EAE1106 - Métodos Computacionais para Economia

Danilo Souza

Índice

Bem-vindo

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

Agradecimentos e reprodução

Esse material deve muito a X, Y e Z

Esse material foi construído bla bla bla. Qualquer erro é de responsabilidade exclusiva do autor. Todo e qualquer feedback é muito bem-vindo.

Introdução

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

Parte I

Alfabetização computacional e lógica básica

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

1 Fundamentos de computação

In summary, this book has no content whatsoever.

1.1 O que é um computador?

Um computador realiza duas coisas e apenas duas coisas: faz operações e guarda de forma eficiente o resultado dessas operações. No entanto, ao fazer essas duas coisas de forma extremamente eficiente essa máquina chamada computador nos permite receber, armazenar, processar e transmitir informações. Em essência, o computador nos serve ao propósito de **resolução de problemas**. Isso significa que, através de uma linguagem formal (especificamente operações de computação), podemos formular problemas e **programar** meios de resolver de forma automatizada tais problemas. No fim das contas, programar é o ato de criar programas, isto é, estabelecer uma sequência de instruções em linguagem formal que especifica como executar uma determinada operação no computador.

Mas antes de avançar na ideia de **programa** e **algoritmos**, quais são as características de um computador como o conhecemos?

1.1.1 Sistema computacional

Um sistema computacional é resultado da integração de componentes atuando como uma entidade, com o propósito de processar dados, isto é, realizar algum tipo de operação lógica envolvendo os dados, de modo a produzir diferentes níveis de informações. Componentes:

- **Hardware**: componente física de um sistema de computação, isto é, todos os equipamentos utilizados pelo usuário nas ações de entrada, processamento, armazenamento e saída de dados.
- **Software**: componente lógica de um sistema de computação, isto é, séries de instruções que fazem o computador funcionar (programas de computador)

E por fim o componente humano do sistema, as pessoas, que utilizam o computador basicamente como ferramenta para atingir determinado fim, para **resolver problemas**.

1.1.2 Arquitetura de um computador

Os computadores, como os conhecemos hoje, são estruturados em cima da lógica proposta inicialmente por John Von Neumann, matemático húngaro que viveu durante a primeira metade do século XX e que contribuiu imensamente em várias dimensões das ciências. Embora tenhamos hoje uma complexidade maior nos elementos que compõe a arquitetura de um computador, a estrutura básica por trás dos computadores mais modernos continua sendo aquela proposta por Von Neumann. Essa arquitetura pode ser representado pelo diagrama abaixo:

Essencialmente, o hadware que compõe as partes da arquitetura de um computador podem ser dividas em 3 partes:

- **Dispositivos de entrada e saída**: é através deles que o usuário dialoga com a máquina. Através do teclado e mouse, por exemplo, o usuário fornece informações ao computador a partir das quais processos serão realizados e seus resultados serão percebidos pelos dispositivos de saída (monitor e impressora, por exemplo).
- **Unidade central de processamento (CPU)**: é o cérebro do sistema computacional, é nele que serão executados cálculos e instruções lógicas dos usuários.
- **Memória**: aqui são armazenados os resultados dos cálculos, dados e instruções.

1.1.3 Capacidade de armazenamento

- Kilobyte: 1024 (2^10) bytes.
  - Capacidade de memória dos computadores pessoais muito antigos.
- Megabyte: aproximadamente, um milhão (2^20) de bytes.
  - Memória de computadores pessoais antigos.
- Gigabyte: aproximadamente, um bilhão (2^30) de bytes.
  - Dispositivos de armazenamento (discos rígidos), especialmente SSD
- Terabyte: aproximadamente, um trilhão (2^40) de bytes.
  - Dispositivos de armazenamento para sistemas muito grandes e servidores.

1.1.4 Como computadores armazenam números

Para entender a diferença entre computação de precisão simples e de precisão dupla, é importante entender o papel da precisão na ciência da computação. Imagine realizar um cálculo usando um número irracional (como π) e incluindo apenas dois dígitos à direita do ponto decimal (3.14). Você obteria um resultado mais preciso se fizesse o cálculo incluindo dez dígitos à direita do ponto decimal (3,1415926535).

Para computadores, esse nível de precisão é chamado de precisão e é medido em dígitos binários (bits) em vez de casas decimais. Quanto mais bits usados, maior a precisão. A representação de grandes números em binários de computador requer um padrão para garantir que não haja grandes discrepâncias nos cálculos. Assim, o Instituto de Engenheiros Elétricos e Eletrônicos (IEEE) desenvolveu o Padrão IEEE para Aritmética de Ponto Flutuante (IEEE 754).

Em geral, um número será representado de forma aproximada com um número fixo de números significantes e escalonado usando um expoente em uma base (usualmente dois, ou 10 ou 16). Esses números possuem a seguinte forma:

$$S \times B^e$$

Em que S é o significando, B é um número inteiro maior ou igual a dois, e e um expoente, um número inteiro. Por exemplo: $1.2345 = 12345 \times 10^{-4}$. O termo Ponto Flutuante se refere ao fato que o ponto decimal (também chamado de ponto binário) pode flutuar; ou seja, pode ser colocado em qualquer lugar em relação aos dígitos significantes do número. Esta posição é indicada pelo expoente, e a representação em ponto flutuante de um número pode ser pensada como uma forma de notação científica.

A precisão nessa representação de ponto flutuante é dada pela quantidade de bits dedicado ao armazenamento do significando e do expoente – quanto maior o significando, mais precisa é a representação (menor é o espaço entre dois números), e quanto maior o expoente, mais alto é o maior valor que pode ser representado.

Existem três componentes do IEEE 754:

- O sinal - 0 representa um número positivo; 1 representa um número negativo.
- O expoente - O expoente é usado para representar expoentes positivos e negativos.
- O significando - Também conhecida como mantissa, a mantissa representa os bits de precisão do número.

Usando esses componentes, o IEEE 754 representa números de ponto flutuante de duas maneiras: formato de precisão simples e formato de precisão dupla. Embora ainda existam várias maneiras de representar números de ponto flutuante, o IEEE 754 é o mais comum porque geralmente é a representação mais eficiente de valores numéricos.

1.1.5 Formato de ponto flutuante de precisão simples

O formato de ponto flutuante de precisão simples usa 32 bits (um bit pro sinal, 8 pro expoente e 24 pro significando) de memória do computador e pode representar uma ampla gama de valores numéricos. Muitas vezes referido como FP32, esse formato é melhor usado para cálculos que não sofrem um pouco de aproximação.

1.1.6 Formato de ponto flutuante de precisão dupla

O formato de ponto flutuante de precisão dupla, por outro lado, ocupa 64 bits (um bit pro sinal, 11 pro expoente e 52 pro significando) da memória do computador e é muito mais preciso do que o formato de precisão simples. Esse formato é frequentemente chamado de FP64 e usado para representar valores que exigem um intervalo maior ou um cálculo mais preciso.

Embora a precisão dupla permita mais precisão, ela também requer mais recursos computacionais, armazenamento de memória e transferência de dados. O custo de usar esse formato nem sempre faz sentido para todos os cálculos.

1.1.7 Mas qual a importância disso tudo?

Saber como os computadores armazenam números, ou qual a importância da memória no processo de fornecimento de parâmetros de entrada em busca de uma saída específica, nos ajuda enormemente a entender porque, por exemplo, um sistema 32 bits só é capaz de nos fornecer 4GB de memória RAM para armazenamento de cálculos e operações ou mesmo porque podemos ter dificuldade ao tentar abrir no nosso computador local um arquivo com alguns milhões de linhas.

É importante ter o conhecimento mínimo de como funciona um computador e como os programas que iremos utilizar interagem com as partes da arquitetura do computador para entendermos o que está ocorrendo atrás das cortinas. Dessa forma, podemos nos concentrar exclusivamente no que nos interessa, isto é, utilizar o computador como ferramenta para **resolver problemas** e não como um fim em si (afinal estamos em um curso de economia e não de ciência da computação). Para tal precisamos de alguma linguagem lógica/formal que nos permita enviar as intruções para o computador e obter as respostas desejadas. Como já deve ter ficado claro, nesse curso trabalharemos com a linguagem **Python**.

1.2 Linguagens de programação

As linguagens naturais são os idiomas que as pessoas falam, como inglês, espanhol e francês. Elas não foram criadas pelas pessoas (embora as pessoas tentem impor certa ordem a elas); desenvolveram-se naturalmente. As linguagens formais são linguagens criadas pelas pessoas para

aplicações específicas. Por exemplo, a notação que os matemáticos usam é uma linguagem formal especialmente boa para denotar relações entre números e símbolos. Os químicos usam uma linguagem formal para representar a estrutura química de moléculas. E o mais importante:

As linguagens de programação são idiomas formais criados para expressar operações de computação.

As linguagens formais geralmente têm regras de sintaxe estritas que governam a estrutura de declarações. Por exemplo, na matemática a declaração \$3 + 3 = \$6 tem uma sintaxe correta, mas não 3+ = 3\$6. Na química, H2O é uma fórmula sintaticamente correta, mas 2Zz não é. As regras de sintaxe vêm em duas categorias relativas a símbolos e estrutura. Os símbolos são os elementos básicos da linguagem, como palavras, números e elementos químicos. Um dos problemas com 3+ = 3\$6 é que o '\$' não é um símbolo legítimo na matemática (pelo menos até onde eu sei). De forma similar, 2Zz não é legítimo porque não há nenhum elemento com a abreviatura Zz.

O segundo tipo de regra de sintaxe refere-se ao modo no qual os símbolos são combinados. A equação 3 + = 3 não é legítima porque, embora + e = sejam símbolos legítimos, não se pode ter um na sequência do outro. De forma similar, em uma fórmula química o subscrito vem depois do nome de elemento, não antes.

Esta é um@ frase bem estruturada em portuguê\$, mas com s*mbolos inválidos.

Ao ler uma frase em português ou uma declaração em uma linguagem formal, é preciso compreender a estrutura (embora em uma linguagem natural você faça isto de forma subconsciente). Este processo é chamado de análise. Embora as linguagens formais e naturais tenham muitas características em comum – símbolos, estrutura e sintaxe – há algumas diferenças:

1. **Ambiguidade**: as linguagens naturais são cheias de ambiguidade e as pessoas lidam com isso usando pistas contextuais e outras informações. As linguagens formais são criadas para ser quase ou completamente inequívocas, ou seja, qualquer afirmação tem exatamente um significado, independentemente do contexto

2. **Redundância**: para compensar a ambiguidade e reduzir equívocos, as linguagens naturais usam muita redundância. Por causa disso, muitas vezes são verborrágicas. As linguagens formais são menos redundantes e mais concisas.

3. **Literalidade**: as linguagens naturais são cheias de expressões e metáforas. Se eu digo "caiu a ficha", provavelmente não há ficha nenhuma na história, nem nada que tenha caído (esta é uma expressão para dizer que alguém entendeu algo depois de certo período de confusão). As linguagens formais têm significados exatamente iguais ao que expressam.

Como todos nós crescemos falando linguagens naturais, às vezes é difícil se ajustar a linguagens formais. As linguagens formais são mais densas que as naturais, então exigem mais tempo para a leitura. Além disso, a estrutura é importante, então nem sempre é melhor ler de cima para baixo e da esquerda para a direita. Em vez disso, aprenda a analisar o programa primeiro, identificando os símbolos e interpretando a estrutura. E os detalhes fazem diferença. Pequenos

erros em ortografia e pontuação, que podem não importar tanto nas linguagens naturais, podem fazer uma grande diferença em uma língua formal.

