacote de extensão do Python para matrizes multi-dimensionais;
- próximo ao hardware (eficiência);
- desenvolvido para computação científica (conveniência).

```
import numpy as np
```

```
a = np.array([0, 1, 2, 3])  
print(a)
```

Por exemplo:

Uma matriz contendo:

- valores de um experimento/simulação em tempos discretos;
- sinal gravado por um equipamento de medição, por exemplo, ondas sonoras;
- pixels de uma imagem, escala de cinza ou coloridos;
- dados tridimensionais medidos em posições X,Y,Z diferentes, por exemplo, MRI scan;

- entre outros...

Por que é muito útil: Recipiente eficiente na questão da memória que provê operações numéricas rápidas.

```
import numpy as np
import time

L = range(1000000)
[i**2 for i in L]
print(time.time() - a)
a = time.time()
b = np.arange(1000000)
b**2
print(time.time() - a)
```

Ajuda interativa:

```
help(np.array)
```

Procura por algo em específico:

```
np.lookfor('create array')
```

Criando arrays

Unidimensionais:

```
import numpy as np
```



```
a = np.array([0, 1, 2, 3])
a
a.ndim
a.shape
len(a)
```

Bidimensionais:

```
b = np.array([[0, 1, 2], [3, 4, 5]]) # 2 x 3 array
b
b.ndim
b.shape
len(b)
```

Tridimensionais:

```
c = np.array([[[1], [2]], [[3], [4]]])
c
c.shape
```

Na prática, raramente se entra com os itens um por um:

Espaçados igualmente:

```
import numpy as np

a = np.arange(10) # 0 .. n-1
a
b = np.arange(1, 9, 2) # início, fim (exclusive), passo
b
```

Por número de pontos:

```
c = np.linspace(0, 1, 6) # início, fim, número de pontos
```

```
c
d = np.linspace(0, 1, 5, endpoint=False)
d
```

Matrizes comuns:

```
a = np.ones((3, 3)) # lembrete: (3, 3) é uma tupla
a
b = np.zeros((2, 2))
b
c = np.eye(3)
c
d = np.diag(np.array([1, 2, 3, 4]))
d
```

Números aleatórios:

```
a = np.random.rand(4)    # uniforme em [0, 1]
a
b = np.random.randn(4)   # Gaussiana
b
np.random.seed(1234)     # Definindo um início
```

Tipos básicos de dados


Você deve ter notado que, em alguns casos, os elementos de matriz são exibidos com um ponto final (por exemplo, 2. contra 2). Isto é devido a uma diferença no tipo de dados utilizada:

```
a = np.array([1, 2, 3])

a.dtype

b = np.array([1., 2., 3.])

b.dtype
```



Diferentes tipos de dados nos permitem armazenar dados de forma mais compacta na memória, mas na maioria das vezes simplesmente trabalhamos com números de ponto flutuante. Note que, no exemplo acima, o NumPy detecta automaticamente o tipo de dados a partir da entrada.

Você pode especificar explicitamente o tipo de dado que você deseja:

```
c = np.array([1, 2, 3], dtype=float)
```

```
c.dtype
```

O tipo de dado padrão é o ponto flutuante:

```
a = np.ones((3, 3))
```

```
a.dtype
```

Também existem outros tipos:

Complexo:

```
d = np.array([1+2j, 3+4j, 5+6*1j])
```

```
d.dtype
```

Booleano:

```
e = np.array([True, False, False, True])
```

```
e.dtype
```



Strings:

```
f = np.array(['Bonjour', 'Hello', 'Olá',])  
f.dtype
```

Natural Language Toolkit (NLTK):

Pode ser empregada em projetos de linguística computacional e outros campos científicos, auxiliando **cientistas de dados a desenvolverem projetos**, de maneira que os computadores passem a entender a linguagem natural humana.

A biblioteca inclui diversos recursos que ajudam você a executar diversas tarefas de NLP, dentre elas:

- Realizar tarefas básicas de Tokenização N-Gramas, Stemming, Tagging, Part of Speech (para obter as classes gramaticais das palavras), Wrappers e Análise & Raciocínio Semântico e muito mais;
- Classificar, marcar e filtrar termos pela base;
- Analisar a sintaxe e o sentido semântico de palavras;
- Reconhecer de Entidades Nomeadas;
- Encontrar Similaridade de textos;
- Exibir Treebanks (isto é, árvores de análise que são como diagramas de sentenças que revelam partes da fala e dependências).

A biblioteca ainda possui mais de cinquenta corpora e recursos lexicais! Dentre eles: WordNet, Web Text Corpus, NPS Chat, SemCor, FrameNet, SentiWordNet e muitos mais! Adicionalmente, possui um Stopwords Corpus, disponível em diversos idiomas.

Matplotlib:

Matplotlib é uma biblioteca de software para criação de gráficos e visualizações de dados em geral, feita para e da linguagem de programação Python e sua extensão de matemática NumPy.

Dúvidas?

Keep calm and drink coffee...

Obrigado! **Thank you!**

Saiba mais em:

<https://linktr.ee/clovesrocha>



Digitalize-me!