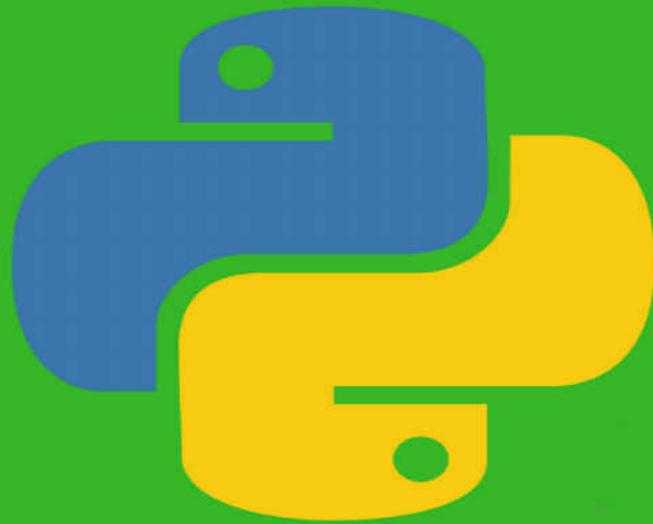


SÉRIE CIENTISTA DE DADOS

ÁLGEBRA LINEAR COM PYTHON

Aprenda na prática os principais
conceitos



RAFAEL FVC SANTOS

Primeira edição

Álgebra Linear com Python

Aprenda na prática os principais conceitos

Primeira edição, 2018.

Autor: Rafael F. V. C. Santos (rafaelfvcs@gmail.com)

Especialista em gestão estratégica de riscos aplicados ao mercado financeiro. Trabalha com o desenvolvimento de estratégias automatizadas de investimentos (Robôs de investimentos - Expert Advisor) utilizando *machine learning* e estatística espacial. Formado em Engenharia Química pela Universidade Federal de Pernambuco (UFPE). Possui mestrado e doutorado em Engenharia Civil (UFPE) nas áreas de caracterização, modelagem e simulação estatística, aplicadas a poços e reservatórios de petróleo. Possui diversos artigos, com o tema de estatística aplicada, publicados em revistas e congressos nacionais e internacionais.

COPYRIGHT

Escrito por Rafael F.V.C. Santos.

Copyright 2018

Todos os direitos reservados. A propriedade intelectual desta obra literária técnica está assegurada ao autor pela Lei Federal nº 9.610/1998. Nenhuma parte desta obra pode ser apropriada, comercializada e estocada em sistema de dados ou processo similar em qualquer forma e meio, seja eletrônico, a fotocópia, a gravação, etc., sem a expressa autorização do autor, titular dos respectivos direitos autorais.

Série Cientista de Dados – Analista Quant

Primeira edição, 2018.

Sumário

1. Introdução
2. Linguagem de Programação Python
 - 2.1. Baixando e instalando o Python
 - 2.2. Ambiente de desenvolvimento (IDE)
 - 2.3. Conhecendo o IDLE Python
 - 2.4. Principais bibliotecas Python
 - 2.5. Instalando o NumPy
 - 2.6. Funcionalidades da Linguagem Python
3. Conceitos básicos da Álgebra Linear
 - 3.1. Escalares, vetores e matrizes
 - 3.2. Tipos de matrizes
 - 3.3. Operações com vetores e escalares
 - 3.4. Operações entre Vetores
 - 3.5. Operações entre Matrizes e escalares
 - 3.6. Operações entre Matrizes e vetores
 - 3.7. Operações entre Matrizes
 - 3.8. Inversa de uma matriz
 - 3.9. Manipulando elementos da matriz
4. Sistemas de Equações Lineares
5. Determinantes de Matrizes
 - 5.1. Calculando o Determinante
6. Autovalores e Autovalores
 - 6.1. Transformação Linear
 - 6.2. Autovalores e autovetores de uma Matriz
7. Aplicações
 - 7.1. Exemplo 1 - Matrizes e Estatísticas
 - 7.2. Exemplo 2 - Matrizes e Criptografia
 - 7.3. Exemplo 3 - Sistemas lineares
 - 7.4. Exemplo 4 - Rankeamento de Busca em sites
 - 7.5. Exemplo 5 - Sequência de Fibonacci
 - 7.6. Agradecimentos

Capítulo 1

1. Introdução

O mundo está se tornando cada vez mais automatizado e inteligente. Existe uma nova rede de autonomia sendo criada por trás de todos os dispositivos que utilizamos. Relógios, televisores, telefones celulares, automóveis, eletrodomésticos em geral, tudo está ganhando uma estrutura de conexão singular e auto dependente com suas próprias partes e cada vez mais independentes de nossa intervenção.

Por trás de tudo isso existe muita ciência e tecnologia envolvida. A matemática, estatística e computação científica são os pilares por trás de toda essa maravilha que surge para maximizar nosso bem-estar e qualidade de vida.

Este e-book pretende oferecer suporte no entendimento dos fundamentos essências para um campo muito interessante e em grande ascensão que é o da inteligência artificial (IA). Iremos estudar **álgebra linear aplicada com o Python**.

Atualmente, estamos vivendo a era da aprendizagem profunda (*deep learning*), que também faz parte da IA. A *deep learning* se propõe a estudar a estrutura de aprendizado profunda de dados, utilizando hierarquias com um conjunto de algoritmos para modelar problemas abstratos. Com esse tipo de técnica, é possível fazer com que máquinas consigam aprender de maneira completamente autônoma utilizando, por exemplo, uma rede neural artificial com várias camadas ocultas. Vale destacar que a álgebra linear tem um papel fundamental por trás de todo esse desenvolvimento.

Portanto, se você está aqui, parabéns pois começou por um bom caminho. Na vida, para sermos os mais plenos e lúcidos possíveis, devemos sempre buscar entender os fundamentos.

Neste e-book iremos estudar os principais assuntos e teoremas da álgebra linear. O leitor, de preferência, poderá acompanhar este material com o auxílio de um livro mais detalhado e abrangente. Isso porque iremos focar nos assuntos de maior relevância.

O conteúdo apresentado é essencial para todos aqueles que desejam mergulhar nos campos da engenharia, economia, estatística, criptomoedas, e principalmente da computação científica com *machine learning* e *deep learning*. A inteligência artificial (IA) é um campo em crescimento e desenvolvimento acelerado que veio para mudar nossas vidas de maneira sem precedentes.

As bases que fundamentam e que dão suporte para as diversas subáreas da IA, fundamentalmente, estão todas ligadas a álgebra linear. A linguagem de programação Python é uma das que mais vem ganhando destaque na comunidade de desenvolvedores ligados a internet das coisas (IoT – *internet of things*), sistemas embarcados e inteligência artificial. Por isso decidimos unir esses dois mundos, a álgebra linear com a poderosa linguagem Python, a fim de apresentar de maneira rápida e simplificada assuntos, teoremas e modelos da álgebra linear.

Grande parte dos projetos *Open Source*, aqueles de código aberto, estão cada vez ganhando destaque entre as grandes empresas. As equipes de desenvolvedores dessas empresas estão de algum modo utilizando a linguagem Python como a mãe geradora de novas tecnologias. Isso porque, Python, ganha em disparado quando o assunto é velocidade na curva de aprendizado da linguagem, manutenção de código e transferência de tecnologia.

Python apresenta uma linguagem de alto nível com sintaxe bastante simplificada e intuitiva. Assim, precisamos de pouquíssimas linhas de códigos para construir grandes e complexos projetos. Se você já está imerso ou ainda está pensando em mergulhar no mundo da engenharia de dados, ciência dos dados, *machine learning*, *deep learning*, internet das coisas, análise de *big data*, dentre outras, Python é, de maneira respeitosa, uma escolha certa.

Existem diversas linguagens excelentes para trabalhar com álgebra linear e consequentemente matrizes e vetores, como por exemplo: *Matlab* e *Octave*. No entanto, ou essas linguagens são pagas e custam caro (caso do *Matlab*) ou possuem uma comunidade de desenvolvedores ainda limitada (caso do *Octave*). Entretanto Python

está na posição de número 5 segundo o Tiobe (lista ordenada de linguagens de programação) quando o assunto é praticidade e utilização entre os desenvolvedores. Isso porque apresenta uma grande variedade de bibliotecas e pacotes implementados para solucionar os mais variados problemas científicos e tecnológicos.

Quando estamos comprando um livro hoje em dia, com o advento da internet, Wikipédia, blogs e fóruns, estamos investindo em algo além do conteúdo, mas também comprando organização e direcionamento de estudos. É por isso que a série cientista de dados - analista Quant está oferecendo a você um conteúdo resumido e organizado com o mínimo necessário de conhecimento que devemos possuir nas diversas áreas de grande importância e, consequentemente, demanda do momento.

Por exemplo, conhecimentos aprofundados de álgebra linear são extremamente necessários para entendermos as estruturas de funcionamento por trás das bibliotecas mais utilizadas atualmente no Python que são: Scikit-Learn, Keras, NumPY, NLTK, Scrapy e Pandas.

Portanto, este e-book tem a pretensão de mostrar, utilizando o Python na prática, como automatizar delegando para o computador pessoal as operações algébricas que antes fazíamos com papel e caneta. Nosso tempo é precioso e é interessante gastá-lo com aquilo que realmente importa.

Aprender alguma linguagem de programação é aprender a delegar as atividades repetitivas e desgastantes para as máquinas. Muitas ideias já estão implementadas, prontas para serem encontradas e utilizadas. Nós do mundo científico e tecnológico devemos estar alinhados a utilização de tudo isso o quanto antes.

Não obstante, precisamos entender os fundamentos teóricos e utilizar as ferramentas computacionais para trabalhar nas etapas repetitivas e maçantes. Após a leitura deste livro você irá perceber o quão fácil, rápido e simples é, uma vez conhecendo a teoria, utilizar Python para resolver os problemas algébricos.

Vamos a um breve resumo do que você irá aprender após a leitura deste e-book:

- Como baixar e instalar o Python 3.7;
- Como instalar pacotes no Python;
- Uma breve introdução à sintaxe de funcionamento da linguagem Python;
- Conhecer algumas bibliotecas Python para trabalhar com matrizes;
- A diferença entre escalares, vetores e matrizes no Python;
- O que são matrizes, como representá-las, seus tipos, e suas principais aplicações na engenharia, ciência naturais e computação científica;
- O que são determinantes e suas aplicações;
- Dependência e independência linear;
- O que são transformações lineares;
- Como efetuar operações com matrizes, escalares e vetores no Python
- O que são e a importância dos autovalores e autovetores
- Como representar matricialmente e resolver sistemas lineares complexos com Python;
- Como criar um método de criptografia de mensagens utilizando matrizes;
- Tudo isso e muito mais utilizando exemplos práticos.

Vamos conhecer os capítulos deste e-book. Cada um dos capítulos segue uma ordem didática evolutiva. Mas nada impede que o leitor siga seu próprio caminho de leitura.

No entanto, é extremamente importante fazer a leitura rápida do Capítulo 2, pois nele aprenderemos tudo aquilo necessário para entender os exemplos do livro aplicados com o Python.

No capítulo 3 iremos estudar os principais elementos algébricos como: escalares, vetores e matrizes. Iremos também aprender a efetuar diversas operações matemáticas com o Python entre esses elementos.

O capítulo 4 apresenta os sistemas lineares e como eles podem ser representados e solucionados a partir de matrizes.

No capítulo 5 iremos estudar sobre determinantes de matrizes. Iremos testar com o Python cada uma das propriedades algébricas desse assunto.

O capítulo 6 apresenta conceitos básicos de transformações lineares e está diretamente interessado em apresentar o que são e como funcionam os autovalores e autovetores.

Por fim, no capítulo 7 temos vários exemplos práticos que utilizam a álgebra linear como fundamento que subsidia a solução dos problemas.

Aos estudos!

Capítulo 2

2. Linguagem de Programação Python

O Python é uma linguagem de programação de alto nível multiparadigma (podemos utilizar programação funcional e/ou orientada a objetos). Vale ressaltar que a linguagem se chama de alto nível pois está bastante próxima do entendimento da linguagem humana, diferentemente da linguagem Assembly, por exemplo, que é tida como uma linguagem de baixo nível (linguagem de máquina).

A linguagem Python apresenta tipagem dinâmica, permitindo que os códigos fiquem menos verbosos e mais claros. Claro que nem tudo são flores, essas características de tipagem abrem portas para possíveis erros de armazenamento de variáveis.

No entanto, os benefícios trazidos com essa linguagem nos permitem ganhar tempo e focar realmente naquilo que interessa que é a lógica do nosso problema. Deixaremos de passar horas e horas entendendo sintaxes complexas e extensas para apenas focar naquilo que pode solucionar os nossos problemas.

Como sabemos, programar é delegar ações para que as máquinas possam executar de maneira mais eficiente e eficaz. Os elementos em jogo são o tempo (velocidade) e memória de armazenamento. Nossas limitações de processamento de cálculos complexos e armazenamento de memória de curto prazo são sanados a partir da utilização de alguma linguagem de programação como Python.

Felizmente o Python é um projeto *Open Source* (<https://github.com/python>) e diversos desenvolvedores pelo mundo estão ativamente criando melhorias e novas bibliotecas que estarão prontas para solucionar os mais diversos problemas de inúmeras áreas.

O criador da linguagem Python foi Guido Van Rossum (1956-atual), que o fez em 1991. Rossum tinha em mente fazer algo prático para que tivesse avanços expressivos de produtividade e legibilidade em seus códigos computacionais. E assim ele criou a linguagem Python visando produzir e manter códigos sofisticados de maneira fácil e rápida.

O nome Python é originário de uma série de TV humorística britânica chamada de *The Monty Python*. Existem também alguns filmes da série que, inclusive, possuem um conteúdo muito interessante e divertido.

A linguagem é uma das mais populares, como podemos ver abaixo no site Tiobe. Python se encontra entre as cinco linguagens de programação mais populares:

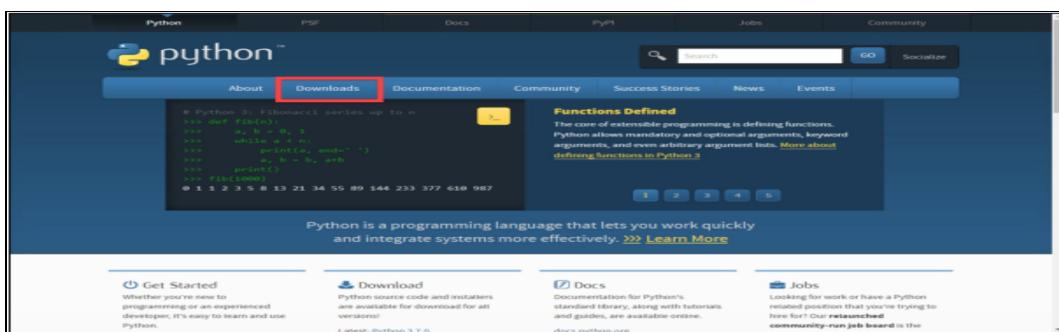
Jul 2018	Jul 2017	Change	Programming Language	Ratings	Change
1	1		Java	16.139%	+2.37%
2	2		C	14.662%	+7.34%
3	3		C++	7.615%	+2.04%
4	4		Python	6.361%	+2.62%
5	7	▲	Visual Basic .NET	4.247%	+1.20%
6	5	▼	C#	3.795%	+0.28%
7	6	▼	PHP	2.832%	-0.26%
8	8		JavaScript	2.831%	+0.22%
9	-	▲	SQL	2.334%	+2.33%
10	18	▲	Objective-C	1.453%	-0.44%

Referência.: <https://www.tiobe.com/tiobe-index/>

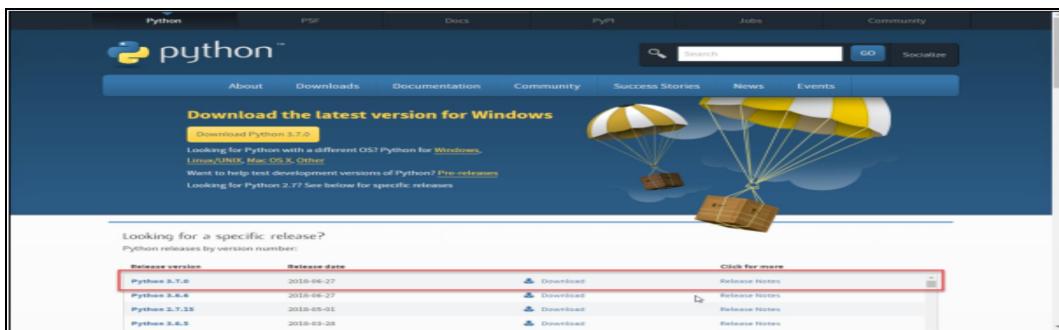
Diversas importantes e grandes empresas estão utilizando o Python nos seus projetos, tais como: YouTube, Google, Instagram, Dropbox, Spotify, BitTorrent, dentre outras.