1.2.1 Alto nível X Baixo nível

Na ciência da computação, uma **linguagem de programação de alto nível** é uma linguagem de programação com forte abstração dos detalhes do computador. São linguagens mais próximas das linguagens humanas e mais distantes das linguagens de máquina. Em contraste com as linguagens de programação de baixo nível, ela pode usar elementos de linguagem natural, ser mais fácil de usar, ou pode automatizar (ou até ocultar totalmente) áreas significativas de sistemas de computação (por exemplo, gerenciamento de memória), tornando o processo de desenvolvimento de um programa mais simples e eficiente. Exemplos de linguagens de alto nível: Python, JavaScript.

Por outro lado, uma **linguagem de baixo nível** é uma linguagem de programação que fornece pouca ou nenhuma abstração de conceitos de programação e está muito próxima de escrever instruções de máquina reais. A palavra "baixo" refere-se à pequena ou inexistente quantidade de abstração entre a linguagem e a linguagem de máquina; por causa disso, as linguagens de baixo nível são às vezes descritas como "próximas do hardware". Programas escritos em linguagens de baixo nível tendem a ser relativamente não portáteis e dependentes do computador para o qual foram escritas. Exemplo de linguagem de baixo nível: Assembly

1.2.2 Linguagem Compilada X Interpretada

As **linguagens compiladas** são convertidas diretamente em código de máquina que o processador pode executar. Como resultado, elas tendem a ser mais rápidos e eficientes de executar do que linguagens interpretadas. Eles também dão ao desenvolvedor mais controle sobre os aspectos de hardware, como gerenciamento de memória e uso da CPU. As linguagens compiladas precisam de uma etapa de "construção" – elas precisam ser compiladas manualmente primeiro. Você precisa "reconstruir" o programa toda vez que precisar fazer uma alteração. Exemplos de linguagens compiladas puras: C, C++, Go.

Uma **linguagem interpretada** é uma linguagem de programação que geralmente é interpretada, sem compilar um programa em instruções de máquina. É aquela em que as instruções não são executadas diretamente pela máquina de destino, mas lidas e executadas, linha por linha, por algum outro programa, um intérprete. As linguagens interpretadas já foram significativamente mais lentas do que as linguagens compiladas. Mas, com o desenvolvimento da compilação just-in-time, essa lacuna está diminuindo. Exemplos de linguagens interpretadas: Python e JavaScript

1.2.3 Paradigmas

Podemos dizer que paradigmas de programação são diferentes formas ou estilos em que um determinado programa ou linguagem de programação pode ser organizado. Cada paradigma consiste em certas estruturas, recursos e opiniões sobre como problemas comuns de programação devem ser abordados. Entre os principais paradigmas de programação podemos citar **programação imperativa**, **programação declarativa** e **programação orientada a objetos**. Na programação imperativa, por exemplo, o programador diz como, o quê e em qual ordem exatamente um programa ou rotina deve realizar. É neste paradigma que surgiram os famosos laços de repetição, estruturas condicionais, atribuição de valor à variáveis e controle de estado. Por outro lado, a programção declarativa não há preocupação na maneira ou método de execução de uma determinada rotina, o que importa é que a instrução seja realizada e não a forma como o será. Mais detalhes sobre paradigmas de programação podem ser obtidos aqui.

Python é considerado uma linguagem de programação multi-paradigma, pois suporta orientação de objeto, programação imperativa e, em menor escala, programação funcional.

1.2.4 Tipagem

O Python utiliza tipagem dinâmica e forte, isso significa que o próprio interpretador do Python infere o tipo dos dados que uma variável recebe, sem a necessidade que o usuário da linguagem diga de que tipo determinada variável é. Hoje existem inúmeras linguagens no mercado que são fortemente tipadas, referenciando especificamente o Python para explicar a questão: tipagem forte significa que o interpretador do Python avalia as expressões por conta própria e não faz coerções automáticas (conversões de valores) entre tipos de dados não compatíveis. Ao fazer operações com tipos incompatíveis, o Python não converte automaticamente esses tipos pra você, ele vai dar erro. Isso é bom, pois assim você terá a certeza que o seu resultado é mais consistente.

2 Primeiros passos no Python

In summary, this book has no content whatsoever.

2.1 O Python

Python foi criado no final dos anos oitenta(1989) por Guido van Rossum no Centro de Matemática e Tecnológia da Informação (CWI, Centrum Wiskunde e Informatica), na Holanda, como sucessor da linguagem de programação ABC, capaz de lidar com exceções e interagir com o sistema operacional Amoeba. O nome da língua vem do gosto de seu criador pelos humoristas britânicos Monty Python (fonte: https://blog.vulpi.com.br/python-como-surgiu/)

Python é uma linguagem de programação de alto-nível, interpretada, multi-paradigma, e que usa tipagem dinâmica e forte.

bla bla bla

2.1.1 Porque escolher

Assim como tudo na vida, o Python é uma linguagem de programação com muitas qualidades mas que também possui seus defeitos (em geral ligados à velocidade de execução dos programas). Em sendo uma linguagem de alto nível, é uma linguagem bastante indicada, por exemplo, para aplicações que demandam replicabilidade, dada a facilidade de escrever e interpretar o código. Por outro lado, aplicações que demandam uma interação mais eficiente entre o código e o gerenciamento de memória da máquina, por exemplo, podem se beneficiar de outras linguagens.

No entanto, o conjunto de qualidades do Python e as inúmeras bibliotecas que foram desenvolvidas nos últimos anos parecem ter mais do que compensado as falhas da linguagem. Hoje o Python é uma das principais linguagens de programação (senão a principal) quando o assunto é ciência dos dados e aprendizado de máquina (*data science* e *machine learning*, respectivamente). É também uma das principais linguagens de programação por trás de vários dos sistemas de grandes empresas, como Uber, GoldmanSachs, Netflix e Google (fonte: link).

Por todos esses fatores, Python é uma das linguagens com a comunidade mais ativa em fóruns online voltados à programação e é a **linguagem que mais cresce no mundo**. As figuras abaixo ilustram bem esse crescimento da linguagem nos últimos anos:

2.1.2 Como instalar

Existe mais de uma forma de instalar o Python no seu sistema e ainda mais formas de interagir com a linguagem. Você pode baixar o Python *puro* (https://www.python.org/downloads/) e instalá-lo diretamente em sua máquina, porém, essa distribuição vem com poucos pacotes já instalados e instalá-los um a um requer paciência. Além disso, a instalação do Python puro pode demandar algumas alterações em configurações do sistema via prompt de comando do Windows, por exemplo, para que seja possível dialogar com a linguagem.

Uma forma mais simples de instalar o Python em seu computador é através da distribuição Anaconda (https://www.anaconda.com/). Além de ser gratuita e conter vários dos pacotes que iremos utilizar, ela nos fornece diversas ferramentas que facilitam nossa interação com a linguagem. É hoje uma das distribuições do Python mais populares no mundo! **Recomendo fortemente utilizar esse caminho**.

O passo a passo a seguir foi feito para o sistema Windows 10. Embora o caminho seja parecido, podem haver algumas divergências em relação ao passo a passo para sistemas Linux e MacOS.

1. Baixar e instalar o Anaconda é fácil. Basta acessar a aba de downloads no site do Anaconda (https://www.anaconda.com/download), fornecer um email para cadastro e baixar o arquivo executável necessário para a instalação a depender do seu sistema operacional.

2. Depois de baixado, basta clicar duas vezes sobre o arquivo e ir acompanhando o instalador, mantendo sempre as opções padrão e escolhendo a opção de instalar para todos os usuários do computador, o que requer acesso de administrador.

3. Pronto, o Anaconda está instalado e os principais pacotes e programas que utilizaremos também!

2.1.3 Anaconda e suas particularidades

Como dito acima, anaconda é uma distribuição do Python para computação científica (e.g, ciência de dados e aprendizado de máquina), que visa simplificar o gerenciamento e a implantação de pacotes. As versões de pacotes no Anaconda são gerenciadas pelo sistema de gerenciamento de pacotes *conda*. Através do *conda* é possível criar ambientes e instalar pacotes distintos de forma a evitar incompatibilidades entre versões de pacotes já instalados.

Além disso, o anaconda traz consigo o Anaconda Navigator, um ambiente *user-friendly* para a gestão de pacotes e aplicativos necessários para a utilização do Python sem que seja preciso ter

qualquer conhecimento acerca da utilização correta de terminais (e.g, *prompt de comando* do Windows). Ao abrir o Anaconda Navigator no Windows, você deve ver algo assim

É possível instalar pacotes e fazer muito mais coisa diretamente pelo Anaconda Navigator. Para os que tiverem interesse em saber um pouco mais, esse vídeo é um bom começo.

2.1.4 Instalando pacotes

Para instalar novos pacotes para o Python precisamos fazê-lo via prompt de comando. O Anaconda nos fornece um prompt de comando próprio, o **Anaconda Prompt** que facilita a instalação desses pacotes através dos comandos `conda` ou `pip`. Vamos instalar como exemplo o pacote `tqdm`, que nos permite criar barras de progresso em atividades repetidas.

1. Abra o **Anaconda prompt**.
2. Veja se o pacote já não está instalado: digite `conda list`.
3. Você pode utilizar 2 comandos distintos para instalar o pacote: `conda install tqdm` ou `pip install tqdm`.

Note também que alguns dos principais pacotes sobre os quais falarei ao longo do curso, *NumPy, SciPy, Matplotlib e Pandas*, já vem instalados com o Anaconda, o que não é verdade no Python puro. Mais do que mostrar o que temos instalado na nossa máquina local, o comando `conda list` nos mostra as versões de cada uma das bibliotecas disponíveis e qual o ambiente utilizado em sua instalação.

Diferença entre *conda* e *pip*: falar mais sobre a diferença entre esses dois comandos populares para instalação de pacotes vai um pouco além do escopo desse curso. De forma bastante resumida, podemos dizer que o `pip` é um gerenciador de pacotes que nos permite instalar qualquer biblioteca escrita em Python e disponível no *Python Package Index* (PyPI), o principal repositório de pacotes para a linguagem. O `conda`, no entanto, é mais do que um simples gerenciador de pacotes já que é possível fazer diferentes instalações em diferentes ambientes de modo a reduzir problemas de incompatibilidade entre versões. Além disso, através do `conda` podemos instalar pacotes escritos em outras linguagens também, como R e C++. Por outro lado, é possível que você demore mais e tenha problemas para instalar alguns pacotes utilizando o `conda`. Aos que quiserem saber um pouco mais sobre a diferença entre esses dois métodos para instalação, esses links (1 e 2) podem ser um bom começo.

2.1.5 IDEs

IDE é um acrônimo para Integrated Development Environment, em português, Ambiente de Desenvolvimento Integrado. Ele é um programa que reúne ferramentas necessárias para a construção de outros softwares. A utilização de um IDE ajuda muito os programadores e empresas, pois torna mais rápido o desenvolvimento de aplicações, aumentando a produtividade

e reduzindo custos. Existem IDE's específicas para plataformas, e outras que são mais flexíveis. As mais famosas são o Sublime Text e o Visual Studio Code.

Quais são os componentes típicos de uma IDE?

- **Editor de código-fonte**: Permite edição do código nas linguagens de programação suportadas pelo IDE.
- **Preenchimento inteligente**: Esse é um recurso trazido pelos IDEs que agiliza o desenvolvimento, pois escreve automaticamente trechos do código como, por exemplo, comandos de função.
- **Compilador**: O compilador que você escreveu em uma determinada linguagem de programação para a linguagem de máquina, de modo que os computadores o entendam.
- **Debugger**: É outra ferramenta que contribui para no código-fonte, melhorando o desempenho do programa.
- **Geração de código**: Com esse recurso, é possível predefinir trechos de códigos para serem usados de modelo em outros projetos, agilizando o desenvolvimento de trabalhos futuros.

Como podemos ver, os IDEs reúnem diversas ferramentas que tornam mais simples a vida dos programadores. Dentre todos os benefícios que a utilização desses programas pode trazer para seus projetos, podemos citar alguns:

- Reduz o tempo gasto em cada aplicação;
- Permite o desenvolvimento de um código mais limpo, organizado e legível;
- Aumenta a produtividade dos desenvolvedores e das empresas;
- Reduz a quantidade de bugs e falhas no código-fonte;
- Reúne diversas ferramentas em um só lugar.

2.1.5.1 Spyder

O Spyder é uma ferramenta leve, simples e ao mesmo tempo poderosa. É um IDE Python de código aberto que conta com elementos avançados de edição, depuração e testes interativos. Ele é bastante utilizado para o aprendizado de Data Science, apesar de não fornecer ferramentas tão avançadas nesse sentido como outras disponíveis. Mas ele é prático e seu depurador destaca bem funções, variáveis e erros. Conta também com um recurso de exploração de variáveis, que exibe os conteúdos armazenados dentro de cada uma. Isso poupa a escrita de comandos de impressão de variáveis na tela.

Disponível com a instalação do Anaconda, uma das maiores e mais utilizadas plataformas de distribuição do Python.

2.1.5.2 Jupyter

É um IDE Python gratuito, utilizado principalmente na análise e ciência de dados. Ele é fácil e intuitivo, proporcionando um bom ambiente para iniciantes em Python. Também conta com muitos materiais de referência, tornando-se um dos IDEs mais utilizados pela comunidade. Ele trabalha muito bem com grandes conjuntos de dados. Além disso, é ótimo para a estética do código e atua como uma. É possível visualizar e editar facilmente seu código para deixá-lo mais atraente e apresentável.

Além de tudo isso, a estrutura do Jupyter serve muito ao propósito de tornar o código mais facilmente replicável, já que é possível intercalar células de comentários e explicações em linguagem Markdown com células de código propriamente dito, seguidas do resultado (ou erros) das instruções. Ele possui ainda integrações com HTML, por exemplo, que fazem a diferença principalmente na hora de apresentar projetos ou utilizá-los para o aprendizado. O material aula a aula do nosso curso, por exemplo, foi inteiramente desenvolvido e criado no Jupyter!

2.1.5.3 Google Colaboratory

O Google Colaboratory, carinhosamente chamado de Colab, é um serviço de nuvem gratuito hospedado pelo próprio Google para incentivar a pesquisa de Aprendizado de Máquina e Inteligência Artificial.

É uma ferramenta que permite que você misture código fonte (geralmente em python) e texto rico (geralmente em markdown) com imagens e o resultado desse código, assim como o próprio Jupyter e sua estrutura de *notebooks* ("cadernos" em inglês). Uma diferença importante é que no caso do Colab os recursos computacionais utilizados para a execução do código são os da Google e não do seu computador. Apesar da versão gratuita disponibilizar apenas algo como 12GB de RAM, é possível ampliar essa capacidade de processamento da máquina virtual por uma conta paga que começa em $5 mensais.

2.1.6 Spyder X Jupyter

Alguns dos pacotes mais utilizados no Python hoje em dia, como *Pandas* e *Matplotlib*, já vem instalados com o Anaconda, o que não é verdade na distribuição pura do Python. Além disso, alguns dos programas mais utilizados para interagir com a linguagem, como o *Spyder* e o *Jupyter Notebook* também. Tanto o *Spyder* quanto o *Jupyter* são ótimas IDEs para trabalhar com o Python, embora cada um tenha suas particularidades.

Existem vários comparativos entre essas duas formas (e tantas outras) de trabalhar com o Python (1, 2 e 3), mas no nosso caso talvez seja mais produtivo se formos direto para ambos os programas!

Acabamos de ver que uma das diferenças mais relevantes para nós, iniciantes na linguagem, ao usar o *Spyder* ou o *Jupyter* é a dificuldade, no segundo caso, de fazer uma gestão mais direta das variáveis criadas e da memória utilizada por cada uma delas. É possível, no entanto, superar essa dificuldade através de uma biblioteca que pode ser bastante útil para nós, o `nbextensions` (documentação aqui e aqui).

Vamos primeiro instalar essa biblioteca e mais uma outra biblioteca auxiliar que nos permitirá ter um controle maior sobre as extensões. Rode os comandos abaixo no *Anaconda Prompt*

1. `pip install jupyter_contrib_nbextensions`
2. `pip install jupyter_nbextensions_configurator`

Note que uma nova aba *Nbextensions* foi criada na "Home Page" do Jupyter Notebook. Ali é possível ativar e desativar as extensões que possam ser interessante ao nosso propósito. Agora já podemos brincar um pouco com tudo que ela pode nos fornecer!

2.2 Construindo o primeiro programa

Tradicionalmente, o primeiro programa que se escreve em uma nova linguagem chama-se "Hello, World!", porque tudo o que faz é exibir as palavras "Hello, World!" na tela. No Python, ele se parece com isto:

```
print('Hello, World!')
```

Este é um exemplo de uma instrução print (instrução de impressão), embora na realidade ela não imprima nada em papel. Ela exibe um resultado na tela. As aspas apenas marcam o começo e o fim do texto a ser exibido; elas não aparecem no resultado. Os parênteses indicam que o print é uma função. No Python 2, a instrução print é ligeiramente diferente; ela não é uma função, portanto não usa parênteses.

2.2.1 Operadores aritméticos

Depois do "Hello, World", o próximo passo é a aritmética. O Python tem operadores, que são símbolos especiais representando operações de computação, como adição e multiplicação. Os operadores +, - e * executam a adição, a subtração e a multiplicação, Finalmente, o operador ** executa a exponenciação; isto é, eleva um número a uma potência, como nos seguintes exemplos:

```
40 + 2
```

```
43 - 1
```

```
6 * 7
```

```
84 / 2
```

```
6 ** 2 + 6
```

Em algumas outras linguagens, o ^ é usado para a exponenciação, mas no Python é um operador bitwise, chamado XOR. Se não tiver familiaridade com operadores bitwise, o resultado o surpreenderá. Não abordaremos operadores bitwise neste curso, mas você pode ler sobre eles em http://wiki.python.org/moin/BitwiseOperators.

```
6 ^ 2
```

Um valor é uma das coisas básicas com as quais um programa trabalha, como uma letra ou um número. Alguns valores que vimos até agora foram 2, 42.0 e 'Hello, World!'.

Esses valores pertencem a tipos diferentes: 2 é um número inteiro, 42.0 é um número de ponto flutuante e 'Hello, World!' é uma string, assim chamada porque as letras que contém estão em uma sequência em cadeia.

Se não tiver certeza sobre qual é o tipo de certo valor, o interpretador pode dizer isso a você:

```
type(2)
```

```
type(42.0)
```

```
type('Hello World')
```

Nesses resultados, a palavra "class" (classe) é usada no sentido de categoria; um tipo é uma categoria de valores. Como se poderia esperar, números inteiros pertencem ao tipo *int*, strings pertencem ao tipo *str* e os números de ponto flutuante pertencem ao tipo *float*.

E valores como '2' e '42.0'? Parecem números, mas estão entre aspas como se fossem strings:

```
type('2')
```

Um dos recursos mais eficientes de uma linguagem de programação é a capacidade de manipular variáveis. Uma variável é um nome que se refere a um valor.

2.2.2 Instruções de atribuição

Uma instrução de atribuição cria uma nova variável e dá um valor a ela:

```
message = 'And now for something completely different'
n = 17
pi = 3.141592653589793
```

Esse exemplo faz três atribuições. A primeira atribui uma string a uma nova variável chamada message; a segunda dá o número inteiro 17 a n; a terceira atribui o valor (aproximado) de π a pi. Uma forma comum de representar variáveis por escrito é colocar o nome com uma flecha apontando para o seu valor. Este tipo de número é chamado de diagrama de estado porque mostra o estado no qual cada uma das variáveis está (pense nele como o estado de espírito da variável).

2.2.3 Nomes de variáveis

Os programadores geralmente escolhem nomes significativos para as suas variáveis – eles documentam o uso da variável. Nomes de variáveis podem ser tão longos quanto você queira. Podem conter tanto letras como números, mas não podem começar com um número. É legal usar letras maiúsculas, mas a convenção é usar apenas letras minúsculas para nomes de variáveis. O caractere de sublinhar (_) pode aparecer em um nome. Muitas vezes é usado em nomes com várias palavras, como *your_name* ou *airspeed_of_unladen_swallow*.

Se você der um nome ilegal a uma variável, recebe um erro de sintaxe:

```
76trombones = 'big parade'
```

```
more@ = 1000000
```

```
class = 'Advanced Theoretical Zymurgy'
```

`76trombones` é ilegal porque começa com um número. `more@` é ilegal porque contém um caractere ilegal, o `@`. Mas o que há de errado com `class`?

A questão é que class é uma das palavras-chave do Python. O interpretador usa palavras-chave para reconhecer a estrutura do programa e elas não podem ser usadas como nomes de variável. O Python 3 tem estas palavras-chave:

```
and         del         from        None        True
as          elif        global      nonlocal    try
assert      else        if          not         while
break       except      import      or          with
class       False       in          pass        yield
continue    finally     is          raise
def         for         lambda      return
```

Você não precisa memorizar essa lista. Na maior parte dos ambientes de desenvolvimento, as palavras-chave são exibidas em uma cor diferente; se você tentar usar uma como nome de variável, vai perceber.

2.2.4 Expressões e instruções

Uma expressão é uma combinação de valores, variáveis e operadores. Um valor por si mesmo é considerado uma expressão, assim como uma variável, portanto as expressões seguintes são todas legais:

```
n
```

```
n + 25
```

Quando você digita uma expressão no prompt, o interpretador a avalia, ou seja, ele encontra o valor da expressão. Neste exemplo, o n tem o valor 17 e n + 25 tem o valor 42.

Uma instrução é uma unidade de código que tem um efeito, como criar uma variável ou exibir um valor.

```
>>> n = 17
>>> print(n)
```

A primeira linha é uma instrução de atribuição que dá um valor a n. A segunda linha é uma instrução de exibição que exibe o valor de n. Quando você digita uma instrução, o interpretador a executa, o que significa que ele faz o que a instrução diz. Em geral, instruções não têm valores.

2.3 Exercícios

1) Inicialize o interpretador do Python e use-o como uma calculadora. Responda as seguintes perguntas:

   a) Quantos segundos há em 42 minutos e 42 segundos?
   b) Quantas milhas há em 10 quilômetros? Dica: uma milha equivale a 1,61 quilômetro.
   c) Se você correr 10 quilômetros em 42 minutos e 42 segundos, qual é o seu passo médio (tempo por milha em minutos e segundos)? Qual é a sua velocidade média em milhas por hora?

2) Crie uma conta no StackOverflow e monte seu perfil.

3 Programação e IA

In summary, this book has no content whatsoever.

3.1 StackOverflow

Stack Overflow é um site de perguntas e respostas, um fórum, para programadores profissionais e entusiastas. É o site principal da Stack Exchange Network. Foi criado em 2008 por Jeff Atwood e Joel Spolsky.Ele apresenta perguntas e respostas sobre uma ampla gama de tópicos em programação de computadores. Ele foi criado para ser uma alternativa mais aberta aos sites anteriores de perguntas e respostas, como o Experts-Exchange. O Stack Overflow foi vendido para a Prosus, um conglomerado de internet para consumidores com sede na Holanda, em 2 de junho de 2021 por US$ 1,8 bilhão.