2.1. Baixando e instalando o Python

Podemos fazer o download da linguagem Python para vários sistemas operacionais, como: Linux, Mac e Windows. Basta acessar o site: <https://www.python.org/> e clicar na aba de downloads.

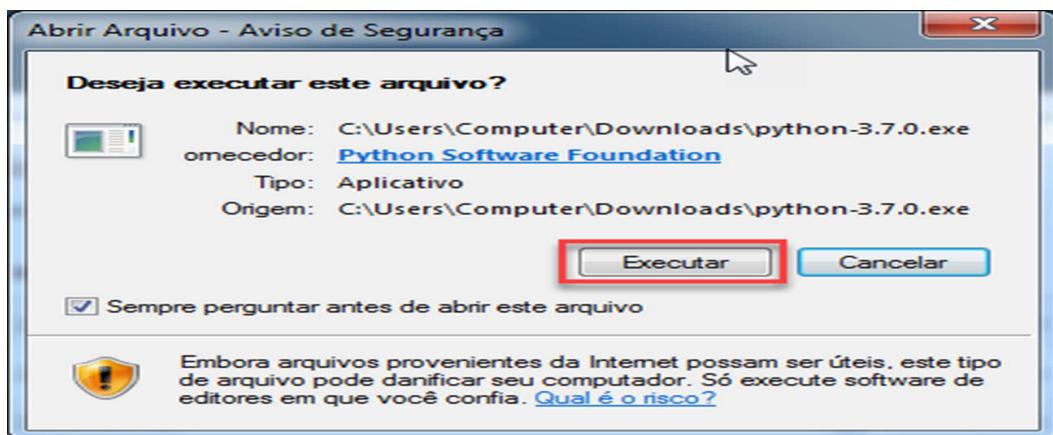


Iremos fazer o download para instalação no Windows. Fato interessante é que os sistemas operacionais Linux e Mac já vêm com Python previamente instalados. Atualmente (junho de 2018) a versão mais recente é a Python 3.7.0.

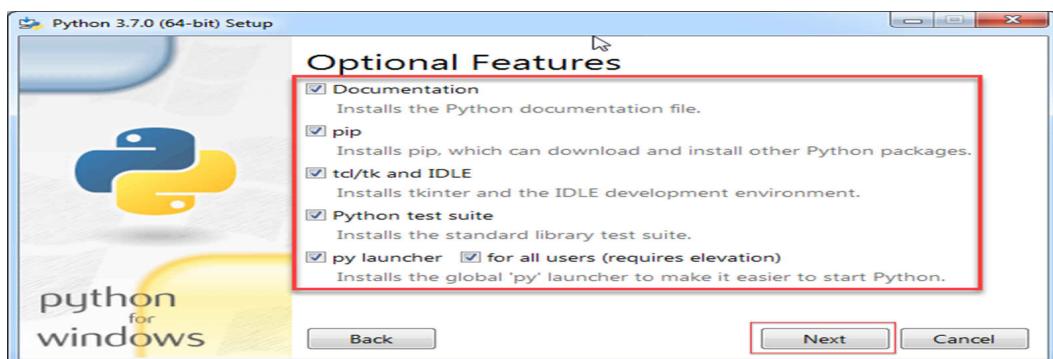


Release version	Release date	Click for more
Python 3.7.0	2018-06-27	Download
Python 3.6.6	2018-06-27	Download
Python 3.7.0rc1	2018-05-21	Download
Python 3.6.5	2018-03-08	Download

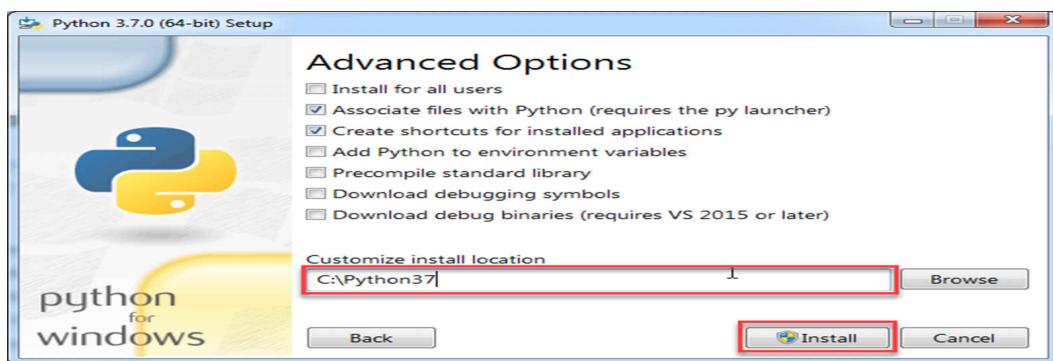
O programa para download é bastante leve, apresentando um pouco menos do que 30 mb. Após o download, a instalação é bastante simplificada. Tente seguir as figuras abaixo:



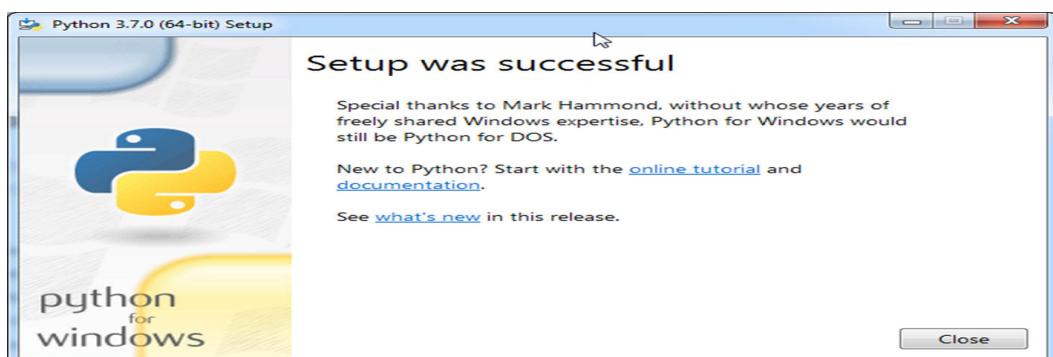
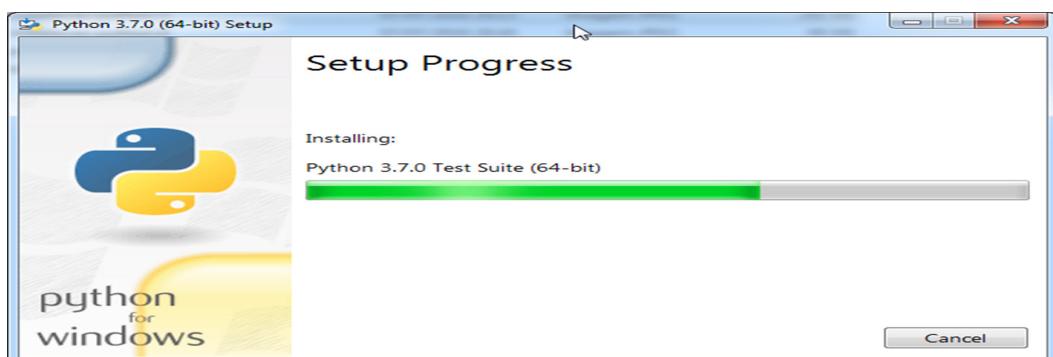
Após executar estaremos diante da seguinte janela abaixo.



Clicando em Next. Vamos customizar nossa instalação para facilitar onde encontrar o Python instalado. Escolher um caminho de instalação curto é interessante para que depois possamos fazer modificações e instalações de novas bibliotecas como será mostrado nos próximos capítulos. Veja abaixo o caminho de nossa instalação para os exemplos deste livro:



Devemos permitir o processo de instalação a partir do administrador e aguardar.



2.2. Ambiente de desenvolvimento (IDE)

Um IDE (*Integrated Development Environment*) ou ambiente integrado de desenvolvimento de linguagem de programação é bastante utilizado pela praticidade e dinamicidade da criação dos algoritmos. Atualmente temos acesso a uma infinidade de IDEs gratuitas e pagas. Não vale a pena discutir sobre a melhor IDE para se programar. Cada programador vai ter sua preferida e defendê-la com unhas e dentes.

Todavia, nosso interesse com esse livro é mais conceitual, ou seja, não iremos desenvolver projetos complexos que necessitem de um ambiente para estruturar e organizar devidamente o nosso projeto. Por isso estaremos deixando de lado o uso de uma IDE mais completa. Contudo, para fins de conhecimento, é importante que saibamos as principais e mais utilizadas IDEs do mercado. Vamos às principais:

Anaconda - Spyder(<https://anaconda.org/anaconda/python>)

PyCharm (<https://www.jetbrains.com/pycharm/>)

Sublime (<https://www.sublimetext.com/>)

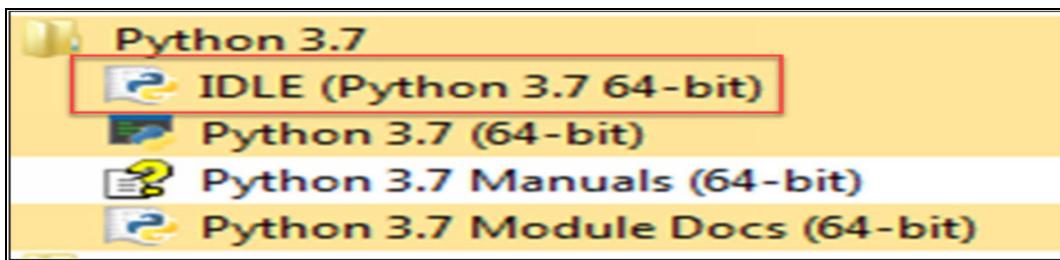
Atom (<https://atom.io/packages/ide-python>)

PyDev (<http://www.pydev.org/>)

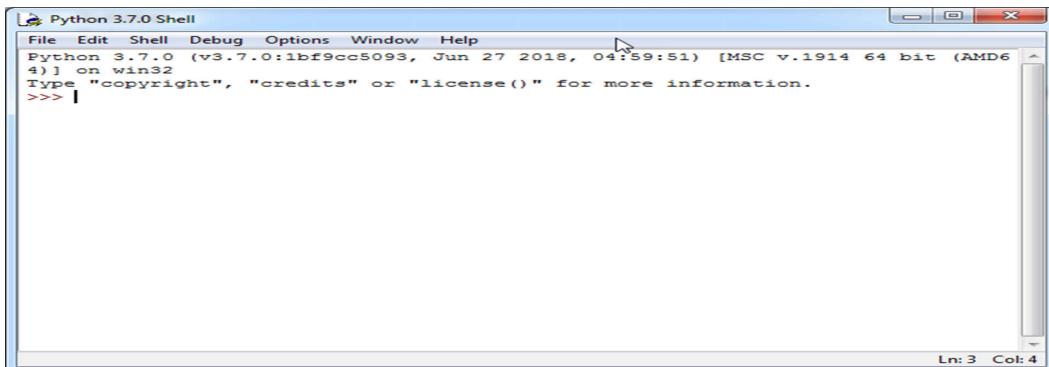
Após o download e instalação do Python 3.7, iremos ter acesso ao IDLE (Python GUI) que é uma interface gráfica bastante útil para nos ajudar no desenvolvimento de projetos pequenos e médios com o Python. Assim, para os objetivos deste livro, o IDLE será suficiente.

2.3. Conhecendo o IDLE Python

Quando fizemos a instalação do Python, foram criados alguns módulos de uso. Podemos encontrá-los no Menu iniciar e procurar pelo Python 3.7 instalado. Assim teremos às seguintes opções de uso:



Iremos desenvolver nossos códigos a partir do IDLE que é um ambiente de desenvolvimento bastante modesto, porém útil para os objetivos deste livro.



Então vamos dar o seguinte comando para termos acesso a filosofia de desenvolvimento com o Python (*The Zen of Python*). O comando está em destaque mostrado na figura abaixo.

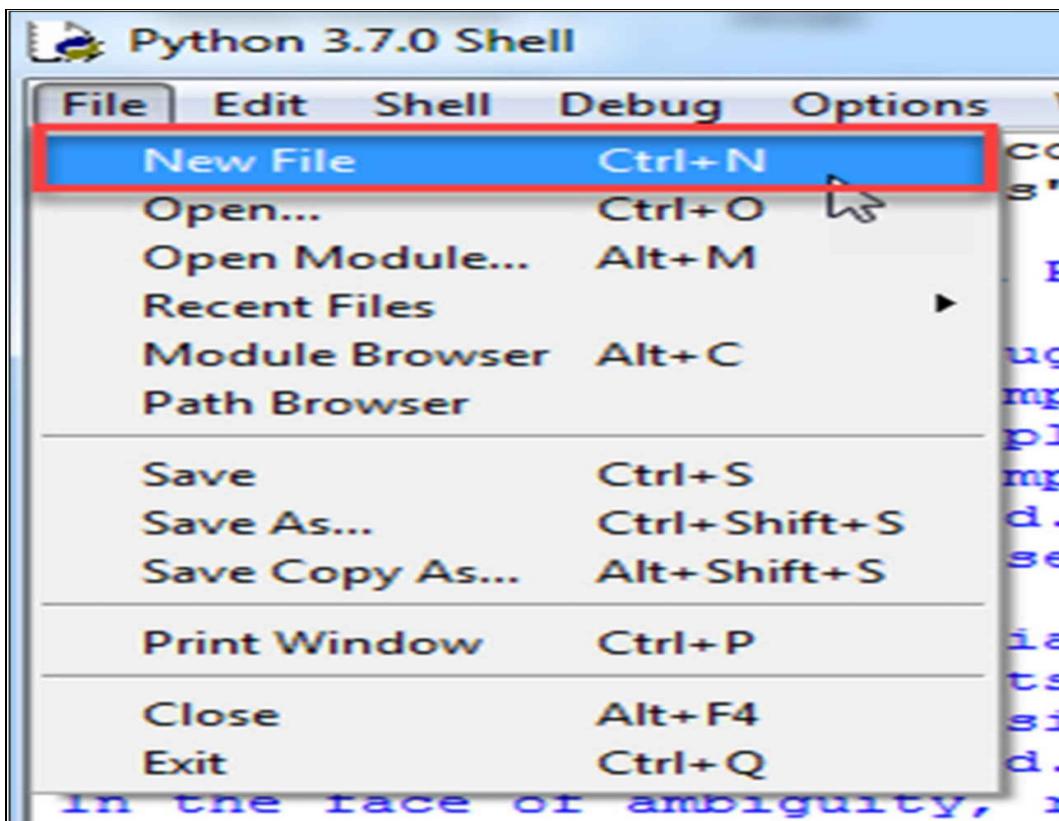
The screenshot shows the Python 3.7.0 Shell window. The title bar reads "Python 3.7.0 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. A status bar at the bottom right shows "Ln: 25 Col: 4". The main window displays the "Zen of Python" text, which is a collection of 19 aphorisms about good programming style. The first few lines are:

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although simple is better than complex.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than "right" now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!  
>>> |
```

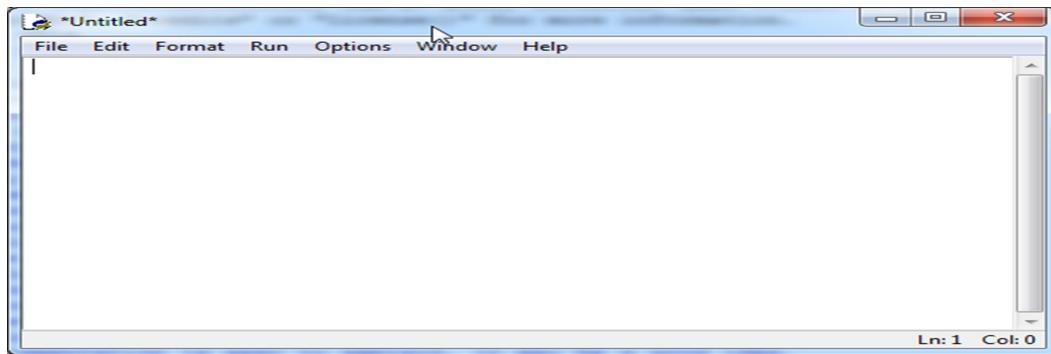
É bastante importante e curioso ler cada uma dessas filosofias de programação com Python.

Muitos dos exemplos deste livro serão efetuados no *prompt* (são as três setinhas >>>) de comando da figura acima.

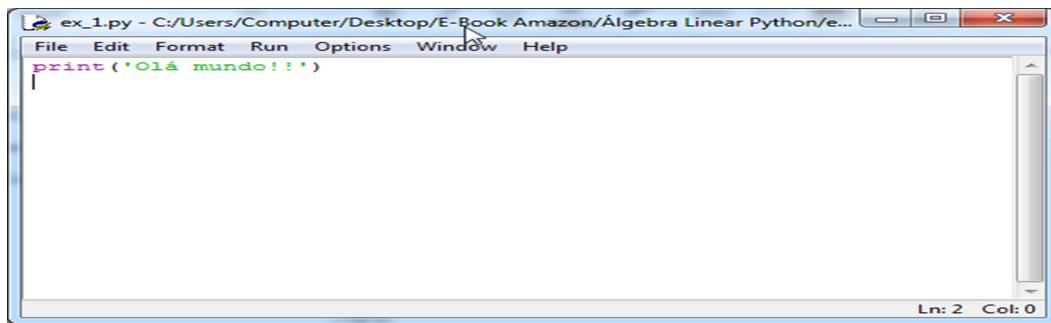
Agora, vamos abrir um novo arquivo para que possamos trabalhar com os códigos que iremos desenvolver:



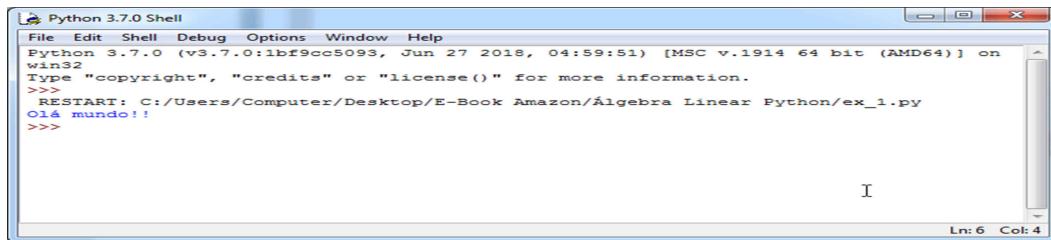
Esse novo arquivo nos permitirá escrever os comandos de maneira mais flexível e organizada. Nosso ambiente de desenvolvimento será:



Vamos dar um nome ao nosso arquivo Python e escrever nossas boas-vindas ao mundo com essa linguagem:



Após salvar o arquivo em algum diretório específico devemos ir no menu **Run -> Run Módulo** (ou apertar F5). Assim teremos:



2.4. Principais bibliotecas Python

A linguagem Python fornece uma gama enorme de bibliotecas para trabalhar nas mais diversas áreas científicas e tecnológicas. As principais e mais utilizadas bibliotecas são:

Básicas:

NumPy -(<http://www.numpy.org/>)

SciPy (<https://www.scipy.org/>)

Pandas (<https://pandas.pydata.org/>)

Gráficas:

Matplotlib (<https://matplotlib.org/>)

Plotly (<https://plot.ly/python/>)

Bokeh (<https://bokeh.pydata.org/en/latest/>)

VPython - para gráficos animados em 3D (<http://vpython.org/>)

Álgebra simbólica:

SymPy (<http://www.sympy.org/pt/index.html>)

Estatística:

Statsmodels (<https://www.statsmodels.org>)

Scikit-Learn (<http://scikit-learn.org/stable/>)

PyMC - para análise Baysiana de Dados (<https://github.com/pymc-devs>)

Pystan (<http://pystan.readthedocs.io/en/latest/>)

Keras (<https://keras.io/>)

Outras

NetworkX (<https://networkx.github.io/>)

Wakari - para desenvolvimentos em computação na nuvem
(<https://anaconda.org/wakari>)

PyCuda, PyOpenCL - Processamento paralelo
(<https://documentacion.de/pycuda/>)

Theano - processamento paralelo e otimização
(<http://deeplearning.net/software/theano/>)

CVXPY - otimização convexa com Python (<http://www.cvxpy.org/>)

PyTables - manipulação de dados em tabelas
(<https://www.pytables.org/>)

Numba - permite que o Python execute processos em velocidade de código em linguagem de máquina nativa (baixo nível)
(<https://numba.pydata.org/>)

Jupyter - permite utilizar o Python diretamente de seu browser
(<http://jupyter.org/>)

Tensor Flow - ampla biblioteca para trabalhar com inteligência artificial (<https://www.tensorflow.org/?hl=pt-br>)

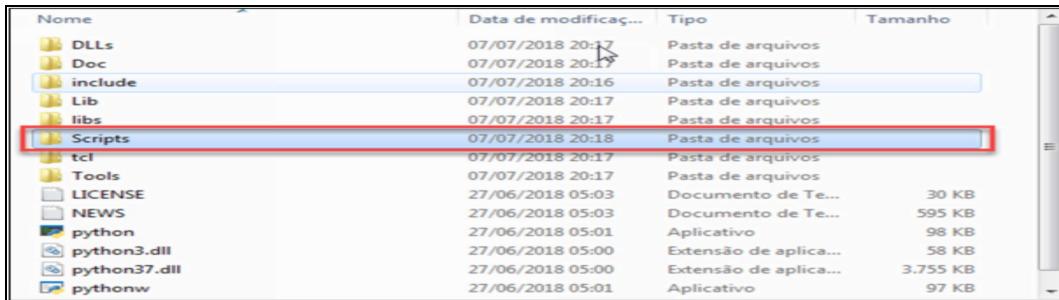
Neste livro iremos fazer uso apenas da biblioteca NumPy (<http://www.numpy.org/>) que apresenta um conjunto de funcionalidade para trabalhar com computação científica tais como: vetores multidimensionais, ferramentas para integrar códigos escritos em C/C++ e Fortran, geração de números aleatórios e principalmente para trabalhar com álgebra linear que é nosso foco principal de estudos.

2.5. Instalando o NumPy

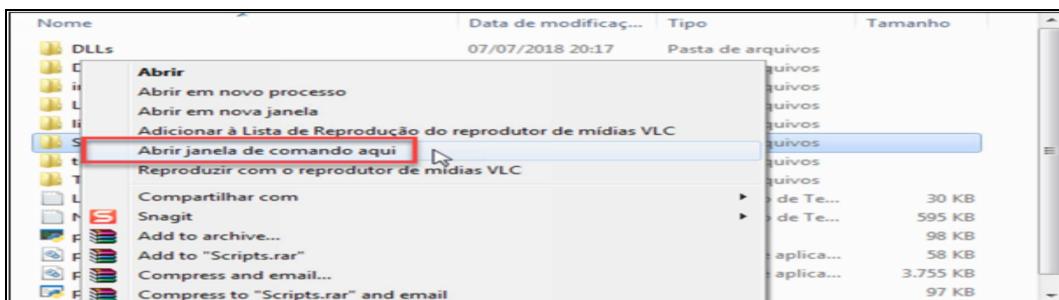
Podemos fazer a instalação do Numpy no Python 3.7 de diversas maneiras. Muitas vezes instalar pacotes e bibliotecas pode se tornar um problema. Mas você irá aprender uma maneira bastante simplificada e rápida utilizando o **pip**. Assim, caso tenhamos mais de uma versão do Python instaladas em nossas máquinas, o seguinte modo de instalação que iremos conhecer a seguir irá facilitar bastante nossas vidas.

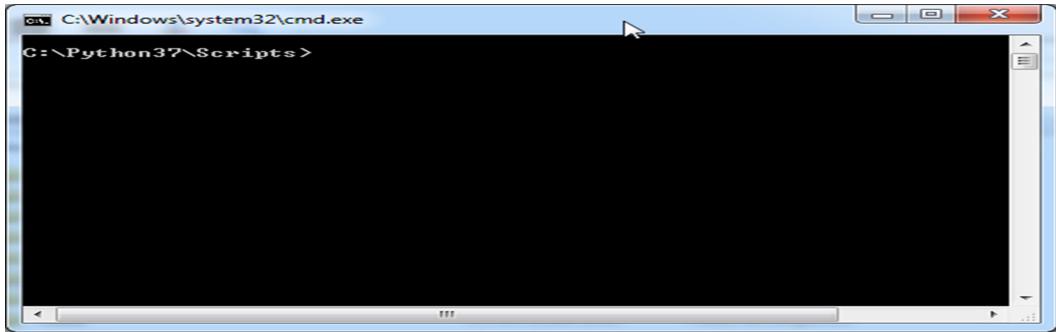
Pois então vamos seguir os seguintes passos:

- 1 - Vamos no diretório onde instalamos o Python 3.7. No nosso caso C:\Python37



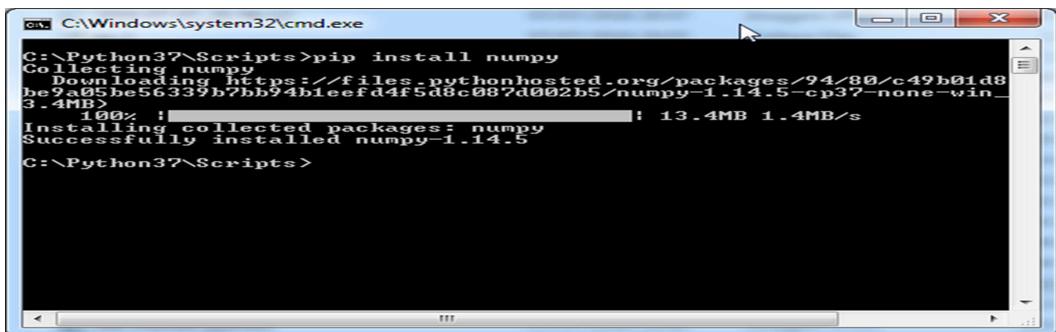
- 2 - Com o botão direito e segurando a tecla *shift* do teclado vamos clicar em cima da pasta Scripts e clicar em Abrir janela de comando aqui, como mostra a figura abaixo:





3 – Em seguida vamos dar o seguinte comando nesta janela: **>> pip install numpy**

E clicar **Enter**:



Tudo certo. Agora temos nossa biblioteca **NumPy** devidamente instalada. Vamos conhecer agora algumas funcionalidades do Python para facilitar a compreensão dos capítulos subsequentes.

2.6. Funcionalidades da Linguagem Python

Se o leitor tiver conhecimento na linguagem Python^[1], poderá pular esse tópico. Aqui serão apresentadas as principais funcionalidades e sintaxe básica da linguagem Python. É básica pois iremos aprender apenas o necessário para a compreensão dos exemplos e aplicações do livro. Também se o leitor tiver algum conhecimento prévio em lógica de programação esse tópico se tornará mais leve e fluido.

De qualquer maneira, vamos a essa breve introdução.

Import de Bibliotecas

Neste livro iremos trabalhar exaustivamente com a biblioteca NumPy. E, para tanto, precisamos fazer a instalação e importação nos nossos scripts de códigos. O Python permite importar bibliotecas a partir do comando ‘*import*’ que deve ser colocado no topo do arquivo. Veja a figura abaixo:



```
File Edit Format Run Options Window Help
import numpy as np
```

Podemos também importar métodos e pacotes específicos de grandes bibliotecas a partir do comando ‘*from*’. Esse tipo de ação é interessante caso queiramos utilizar apenas um conjunto específico de pacotes ou métodos e não estamos interessados em todas as funcionalidades da biblioteca. Veja abaixo como fazer esse tipo de *import*:

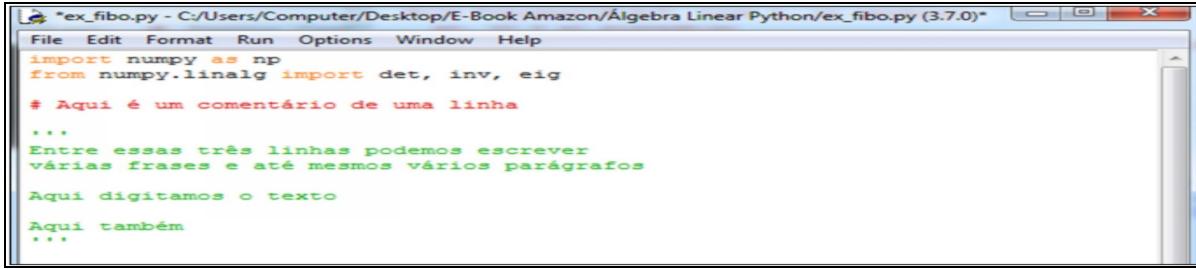


```
File Edit Format Run Options Window Help
import numpy as np
from numpy.linalg import det, inv, eig
```

No caso acima, estamos importando os métodos **det()**, **inv()** e **eig()**. No decorrer do livro iremos aprender mais a respeito dos mesmos.

Comentários no Python

É muito importante ter um código limpo e bem documentado. Para isso precisamos utilizar caracteres especiais que fazem com que o Python não execute no compilador linhas comentadas. Existem duas principais maneiras de fazer comentário com a linguagem. Veja abaixo:



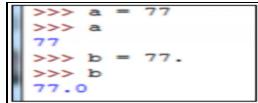
The screenshot shows a Windows-style code editor window titled "ex_fibo.py - C:/Users/Computer/Desktop/E-Book Amazon/Álgebra Linear Python/ex_fibo.py (3.7.0)". The code in the editor is as follows:

```
File Edit Format Run Options Window Help
ex_fibo.py - C:/Users/Computer/Desktop/E-Book Amazon/Álgebra Linear Python/ex_fibo.py (3.7.0)
import numpy as np
from numpy.linalg import det, inv, eig
# Aqui é um comentário de uma linha
...
Entre essas três linhas podemos escrever
várias frases e até mesmos vários parágrafos
Aqui digitamos o texto
Aqui também
... 
```

Parágrafos entre três aspas simples ("....") ou frases após cerquinha (# jogo da velha) são tidos como comentários, portanto não interpretados pelo compilador.

Declaração de variáveis

Na linguagem Python a declaração do tipo de variáveis é dinâmica, ou seja, não precisamos especificar o tipo de variável que nossa variável tem assim como no Java, C++, C# dentre outras. Por exemplo, se quisermos declarar uma variável 'a' com um número inteiro 77 automaticamente o Python reconhecerá a variável 'a' como sendo inteira. Por outro lado, se fizermos a declaração 'a = 77.' (colocarmos um ponto depois do número) teremos uma representação de decimal de ponto flutuante (*float* ou *double*).

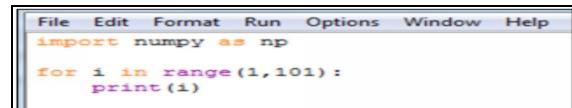


```
>>> a = 77
>>> a
77
>>> b = 77.
>>> b
77.0
```

Laço de repetição - For

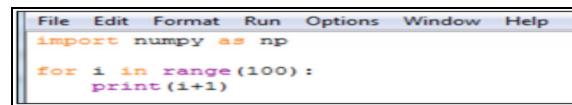
Muitas vezes queremos que o computador execute atividades repetitivas a partir de um laço iterativo. Para isso, praticamente toda linguagem de programação oferece uma estrutura de repetição do tipo ‘**for**’. Isto é, podemos executar comandos dentro de um bloco específico de códigos de maneira repetitiva até que uma condição lógica seja desrespeitada.

Vamos a um exemplo para esclarecer melhor tudo isso. Supondo que queiramos pedir para o Python escrever na tela (**print()**) números de 1 a 100. Poderíamos colocar o comando **print** 100 vezes: **print(1)**, **print(2)** ,..., **print(100)**. No entanto, o ‘**for**’ facilita nosso trabalho. Veja abaixo como fica o código em Python:



```
File Edit Format Run Options Window Help
import numpy as np
for i in range(1,101):
    print(i)
```

A função ‘range(inicio,fim)’ permite criar uma lista de valores que inicia a partir do primeiro parâmetro (inicio) e vai até o ultimo número pedido (fim) adicionando uma unidade. Essa função é muito utilizada dentro do laço de repetição ‘for’. O índice ‘i’ vai percorrer cada um dos valores dentro da lista criada pelo **range()**. Vale ressaltar que se colocarmos apenas um parâmetro dentro do método por exemplo: **range(100)**, iremos ter uma varredura de valores sempre começando por 0 (zero) acrescentando uma unidade até completar o número de valores escolhido como parâmetro, ou seja, como temos **range(100)**: 0,1,2,3...,97,98,99 (vai até 99 pois iniciamos com zero!). O código abaixo, fornece o mesmo resultado que o anterior.