O site serve como uma plataforma para os usuários fazerem e responderem perguntas e, por meio de associação e participação ativa, votarem em perguntas e respostas semelhantes ao Reddit e editarem perguntas e respostas de maneira semelhante a um wiki. Os usuários do Stack Overflow podem ganhar pontos de reputação e "emblemas"; por exemplo, uma pessoa recebe 10 pontos de reputação por receber um voto "para cima" em uma pergunta ou resposta a uma pergunta, e pode receber medalhas por suas valiosas contribuições, o que representa uma gamificação do tradicional Q&A local na internet. Os usuários desbloqueiam novos privilégios com um aumento na reputação, como a capacidade de votar, comentar e até editar as postagens de outras pessoas.

Em março de 2021, o Stack Overflow tinha mais de 14 milhões de usuários registrados e recebeu mais de 21 milhões de perguntas e 31 milhões de respostas. Esse fórum, juntamente com outros sites de perguntas e respostas de programação semelhantes, substituíram globalmente principalmente os livros de programação para referência de programação do dia-a-dia nos anos 2000, e hoje são uma parte importante da programação de computadores. Vocês lembram do nosso gráfico de participação de algumas das principais linguagens no total de perguntas do StackOverflow? Qual linguagem está em 1º lugar desde o fim de 2018?

Mas pera lá, o que significa esse monte de adjetivo? (fontes: computersciencewiki.org, geeksforgeeks.org, freecodecamp.org)

4 Tipos primitivos e objetos básicos

In summary, this book has no content whatsoever.

4.1 Strings

Strings não são como números inteiros ou de ponto flutuate. No Python, uma string nada mais é do que uma sequência ordenada de caracteres unicode. Eles são delimitados sempre por aspas (simplas ou duplas). Relembrando nosso primeiro "programa" da aula passada podemos atribuir a uma variável a string *Hello, World* usando aspas.

```
str1 = 'Hello, World'

print(str1)
print(type(str1))
```

```
Hello, World
<class 'str'>
```

Usando aspas duplas o resultado seria o mesmo

```
str2 = "Hello, World"

print(str2)
print(type(str2))
```

```
Hello, World
<class 'str'>
```

Note que podemos também usar uma sequência de três aspas duplas e escrever strings que percorrem várias linhas.

```
str3 = """Das Utopias

Se as coisas são inatingíveis...ora!
Não é motivo para não querê-las...
Que tristes os caminhos, se não fora
A presença distante das estrelas!

Mario Quintana
"""

print(str3)
print(type(str3))
```

```
Das Utopias

Se as coisas são inatingíveis...ora!
Não é motivo para não querê-las...
Que tristes os caminhos, se não fora
A presença distante das estrelas!

Mario Quintana

<class 'str'>
```

4.1.1 Operações básicas

Por ser uma sequência de caracteres e não um número, operações aritméticas (em geral) não são permitidas, mas outras operações, como *slicing*, o são. É possível acessar um caracter específico da sequência utilizando a posição desse caractere, utilizando o que chamamos de **índice**. No caso do string *'Hello, World'*, para acessar a segunda letra podemos utilizar colchetes após o string e dentro dele o índice referente à posição do 'e' no string:

```
print(str1)
```

```
Hello, World
```

```
str1[2]
```

```
'l'
```

Ué, mas porque que obtivemos como resposta o caractere 'l', que é o 3º na sequência, e não o 'e', que é o 2º elemento? Aqui vai uma particularidade da sintaxe do Python: **em se tratando de índices, o Python sempre começa a contagem em 0!**. Dessa forma, para acessar o primeiro caractere de *'Hello, World'* devemos pedir `str1[0]`, para acessar o segundo é preciso pedir `str1[1]` e assim por diante.

```
str1[1]
```

```
'e'
```

Podemos acessar contando de trás para frente e usando um índice negativo

```
str1[-11]
```

```
'e'
```

É possível que o índice seja uma expressão, mas deve sempre ser um valor inteiro.

```
n=0
str1[n+1]
```

```
'e'
```

```
str1[1.5]
```

```
TypeError: string indices must be integers, not 'float'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 str1[1.5]
TypeError: string indices must be integers, not 'float'
```

Para acessar uma **fatia** (*slice*) do string podemos utilizar um intervalo de índices (inclusive o índice inicial e exclusive o índice final). Por exemplo,

```
str1[0:5]
```

```
'Hello'
```

Além de ser possível calcular o número de caracteres em uma sequência string utilizando a função integrada `len()`, podemos "concatenar" ou "somar" strings utilizando apenas o sinal de +

```
len(str1)
```

```
12
```

```
str1 + ' --- ' + str2
```

```
'Hello, World --- Hello, World'
```

Note, porém, que strings são **imutáveis**, de modo que caso queira substituir um dos caracteres dentro de um string é preciso utilizar uma função específica para isso ou criar um novo string derivado do 1º. Apenas tentar substituir um dos caracteres de um string já definido não é permitido.

```
# Tentemos substituir o 'e' do str1 por 'a'
str1[1] = 'a'
```

```
TypeError: 'str' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[13], line 2
      1 # Tentemos substituir o 'e' do str1 por 'a'
----> 2 str1[1] = 'a'
TypeError: 'str' object does not support item assignment
```

4.1.2 Métodos de strings

As strings oferecem métodos que executam várias operações úteis. Um método é em essência uma sequência de instruções encapsuladas dentro de um único comando que recebe argumentos e devolve um valor. Embora a sintaxe seja diferente, a ideia é a mesma quando falamos de funções, objeto do nosso estudo daqui algumas aulas.

No caso dos métodos, temos que passar o nome da string que foi definida anteriormente seguida de '.' e depois do comando relacionado ao método específico. Dentre os principais métodos aplicáveis a strings e suas funcionalidades podemos citar:

1. **str_example.upper()**: devolve a string 'str_example' toda em letras maiúsculas.
2. **str_example.lower()**: devolve a string 'str_example' toda em letras minúsculas.

3. **str_example.strip()**: devolve a string 'str_example' retirando possíveis espaços em branco no início e no fim da string.
4. **str_example.startswith('xyz')**: testa se a string 'str_example' começa com a string 'xyz'.
5. **str_example.endswith('xyz')**: testa se a string 'str_example' termina com a string 'xyz'.
6. **str_example.find('xyz')**: procura a string 'xyz' dentro de 'str_example' e retorna o primeiro índice onde 'xyz' começa ou retorna -1 se nada for encontrado.
7. **str_example.replace('old','new')**: retorna uma string nova onde todas as ocorrências de 'old' encontradas em 'str_example' serão substituídas por 'new'.

Além desses principais, existem outros vários métodos para strings. Esse link é um bom ponto de partida para quem quiser conhecer outros exemplos.

```
str1 = ' Hello, World '

print(str1.upper())
print(str1.lower())
```

```
 HELLO, WORLD
 hello, world
```

```
print(str1.strip())
```

```
Hello, World
```

```
print(str1.startswith(' '))
print(str1.endswith('d'))
```

```
True
False
```

```
print(str1.find(','))
```

```
6
```

```
print(str1.replace('Hello','World'))
```

```
 World, World
```

4.1.3 Formatação e a instrução print

Um método sobre o qual não falamos, mas que é bastante interessante quando queremos, por exemplo, printar bonitinho o resultado de determinada operação é `s.format()`. Com esse método podemos converter uma variável numérica para uma formato específico e printá-la dentro de um string maior. Imagine, por exemplo, que estejamos interessados em printar o valor de π arredondado para 2 casas decimais apenas dentro de um string que diz isso. Podemos implementar isso da seguinte forma

```
pi = 3.1415926535
print('O valor de pi arredondado para 2 casas decimais é {:.2f}. Interessante, não?'.format(p
```

```
O valor de pi arredondado para 2 casas decimais é 3.14. Interessante, não?
```

Os colchetes dentro do string mostram onde que o número deve aparecer. Mais do que isso, definimos dentro do colchete o formato do número. Após os ':', o '.2' significa que queremos 2 casas decimais enquanto 'f' significa que queremos um formato de ponto fixo. Não vou entrar nos detalhes de todas as formatações possíveis, mas podemos ver um pouco mais disso aqui.

Podemos também usar mais de um número dentro do mesmo método format. Por exemplo,

```
print('O valor de pi arredondado para 2 casas decimais é {:.2f}. Com 4 casas decimais, no ent
```

```
O valor de pi arredondado para 2 casas decimais é 3.14. Com 4 casas decimais, no entanto, o r
```

4.1.4 Introdução a expressões regulares

Por vezes queremos encontrar um padrão específico de texto (e.g., placas de carro, e-mails ou números de telefone) dentro de um texto maior, para realizar algum tipo de coleta, limpeza ou mesmo substituição que um simples `str.replace()` não dá conta. Para realizar tal ação podemos utilizar as famosas **Expressões Regulares**, também conhecidas como *Regular Expressions* no inglês, ou simplesmente *Regex*.

As expressões regulares são em essência uma potente linguagem para especificar padrões de texto. De forma mais detalhada, é uma composição dos chamados **metacaracteres**, caracteres com funções especiais, que, agrupados entre si e em conjunto com caracteres literais, formam

uma sequência, uma expressão. Essa expressão é interpretada como uma regra que indicará sucesso se uma entrada de dados qualquer casar com essa regra, ou seja, obedecer exatamente a todas as suas condições.

Imagine que você tenha o string abaixo, que mostra o texto de um trecho de uma notícia da CNN sobre o resultado da Pesquisa Datafolha para presidente divulgada em 18/08/2022 (link para a matéria completa aqui).

```
pesquisa = """
Pesquisa Datafolha divulgada nesta quinta-feira (18) mostra o ex-presidente Luiz Inácio Lula
com 47% das intenções de voto na corrida pelo Palácio do Planalto. O presidente Jair Bolsona
O primeiro turno das eleições acontece em 2 de outubro.

Na sequência, aparecem Ciro Gomes (PDT), com 7%; Simone Tebet (MDB), com 2%, e Vera Lúcia (P
"""

print(pesquisa)
```

```
Pesquisa Datafolha divulgada nesta quinta-feira (18) mostra o ex-presidente Luiz Inácio Lula
com 47% das intenções de voto na corrida pelo Palácio do Planalto. O presidente Jair Bolsona
O primeiro turno das eleições acontece em 2 de outubro.

Na sequência, aparecem Ciro Gomes (PDT), com 7%; Simone Tebet (MDB), com 2%, e Vera Lúcia (P
```

E se quiséssemos, por exemplo, substituir todas as porcentagens de intenção de voto por 'ZZZ'? Note que as porcentagens são diferentes e não há uma repetição dos números que nos permita usar o `str.replace()` de uma vez só. No entanto, todas as porcentagens são representadas por 1 ou 2 números inteiros seguidos do símbolo %. Nesse caso, o mais indicado é utilizar as expressões regulares.

Os módulos e funções nativas do Python não nos trazem muito material para trabalhar com expressões regulares. Para operar com elas utilizaremos uma biblioteca de comandos chamada `re`. Para trazer para dentro do Python as funcionalidades dessa biblioteca precisamos utilizar a função `import` seguida do nome da biblioteca.

```
import re
```

Falaremos mais sobre importação de bibliotecas de comandos e funções mais a frente, mas por hora tenha na cabeça que para utilizar um conjunto de instruções disponível em alguma biblioteca importada é preciso utilizar o nome da biblioteca seguido de ponto e do nome da função dessa biblioteca que você quer utilizar. No caso, para realizar a substituição das

porcentagens no string *pesquisa* utilizaremos a função `sub()` de dentro da biblioteca `re`. Mas o que devemos colocar como input dessa função?

O mundo das expressões regulares é um mundo gigante e à parte, com conteúdo suficiente para preencher um outro curso. De forma geral, a combinação entre metacaracteres e caracteres literais é o que dá o padrão do texto pelo qual procuramos. Alguns dos metacaracteres-padrão são `. ? * + ^ | [ ] { } ( ) \`, cada um realizando uma função específica. Para o nosso caso utilizaremos basicamente a expressão regular dada por

```
[0-9]{1,2}%
```

Mas o que essa coisa bizarra diz de fato? A função buscará todo e qualquer elemento dentro dos colchetes (no caso os dígitos numéricos) que apareça uma ou duas vezes (código dentro dos colchetes) e que seja seguido pelo símbolo de porcentagem. Note que esse é o padrão de qualquer uma das porcentagens no nosso string *pesquisa*. Vamos ver o que acontece se usarmos isso dentro de `re.sub()`.

```
pesquisa2 = re.sub('[0-9]{1,2}%','ZZZ',pesquisa)
```

```
print(pesquisa)
```

```
Pesquisa Datafolha divulgada nesta quinta-feira (18) mostra o ex-presidente Luiz Inácio Lula
com 47% das intenções de voto na corrida pelo Palácio do Planalto. O presidente Jair Bolsonar
O primeiro turno das eleições acontece em 2 de outubro.

Na sequência, aparecem Ciro Gomes (PDT), com 7%; Simone Tebet (MDB), com 2%, e Vera Lúcia (PS
```

```
print(pesquisa2)
```

```
Pesquisa Datafolha divulgada nesta quinta-feira (18) mostra o ex-presidente Luiz Inácio Lula
com ZZZ das intenções de voto na corrida pelo Palácio do Planalto. O presidente Jair Bolsonar
O primeiro turno das eleições acontece em 2 de outubro.

Na sequência, aparecem Ciro Gomes (PDT), com ZZZ; Simone Tebet (MDB), com ZZZ, e Vera Lúcia
```

Conseguimos exatamente o que a gente queria. Boa, time!

Pare um pouco e pense sobre a aplicabilidade dessa ferramenta dentro de um mundo repleto de dados não-estruturados, como tweets e notícias. **O potencial de uso é gigante!** Mas como dissemos, o mundo de *regex* é muito grande e existem cursos e livros que se dedicam

integralmente a estudar essa linguagem de padrões textuais. Uma boa referência introdutória é o livro Expressões Regulares: Uma Abordagem Divertida.

Se eu pudesse dar um conselho para o meu eu de 15 anos atrás seria: beba água e estude expressões regulares.

4.2 Listas

Como uma string, uma lista é uma sequência de valores. Em uma string, os valores são caracteres; em uma lista, eles podem ser de qualquer tipo. Podemos ter uma lista de strings, uma lista de valores numéricos ou mesmo uma lista de listas, combinando strings, números e mesmo outros tipos de objetos que ainda veremos, como tuplas, dicionários e dataframes.

Uma lista é delimitada por colchetes e os elementos, ou itens, pertencentes a ela são separados por vírgula. Para definir uma lista com 5 números inteiros, ordenados de forma sequencial e começando em 1 devemos escrever a seguinte linha de código:

```
lista1 = [1,2,3,4,5]

print(lista1)
print(type(lista1))
```

```
[1, 2, 3, 4, 5]
<class 'list'>
```

Podemos definir uma lista de strings, uma lista mista de strings e números, e uma lista composto por outras listas (lista aninhada):

```
lista2 = ['Danilo Souza','Claudio Lucinda']
lista3 = ['Turma 2024201',46.0,'Turmas 2024202',49,'Turmas 2024221',81]
lista4 = [lista1, lista2]

print(lista2)
print(lista3)
print(lista4)
```

```
['Danilo Souza', 'Claudio Lucinda']
['Turma 2024201', 46.0, 'Turmas 2024202', 49, 'Turmas 2024221', 81]
[[1, 2, 3, 4, 5], ['Danilo Souza', 'Claudio Lucinda']]
```

De forma análoga à listas com elementos não vazios, é possível definir uma lista vazia utilizando apenas os colchetes:

```
lista_vazia = []

print(lista_vazia)
```

```
[]
```

4.2.1 Operações básicas

A sintaxe para acessar os elementos de uma lista é a mesma que para acessar os caracteres de uma string: o operador de colchete. A expressão dentro dos colchetes especifica o índice ou o intervalo de índices. **Lembrando que o índice no Python começa em ZERO e não em UM**

```
print(lista2[0])
print(lista2[1])
print(lista3[-1])
print(lista3[0:2])
```

```
Danilo Souza
Claudio Lucinda
81
['Turma 2024201', 46.0]
```

Diferente das strings, listas são mutáveis. Quando o operador de colchete aparece do lado esquerdo de uma atribuição, ele identifica o elemento da lista que será atribuído:

```
numbers = [42, 123]
numbers[1] = 5
numbers
```

```
[42, 5]
```

O segundo elemento de numbers (índice 1), que era 123, agora é 5.

Índices de listas funcionam da mesma forma que os índices de strings:

- Qualquer expressão de números inteiros pode ser usada como índice.

- Se tentar ler ou escrever um elemento que não existe, você recebe um `IndexError`.
- Se um índice tiver um valor negativo, ele conta de trás para a frente, a partir do final da lista.

O operador `in`, que serve ao propósito de testar a existência de determinado elemento dentro de um objeto específico, também funciona com listas:

```
cheeses = ['Cheddar', 'Gorgonzola', 'Gouda']
'Edam' in cheeses
```

```
False
```

```
'Brie' in cheeses
```

```
False
```

Assim como com strings, é possível calcular o número de elementos em uma lista utilizando a função integrada `len()` e "concatenar" ou "somar" listas utilizando apenas o sinal de +

```
print(len(cheeses))
```

```
3
```

```
lista_soma = lista2 + cheeses
print(lista_soma)
```

```
['Danilo Souza', 'Claudio Lucinda', 'Cheddar', 'Gorgonzola', 'Gouda']
```

4.2.2 Métodos de listas

As listas também possuem métodos bastante úteis, que nos facilitam a vida em várias dimensões. Dentre os principais métodos aplicáveis a listas e suas funcionalidades podemos citar:

1. **lista_example.append()**: adiciona um novo elemento ao fim da lista "lista_example".
2. **lista_example.extend(lista2)**: toma a lista "lista_example" como argumento e adiciona todos os elementos de lista2 como novos elementos da lista inicial.
3. **lista_example.sort()**: classifica os elementos de "lista_example" em ordem ascendente.

4. **lista_example.remove(x)**: exclui o elemento igual a "x" de "lista_example". É um método bastante útil quando queremos excluir um elemento específico, mas não sabemos sua posição dentro da lista.

A maior parte dos métodos de listas são nulos; eles alteram a lista e retornam None. Se você escrever `t = t.sort()` por acidente, ficará desapontado com o resultado.

```
t = ['a', 'b', 'c']
t.append('d')
print(t)
```

```
['a', 'b', 'c', 'd']
```

```
t1 = ['a', 'b', 'c']
t2 = ['d', 'e']
t1.extend(t2)
print(t1)
```

```
['a', 'b', 'c', 'd', 'e']
```

```
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
print(t)
```

```
['a', 'b', 'c', 'd', 'e']
```

Assim como já explicamos, uma lista é uma sequência de valores e uma string é uma sequência de caracteres, mas uma lista de caracteres não é a mesma coisa que uma string. Para converter uma string em uma lista de caracteres, você pode usar o comando `list`:

```
s = 'spam'
t = list(s)
print(t)
```

```
['s', 'p', 'a', 'm']
```

A função `list` quebra uma string em letras individuais. Se você quiser quebrar uma string em palavras, você pode usar o método `split()`. Esse método admite um argumento adicional, chamado *delimiter*, que especifica quais caracteres podem ser usados para demonstrar os limites das palavras. Isso é muito útil, por exemplo, quando queremos separar um texto em palavras e fazer a contagem de palavras que mais se repetem. Qual caracter deveríamos passar como argumento em `split()` nesse caso?

```
s = 'pining for the fjords'
t = s.split(' ')
print(t)
```

```
['pining', 'for', 'the', 'fjords']
```

```
s = 'spam-spam-spam'
t = s.split('-')
t
```

```
['spam', 'spam', 'spam']
```

O método `join()` é o contrário de `split()`. Ele toma uma lista de strings e concatena os elementos. `join()`, no entanto, é um método de string, então é preciso invocá-lo no string delimitador (por exemplo, " - ") e passar a lista de strings como parâmetro:

```
t = ['spam','spam','spam']
s = ' - '.join(t)
print(s)
```

```
spam - spam - spam
```

Além desses principais, existem outros vários métodos para listas. Esse link é um bom ponto de partida para quem quiser conhecer outros exemplos.

4.2.3 Operadores lógicos e variáveis booleanas

Agora faremos um pequeno desvio, que será bem útil daqui para frente, para falar de expressões e variáveis **booleanas**. Em resumo, uma expressão booleana é uma expressão que pode ser verdadeira ou falsa e dessa forma assumir apenas dois valores como resultado: **True** e **False**. Os exemplos seguintes usam o operador de igual no Python (==) que compara dois operandos e produz True se forem iguais e False se não forem:

```
5 == 5
```

```
True
```

```
5 == 6
```

```
False
```

```
lista1 = [1,2,3,4,5]
lista2 = [5,4,3,2,1]

lista1 == lista2
```

```
False
```

True e *False* são valores especiais que pertencem ao tipo *bool*; não são strings. Além disso, é possível fazer operações aritméticas com variáveis do tipo *bool* já que o Python entende *True* como sendo equivalente ao número 1 e *False* como sendo equivalente ao número 0.

```
print(type(True))
print(type(False))
```

```
<class 'bool'>
<class 'bool'>
```

```
print(True + True)
print(True + False)
print(False + False)
```

```
2
1
0
```

O operador == é um dos operadores relacionais dentro do Python, os outros são:

1. **x != y**: x não é igual a y
2. **x > y**: x é maior que y
3. **x < y**: x é menor que y
4. **x >= y**: x é maior ou igual a y

5. `x <= y`: x é menor ou igual a y

Embora essas operações provavelmente sejam familiares para você, os símbolos do Python são diferentes dos símbolos matemáticos. Um erro comum é usar apenas um sinal de igual (=) em vez de um sinal duplo (==). Lembre-se de que = é um operador de atribuição e == é um operador relacional. Não existe => ou =<.

Há três operadores lógicos: `and`, `or` e `not`. A semântica (significado) destes operadores é semelhante ao seu significado em inglês. Por exemplo, `x>=0 and x<=10` só é verdade se x for maior que 0 **e** menor que 10. `n%2 == 0 or n%3 == 0` é verdadeiro se uma ou as duas condição(ões) for(em) verdadeira(s), isto é, se o número n for divisível por 2 ou 3 (caso você não esteja familiarizado com a divisão pelo piso e o operador módulo, tente brincar um pouco com `//` e com `%`). Finalmente, o operador `not` nega uma expressão booleana, então `not (x > y)` é verdade se `x > y` for falso, isto é, se x for menor que ou igual a y.

Falando estritamente, os operandos dos operadores lógicos devem ser expressões booleanas, mas o Python não é muito estrito. Qualquer número que não seja zero é interpretado como *True*:

```
42 and True
```

```
True
```

Esta flexibilidade tem sua utilidade, mas há algumas sutilezas relativas a ela que podem ser confusas. Assim, pode ser uma boa ideia evitá-la (a menos que você tenha certeza absoluta do que está fazendo).