```
File Edit Format Run Options Window Help
import numpy as np
for i in range(100):
    print(i+1)
```

Listas e dicionários

Constantemente estamos criando variáveis onde iremos armazenar dados. Um tipo de variável muito importante são as listas, onde podemos acrescentar valores à medida em que temos necessidade. No Python podemos declarar uma variável do tipo lista da seguinte maneira:

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

Podemos adicionar vários tipos de elementos a lista. Para fazer isso, podemos usar o método **append()**. Ver exemplo abaixo:

```
>> a.append(77)
>> a.append(123)
>> a
[77, 123]
>> a.append('texto')
>> a
[77, 123, 'texto']
>> len(a) # o método ou função len() diz o tamanho da lista
3
```

Vamos falar agora sobre dicionários em Python. Iremos precisar desse modo de armazenamento de dados no nosso exemplo prático de criptografia com matrizes.

O dicionário é um tipo particular de coleção de dados. Ele é um tipo de mapeamento nativo a linguagem. Temos no caso uma associação dual do tipo chave – valor. Para declarar um dicionário em Python devemos fazer o seguinte:

```
>> port2eng = {} # abrir e fechar chaves!
>> type(port2eng)
<class 'dict'>
```

Vejamos agora como adicionar elementos a um dicionário:

```
>> port2eng['UM'] = 'ONE'
```

```
>> port2eng['DOIS'] = 'TWO'
>> port2eng['TRES'] = 'THREE'
>> port2eng['QUATRO'] = 'FOUR'
>> port2eng
{'UM': 'ONE', 'DOIS': 'TWO', 'TRES': 'THREE', 'QUATRO': 'FOUR'}
>> len(port2eng) # o método len() diz o tamanho do dicionário
4
```

Portanto, temos no dicionário ‘port2eng’ uma associação entre palavras.

Para recuperar algum valor dessa associação basta fazer o seguinte:

```
>> texto = 'Em inglês, o número 4 é escrito da seguinte forma:
'+port2eng['QUATRO']
>> texto
'Em inglês, o número 4 é escrito da seguinte forma: FOUR'
```

Declarando Funções

O Python nos permite definir nossas próprias funções. Vimos a utilização da função `len()` usada para encontrar o tamanho de uma lista ou dicionário. Agora vamos aprender como criar nossas próprias funções.

No Python quando queremos declarar uma função devemos utilizar a instrução `def`. Uma função é algo que possui entradas, processamentos e saída. Vamos ver como declarar uma função que apresenta na tela um texto com uma mensagem do tamanho de uma lista ou dicionário:

```
a = []
a.append(12)
a.append(77)
a.append('python')

def dizTamanhoLista(lista):
    resposta = 'A lista tem '+str(len(lista))+ ' elementos!'
    return resposta

print(a)
print(dizTamanhoLista(a))
```

Atenção quando escrever as funções pois devemos respeitar a tabulação (espaços) na declaração das mesmas. Vejamos outro exemplo onde temos dois parâmetros de entrada, com um processamento de soma e retorno dos valores de entrada somados:

```
| def soma(a,b):  
|     soma = a + b  
|     return soma  
  
| print(soma(22,11))
```

Chegamos ao final do capítulo e vale ressaltar que fizemos apenas uma breve pincelada nos principais conceitos que iremos utilizar no decorrer do livro. O leitor deverá se aprofundar no assunto a partir de livros específicos e pesquisas na internet.

Capítulo 3

3. Conceitos básicos da Álgebra Linear

Entender os fundamentos é uma atitude inteligente para iniciar qualquer atividade intelectual. Na era da computação científica quem nos dita as regras são os fundamentos da matemática e estatística. A álgebra linear é um ramo da matemática que surgiu, dentre outras coisas, para solucionar sistemas de equações lineares algébricas ou diferenciais. Praticamente tudo a nossa volta pode ser modelado e, portanto, simplificado a partir de equações algébricas ou diferenciais.

A álgebra linear está à nossa disposição fornecendo conceitos e teorias para trabalharmos com os tijolos dos problemas algébricos que são: os vetores, espaços vetoriais, transformações lineares, matrizes e tensores. O escopo deste e-book limita-se ao estudo pouco aprofundado desses conceitos, focando mais em apresentar como o Python pode nos ajudar com a automatização das operações algébricas.

Quando o assunto é inteligência artificial (IA) e suas aplicações, o conhecimento de Álgebra Linear se torna indispensável para aqueles que procuram entender e desenvolver essa tão atual e importante área. Infelizmente, muitos livros são apenas teóricos ou mostram poucas aplicações práticas. Não obstante, neste livro estaremos minimizando esse problema a partir da apresentação e aplicação imediata das teorias, através da linguagem de programação Python.

3.1. Escalares, vetores e matrizes

É muito importante entendermos as diferenças entre os principais elementos modeladores dos nossos problemas. O mundo precisa ser simplificado e representado de maneira tal que problemas iterativos e complexos possam ser solucionados.

Escalares

Os escalares são variáveis que admitem apenas um valor específico. Esses tipos de variáveis são úteis para modelar problemas principalmente em uma dimensão. Portanto, um escalar é um número qualquer (inteiro, real, natural e até mesmo complexo).

Vamos a exemplos de escalares com o Python:

```
>> T = 37      # (escalar que representa a temperatura em graus Celsius)
>> Vel = 80    # (escalar que pode representar a velocidade instantânea em m/s)
>> c = 150,00 # (escalar que pode representar algum valor monetário em real R$)
>> type(T)    # vai retornar um tipo inteiro (int)
```

Vetores

Os vetores são variáveis que tentam representar (modelar ou armazenar) dimensões mais elevadas do que os escalares. Com os vetores podemos representar conjuntos de dados. Podemos pensar que os vetores são úteis para armazenar coleções de escalares. Esses dados podem estar representando um problema de n-dimensões (multidimensional).

Os vetores podem ser representados em uma linha ou em uma coluna como mostra abaixo:

$$v1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \text{ (vetor coluna) ou}$$

$$v2 = [1 \quad 2 \quad 3] \text{ (vetor linha)}$$

Vejamos alguns exemplos de vetores com o Python. Precisamos primeiro importar a biblioteca **NumPy** como mostra abaixo:

```
>> import numpy as np    # carrega a biblioteca que iremos utilizar  
para trabalhar com matrizes e vetores  
>> VT = np.array([27, 37]) # vetor com duas dimensões  
>> VV = np.array([20, 15, 100]) # vetor com três dimensões  
>> VM = np.array([100, 205, 777, 111]) # vetor com quatro  
dimensões  
>> type(VT) # retorna um vetor numpy (numpy.ndarray)
```

Daqui para frente iremos sempre ter que carregar a biblioteca NumPy. Portanto, sempre ao reproduzir os exemplos deste capítulo assegure-se de fazer o carregamento a partir do (*import numpy as np*).

Matrizes

Armazenar dados em forma de matrizes é uma das maneiras mais simples e eficientes de modelar os problemas do cotidiano, isso porque já foram feitos muitos estudos com respeito a manipulação de matrizes. Somos indivíduos racionais, claro, e por isso procuramos obter maneiras de melhor executar as atividades para solucionar problemas que nos deparamos.

Quando estamos interessados em representar coleções de vetores, as matrizes são uma maneira inteligente e simplificada que nos

auxiliam nessa atividade. Os elementos das matrizes podem ser números reais, números complexos, expressões algébricas e até mesmo funções. Neste livro iremos trabalhar apenas com os elementos sendo representados por números reais.

Veja abaixo alguns exemplos de matrizes:

$$A_1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, A_2 = \begin{bmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{bmatrix}, A_3 = \begin{bmatrix} 2 & 1+i \\ 3i & -i \end{bmatrix}$$

Portanto, podemos entender uma matriz como uma coleção de vetores linhas ou vetores colunas. Uma matriz possui a seguinte representação geral ($m \times n$):

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Considerando que os elementos das linhas sejam representados por 'i' até 'm' e os elementos das colunas por 'j' até 'n', chamamos de diagonal principal os elementos: a_{ij} onde $i=j$. É importante conhecermos a diagonal principal, pois esta é referência para vários conceitos e teoremas algébricos.

No Python podemos trabalhar com matrizes de diversas maneiras. Isso porque temos a nossa disposição inúmeras bibliotecas prontas para manipular e executar operações com matrizes. Vale lembrar que estaremos utilizando apenas a biblioteca **NumPy**.

Vamos prestar atenção no exemplo abaixo:

```
>> import numpy as np  
>> A = np.array(([1,2,3],[4,5,6],[7,8,9])) # representação de uma  
matriz 3x3  
>> A  
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

Podemos utilizar o método ou função np.matrix()

```
>> B = np.matrix(([1,2,3],[4,5,6]))  
matrix([[1, 2, 3],  
       [4, 5, 6]])
```

A variável da matriz pode possuir o tipo **array** ou **matrix**. Com ambos os tipos podemos fazer as operações algébricas que serão estudadas. Vamos conferir mais alguns exemplos de matrizes com o Python.

$$A_1 = \begin{bmatrix} 1 & 2 & 4 \\ 0 & 5 & -1 \\ 7 & 0 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 2 & 7 \\ 0 & -1 \end{bmatrix}$$

A representação desse exemplo no Python fica:

```
>> A1 = np.array(([1,2,4],[0,5,-1],[7,0,0]))  
>> A2 = np.array(([2,7],[0,-1]))  
>> type(A1)  
<class 'numpy.ndarray'>
```

No caso apresentado acima temos um vetor (array) multidimensional de três dimensões. No entanto, podemos representar A1 e A2 como matrizes propriamente ditas utilizando NumPy. Veja como fazer abaixo:

```
>> A1 = np.array(([1,2,4],[0,5,6],[0,0,3]))  
>> A2 = np.array(([2,8],[0,1]))  
>> A1 = np.matrix(A1) # estamos transformando o vetor  
multidimensional em matrix  
>> A2 = np.matrix(A2)
```

```
>> type(A1)
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Fazer essa transformação para **matrix** nos permite distinguir vetores multidimensionais de matrizes propriamente ditas. Porém, como já mencionado antes, a forma de trabalhar com as operações matriciais é praticamente a mesma. Por praticidade, iremos utilizar, predominantemente, os vetores multidimensionais como representações de matrizes.

3.2. Tipos de matrizes

Matriz diagonal e triangular

Temos uma matriz quadrada quando o tamanho das linhas for igual ao tamanho das colunas: $m=n$.

Uma matriz quadrada é dita diagonal quando todos os elementos fora da diagonal principal forem nulos, isto é, $a_{ij} = 0$ para qualquer 'i' diferente de 'j'.

No Python temos o método **diag()** para facilmente escrever uma matriz diagonal. Veja o exemplo abaixo:

```
>> np.diag([1, 5, 7, 9, 10, 1])
array([[ 1,  0,  0,  0,  0,  0],
       [ 0,  5,  0,  0,  0,  0],
       [ 0,  0,  7,  0,  0,  0],
       [ 0,  0,  0,  9,  0,  0],
       [ 0,  0,  0,  0, 10,  0],
       [ 0,  0,  0,  0,  0,  1]])
```

Uma matriz quadrada $A = [a_{ij}]$ é dita triangular superior se $a_{ij} = 0$ quando $i > j$, ou seja, os elementos abaixo da diagonal principal são todos iguais a zero.

Uma matriz quadrada $A = [a_{ij}]$ é dita triangular inferior se $a_{ij} = 0$ para todo $i < j$, ou seja, os elementos acima da diagonal principal são todos iguais a zero.

Exemplos:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 5 & 6 & 0 \\ 1 & 3 & 2 \end{bmatrix} \quad \text{e} \quad A_2 = \begin{bmatrix} 3 & 0 \\ 9 & 1 \end{bmatrix}$$

Matriz linha ou coluna

Podemos ter uma matriz linha dada por:

```
>> B = np.array([[2, 4, 6]]) # obs: devemos colocar um conjunto a mais de colchetes ou parênteses
```

$$B = [2 \quad 4 \quad 6]$$

E uma matriz coluna:

```
>> D = np.array([[2],[-4],[6],[-10]])
>> D
array([[ 2],
       [-4],
       [ 6],
       [-10]])
```

$$D = \begin{bmatrix} 2 \\ -4 \\ 6 \\ -8 \end{bmatrix}$$

Poderíamos ter obtido a matriz coluna D utilizando a função `np.transpose([[2,-4,6,-10]])`. ver abaixo:

```
>> Dt = np.transpose([[2,-4,6,-10]])
>> Dt
array([[ 2],
       [-4],
       [ 6],
       [-10]])
```

É importante não esquecer de adicionar colchetes a mais, pois se não fizermos isso a representação será de um vetor, não de uma

matriz (vetor multidimensional).

Matriz Nula

Considerando $A=[a_{ij}]$ uma matriz de ordem $m \times n$, temos que a matriz A é nula se e somente se todos os seus elementos forem iguais a zero. Em outras palavras, se $a_{ij}=0$, para todo $i = 1, \dots, m$ e $j = 1, \dots, n$.

Temos a matriz nula definida por:

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} B = [0 \quad 0]$$

No Python temos uma função específica dedicada para fazer uma matriz nula **np.zeros()**:

```
>> N = np.zeros([2,2]) # ou podemos fazer N = np.zeros((2,2))
>> N
array([[0., 0.],
       [0., 0.]])
```

Lembrando que uma matriz quadrada é definida quando temos sempre $m = n$.

Matriz identidade

Como viemos estudando, a depender de como os elementos das matrizes estão organizados e de seus valores temos nomes distintos para representá-las. Por exemplo, um tipo de matriz muito importante é a matriz chamada de identidade.

A matriz identidade é muito utilizada para ajudar a solucionar problemas de sistemas de equações lineares. Iremos estudá-las em capítulos mais à frente. No momento, vamos entender sua representação. Assim, uma matriz identidade é aquela em que a diagonal principal possui todos os valores iguais a 1 e os demais iguais a zero. Veja exemplos abaixo:

$$I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Portanto, para uma matriz identidade do tipo geral: $I_n = (a_{ij})_{n \times n}$, a mesma deve respeitar as seguintes condições abaixo:

$$a_{ij} = \begin{cases} 1, & \text{se } i = j \\ 0, & \text{se } i \neq j \end{cases}$$

No Python, a representação fica determinada a partir da função **np.identity()**. Vaja um exemplo abaixo:

```
>> MI = np.identity(4) # o número 4 dentro do método representa a
dimensão da matriz
>> print('MI = ', MI)
MI = [[1. 0. 0. 0.]
[0. 1. 0. 0.]
[0. 0. 1. 0.]
[0. 0. 0. 1.]]
```

Matriz Transposta

Um tipo de matriz bastante importante que precisamos conhecer é a matriz transposta. Nesse caso, para a transposta de uma matriz A, por exemplo, troca-se ordenadamente as linhas por colunas ou vice-versa.

Denotamos uma matriz transposta de A por A^T . A transposta é uma permutação das linhas com as colunas da matriz A.

Uma observação interessante é que, quando a matriz A for quadrada, teremos a diagonal principal de A e A^T são invariantes, isso mesmo, a **transposição não altera a diagonal principal da matriz quadrada**.

Exemplos:

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 1 & 4 \end{bmatrix}, \text{ então } A^t = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}, B^t = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}$$

Se uma **matriz A** é de ordem $m \times n$, a matriz A^t transposta de A, é de ordem $n \times m$.

Vejamos algumas propriedades importantes da transposta.

Considerem A e B matrizes de ordem $m \times n$ e α pertencente aos reais. Assim:

- (i) $(A + B)^T = A^T + B^T$
- (ii) $(A^T)^T = A$
- (iii) $(\alpha A)^T = \alpha A^T$
- (iv) $(AB)^T = B^T A^T$

Note que a propriedade (iv) expressa que a transposta de um produto é igual ao produto das transpostas, todavia em ordem inversa. Essa propriedade é muito utilizada quando estamos fazendo deduções de teoremas ou simplificações algébricas nas diversas áreas da ciência da computação. Portanto tome nota.

A representação no Python fica:

```
>> A = np.array([[1, 0, 3],[2,1,4]])
>> print('A = ', A)
A = [[1 0 3]
[2 1 4]]
>> At = np.transpose(A)
>> print('At=' , At)
At= [[1 2]
[0 1]
[3 4]]
```

Para obtermos a matriz transposta no Python podemos, também, utilizar um método ‘.T’

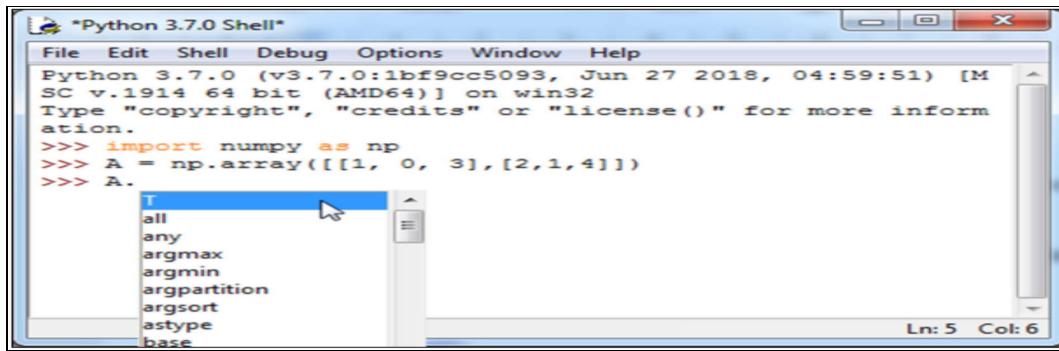
```
>> A = np.array([[1, 0, 3],[2,1,4]])
>> At = A.T # esse .T calcula a transposta imediata da matriz
>> print('At=' , At)
At= ([[1, 2],
 [0, 1],
 [3, 4]])
```

Vejamos a validação de cada uma das propriedades da matriz transposta com o Python:

```
>> A = np.array(([1,2],[3,4]))
>> B = np.array(([5,6],[7,8]))
>> A == B
array([[False, False],
       [False, False]])
>> (A+B).T == A.T + B.T
array([[ True,  True],
       [ True,  True]])
>> (A.T).T == A
array([[ True,  True],
```

```
[ True, True]])
>> (77*A).T == 77*A.T
array([[ True, True],
       [ True, True]])
>> (A*B).T == B.T*A.T
array([[ True, True],
       [ True, True]])
```

Se você estiver no **shell** do IDLE e digitar um ponto ‘.’ após o nome da matriz e apertar a tecla Tab irá ter acesso a um conjunto de métodos para trabalhar dentro da matriz em questão.



3.3. Operações com vetores e escalares

É muito comum termos um conjunto de escalares, um vetor, e precisarmos executar operações de soma, divisão e/ou multiplicação de escalares com esses vetores. Vamos acompanhar o exemplo abaixo

```
>> import numpy as np  
>> a = 2      # escalar  
>> v1 = np.array([1, 2, 3, 4, 5]) # vetor  
>> r1 = v1 + 2  # adiciona o escalar 2 a cada elemento do vetor  
>> print(' r1 = ', r1)  
r1 = [2, 4, 5, 6, 7]
```

Como podemos ver com o Python, podemos fazer subtrações, multiplicações e divisões por escalares, de modo bastante rápido e simplificado. Vejamos outros exemplos:

```
>> b = 3  
>> c = 10  
>> r2 = (v1*b)/c  
>> print('r2=',r2)  
r2 = [0.3, 0.6, 0.9, 1.2, 1.5]
```

Ou seja, podemos executar qualquer tipo de operação com cada um dos elementos de um vetor a partir de um escalar de referência.

3.4. Operações entre Vetores

O Python nos ajuda a trabalhar com estruturas vetoriais. Operações matemáticas entre vetores ficam bastante simplificadas. Vamos supor que precisamos multiplicar cada um dos elementos escalares de um vetor (v_1) pelos respectivos elementos de outro vetor (v_2) e obter assim um vetor v_3 . Vejamos como fica no Python:

```
>> a = 2
>> v1 = np.array([1, 2, 3, 4, 5]) # vetor linha
>> v2 = np.array([2, 3, 5, 1, 4])
>> v3 = v1 * v2
>> v4 = v1 + v2 - a*v1
>> print('v3 = ', v3)
v3 = [2, 6, 15, 4, 20]
>> print('v4 = ', v4)
v4 = [1, 1, 2, -3, -1]
```

Como sabemos, temos com uma matriz uma coleção que condensa vetores preenchidos por escalares. Iremos daqui em diante trabalhar com o estudo das matrizes, pois estas podem ser utilizadas para representar sistemas de equações lineares, por exemplo. Com as equações lineares podemos solucionar uma infinidade de problemas nas áreas de economia, administração, engenharia, medicina etc.

Devido a sua grande importância para solucionar problemas em geral, iremos estudar os principais tipos e propriedades das matrizes utilizando o Python. Vamos a elas.

3.5. Operações entre Matrizes e escalares

É muito comum precisarmos fazer operações entre matrizes e escalares. Por exemplo, multiplicar, somar, subtrair ou dividir todos os elementos de uma matriz por um número. Vejamos como fica no Python:

```
>> c = 2
>> d = 10
>> A = np.array(([1,1,1],[2,2,2],[3,3,3]))
>> B = c*A
>> print('B = ', B)
B = [[2 2 2]
[4 4 4]
[6 6 6]]
>> D = A - c*d
>> print('D=',D)
D= [[-19 -19 -19]
[-18 -18 -18]
[-17 -17 -17]]
```

3.6. Operações entre Matrizes e vetores

Quando formos trabalhar com resolução de sistemas lineares, iremos precisar saber como fazer operações entre matrizes e vetores. Nesse caso de operação precisamos ter cuidado, pois não podemos fazê-la sem que as dimensões estejam adequadas. No caso da multiplicação de um vetor por uma matriz, precisamos que a dimensão do vetor seja compatível com a dimensão da matriz. Para esclarecer melhor isso, veja o exemplo abaixo:

```
>> L = np.array([1,2])
>> A = np.array([1,2,3]) # A é um vetor, temos apenas um colchetes
>> B = np.array(([1,1,1],[2,2,2])) # B é uma matriz
>> C = A*B
>> print('C=',C)
C= [[1 2 3]
 [2 4 6]]
```

No exemplo acima, podemos perceber que cada um dos elementos do vetor A foi multiplicado pelos elementos das linhas da matriz B.

```
>> F = L*A # aqui iremos ter um erro de compilação!
```

O erro de compilação é proveniente das dimensões do vetor L com relação a matriz A. O vetor L tem duas dimensões e, portanto, dois valores de escalares. Por sua vez, estes não encontram correspondentes com a matriz A que em cada linha possui dimensão três ($n=3$).

```
>> B = np.array(([1,1,1],[2,2,2]))
>> A = np.array([1,2,3])
>> G = B - A
>> print('G=',G)
G= [[ 0 -1 -2]]
```

```
[ 1 0 -1]]  
>> GI = A - B  
>> print('GI = ', GI)  
GI= [[ 0  1  2]  
     [-1  0  1]]  
>> D = A / B # divisão dos elementos  
>> print('D=',D)  
D= [[1.  2.  3.]  
    [0.5 1.  1.5]]
```

Tente entender com cuidado o exemplo acima antes de passar para o próximo tópico.

3.7. Operações entre Matrizes

Agora chegou a hora de estudar as operações entre matrizes. Esta é a parte mais interessante, pois a resolução dos sistemas lineares vai depender fortemente do entendimento de como funcionam as operações entre matrizes.

Aqui, também precisamos garantir consistência de dimensões entre as matrizes para que as operações matemáticas aconteçam corretamente.

Vamos estudar a soma, subtração e divisão de matrizes. Para isso precisamos que as matrizes tenham a mesma dimensão. Isso porque as operações se darão elemento a elemento.

```
>> A = np.array(([1,1],[2,2]))  
>> B = np.array(([20,20],[40,40]))  
>> C = np.array(([1,2,3],[4,5,6]))  
>> S = A + B # podemos fazer o mesmo com subtrações  
>> print('S=',S)  
S= [[21 21]  
     [42 42]]  
>> D = A/B  
>> print('D=',D)  
D= [[0.05 0.05]  
     [0.05 0.05]]  
>> M = A*B # atenção: aqui não é o produto entre matrizes (este  
iremos estudar mais a frente), mas sim entre os elementos
```

Para o caso de multiplicação dos elementos entre as matrizes podemos utilizar também a função `np.multiply()`.

```
>> M = np.multiply(A,B) # possui o mesmo resultado de que A*B
```

Quando formos fazer operações de multiplicação de matrizes, teremos um processo diferenciado e pouco intuitivo. Portanto, é bom prestar bastante atenção no funcionamento do produto entre matrizes.

Só podemos fazer o produto entre matrizes quando o número de colunas da primeira matriz for igual ao número de linhas da segunda. Isso implica que a ordem dos produtos entre as matrizes importa. Não temos comutatividade, ou seja, $M1 \cdot M2$ é diferente de $M2 \cdot M1$. Então, muita atenção.

Para fazermos o produto entre matrizes devemos utilizar a função `np.dot(m1, m2)`.

A figura abaixo mostra as condições e resultado final das dimensões das matrizes no processo de multiplicação.

$$A_{m \times n} \times B_{n \times p} = (AB)_{m \times p}$$

Primeiro vamos entender o que essa função faz. Cada um dos elementos das linhas da primeira matriz deve ser multiplicado e somado pelos correspondentes elementos da coluna da segunda matriz (por isso que devemos ter a mesma dimensão). Devemos fazer esse processo linha a linha com suas colunas correspondentes. Veja a figura esquemática abaixo, que mostra todo o processo da multiplicação de uma matriz A(3x2) por uma B(2x3).

$$\begin{bmatrix} 2 & 1 \\ 5 & 3 \\ 4 & 2 \end{bmatrix} \times \begin{bmatrix} 3 & 1 & 0 \\ 4 & 2 & 1 \end{bmatrix} \quad \text{multiplicar linhas por colunas}$$

$$= \begin{bmatrix} 2 \cdot 3 + 1 \cdot 4 \\ 5 \cdot 3 + 3 \cdot 4 \\ 4 \cdot 3 + 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 + 1 \cdot 2 & 2 \cdot 0 + 1 \cdot 1 \\ 5 \cdot 1 + 3 \cdot 2 & 5 \cdot 0 + 3 \cdot 1 \\ 4 \cdot 1 + 2 \cdot 2 & 4 \cdot 0 + 2 \cdot 1 \end{bmatrix}$$

$$\begin{bmatrix} 6 + 4 & 2 + 2 & 0 + 1 \\ 15 + 12 & 5 + 6 & 0 + 3 \\ 12 + 8 & 4 + 4 & 0 + 2 \end{bmatrix} = \begin{bmatrix} 10 & 4 & 1 \\ 27 & 11 & 3 \\ 20 & 8 & 2 \end{bmatrix}$$

No Python temos:

```
>> A = np.array(([2,1],[5,3],[4,2]))  
>> B = np.array(([3,1,0],[4,2,1]))  
>> R = np.dot(A,B) # multiplicação entre matrizes  
R= [[10 4 1]  
[27 11 3]  
[20 8 2]]  
>> R2 = np.dot(B,A)  
>> print('R2=',R2)  
R2= [[11 6]  
[22 12]]
```

Perceba a diferença entre: $A \times B \neq B \times A$

```
>> T = A*B # teremos um erro de compilação, pois as dimensões  
não conferem para multiplicação elemento a elemento
```

Temos, abaixo, uma multiplicação bem interessante de uma matriz linha por uma matriz coluna com dimensões compatíveis claro:

```
>> M1 = np.array([[1,2,3,4]])  
>> M2 = np.array([[20,30,40,50]])  
>> M3 = M1*M2  
M3= [[ 20 60 120 200]]  
>> M4 = np.dot(M1,M2) # aqui teremos um erro de compilação pois  
o número de linhas de M1 é diferente do número de colunas de M2.
```

Entretanto podemos fazer a seguinte operação:

```
>> M4 = np.dot(np.transpose(M1),M2) # np.transpose(M1) - faz com  
que o número de linhas de M1 fique igual ao número de colunas de  
M1  
>> print('M4=',M4)
```

```
M4= [[ 20 30 40 50]
[ 40 60 80 100]
[ 60 90 120 150]
[ 80 120 160 200]]
```

Traço da matriz

Por vezes, para compreender alguns teoremas, iremos precisar entender o que é o traço de uma matriz. Em álgebra linear, o traço de uma matriz quadrada é definido como a soma dos elementos da sua diagonal principal. Nos casos onde a matriz não é quadrada teremos a soma dos elementos que fazem parte de uma diagonal principal referente a uma matriz quadrada. Veja o exemplo abaixo:

Assim se tivermos $A = [a_{ij}]$, então o traço é dado por:

$$\text{tr}(A) = a_{11} + a_{22} + \dots + a_{nn}.$$

```
>> A = np.array(([1,1],[2,2]))
>> B = np.array(([1,1,1],[3,3,3]))
>> tr_A = np.trace(A) # o resultado será a soma de 1 com 2
>> print('tr_A=', tr_A )
tr_A= 3
>> tr_B = np.trace(B) # o resultado será a soma de 1 com 3
tr_B= 4
```

Agora vamos conhecer algumas propriedades interessantíssimas do traço de uma matriz.

- $\text{tr}(A+B) = \text{tr}(A) + \text{tr}(B)$
- $\text{tr}(\lambda * A) = \lambda * \text{tr}(A)$
- O traço de uma matriz quadrada é igual ao traço de sua transposta. $\text{tr}(A) = \text{tr}(A^T)$

- $\text{tr}(AB) = \text{tr}(BA)$ # o traço de um produto de matrizes quadradas não depende da ordem do produto

Por fim, mas não menos importante, é a propriedade seguinte. Todavia, iremos entendê-la melhor nos próximos capítulos:

- O traço de uma matriz simétrica é igual à soma dos seus valores próprios (autovalores).

Agora vejamos a validação dessas propriedades com o Python:

```
>> A = np.array(([1,1],[2,2]))
>> B = np.array(([3,3],[4,4]))
>> tr_AB = np.trace(A*B)
>> print('tr_AB=',tr_AB)
tr_AB = 11
>> tr_BA = np.trace(B*A)
>> print('tr_BA=',tr_BA)
tr_BA = 11
>> tr_A_mais_B = np.trace(B+A)
>> print('tr_A_mais_B=',tr_A_mais_B)
tr_A_mais_B=10
>> np.trace(B)+np.trace(A)
10
>>np.trace(A)
3
>>np.trace(np.transpose(A))
3
```

3.8. Inversa de uma matriz

Obter a inversa de uma matriz é uma das operações mais importantes e utilizadas na computação científica. São muitas as aplicações que utilizam matrizes inversas. Uma aplicação bastante interessante utilizando esse tipo de matriz é apresentada no último capítulo do livro. Nesse exemplo, veremos como as matrizes inversas podem ser utilizadas na criptografia de mensagens.

Mas, por hora, vamos a definição de uma matriz inversa:

Seja A uma matriz $m \times n$, esta matriz é dita invertível ou não singular se existe uma matriz B que satisfaça a seguinte condição abaixo:

$$AB = BA = I_n$$

(I_n é a matriz identidade)

Nesse caso a matriz B é uma inversa multiplicativa de A . Caso B não exista dizemos que A é singular ou não invertível.

A biblioteca **NumPy** é realmente bastante completa e, portanto, temos uma função especial para encontrar a matriz inversa.

Algo muito útil é que dentro da NumPy temos um pacote de métodos disponíveis para trabalhar exclusivamente com álgebra linear. Esse pacote se chama (**numpy.linalg**). Dentro dele podemos encontrar a função para calcular a inversa de uma matriz (chamada de **inv()**).

Sem mais delongas, vamos a prática com o Python:

```
>> import numpy as np  
>> from numpy.linalg import inv  
>> A = np.array(([1,2],[3,4]))  
>> B = inv(A)
```

```
>> A*B # cuidado esse não é o tipo de multiplicação de matrizes,  
mas sim dos elementos
```

```
array([[-2. ,  2. ],  
      [ 4.5, -2. ]])
```

Devemos fazer o seguinte para verificar B como inversa da matriz A:

```
>> np.dot(B,A)  
array([[1.00000000e+00, 0.00000000e+00],  
      [2.22044605e-16, 1.00000000e+00]])
```

ou

```
>>B.dot(A)  
array([[1.00000000e+00, 0.00000000e+00],  
      [2.22044605e-16, 1.00000000e+00]])
```

Vamos agora às propriedades da matriz inversa:

i) Caso A seja uma matriz inversa, então A^{-1} também é invertível:

$$(A^{-1})^{-1} = A$$

```
>> A = np.array(([1.,2.],[3.,4.]))  
>> inv(inv(A)) == A # se fizermos isso obteremos falso. Isso por  
motivos de arredondamento e tipo de variável
```

```
array([[False, False],  
      [False, False]])
```

entretanto, sabemos que:

```
>> inv(inv(A)) # igual a A  
array([[1., 2.],  
      [3., 4.]])
```

ii) Caso A e B forem matrizes invertíveis, então AB também é invertível

$$(AB)^{-1} = B^{-1}A^{-1}$$

```
>> B = inv(A)
>> inv(A.dot(B))
array([[ 1.0000000e+00,  0.0000000e+00],
       [-8.8817842e-16,  1.0000000e+00]])
>> inv(B).dot(inv(A))
array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+00]])
```

iii) Se A é uma matriz invertível, então a seguinte relação é válida:

$$(A^T)^{-1} = (A^{-1})^T$$

```
>> inv(A.T)
array([[-2. ,  1.5],
       [ 1. , -0.5]])
>> (inv(A)).T
array([[-2. ,  1.5],
       [ 1. , -0.5]])
```

Podemos expandir tudo o que foi apresentado para uma matriz qualquer ($m \times n$), desde que o computador tenha velocidade de processamento e memória mínima para executar as operações. O bom de tudo isso é que é muito prático e intuitivo aprender álgebra linear com Python.

3.9. Manipulando elementos da matriz

Às vezes queremos fazer manipulações com parte ou elementos específicos de uma matriz. O Python permite fazer a extração de elementos de uma maneira bastante intuitiva. Vamos estudar essas questões utilizando a seguinte matriz como referência:

```
>> A = np.array(([1,2,3,4],[5,6,7,8],[9,10,11,12]))  
>> A  
array([[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12]])
```

A referênciação de linhas e colunas para o Python começa a partir do número zero. Em outras palavras, o elemento da linha 1 e coluna 1 apresenta referênciação de $A[0,0]$. Então se quisermos obter o elemento $A(1,1)$ devemos fazer o seguinte:

```
>> a11 = A[0,0]  
>> a11  
1
```

Se quisermos extrair a primeira linha inteira da matriz A devemos utilizar o operador ‘:’ e fazer o seguinte:

```
>> A1n = A[0,:] # o índice para primeira linha com  
>> print('A1n=',A1n)  
A1n = [5 6 7 8]
```

Para coletar a segunda coluna inteira da matriz A, temos:

```
>> Am2 = A[:,1]
```

```
>> print(Am2 ='Am2)
```

```
Am2 = [ 2, 6, 10]
```

Capítulo 4

4. Sistemas de Equações Lineares

Os fenômenos naturais são predominantemente não lineares, no entanto podemos representá-los a partir de sistemas lineares fazendo as devidas considerações e simplificações. O importante é que tenhamos ao menos algo que possa representar o fenômeno de interesse em algum modelo matemático que possamos solucionar sem grandes dificuldades.

Sistemas lineares são ótimos para modelar e simplificar problemas do nosso dia a dia. Podemos utilizar os sistemas lineares nas mais diversas áreas do conhecimento humano. Atualmente os sistemas lineares vem ganhando destaque graças aos diversos ramos da inteligência artificial como o *machine learning* e *deep learning*.

Por exemplo, as redes neurais artificiais (RNAs), muito utilizadas para aprendizado de máquinas e sistemas autônomos, são representadas por sistemas lineares.

Caso o leitor se interesse em conhecer mais a respeito dessas áreas pode procurar por livros do mesmo autor na série cientista de dados – analista quant.

Representação matemática

Um conjunto de equações lineares devem respeitar a seguinte estrutura matemática:

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{array} \right.$$

com a_{ij} , $1 \leq i \leq m$, $1 \leq j \leq n$, números reais ou complexos.

Uma solução para o sistema acima é um vetor (ou matriz linha ou coluna) dos elementos (x_1, x_2, \dots, x_n) que satisfaz simultaneamente as 'm' equações.

A equivalência entre dois sistemas lineares é admitida quando a solução para um sistema é exatamente a mesma para o outro.

Podemos facilmente representar um sistema linear através de matrizes, fazendo como mostra abaixo:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Temos:

$$A^*X = B$$

$$AX = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix}$$

A matriz A é chamada de matriz dos coeficientes do sistema linear. Por outro lado, a matriz coluna X é chamada de matriz das variáveis. E B é a matriz dos termos independentes.

Chegamos a algo muito interessante, pois podemos solucionar sistemas de equações lineares a partir de operações manipulando as matrizes. Esse é um tipo de tecnologia matemática (algébrica) muito importante.

Vamos a um exemplo prático. Precisamos encontrar os valores das variáveis x_1 e x_2 que respeite simultaneamente as igualdades do sistema linear abaixo:

$$\begin{cases} -x_1 + 2x_2 = -1 \\ 2x_1 + x_2 = 5 \end{cases}$$

Assim podemos representar matricialmente o sistema acima da seguinte maneira:

$$\begin{bmatrix} -1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 5 \end{bmatrix}$$

Para solucionar esse problema em Python temos algumas bibliotecas para lidar com solução de sistemas lineares. No entanto, podemos utilizar os conceitos de matrizes estudados até o presente momento e resolver o sistema. De qualquer maneira, vamos fazer a solução do sistema utilizando conceitos básicos e bibliotecas especializadas.

No nosso exemplo, devemos ter um sistema de equações lineares em que o número de equações é igual ao número de incógnitas.

Repare o seguinte sistema geral onde o número de equações é igual ao número de incógnitas:

$$\begin{bmatrix} a_{11} + \dots + a_{1n} \\ \vdots \\ a_{n1} + \dots + a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Para a solução adequada do sistema devemos admitir que o **det(A)** é diferente de zero e, portanto, a matriz A possui inversa A^{-1} . Iremos estudar com mais detalhes a respeito dos determinantes de uma matriz no próximo capítulo.

Assim temos:

$$A^{-1}(AX) = A^{-1}B$$

$$(A^{-1}A)X = A^{-1}B$$

$$I_n X = A^{-1}B$$

$$X = A^{-1}B$$

Na forma matricial temos:

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}^{-1} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Portanto, como já sabemos obter uma matriz inversa fica fácil solucionar o sistema utilizando o NumPy.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} -1 \\ 5 \end{bmatrix}$$

```
>> import numpy as np
>> from numpy.linalg import inv
>> A = np.array([-1,2],[2,1])
>> B = np.array([-1],[5])
>> det_A = np.linalg.det(A) # se diferente de zero
>> print('det_A=',det_A)
det_A = -5.000000000000001 (portanto diferente de zero!)
>> A_inv = inv(A)
>> X = A_inv.dot(B)
>> print('X=',X)
X= [[2.2]
[0.6]]
```

Se o determinante da matriz dos coeficientes pelo método (`np.linalg.det()`) for igual a zero, a matriz é singular, isto é, não possui inversa. Assim o método ‘`inv()`’ irá retornar um erro

Veja o exemplo abaixo:

```
>> S = np.array([1,2],[2,4])
>> det_S = np.linalg.det(S)
>> print('det_S=',det_S)
det_S=0.0
>> inv_S = inv(S) # teremos um erro LinAlgError()
```

`numpy.linalg.LinalgError: Singular matrix`

Podemos utilizar o mesmo raciocínio para um sistema de equações lineares com muito mais equações e incógnitas. Existe uma maneira mais rápida e eficiente de resolver sistemas lineares com o NumPy utilizando o método ‘**solve()**’. Veja, abaixo, como funciona:

```
>> import numpy as np  
>> A = np.array(([[-1,2],[2,1]]))  
>> B = np.array(([[-1],[5]]))  
>> X = np.linalg.solve(A,B)  
>> print('X=',X)  
X= [[2.2]  
[0.6]]
```

Caso você precise obter alguma solução de sistemas lineares em seus projetos pessoais, é aconselhável utilizar a função **solve()**. Isso porque ela possui otimizações em seu algoritmo fazendo com que os cálculos para sistemas grandes sejam mais eficientes.

Capítulo 5

5. Determinantes de Matrizes

Tanto no ocidente quanto no oriente havia uma indignação de como a partir da utilização de matrizes poderíamos solucionar sistemas de equações lineares. Os orientais foram os primeiros a se questionar a respeito de como simplificar e facilitar a solução de sistemas de equações. Entretanto, no ocidente esse assunto só veio ganhar destaque a partir do século XVII.

Foi assim que surgiram os trabalhos de Leibniz (1646-1716) e G. Cramer (1704-1752). Um trabalho muito importante publicado em 1750, por Cramer, propunha a solução de sistemas de equações através de determinantes. Outros estudiosos contemporâneos de Cramer, como C. Machlaurin (1698-1746) e J.L. Lagrange (1736-1813) provavelmente já conheciam as técnicas publicadas por ele.

Os estudos de A.L. Cauchy no século XIX fizeram com que os determinantes ganhassem notoriedade frente a solução de sistemas de equações. Então, vamos estudar os principais conceitos por trás dos determinantes.

Determinantes

Afinal, o que vem a ser um determinante de uma matriz? Para responder, vamos ao seguinte problema: precisamos encontrar o valor de x da equação: $ax=b$ com ‘ a ’ diferente de zero. A solução é $x = b/a$.

Observe que o denominador se refere a matriz dos coeficientes da equação.

Vamos agora expandir para um sistema de equações lineares e procurar a solução do problema. Primeiro, analise o sistema de equações lineares abaixo:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases}$$

Estamos à procura de x_1 e x_2 . Para isso podemos isolar um desses valores e substituir na outra equação. Por exemplo, vamos isolar x_2 da primeira equação e substituir na segunda equação:

$$x_2 = \frac{b_1 - a_{11}x_1}{a_{12}}$$

Substituindo a equação acima na segunda equação do sistema, temos:

$$a_{12}x_1 + a_{22} \frac{b_1 - a_{11}x_1}{a_{12}} = b_2$$

Agora, isolando x_1 podemos encontrar:

$$x_1 = \frac{b_1 a_{22} - b_2 a_{12}}{a_{11} a_{22} - a_{12} a_{21}}$$

Executando raciocínio semelhantes para o x_2 chegaremos a:

$$x_2 = \frac{b_2 a_{11} - b_1 a_{21}}{a_{11} a_{22} - a_{12} a_{21}}$$

Podemos notar que o denominador das equações de x_1 e x_2 são os mesmos ($a_{11}a_{22}-a_{12}a_{21}$). Também esse denominador é formado apenas pelos termos da matriz dos coeficientes da equação do sistema linear. Podemos agora fazer um paralelo com o primeiro exemplo da equação simples ($ax=b$) mostrada no início do capítulo.

Contudo, o termo do denominador fica constante (imutável), para um determinado tamanho de sistema, independe do número de variáveis e equações do sistema. Veja um exemplo para um sistema 3×3 :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \end{cases}$$

Indo à procura das variáveis x_1 , x_2 e x_3 de maneira semelhante como fizemos para o sistema 2×2 , encontraremos também um denominador constante igual a $(a_{11}a_{22}a_{33} - a_{11}a_{21}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{12}a_{22}a_{31})$ que está associado a uma matriz dos coeficientes do sistema:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Essas constantes que aparecem no denominador das matrizes (que precisam ser quadradas) são chamadas de **determinantes** de uma matriz. Agora precisamos aprender como calcular o determinante de maneira mais prática. Ainda bem que vários estudiosos já fizeram o trabalho árduo de descobrir como fazer isso. Precisamos apenas entender a descoberta.

5.1. Calculando o Determinante

Existem diversas maneiras para calcular o determinante de uma matriz quadrada, ou seja, encontrar um número constante associado à matriz dos coeficientes de um sistema de equações. Aqui, apenas para fins de conhecimento, iremos aprender a principal chamada de regra de **Sarrus** (serve apenas para matrizes 3x3).

Nos dias atuais, felizmente não precisamos encontrar o determinante na mão (com papel e lápis). Temos o Python que, com apenas um comando, executa essa trabalheira para nós. Contudo, é importante saber o que está sendo feito por trás dos bastidores. Então vamos nessa.

Seja uma matriz quadrada $A [a_{ij}]$. O determinante dessa matriz é, principalmente, representado por: $\det A$ ou $|A|$.

O determinante de uma matriz com 1×1 (uma coluna e uma linha, ou seja, um escalar) é o próprio escalar:

$$\det[a] = a$$

O cálculo do determinante de uma matriz 2×2 é bastante simples dado pela subtração do produto entre a diagonal principal com a diagonal secundária como mostra abaixo:

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

Diagonal secundária Diagonal principal

Para uma matriz 3×3 o determinante pode ser obtido a partir da equação abaixo:

$$\begin{aligned} \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} &= \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \\ &= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} \\ &\quad - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} \end{aligned}$$

Felizmente, não precisamos decorar o determinante mostrado na matriz acima. Existe uma regra ou método, chamado de Sarrus, utilizado para calcular o determinante de uma matriz de ordem 3x3.

Para aplicar o método, primeiro deveremos repetir ao lado da matriz as duas primeiras colunas (como mostra a figura abaixo). Depois devemos multiplicar os elementos da diagonal principal e somá-los subtraindo a multiplicação dos elementos da diagonal secundária.

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}$$

Vamos fazer um exemplo prático manual para entender a aplicação do método de Sarrus e logo em seguida utilizando o Python.

Seja a matriz 3x3 dada por:

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 5 & 3 & -1 \\ 7 & 2 & 0 \end{bmatrix}$$

Vamos encontrar o determinante a partir do método de Sarrus. Como dito antes, vamos repetir as duas primeiras colunas e somar os produtos da diagonal principal e subtrair os produtos da diagonal secundária. Ver cálculos na figura abaixo:

$$\left| \begin{array}{ccc|cc} 1 & 2 & 4 & 1 & 2 \\ 5 & 3 & -1 & 5 & 3 \\ 7 & 2 & 0 & 7 & 2 \end{array} \right|$$

$$= 1 * 3 * 0 + 2 * (-1) * 7 + 4 * 5 * 2$$

$$- 4 * 3 * 7 - 1 * (-1) * 2 - 2 * 5 * 0$$

Após fazer as contas chegamos em $\det(A) = -56$.

Vamos agora ver como obter o determinante de uma matriz utilizando o NumPy do Python.

```
>> import numpy as np
>> from numpy.linalg import det
>> A = np.array(([1,2,4],[5,3,-1],[7,2,0]))
>> det_A = det(A)
>> print('det_A=',det_A)
det_A = -56.00000000000002
```

Podemos utilizar outros teoremas para obter o determinante de uma matriz como por exemplo o teorema de Laplace, decomposição LU, eliminação gaussiana dentre outros. O teorema de Laplace permite obter o determinante de matrizes maiores do que 3x3. Antes dos computadores pessoais não serem barateados e popularizados era preciso muitas vezes fazer os cálculos determinantes com o método de Laplace. Atualmente, basta sabermos que existe e como funciona na teoria, porque na prática sempre iremos fazer uso de linguagens de programação.

Vamos a outro exemplo.

```
>> B = np.array(([1,2,4,5,6,7],[5,3,-1,5,1,2],[7,2,0,1,4,9]))
>> B
array([[ 1,  2,  4,  5,  6,  7],
       [ 5,  3, -1,  5,  1,  2],
```

```
[ 7, 2, 0, 1, 4, 9]])  
>> det(B)  
numpy.linalg.LinAlgError: Last 2 dimensions of the array must  
be square
```

Perceba que o determinante só pode ser calculado em uma matriz quadrada. Vamos completar a matriz B para que ela se torne quadrada e possamos efetuar o determinante.