4.2.4 Execução condicional

Para escrever programas úteis, quase sempre precisamos da capacidade de verificar condições e mudar o comportamento do programa de acordo com elas. Instruções condicionais nos dão esta capacidade. A forma mais simples é a instrução `if`:

```
x=5

if x > 0:
    print('x é positivo')
```

```
x é positivo
```

A expressão booleana depois do `if` é chamada de condição. Se for verdadeira, a instrução indentada é executada. Se não, nada acontece. *Aqui vale mais um adendo*: uma característica muito importante da sintaxe do Python é justamente a **identação**. Diferentemente de outras linguagens de programação, a identação exerce papel importante aqui, já que é através dela que se determina onde se inicia e onde termina um bloco de código, que pode ser uma expressão condicional ou mesmo uma função, como veremos mais a frente. Além de economizar várias chaves ("{" e "}") e vários "end", a identação exerce um papel estético importante ao permitir uma melhor visualização do código como um todo.

Não há limite para o número de instruções que podem aparecer no corpo de uma instrução if, mas deve haver pelo menos uma. Ocasionalmente, é útil ter um corpo sem instruções (normalmente como um espaço reservado para código que ainda não foi escrito). Neste caso, você pode usar a instrução pass, que não faz nada.

```
if x < 0:
    pass
```

4.2.5 Execução alternativa

Uma segunda forma da instrução if é a "execução alternativa", na qual há duas possibilidades e a condição determina qual será executada. A sintaxe pode ser algo assim:

```
x = 5

if x % 2 == 0:
    print('x é par')
else:
    print('x é ímpar')
```

```
x é ímpar
```

Se o resto quando x for dividido por 2 for 0, então sabemos que x é par e o programa exibe uma mensagem adequada. Se a condição for falsa, o segundo conjunto de instruções é executado. Como a condição deve ser verdadeira ou falsa, exatamente uma das alternativas será executada. As alternativas são chamadas de ramos (branches), porque são ramos no fluxo da execução.

Podemos usar a instrução if para testar o tamanho de uma lista também:

```
lista1 = [1,2,3,4,5]

if len(lista1) > 2:
```

```
    print('lista1 tem mais de dois elementos')
else:
    print('lista1 tem menos de dois elementos')
```

```
lista1 tem mais de dois elementos
```

4.2.6 Condicionais encadeadas

Às vezes, há mais de duas possibilidades e precisamos de mais que dois ramos. Esta forma de expressar uma operação de computação é uma condicional encadeada:

```
x = 5
y = 6

if x < y:
    print('x é menor do que y')
elif x > y:
    print('x é maior do que y')
else:
    print('x e y são iguais')
```

```
x é menor do que y
```

`elif` é uma abreviatura de "else if". Novamente, exatamente um ramo será executado. Não há nenhum limite para o número de instruções elif. Se houver uma cláusula else, ela deve estar no fim, mas não é preciso haver uma. Cada condição é verificada em ordem. Se a primeira for falsa, a próxima é verificada, e assim por diante. Se uma delas for verdadeira, o ramo correspondente é executado e a instrução é encerrada. Mesmo se mais de uma condição for verdade, só o primeiro ramo verdadeiro é executado.

4.3 Tuplas

Agora falaremos de mais um tipo de objeto básico do Python, a tupla. Uma tupla é uma sequência de valores. Os valores podem ser de qualquer tipo, e podem ser indexados por números inteiros, portanto, nesse sentido, as tuplas são muito parecidas com as listas. A diferença importante é que as tuplas são **imutáveis**, assim como os strings.

Em resumo, uma tupla é uma lista de valores separados por vírgulas e em geral delimitado por parênteses (lembre que listas são delimitadas por colchetes):

```
t = ('a', 'b', 'c', 'd', 'e')

print(type(t))
```

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

Um único valor entre parênteses não é uma tupla. Para criar uma tupla com um único elemento, é preciso incluir uma vírgula final após o elemento, com ou sem os parênteses.

```
t1 = ('a')
t2 = 'a',
t3 = ('a',)

print(type(t1))
print(type(t2))
print(type(t3))
```

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

Outra forma de criar uma tupla é com a função integrada `tuple`. Sem argumentos, cria uma tupla vazia. Se os argumentos forem uma sequência (string, lista ou tupla), o resultado é uma tupla com os elementos da sequência:

```
t = tuple()
print(t)
```

```
()
```

```
t = tuple('lupins')
print(t)
```

```
('l', 'u', 'p', 'i', 'n', 's')
```

4.3.1 Operações básicas

A maior parte dos operadores de lista também funciona em tuplas. O operador de colchetes indexa um elemento e o operador de fatia seleciona vários elementos.

```
t = ('a', 'b', 'c', 'd', 'e')
print(t[0])
print(t[1:3])
```

```
a
('b', 'c')
```

Entretanto, se tentar alterar um dos elementos da tupla, vai receber um erro:

```
t[0] = 'A'
```

```
TypeError: 'tuple' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[58], line 1
----> 1 t[0] = 'A'
TypeError: 'tuple' object does not support item assignment
```

Como tuplas são imutáveis, você não pode alterar os elementos, mas pode substituir uma tupla por outra.

```
t = ('A',) + t[1:]
print(t)
```

```
('A', 'b', 'c', 'd', 'e')
```

Os operadores relacionais funcionam com tuplas e outras sequências. O Python começa comparando o primeiro elemento de cada sequência. Se forem iguais, vai para os próximos elementos, e assim por diante, até que encontre elementos que sejam diferentes. Os elementos subsequentes não são considerados (mesmo se forem muito grandes).

```
(0, 1, 2) < (0, 3, 4)
```

```
True
```

```
(0, 1, 2000000) < (0, 3, 4)
```

```
True
```

4.3.2 Atribuição de tuplas

Muitas vezes, é útil trocar os valores de duas variáveis. Com a atribuição convencional, é preciso usar uma variável temporária. Por exemplo, trocar a e b.

```
a=5
b=6

temp = a
a = b
b = temp

print(b)
```

```
5
```

Essa solução é trabalhosa; a atribuição de tuplas é mais elegante:

```
a=5
b=6

a, b = b, a

print(b)
```

```
5
```

O lado esquerdo é uma tupla de variáveis e o lado direito é uma tupla de expressões. Cada valor é atribuído à sua respectiva variável. Todas as expressões no lado direito são avaliadas antes de todas as atribuições.

O número de variáveis à esquerda e o número de valores à direita precisam ser iguais:

```
a, b = 1, 2, 3
```

```
ValueError: too many values to unpack (expected 2)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[64], line 1
----> 1 a, b = 1, 2, 3
ValueError: too many values to unpack (expected 2)
```

De forma geral, o lado direito pode ter qualquer tipo de sequência (string, lista ou tupla). Por exemplo, para dividir um endereço de email em um nome de usuário e um domínio, você poderia escrever:

```
addr = 'monty@python.org'
uname, domain = addr.split('@')
```

O valor de retorno do `split()` é uma lista com dois elementos; o primeiro elemento é atribuído a `uname`, o segundo a `domain`:

```
print(uname)
print(domain)
```

```
monty
python.org
```

4.3.3 Tuplas como valores de retorno

Falando estritamente, uma função só pode retornar um valor, mas se o valor for uma tupla, o efeito é o mesmo que retornar valores múltiplos. Por exemplo, se você quiser dividir dois números inteiros e calcular o quociente e resto, não é eficiente calcular `x/y` e depois `x%y`. É melhor calcular ambos ao mesmo tempo. A função integrada `divmod` toma dois argumentos e devolve uma tupla de dois valores: o quociente e o resto da divisão do primeiro termo pelo segundo termo. Você pode guardar o resultado como uma tupla:

```
print(7//3)
print(7%3)
```

```
2
1
```

```
t = divmod(7, 3)

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

```
(2, 1)
<class 'tuple'>
```

4.3.4 Operações nativas com tuplas

`zip` é uma função integrada que recebe duas ou mais sequências e devolve uma lista de tuplas onde cada tupla contém um elemento de cada sequência. O nome da função tem a ver com o zíper, que se junta e encaixa duas carreiras de dentes.

Este exemplo encaixa uma string e uma lista:

```
s = 'abc'
t = [0, 1, 2]
zip(s,t)
```

```
<zip at 0x18630538700>
```

O resultado é um objeto `zip` que sabe como percorrer os pares. O uso mais comum de `zip` é em um loop `for` (falaremos sobre loops mais adiante no curso):

```
for pair in zip(s, t):
    print(pair)
```

```
('a', 0)
('b', 1)
('c', 2)
```

Um objeto `zip` é um tipo de iterador, ou seja, qualquer objeto que percorre ou itera sobre uma sequência. Iteradores são semelhantes a listas em alguns aspectos, mas, ao contrário de listas, não é possível usar um índice para selecionar um elemento de um iterador. Se quiser usar operadores e métodos de lista, você pode usar um objeto `zip` para fazer uma lista:

```
list(zip(s, t))
```

```
[('a', 0), ('b', 1), ('c', 2)]
```

O resultado é uma lista de tuplas. Neste exemplo, cada tupla contém um caractere da string e o elemento correspondente da lista. Se as sequências não forem do mesmo comprimento, o resultado tem o comprimento da mais curta:

```
list(zip('Anne', 'Elk'))
```

```
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

4.4 Dicionários

Dicionários são um outro tipo de objeto bastante importante e útil em nossas aplicações. Dicionários são um dos melhores recursos do Python, eles são os blocos de montar de muitos algoritmos eficientes e elegantes.

Um dicionário se parece com uma lista, mas é mais geral. Em uma lista os índices têm que ser números inteiros, em um dicionário eles podem ser de (quase) qualquer tipo. Um dicionário contém uma coleção de índices, que se chamam **chaves** e uma coleção de valores. **Cada chave é associada com um único valor**. A associação de uma chave e um valor chama-se par chave-valor ou item.

Em linguagem matemática, um dicionário representa um mapeamento de chaves a valores, para que você possa dizer que cada chave "mostra o mapa" a um valor. Como exemplo, vamos construir um dicionário que faz o mapa de palavras do inglês ao espanhol, portanto as chaves e os valores são todos strings.

A função `dict` cria um novo dicionário sem itens. Como `dict` é o nome de uma função integrada, você deve evitar usá-lo como nome de variável.

```
eng2sp = dict()
eng2sp
```

```
{}
```

As chaves {} representam um dicionário vazio. Para acrescentar itens ao dicionário, você pode usar colchetes:

```
eng2sp['one'] = 'uno'
```

Esta linha cria um item que mapeia da chave 'one' ao valor 'uno'. Se imprimirmos o dicionário novamente, vemos um par chave-valor com dois pontos entre a chave e o valor:

```
eng2sp
```

```
{'one': 'uno'}
```

Este formato de saída também é um formato de entrada. Por exemplo, você pode criar um dicionário com três itens:

```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
eng2sp
```

```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Ué, mas a ordem parece diferente daquela que definimos, não? Se você digitar o mesmo exemplo no seu computador, pode receber um resultado diferente. Em geral, a ordem dos itens em um dicionário é imprevisível. No entanto, isso não é um problema porque os elementos de um dicionário nunca são indexados com índices de números inteiros. Em vez disso, você usa as chaves para procurar os valores correspondentes. A chave `'two'`, por exemplo sempre mapeia ao valor `'dos'`, assim a ordem dos itens não importa.

```
eng2sp['two']
```

```
'dos'
```

Se a chave não estiver no dicionário, você recebe uma exceção:

```
eng2sp['four']
```

```
KeyError: 'four'
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[78], line 1
----> 1 eng2sp['four']
KeyError: 'four'
```

Assim como em strings e listas, a função `len` também é compatível com dicionários. Nesse caso ela devolve o número de pares chave-valor.

```
len(eng2sp)
```

```
3
```

O operador `in` funciona em dicionários também. Ele acusa se algo aparece como chave no dicionário (aparecer como valor não é o suficiente).

```
'one' in eng2sp
```

```
True
```

```
'uno' in eng2sp
```

```
False
```

Para ver se algo aparece como um valor em um dicionário, você pode usar o método `values()`, que devolve uma coleção de valores, e então usar o operador `in`:

```
vals = eng2sp.values()
'uno' in vals
```

```
True
```

4.4.1 Operações básicas

Podemos adicionar um novo par de chave e valor ao dicionário usando colchetes, mas caso a chave já exista no dicionário, podemos atualizar o valor.

```
y = {}
y['one'] = 1
y['two'] = 2
print(y)
```

```
{'one': 1, 'two': 2}
```

```
y['two'] = 'dos'
print(y)
```

```
{'one': 1, 'two': 'dos'}
```

O valor-chave deve ser um tipo de dados imutável, por essa razão, se você tentar definir um dicionário com um tipo de dados mutável, o Python levantará um erro de exceção.

A instrução `del` pode ser usada para remover uma entrada (par de valor de chave) de um dicionário.

```
y = {'one': 1, 'two': 2}
print(y)
```

```
{'one': 1, 'two': 2}
```

```
del y['two']
print(y)
```

```
{'one': 1}
```

Mais uma vez, podemos utilizar a função nativa `len` para retornar o número de entradas (pares de valor de chave) em um dicionário.

```
x = {'one': 0, 'two': 2}
print(len(x))
```

```
2
```

Tente acessar uma chave que não está em um dicionário e você receberá um erro de exceção do Python. Para lidar com essa exceção, você pode usar mais uma vez o operador `in` que testa se existe uma chave em um dicionário. Esse operador retorna *True* se o dicionário tiver um valor armazenado sob a chave fornecida e *Falso* caso contrário.

```
y = {'one': 1, 'two': 2}
del y['three']
```

```
KeyError: 'three'
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[88], line 2
      1 y = {'one': 1, 'two': 2}
----> 2 del y['three']
KeyError: 'three'
```

```
'three' in y
```

```
False
```

4.4.2 Métodos de dicionários

Dentre os principais métodos aplicáveis a dicionários e suas funcionalidades podemos citar:

1. **dict_example.update(x)**: atualiza o dicionário "dict_example" com todos os pares de valor-chave de um segundo dicionário "x". Os valores de chaves, que são comuns a ambos os dicionários, do segundo dicionário irão se sobrepor aos valores das chaves do primeiro dicionário.
2. **dict_example.keys(x)**: permite que você obtenha todas as chaves no dicionário. Muitas vezes é usado dentro de uma série de instruções repetida várias vezes para iterar sobre o conteúdo de um dicionário.
3. **dict_example.items()**: retorna todas as chaves do dicionário "dict_example" e seus valores associados como uma sequência de tuplas.
4. **dict_example.get(x,y)**: devolve o valor associado a uma chave "x" se o dicionário contiver essa chave. Caso o dicionário não contenha a chave, você pode especificar um segundo argumento opcional "y" para retornar um valor padrão (se o argumento não estiver incluído o método retornará *None*).
5. **dict_example.setdefault(x,y)**: é semelhante ao método `get()`: ele retorna o valor associado a uma chave "x" se o dicionário contiver essa chave, mas caso o dicionário não contenha a chave, este método criará um novo elemento no dicionário (par de valor de chave), onde o primeiro argumento neste método é a chave, e o segundo argumento ("y") é o valor. O segundo argumento é opcional, mas se isso não for incluído, o valor atrelado a essa nova chave será *None*.

Além desses principais, existem outros vários métodos para dicionários. Esse link é um bom ponto de partida para quem quiser conhecer outros exemplos.

```
x = {'one': 0, 'two': 2}
y = {'one': 1, 'three': 3}
print(x)

x.update(y)
print(x)
```

```
{'one': 0, 'two': 2}
{'one': 1, 'two': 2, 'three': 3}
```

```
x = {'one': 1, 'two': 2}
print(x.keys())
```

```
dict_keys(['one', 'two'])
```

```
x = {'one': 1, 'two': 2}
print(x.items())
```

```
dict_items([('one', 1), ('two', 2)])
```

```
y = {'one': 1, 'two': 2}
print(y.get('one'))
print(y.get('three'))
print(y.get('three', 'The key does not exist.'))
```

```
1
None
The key does not exist.
```

```
y = {'one': 1, 'two': 2}
print(y.setdefault('three', '3'))
print(y.setdefault('two', 'dos'))
print(y)
```

```
3
2
{'one': 1, 'two': 2, 'three': '3'}
```

```
print(y.setdefault('four'))
print(y)
```

```
None
{'one': 1, 'two': 2, 'three': '3', 'four': None}
```

4.5 Exercícios

1. Considere a seguinte lista: `fruit = ['pear', 'orange', 'apple', 'grapefruit', 'apple', 'pear']`. Use uma função de lista para dizer o índice da primeira ocorrência de `apple`
2. Usando a lista do exercício anterior, use uma função de lista para retornar o número de veze que `apple` ocorre.

3. Existem duas listas abaixo. Escreva um programa que as converta em um dicionário em que o item de `keys` é a chave e o item de `values` é o valor

```
keys = ['Ten', 'Twenty', 'Thirty']
values = [10, 20, 30]
```

Output esperado do último exercício

```
{'Ten': 10, 'Twenty': 20, 'Thirty': 30}
```

```
{'Ten': 10, 'Twenty': 20, 'Thirty': 30}
```

5 Controle de fluxo e iteração

X bla bla bla

5.1 Repetição simples e a instrução *for*

Os computadores muitas vezes são usados para automatizar tarefas repetitivas. A repetição de tarefas idênticas ou semelhantes sem cometer erros é algo que os computadores fazem muito bem e as pessoas não. Em um programa de computador, a repetição também é chamada de iteração.

De maneira um pouco mais formal, iteração significa executar o mesmo bloco de código repetidamente, potencialmente muitas vezes. Uma estrutura de programação que implementa a iteração é chamada de *loop*. A forma mais simples de iteração é a chamada *iteração definida*, em que o número de vezes que o bloco designado será executado é especificado explicitamente no momento em que o *loop* é iniciado.

Para atingir o objetivo proposto pela *iteração definida* utilizaremos no Python a instrução `for`. Um loop `for` tem duas partes: um cabeçalho especificando a iteração, que termina em dois pontos, e um corpo **identado** que é executado uma vez por iteração. O corpo pode conter qualquer número de instruções, mas o Python só reconhecerá como parte do corpo o código que estiver identado em relação ao cabeçalho.

No Python o `for` tem a cara abaixo

```
for <elemento> in <objeto>:
    <instruções>
```

Note que `<objeto>` em geral se refere à uma sequência, seja ela uma lista, uma tupla, um string ou qualquer outro objeto pelo qual seja possível percorrer. Para cada `<elemento>` dentro da sequência `<objeto>` o conjunto de instruções em `<instruções>`será executado. Após percorrer todos os elementos pertencentes à sequência `<objeto>`, o Python para a execução do *loop*.

Antes de partir para os exemplos, porém, vamos dar um passo atrás e falar mais sobre algo que apenas citamos algumas aulas atrás que é a importação de bibliotecas de comando e de conjuntos de funções que não estão contidas nas funções e operações nativas do Python.

5.1.1 Importando bibliotecas de comandos e conjuntos de funções

Sempre que quisermos trazer algo "de fora", devemos carregar as bibliotecas específicas, seja por completo ou apenas um subconjunto de suas funções. Fazemos isso por meio do comando `import`. Vamos utilizar como exemplo a biblioteca `NumPy`, uma das principais bibliotecas de comando do Python em se tratando de operações algébricas e matriciais.

Dedicaremos uma aula inteira para trabalhar com o `NumPy` daqui algumas semanas, mas por hora trabalharemos apenas com as funções geradoras de números aleatórios, em especial a função geradora de números aleatórios distribuídos de acordo com uma Distribuição Uniforme padrão (vocês devem se lembrar das características dessa distribuição das aulas de estatística, mas caso ainda restem dúvidas sempre existe o Wikipedia).

Há 3 formas de trabalhar com essa função específica, contida no `NumPy`:

1. Podemos importar a biblioteca inteira

```
import numpy
```

2. Podemos importar a biblioteca inteira mas dando-lhe um "apelido"

```
import numpy as np
```

3. Podemos importar apenas a função que nos interessa nesse caso

```
from numpy.random import uniform
```

Qual a diferença entre esses métodos? No Python, sempre que formos utilizar uma função de algum pacote "de fora" é preciso dizer ao Python de onde que essa função está vindo. Se quisermos usar a função `uniform()` através do 1º método, por exemplo, é preciso chamá-la utilizando o nome do pacote, nesse caso `numpy.random.uniform()`. O 2º método encurta esse nome de tal forma que é possível chamar a mesma função usando `np.random.uniform()`. Por fim, no 3º caso basta chamar a função diretamente, isto é, `uniform()`.

Mas se o 3º caso é mais direto, porque não usá-lo sempre? Por duas razões bem simples: (i) usar a sintaxe dos 2 primeiros casos torna o código mais compreensível e depurável, já que no caso de algum problema de execução sabemos onde procurar a resposta, e (ii) é comum pacotes distintos usarem o mesmo nome para funções que fazem operações distintas, o que pode gerar problemas de incompatibilidade e/ou de executarmos algo diferente daquilo que gostaríamos de executar.

Na maioria das vezes, acabamos utilizando o 2º método, já que isso permite reduzir linhas desnecessárias de código ao mesmo tempo que facilita a replicabilidade e a atividade de depuração do código.

Voltemos então ao nosso exercício de gerar números aleatórios uniformemente distribuídos:

```
import numpy as np
```

Imagine agora que estejamos interessados em criar um `for` loop que printa 10 números aleatórios sorteados de uma distribuição uniforme que vai de 0 a 1. Utilizando o objeto `range()` e a lógica de listas:

```
print(range(0,10))
```

```
range(0, 10)
```

```
print(list(range(0,10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
for i in list(range(0,10)):

    u = np.random.uniform()
    print(u)
```

```
0.08746792467565734
0.14627934888861915
0.8667009589945069
0.1507023416127552
0.9017874216138307
0.9224517363939674
0.7832500550808872
0.32352150321065076
0.07310844885081003
0.14379188785764674
```

No *loop* acima não utilizamos o elemento `i` em nenhum momento, de modo que a lista nos serviu apenas para ditar o número de vezes que gostaríamos de rodar o bloco de instruções abaixo do cabeçalho, no caso 10. Podemos ir um pouco além.