```
>> C = np.array(([1,2,4,5,6,7],[5,3,-1,5,1,2],[7,2,0,1,4,9],[1,2,1,5,6,5],  
[-7,2,4,-1,6,6],[0,2,5,5,9,7]))  
>> C  
array([[ 1,  2,  4,  5,  6,  7],  
       [ 5,  3, -1,  5,  1,  2],  
       [ 7,  2,  0,  1,  4,  9],  
       [ 1,  2,  1,  5,  6,  5],  
       [-7,  2,  4, -1,  6,  6],  
       [ 0,  2,  5,  5,  9,  7]])  
>> det(C)  
8184.000000000006
```

Para podemos resolver o determinante da matriz C, acima, deveríamos utilizar algum método mais sofisticado como o de Laplace. Esse método é bastante exaustivo para ser efetuado na mão em uma matriz 6x6 como é o caso apresentado. Entretanto com o NumPy em questão de segundos resolvemos o problema.

Capítulo 6

6. Autovalores e Autovalores

Um dos assuntos mais curiosos e importantes de serem estudados na álgebra linear são os autovalores e autovetores. As aplicações utilizando esses conceitos trafegam por todas as áreas científicas e tecnológicas. Antes vamos a uma breve apresentação do que são transformações lineares.

6.1. Transformação Linear

Uma maneira de representar algum tipo de dependência entre variáveis é a partir de funções lineares. Muitos problemas tecnológicos podem ser representados por tais funções. Por exemplo, para cada km rodado o aplicativo Uber cobra R\$ 1,20. Portanto, após percorrer X km teremos o custo da corrida valendo: $C(x) = 1,2 \cdot x$.

Vamos a outro exemplo: Supondo que o percentual de assinantes que opinam sobre o que acharam dos filmes assistidos no Netflix respeita a tabela abaixo:

Opiniões	Comédia	Ação	Drama	Terror	Séries
	0,3	0,4	0,8	0,1	0,7

Assim, para cada 100 pessoas que assistem algum filme de comédia, 30 delas deixam suas opiniões com relação ao filme assistido. Para as séries, em cada 100 pessoas que assistem 70 delas deixam sua opinião e assim por diante.

Portanto, se quisermos saber o número de opiniões dadas na plataforma Netflix para cada uma das 5 categorias da tabela acima, basta resolver a seguinte função linear:

$$Q = [0.3 \ 0.4 \ 0.8 \ 0.1 \ 0.7] \begin{bmatrix} c \\ a \\ d \\ t \\ s \end{bmatrix} = 0.3c + 0.4a + 0.8d + 0.1t + 0.7s$$

Onde c, a, d, t e s são os números de assinantes Netflix que assistem, respectivamente, as categorias de comédia, ação, drama, terror e séries.

Portanto, Q é uma função de transformação linear $R^4 \rightarrow R$, assim é possível saber o número de comentários médio em, por exemplo, um dia.

6.2. Autovalores e autovetores de uma Matriz

Após essa breve explicação de transformação linear vamos ao conceito que origina a ideia de autovalores e autovetores.

Considere uma transformação linear de um espaço vetorial $T:V \rightarrow V$. Estamos interessados em saber quais vetores são levados em um múltiplo de si mesmo, em outras palavras estamos procurando um vetor (v) e um escalar λ (lambda) tais que:

$$T(v) = \lambda * v$$

Portanto, $T(v)$ será um vetor de mesma direção que v . E estamos interessados em saber um $v \neq 0$ que satisfaça a equação acima. O escalar λ é chamado de autovalor (também chamado de valor próprio ou característico) de T e o vetor v um autovetor (vetor próprio ou característico) de T .

No capítulo de aplicações você encontrará exemplos práticos do uso de autovalores e autovetores.

Como calcular

Vamos conhecer como calcular os autovalores e autovetores de uma matriz. Considerando a matriz quadrada, A , de ordem n , os autovalores e autovetores de A estarão associados a matriz A em relação à base canônica, isto é, $TA(v) = A*v$ (forma canônica).

Desse modo, um autovalor λ pertencente aos reais e um autovetor v pertencente ao R^n , são soluções da equação $A/v = \lambda *v$, com $v \neq 0$.

$$A = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix}$$

E dados os vetores da base canônica formadores do R^n $e_1 = (1, 0, \dots, 0)$, $e_2 = (0, 1, 0, 0, \dots, 0)$, ..., $e_n = (0, 0, \dots, 0, 1)$, temos que:

$$A * e_1 = \begin{bmatrix} a_{11} \\ 0 \\ \vdots \\ 0 \end{bmatrix} = a_{11} e_1$$

$$A * e_2 = \begin{bmatrix} 0 \\ a_{22} \\ \vdots \\ 0 \end{bmatrix} = a_{22} e_2$$

e em geral temos: $A^*e_i = a_{ii}e_i$. Assim, os vetores da base canônica do R^n são autovetores para A , e o autovetor e_i é associado ao autovalor a_{ii} .

Temos uma maneira geral bastante interessante para encontrar os autovetores e autovalores a partir do chamado polinômio característico. Vejamos o exemplo abaixo:

$$A = \begin{bmatrix} 4 & 2 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

Estamos à procura dos autovetores e autovalores, isto é, queremos: v pertencente ao R^3 e escalares λ pertencente ao R tais que $A^*v=\lambda^*v$. Iremos fazer uso da matriz identidade I (3x3) para

reescrever a equação acima da seguinte maneira: $Av = (\lambda^*I)v$, ou ainda, $(A-\lambda^*I)v=0$. Assim nosso exemplo fica:

$$\left(\begin{bmatrix} 4 & 2 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix} - \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Organizando os termos, chegamos na seguinte equação:

$$\begin{bmatrix} 4 - \lambda & 2 & 0 \\ -1 & 1 - \lambda & 0 \\ 0 & 1 & 2 - \lambda \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Podemos reescrever a matriz acima em um sistema de três equações com três incógnitas. Uma propriedade muito importante é que, se o determinante da matriz dos coeficientes for diferente de zero, o sistema terá solução única com todos os termos $x=y=z=0$ (zero). Essa é uma propriedade que devemos verificar com cuidado. No entanto, estamos interessados em calcular os autovetores de A , isto é $v \neq 0$, assim queremos $(A-\lambda I)v=0$; Portanto, queremos que o determinante da matriz dos coeficientes seja igual a zero:

$$\det(A-\lambda^*I) = 0$$

$$\det \begin{bmatrix} 4 - \lambda & 2 & 0 \\ -1 & 1 - \lambda & 0 \\ 0 & 1 & 2 - \lambda \end{bmatrix} = 0$$

Como a matriz é 3x3 podemos utilizar a regra de Sarrus apresentada no capítulo de determinantes e encontrar o seguinte polinômio:

$$-\lambda^3 + 7\lambda^2 - 16\lambda + 12 = 0 \text{ (conhecido como polinômio característico de } A)$$

Podemos simplificar o polinômio acima da seguinte maneira:

$$(\lambda-2)^2(\lambda-3) = 0$$

E a partir de então encontrar as seguintes raízes da equação: $\lambda=2$ e $\lambda=3$ que são os autovalores da matriz A. Uma vez conhecendo os autovalores podemos encontrar os autovetores correspondentes resolvendo a seguinte equação: $Av=\lambda v$, para $\lambda=2$ e $\lambda=3$:

1) $\lambda=2$

$$\begin{bmatrix} 4 & 2 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = 2 \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{cases} 4x + 2y = 2x \\ -x + y = 2y \\ y + 2z = 2z \end{cases}$$

A solução do sistema mostra que a partir da terceira equação acima $y=0$ e, portanto, $x=0$ a partir da segunda equação. Assim teremos a variável z sem nenhuma restrição fazendo com que para o autovalor $\lambda=2$ os autovetores associados são do tipo $(0, 0, z)$. O vetor $(0, 0, z)$ está associado a um subespaço vetorial $(0, 0, 1)$.

2) $\lambda=3$

Devemos resolver agora a equação $Av=3v$:

$$\begin{cases} 4x + 2y = 3x \\ -x + y = 3y \\ y + 2z = 3z \end{cases}$$

Do sistema acima podemos facilmente encontrar que $x=-2y$ e $z=y$. Assim, os autovetores associados ao autovalor $\lambda=3$ são do tipo

$(-2y, y, y)$, ou seja, pertencem ao subespaço $\{(-2, 1, 1)\}$.

Este exemplo foi retirado do livro Álgebra Linear de Boldrini e Figueiredo 3^a ed. Editora Harbra^[2]. Caso o leitor queira se aprofundar no assunto poderá consultar tal referência e detalhar os estudos.

Vamos utilizar o NumPy para obter os autovalores e autovetores associados:

```
>> import numpy as np
>> import numpy.linalg as al
>> A = np.array(([4,2,0],[-1,1,0],[0,1,2]))
>> autovalores, autovetores = al.eig(A)
>> autovalores
array([2., 2., 3.])
>>
autovetores
array([[ 0.00000000e+00,  4.44089210e-16, -8.16496581e-01],
       [ 0.00000000e+00, -4.44089210e-16,  4.08248290e-01],
       [ 1.00000000e+00,  1.00000000e+00,  4.08248290e-01]])
```

O método ‘**numpy.linalg.eig**’ retorna dois valores: um primeiro referente aos autovalores (não necessariamente ordenados) e o segundo uma matriz contendo em suas colunas os autovetores correspondentes aos autovalores.

Portanto, conhecemos de maneira breve o que são os autovalores e autovetores. Certamente se você for para o mundo de mineração e análise de dados irá, constantemente, se deparar com esses conceitos. Assim é importante conhecer os fundamentos antes de sair utilizando os métodos do NumPy ou diversas outras bibliotecas e pacotes.

Capítulo 7

7. Aplicações

Mesmo nos anos 250 A.C. havia exemplos com resolução de sistemas de equações utilizando matrizes. Nos dias atuais são praticamente infinitas as aplicações existentes que utilizam conceitos de álgebra linear. Iremos fazer uma comparação injusta, porém apropriada para os dias atuais. Se fossemos eleger para a importância nas aplicações em computação científica entre cálculo (diferencial e integral) e álgebra linear, de longe os assuntos de álgebra linear são os mais importantes. Muitos dos processos e sistemas a nossa volta utilizam procedimentos algébricos para acomodar diversos níveis da problemática tecnológica.

Agora, iremos estudar aplicações práticas dos principais elementos estudados até aqui.

7.1. Exemplo 1 - Matrizes e Estatísticas

Supondo que somos o coordenador de um curso de ciências da computação e estamos avaliando as notas em várias disciplinas para quatro turmas da faculdade. A tabela abaixo apresenta as médias das notas obtidas por disciplina em cada uma das quatro turmas.

	Lógica	Cálculo	Python	Inglês
Turma A	6	5	9	6
Turma B	8	9	8	7
Turma C	9	9	8	6
Turma D	7	8	6	9

Temos uma tabela com quatro linhas e quatro colunas. A representação da tabela em Python fica da seguinte maneira:

```
>> import numpy as np  
>> T = np.array(([6,5,9,6],[8,9,8,7],[9,9,8,6],[7,8,6,9]))  
array([[6, 5, 9, 6],  
       [8, 9, 8, 7],  
       [9, 9, 8, 6],  
       [7, 8, 6, 9]])
```

Vamos tirar a média das notas obtidas nas 4 turmas por disciplinas. Como já aprendemos a extrair partes de uma matriz (colunas ou linhas) fica fácil a partir dos métodos presentes no NumPy encontrar o que desejamos. Precisamos armazenar em uma variável a média de cada uma das colunas da matriz em questão e logo em seguida aplicar o método ‘mean()’. Como mostra abaixo, precisamos manter as colunas fixas e coletar todas as linhas.

```
>> med_logica = T[:,0].mean()  
>> med_calculo = T[:,1].mean ()  
>> med_python = T[:,2].mean ()
```

```
>> med_ingles = T[:,3].mean ()
```

Podemos calcular a média de todas as notas das disciplinas por turmas. Agora precisamos manter as linhas fixas e coletar todas as colunas.

```
>> med_A = T[0,:].mean()  
>> med_B = T[1,:].mean()  
>> med_C = T[2,:].mean()  
>> med_D = T[3,:].mean()
```

Podemos obter o desvio-padrão das médias das notas a partir da função **.std()**. Note o quanto prático e simples, com o NumPy, é fazer operações estáticas dentro dos elementos das matrizes. Existem vários outros métodos estatísticos que podemos utilizar, facilmente podemos encontrá-los com uma rápida pesquisa.

Vejamos como fazer operações utilizando todos os dados de uma matriz.

Por exemplo, se quisermos saber a média geral e seu desvio-padrão de todas as turmas e todas as disciplinas da faculdade inteira basta dar o seguinte comando:

```
>> med_faculdade = T.mean() # média  
>> std_faculdade = T.std() # desvio padrão
```

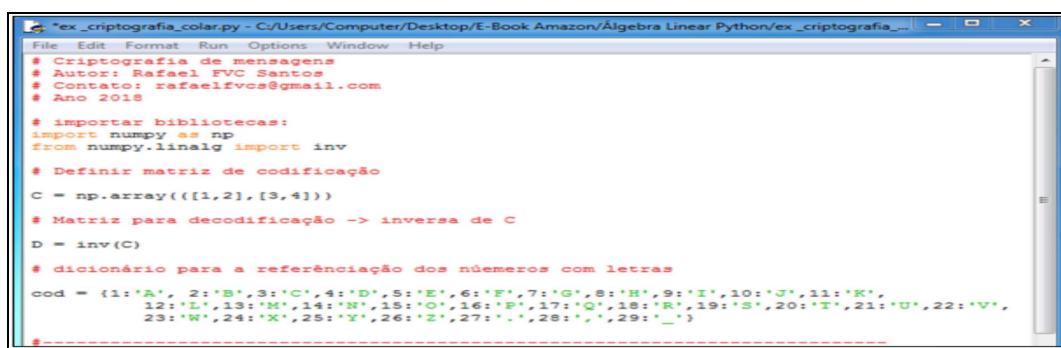

7.2. Exemplo 2 - Matrizes e Criptografia

Este próximo exemplo é bem interessante. Você vai aprender a fazer um código para criptografar mensagens e conversar de maneira secreta com seus amigos. Iremos conhecer uma maneira de codificar e decodificar mensagens utilizando as propriedades das matrizes.

Iremos precisar de basicamente uma matriz quadrada A e de sua inversa para fazermos a criptografia. A matriz A será utilizada para codificar nossa mensagem, por outro lado a inversa da matriz (A^{-1}) nos ajudará a decodificar a mensagem.

Como precisaremos de mais flexibilidade, iremos utilizar o IDLE do Python. Num primeiro momento vamos importar o NumPy e criar um dicionário de codificação de números em caracteres alfabéticos. A maneira ensinada, claro, é apenas uma, você pode utilizar sua criatividade para fazer como bem desejar.

A primeira parte do código compete os *imports* mais a declaração da matriz de codificação (D), da matriz de decodificação (inversa de D⁻¹) bem como de um dicionário que referencia um número a uma letra (*string*). Ver figura abaixo:



The screenshot shows a Python script titled "ex_criptografia_colar.py" running in the IDLE editor. The code is as follows:

```
# ex_criptografia_colar.py - C:/Users/Computer/Desktop/E-Book Amazon/Álgebra Linear Python/ex_criptografia_...
File Edit Format Run Options Window Help
# Criptografia de mensagens
# Autor: Rafael FVC Santos
# Contato: rafaelfvcs@gmail.com
# Ano 2018

# Importar bibliotecas:
import numpy as np
from numpy.linalg import inv

# Definir matriz de codificação
C = np.array([[1,2],[3,4]])

# Matriz para decodificação -> inversa de C
D = inv(C)

# Dicionário para a referência dos números com letras
cod = {1:'A', 2:'B', 3:'C', 4:'D', 5:'E', 6:'F', 7:'G', 8:'H', 9:'I', 10:'J', 11:'K',
       12:'L', 13:'M', 14:'N', 15:'O', 16:'P', 17:'Q', 18:'R', 19:'S', 20:'T', 21:'U', 22:'V',
       23:'W', 24:'X', 25:'Y', 26:'Z', 27:'.', 28:',', 29:'_'}  
#
```

Após isso iremos precisar de pelo menos duas funções: uma para transformar textos em um vetor de números e outra para transformar o vetor de números em texto novamente. Veja abaixo um exemplo de código para solucionar esse problema:

```


-----#
def texto2numeros(texto):
    num = []
    # para deixar a matriz com duas linhas completa:
    if(len(texto)%2 != 0):
        texto = texto+''
        print('Precisamos add _ para completar matriz!')
    # transformar letras em números no dicionário
    for i in texto:
        for j in cod:
            if(cod[j]==i):
                num.append(j)
    return num

def numeros2texto(num):
    Tex = ''
    for i in num:
        Tex = Tex+str(cod[round(i)])
    return Tex


```

Agora que já temos as funções para transformar texto em vetor de números e vetor de números em texto vamos fazer a codificação e decodificação de uma mensagem:

```


#-----#
texto = 'PYTHON_UMA_EXCELENTE_LINGUAGEM_DE_PROGRAMACAO'

VT = texto2numeros(texto)           # vetor com texto em números

MT = np.array(VT)                  # transformar em array

MSG = MT.reshape(2,int(MT.size/2)) # mensagem em matriz

MC = C.dot(MSG)                  # mensagem codificada

MD = D.dot(MC)                   # mensagem decodificada

msg_d = MD.reshape(1,MD.size)     # matriz da msg transformada em vetor

print('==MENSAGEM EM MATRIZ=====') 
print(MSG)
print('==MENSAGEM CODIFICADA====')
print(MC)
print('==MENSAGEM DECODIFICADA==')
print(MD)
print('=MSG DECODIFICA EM VETOR=')
print(msg_d)
print('=====MSG TRANSFORMADA=====')
print(numeros2texto(msg_d[0,:]))


```

Depois de compilar o código (**F5**) obteremos a seguinte resposta:

```

RESTART: C:\Users\Computer\Desktop\E-Book Amazon\Algebra Linear Python\ex_crip
tografia_colar.py
Precisamos add _ para completar matriz!
---MENSAGEM EM MATRIZ---
[[16 25 20  8 15 14 29 21 13  1 29  5 24  3  5 12  5 14 20  5 29 12  9]
 [14  7 21  1  7  5 13 29  4  5 29 16 18 15  7 18  1 13  1  3  1 15 29]]
---MENSAGEM CODIFICADA---
[[ 44 39 62 10 29 24 55 79 21 11 87 37 60 33 19 48 7 40
 22 11 31 42 67]
 [104 103 144 28 73 62 139 179 55 23 203 79 144 69 43 108 19 94
 64 27 91 96 143]]
---MENSAGEM DECODIFICADA---
[[16. 25. 20. 8. 15. 14. 29. 21. 13. 1. 29. 5. 24. 3. 5. 12. 5. 14.
 20. 5. 29. 12. 9.]
 [14. 7. 21. 1. 7. 5. 13. 29. 4. 5. 29. 16. 18. 15. 7. 18. 1. 13.
 1. 3. 1. 15. 29.]]
---MSG DECODIFICA EM VETOR---
[[16. 25. 20. 8. 15. 14. 29. 21. 13. 1. 29. 5. 24. 3. 5. 12. 5. 14.
 20. 5. 29. 12. 9. 14. 7. 21. 1. 7. 5. 13. 29. 4. 5. 29. 16. 18.
 15. 7. 18. 1. 13. 1. 3. 1. 15. 29.]]
---MSG TRANSFORMADA---
PYTHON_UMA_EXCELENTE_LINGUAGEM_DE_PROGRAMACAO_
>>>

```

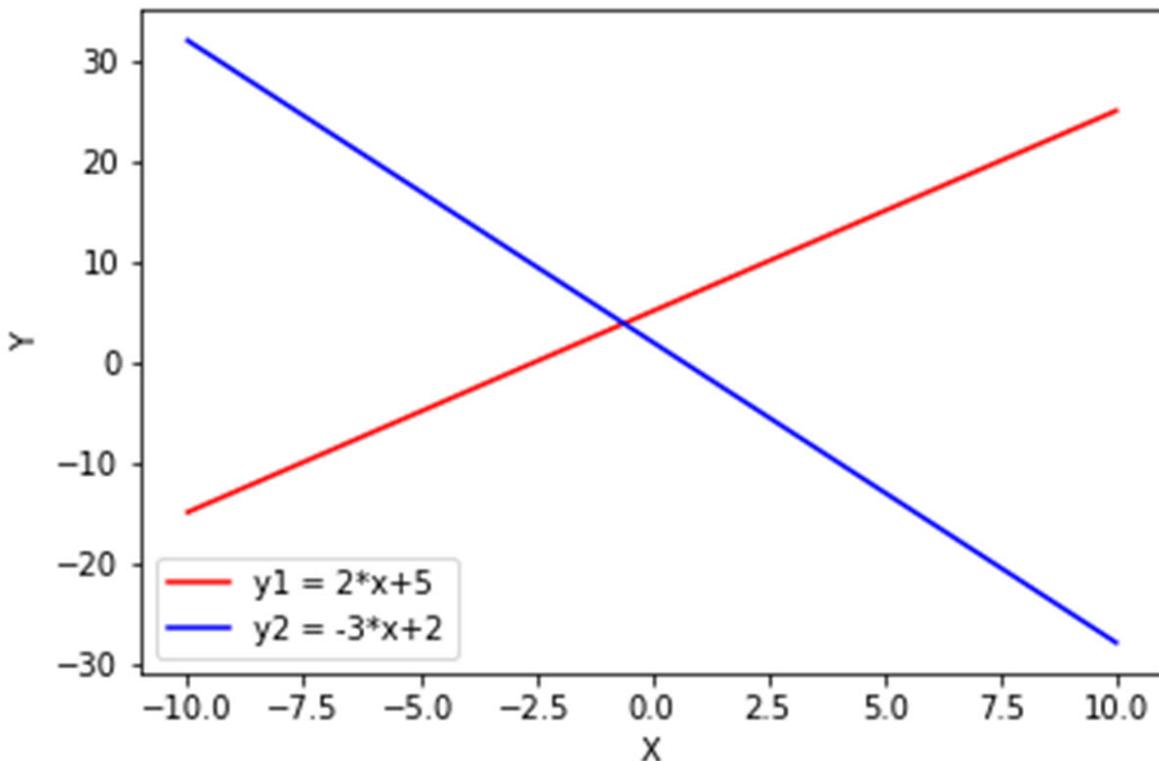
A mensagem codificada, para o nosso caso, sempre estará representada numa matriz $2 \times n$. Isso porque escolhemos nossa matriz de codificação de tamanho 2×2 . Se nossa matriz de codificação tivesse tamanho 3×3 , por exemplo, nossa matriz de mensagem deveria possuir pelo menos 3 linhas e o números de colunas que se adequasse às palavras do texto. Vale ressaltar que poderíamos ter utilizado uma matriz de tamanho qualquer, no entanto mantendo a consistência entre os tamanhos das matrizes para efetuarmos o produto corretamente.

O segredo de nossa criptografia está na matriz de codificação e de sua inversa. Para a troca segura de mensagens devemos manter em segredo essa matriz e o destinatário deverá possuir a inversa para fazer a decodificação da mensagem recebida.

7.3. Exemplo 3 - Sistemas lineares

Vamos estudar aqui o cálculo do cruzamento entre retas. Podemos utilizar sistemas de equações lineares para calcular o ponto em comum entre duas retas, ou seja, o seu cruzamento. Uma vez que duas retas não sejam paralelas teremos assim um ponto em comum a ambas. Esse tipo de problema é muito esclarecedor para mostrar o porquê de matrizes singulares não possuírem inversa. De maneira indireta você irá perceber essa problemática ao término desse exemplo.

Temos duas retas y_1 e y_2 e queremos achar o ponto de cruzamento entre elas. Em outras palavras, queremos achar os valores de (X, Y) que pertencem às duas retas ao mesmo tempo. Veja a figura abaixo:



Podemos então transformar nosso problema em um sistema de duas equações lineares com duas incógnitas da seguinte maneira:

$$\begin{cases} y - 2x = 5 \\ y + 3x = 2 \end{cases}$$

Vamos agora representar o sistema acima na forma matricial:

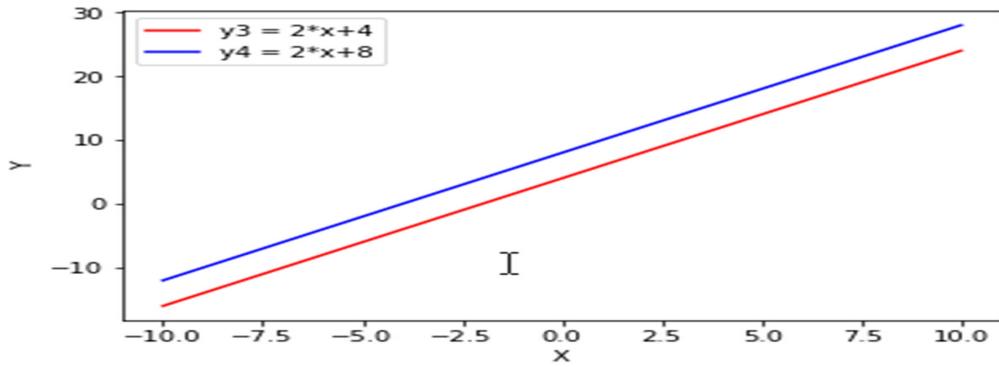
$$\begin{bmatrix} 1 & -2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

Assim precisamos encontrar os valores da matriz coluna [x, y]:

$$\begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} 1 & -2 \\ 1 & 3 \end{bmatrix}^{-1} \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

```
>> import numpy as np
>> from numpy.linalg import inv, det
>> A = np.array(([1,-2],[1,3]))
>> B = np.array(([5],[2]))
>> det_A = det(A) # avaliar se diferente de zero
>> print('det_A=',det_A)
det_A=4.999999999999999 # ok ! podemos encontrar a inversa de
A
>> inv_A = inv(A)
>> X = inv_A.dot(B)
>> print('y = ',X[0], ' x = ', X[1])
y = [3.8] x = [-0.6]
```

Encontramos o par ordenado que representa o cruzamento entre as duas retas do problema. Vamos supor agora que temos duas retas paralelas entre si como mostra a figura abaixo:



A representação matricial supondo que queiramos encontrar o cruzamento entre as retas fornece:

$$\begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} 1 & -2 \\ 1 & -2 \end{bmatrix}^{-1} \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

```
>> A = np.array(([1,-2],[1,-2]))
>> B = np.array(([4],[8]))
>> det_A = det(A) # avaliar se diferente de zero
>> print('det_A=',det_A)
0.0 (portanto temos uma matriz singular)
>> inv_A = inv(A) # teremos um erro LinAlgError("Singular matrix")
```

Fica claro agora porque o sistema linear não tem solução, pois as retas estão paralelas entre si. Existe uma dependência linear que torna nosso sistema linearmente dependente LD (uma equação é combinação linear da outra). Logo, não podemos solucionar um sistema de duas equações com duas incógnitas sendo que uma equação é combinação linear da outra (existe dependência linear).

7.4. Exemplo 4 - Rankeamento de Busca em sites

Como sabemos bem, o site do Google é utilizado para encontrar páginas na internet que contenham uma determinada frase ou palavras-chave digitadas. Logo no início da criação do Google existiam poucos sites e a inovação estava na maneira de fazer a listagem dos sites encontrados, ou seja, na melhor forma de classificar as páginas na internet.

O rankeamento de páginas na internet tem o principal objetivo de ordenar as páginas por importância de um resultado buscado. Podemos definir a importância de uma página como o número de páginas que apontam para ela. Kendall e Wei na década de 1950 propuseram uma estrutura de algoritmo que foi utilizada pelos criadores do Google.

Vamos entender de maneira matricial o funcionamento geral do método proposto pelos dois pesquisadores. Considere x_i , $1 \leq i \leq n$, a importância do i -ésimo site. A hipótese principal do modelo é que a importância de um site é diretamente proporcional à importância dos sites que apontam para ele. Em outras palavras, quanto maior é o número de apontamento, maior é a importância de um determinado site.

Isso nos leva a um sistema de equações descrito abaixo:

$$\left\{ \begin{array}{lcl} x_1 & = & K(x_2 + x_{14} + x_{541}) \\ x_2 & = & K(x_1 + x_{23} + x_{541} + x_{1023}) \\ \vdots & & \\ x_n & = & K(x_{25} + x_{133}) \end{array} \right. ,$$

Onde K é a constante de proporcionalidade que está multiplicada em cada linha. O valor de K está relacionado à importância dos sites que apontam para os x_1, x_2, \dots, x_n .

Podemos representar todo esse sistema a partir da seguinte equação abaixo:

$$\sum_{j=1}^n a_{ij}x_i = \frac{1}{K}x_i \iff Ax = \frac{1}{K}x$$

Perceba que estamos procurando autovetores de coordenadas positivas associada ao autovalor $1/K$ da matriz A. Por trás disso tudo existe o teorema de Perron que garante que sempre existirá um autovetor associado a tal matriz. Uma vez encontrado os autovetores, sua coordenada maior determina o site de maior importância, a segunda maior refere-se ao segundo site mais importante e assim sucessivamente.

Com o NumPy, uma vez encontrado os autovalores com o método ‘numpy.linalg.eig()’ podemos fazer o ordenamento dos mesmos com o método **numpy.argsort()**.

Veja abaixo como fica a aplicação para um caso genérico desse tipo:

```
>> import numpy as np
>> import numpy.linalg as la
>> w, v = la.eig(A) # sendo A a matriz dos coeficientes referentes ao rankeamento
>> ind = np.argsort(w)[::-1] # teremos os índices ordenados decrescente dos autovalores
>> w_dec = w[ind]
>> v_dec = v[ind]
```


7.5. Exemplo 5 - Sequência de Fibonacci

Existem alguns padrões matemáticos no mundo capazes de assustar qualquer mente curiosa. E um dos mais intrigantes padrões presentes na natureza está relacionado com a sequência de Fibonacci. Ela possui esse nome pois foi proposta pelo matemático italiano Leonardo Pisa (1170-1250) mais conhecido como Fibonacci. A sequência é a seguinte:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$$

Aqui temos uma série matemática do tipo: $u_n = u_{n-1} + u_{n-2}$ com valores iniciais $u_1 = 1$, $u_2 = 1$.

Essa sequência é largamente aplicada nas áreas de análise de mercado financeiro, ciências da computação, teoria dos jogos, crescimento populacional, biologia, economia dentre outras.

Vamos representar essa sequência matricialmente. Ver abaixo:

$$\begin{bmatrix} u_n \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{n-1} \\ u_{n-2} \end{bmatrix}$$

Agora vamos definir w_k e A como às seguintes matrizes:

$$w_k = \begin{bmatrix} u_{k+1} \\ u_k \end{bmatrix} \text{ e } A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \text{ tal que } 0 \leq k \leq n-1,$$

de modo que:

$$w_0 = \begin{bmatrix} u_1 \\ u_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, w_1 = \begin{bmatrix} u_2 \\ u_1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \dots, w_{n-2} = \begin{bmatrix} u_{n-1} \\ u_{n-2} \end{bmatrix}, w_{n-1} = \begin{bmatrix} u_n \\ u_{n-1} \end{bmatrix}$$

Portanto, podemos reescrever uma equação geral:

$$W_{n-1} = A W_{n-2}$$

Logo

$$W_1 = AW_0$$

$$W_2 = AW_1 = A(AW_0) = A^2W_0$$

$$W_3 = AW_2 = A(A^2W_0) = A^3W_0$$

...

$$W_{n-1} = A^{n-1}W_0$$

Sendo assim, para encontrar w_n basta calcular A^{n-1} . Vamos agora entender a parte mais interessante do problema. A depender do tamanho de n , calcular o expoente da matriz A pode ser muito custoso computacionalmente. Então, para evitar esse problema vamos utilizar nossos conhecimentos de autovalores e autovetores.

Para encontrar o polinômio característico de A podemos utilizar $\det(A - \lambda I) = 0$:

$$\begin{bmatrix} \lambda - 1 & -1 \\ -1 & \lambda \end{bmatrix} = \lambda^2 - \lambda - 1 = 0$$

Resolvendo a equação do segundo grau (polinômio característico) encontramos as seguintes raízes:

$$\lambda_1 = \frac{1 + \sqrt{5}}{2} \quad e \quad \lambda_2 = \frac{1 - \sqrt{5}}{2}$$

De modo que a matriz formada por essas raízes é do tipo diagonal:

$$D = \begin{bmatrix} \frac{1 + \sqrt{5}}{2} & 0 \\ 0 & \frac{1 - \sqrt{5}}{2} \end{bmatrix}$$

E os autovetores correspondentes são:

$$x_1 = \begin{bmatrix} \frac{1 + \sqrt{5}}{2} \\ 1 \end{bmatrix} \text{ e } x_2 = \begin{bmatrix} \frac{1 - \sqrt{5}}{2} \\ 1 \end{bmatrix}$$

Assim temos P como a matriz dos autovetores:

$$P = \begin{bmatrix} \frac{1 + \sqrt{5}}{2} & \frac{1 - \sqrt{5}}{2} \\ 1 & 1 \end{bmatrix},$$

$$P^{-1} = \begin{bmatrix} \frac{1}{\sqrt{5}} & \frac{-1 + \sqrt{5}}{2\sqrt{5}} \\ \frac{-1}{\sqrt{5}} & \frac{1 + \sqrt{5}}{2\sqrt{5}} \end{bmatrix},$$

Portanto, $A^k = PD^kP^{-1}$, pois D é uma matriz diagonal, assim para calcular D^k , basta elevar os elementos da diagonal principal a k-ésima potência, respeitando a equação abaixo:

$$w_{n-1} = A^{n-1} w_0 = P D^{n-1} P^{-1} w_0 = \\ \begin{bmatrix} \frac{1+\sqrt{5}}{2} & \frac{1-\sqrt{5}}{2} \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \left(\frac{1+\sqrt{5}}{2}\right)^{n-1} & 0 \\ 0 & \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{5}} & \frac{-1+\sqrt{5}}{2\sqrt{5}} \\ \frac{-1}{\sqrt{5}} & \frac{1+\sqrt{5}}{2\sqrt{5}} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Executando as multiplicações chegaremos a uma equação que permite calcular diretamente o u_n . Ver abaixo:

$$u_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^{n-1} - \left(\frac{1 - \sqrt{5}}{2} \right)^{n-1} \right]$$

Assim, temos uma forma geral para calcular a posição do termo ‘n’ de uma sequência de Fibonacci com a solução de uma equação bastante simplificada.

Em Python ficamos:

```
>> import numpy as np
>> n = 1000 # queremos o valor do milésimo termo da sequência de
Fibonacci
>> u = 1/np.sqrt(5)*(((1+np.sqrt(5))/2)**(n+1)-((1-np.sqrt(5))/2)**
(n+1))
>> print(u)
7.033036771142517e+208
```


7.6. Agradecimentos

Obrigado pela confiança em comprar este e-book. Parabéns por chegar até aqui. Caso tenha gostado ou não da didática, favor deixar um comentário na loja da Amazon. Seu feedback é de extrema importância para melhoria contínua deste trabalho.

Fique de olho na série: Cientista de dados – analista Quant. Em breve teremos novos volumes abordando assuntos da área.

Estou à disposição, segue meu e-mail: rafaelfvcs@gmail.com

Bons estudos!

[1] A concordância para a palavra Python por vezes será no feminino e por outras no masculino. Portanto não estranhe quando estivermos falando no Python (projeto como um todo) e na Python (linguagem de programação).

[2] BOLDRINI, J. L. Álgebra Linear. 3 ed. São Paulo: Harbra, 1986.