```
for i in list(range(0,10)):

    u = np.random.uniform()
    print('Essa é a iteração '+str(i+1)+' e o número sorteado foi: {:.2f}'.format(u))
```

```
Essa é a iteração 1 e o número sorteado foi: 0.08
Essa é a iteração 2 e o número sorteado foi: 0.40
Essa é a iteração 3 e o número sorteado foi: 0.88
Essa é a iteração 4 e o número sorteado foi: 0.44
Essa é a iteração 5 e o número sorteado foi: 0.82
Essa é a iteração 6 e o número sorteado foi: 0.33
Essa é a iteração 7 e o número sorteado foi: 0.42
Essa é a iteração 8 e o número sorteado foi: 0.95
Essa é a iteração 9 e o número sorteado foi: 0.17
Essa é a iteração 10 e o número sorteado foi: 0.38
```

Note que nesse caso utilizamos tanto o elemento *iterável* `i`, quanto o resultado de cada sorteio individual `u`.

Como dito anterioremente, o loop `for` funciona com qualquer sequência ou objeto pelo qual seja possível percorrer. Podemos trabalhar, inclusive, com strings e listas como nos exemplos abaixo:

```
str1 = 'Vou nadaaaa'

for c in str1:
    print(c)
```

```
V
o
u

n
a
d
a
a
a
a
```

```
lista1 = ['Minha terra tem palmeiras','onde canta o sabiá','seno A cosseno B','seno B cossen

for f in lista1:
    print(f)
```

```
Minha terra tem palmeiras
```

```
onde canta o sabiá
seno A cosseno B
seno B cosseno A
```

5.1.2 *for* loop e instruções condicionais

Podemos utilizar instruções condicionais dentro de um loop `for` também. Mais do que isso, é possível utilizar esse tipo de instrução para interromper a repetição do código, mesmo antes do Python percorrer todos os elementos da sequência alvo.

```
for i in ['foo','bar','baz','qux']:

    if 'b' in i:
        break
    print(i)
```

```
foo
```

Note que no caso acima, o segundo elemento da lista `['foo','bar','baz','qux']` contém a letra 'b' mas o primeiro não. A instrução `if`, portanto, retorna `False` na primeira iteração e `True` na segunda. No momento em que ela retorna `True` o código identado `break` é executado e o loop para aí! Dessa forma, o único termo que a instrução print vai imprimir será o primeiro da lista, isto é, 'foo'.

5.1.3 Exercício de fixação

O último teorema de *Fermat* diz que não há nenhum número inteiro positivo *a,b,c* tal que

$$a^n + b^n = c^n$$

para quaisquer valores de n maiores do que 2.

Use o que você aprendeu com condicionais, operações aritméticas e instruções print para testar se o teorema se mantém, dada a lista de números inteiros abaixo. Note que ao fim de cada iteração, o programa deve exibir "Holy smokes! Fermat was wrong" caso o teorema não valha e "Fermat was right" caso você não tenha refutado um gênio dos tempos modernos.

```
a = [1,2,3,4,5,6,7,8,9,10]

b = [1,2,3,4,5,6,7,8,9,10]

n = [3,4,5,6,7,8,9,10,37,52,89,100]
```

5.1.4 Usando *enumerate* e *zip*

Às vezes pode ser útil, dentro de um loop `for`, ter uma variável que muda em cada iteração do loop que possibilite fazer um controle mais direto sobre qual iteração está sendo executada em determinado momento. Em vez de criar e incrementar uma variável você mesmo, você pode usar a função nativa `enumerate()` para obter um contador e o valor do objeto iterável ao mesmo tempo! Para deixar claro como funciona a sintaxe nesse caso, vamos trabalhar em cima de uma lista de strings.

```
str1 = 'O Tata é foota, o Tata é foota'

# Algumas linhas de código adicionais para excluir pontuação e transformar o string em uma li
str1 = str1.replace('.','')
str1 = str1.replace(',','')
str1 = str1.replace('?','')
lista1 = str1.split()

for c in lista1:
    print(c)
```

```
O
Tata
é
foota
o
Tata
é
foota
```

Utilizando o `enumerate()` podemos, em uma mesma linha de código, printar não apenas cada string individual da lista, mas também a posição desse string dentro da lista!

```
for count,value in enumerate(lista1):

    print('Elemento '+str(count)+' = '+value)
```

```
Elemento 0 = O
Elemento 1 = Tata
Elemento 2 = é
Elemento 3 = foota
Elemento 4 = o
```

```
Elemento 5 = Tata
Elemento 6 = é
Elemento 7 = foota
```

Uma outra forma de criar esses mesmos resultados seria utilizando a função nativa `zip()`. Essa função nos permite iterar ao longo de duas ou mais sequências ao mesmo tempo, contanto que as sequências possuam o mesmo comprimento.

```
print(lista1)
```

```
['O', 'Tata', 'é', 'foota', 'o', 'Tata', 'é', 'foota']
```

```
lista2 = list(range(0,len(lista1)))
print(lista2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
len(lista1) == len(lista2)
```

```
True
```

```
for count,value in zip(lista2, lista1):

    print('Elemento '+str(count)+' = '+value)
```

```
Elemento 0 = O
Elemento 1 = Tata
Elemento 2 = é
Elemento 3 = foota
Elemento 4 = o
Elemento 5 = Tata
Elemento 6 = é
Elemento 7 = foota
```

Embora tenhamos utilizado a função `zip()` para chegar ao mesmo resultado de `enumerate()`, seu escopo de ação é muito mais geral já que nos permite percorrer dois objetos distintos ao mesmo tempo, sejam quais forem esses objetos. A única obrigatoriedade é que esses objetos tenham o mesmo tamanho, nada mais!

```
professores = ['Danilo Souza','Danilo Souza','Claudio Lucinda']
turmas = ['2024201','2024202','2024221']

for p,t in zip(professores,turmas):

    print('Nesse curso, a turma '+t+' é de responsabilidade do professor '+p)
```

```
Nesse curso, a turma 2024201 é de responsabilidade do professor Danilo Souza
Nesse curso, a turma 2024202 é de responsabilidade do professor Danilo Souza
Nesse curso, a turma 2024221 é de responsabilidade do professor Claudio Lucinda
```

5.1.5 *For* e *list comprehensions*

List comprehensions é uma das várias formas que o Python nos oferece para de criar, alterar e manipular listas. Sua sintaxe é bastante concisa e nos permite olhar para um exerício de iteração através de algo parecido com uma fórmula.

Imagine que estejamos interessados em criar uma lista que contenha a parte inteira da raiz quadrada de todo número de uma outra lista original. Usualmente faríamos isso utilizando um loop `for`:

```
lista1 = [1,4,9,16,25,36,49,64,81,100,121,144,169,196,225]
lista2 = []

for x in lista1:
    lista2.append(int(x**0.5))

print(lista2)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Alternativamente, podemos utilizar *list comprehensions* através da seguinte sintaxe:

```
lista2 = [int(x**0.5) for x in lista1]

print(lista2)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Em termos práticos isso funciona como se fosse um loop definido dentro de uma única linha de código. Mas porque aprender as duas formas? Será que alguma delas é mais eficiente? Vamos fazer um teste para esse exercicío da raiz quadrada, porém para uma lista maior. Utilizaremos também a função `time` do pacote `time` para calcular o tempo utilizado por cada um dos métodos.

```
import time
lista1 = list(range(1,1000000))

# list comprehension
start_comp = time.time()

list_comp  = [int(x**0.5) for x in lista1]

end_comp   = time.time()


# loop
start_loop = time.time()

list_loop = []
for x in lista1:
    list_loop.append(int(x**0.5))

end_loop   = time.time()

ratio = (start_loop - end_loop) / (start_comp - end_comp)
```

Qual abordagem será que levou menos tempo?

```
print('Tempo necessário para a realização dos cálculos utilizando list comprehensions: {:.4f}
print('Tempo necessário para a realização dos cálculos utilizando loop: {:.4f} segundos'.form
print('\nA abordagem de loop demorou {:.1%} mais tempo! Dê uma chance para list comprehensio
```

```
Tempo necessário para a realização dos cálculos utilizando list comprehensions: 0.0898 segun
Tempo necessário para a realização dos cálculos utilizando loop: 0.1227 segundos

A abordagem de loop demorou 36.6% mais tempo! Dê uma chance para list comprehensions ;)
```

5.2 Reatribuição e atualização de variáveis

Pode ser que você já tenha descoberto que é permitido fazer mais de uma atribuição para a mesma variável. Uma nova atribuição faz uma variável existente referir-se a um novo valor (e deixar de referir-se ao valor anterior).

```
x = 5
print(x)
```

```
5
```

```
x = 7
print(7)
```

```
7
```

A primeira vez que exibimos x, seu valor é 5. Na segunda vez, seu valor é 7.

Neste ponto quero tratar de uma fonte comum de confusão. Como o Python usa o sinal de igual (=) para atribuição, é tentador interpretar uma afirmação como `a = b` como uma proposição matemática de igualdade, isto é, a declaração de que a e b são iguais. Mas esta é uma interpretação equivocada.

Em primeiro lugar, a igualdade é uma relação simétrica e a atribuição não é. Por exemplo, na matemática, se `a=7` então `7=a`. Mas no Python, a instrução `a = 7` é legal e `7 = a` não é. Além disso, na matemática, uma proposição de igualdade é verdadeira ou falsa para sempre. Se `a=b` agora, então `a` sempre será igual a `b`. No Python, uma instrução de atribuição pode tornar duas variáveis iguais, mas elas não precisam se manter assim:

```
a = 5
b = a # a e b agora são iguais

print('Valor de a: '+str(a)+'\nValor de b: '+str(b))

a = 3 # a e b não são mais iguais

print('\nValor de a: '+str(a)+'\nValor de b: '+str(b))
```

```
Valor de a: 5
Valor de b: 5

Valor de a: 3
Valor de b: 5
```

Quando redefinimos o valor de `a` para 3, não mudamos o valor de `b`, de modo que as variáveis deixam de ser iguais. A reatribuição de variáveis muitas vezes é útil, mas você deve usá-la com prudência. Se os valores das variáveis mudarem frequentemente, isso pode dificultar a leitura e depuração do código.

5.3 Repetição condicional e a instrução *while*

Em programação existem dois tipos de iteração, indefinidas e definidas:

- Com **iteração definida**, o número de vezes que o bloco designado será executado é especificado explicitamente no momento em que o loop é iniciado. É o exemplo do `for` loop com o qual trabalhamos anteriormente.
- Com **iteração indefinida**, o número de vezes que o loop é executado não é especificado explicitamente com antecedência. Em vez disso, o bloco designado é executado repetidamente enquanto alguma condição for atendida.

Para interação indefinida, a construção típica do Python é o `while`. A forma mais simples desse loop `while` é a seguinte:

```
while <expr>:
    <instruções>
```

Em que `instruções` representam uma ou mais linhas de código a serem executadas. Da mesma forma que nas outras estruturas, a **indentação** determina quais linhas vão ser executadas.

A expressão de controle, `<expr>`, normalmente envolve uma ou mais variáveis que são inicializadas antes de iniciar o loop e depois modificadas em algum lugar no corpo do loop. **Você precisa dessa expressão bem pensada, porque é ela que vai determinar o momento em que o loop para**

Quando um loop `while` é encontrado, `<expr>`é avaliado primeiro no contexto booleano. Se for verdadeiro, o corpo do loop é executado. Em seguida `<expr>`, é verificado novamente e, se ainda for verdadeiro, o corpo é executado novamente. Isso continua até `<expr>` se tornar falso, momento em que a execução do programa prossegue para a primeira instrução além do corpo do loop. **Dentro de `<instruções>` você precisa ter algo que modifique algum elemento de `<expr>`. Senão ou o loop não vai rodar ou vai rodar para sempre.** Uma das formas de interromper **à força** a execução de um loop é por meio das teclas `Ctrl+C`.

Considere o caso abaixo:

```
n = 5

while n > 0:
    n = n-1
    print(n)

print('\nÀ partir de agora as instruções fora do loop serão executadas.')
```

```
4
3
2
1
0

À partir de agora as instruções fora do loop serão executadas.
```

Veja o que está acontecendo neste exemplo:

O valor inicial de `n` é 5. A expressão no cabeçalho da instrução `while` na linha 2 é `n > 0`, o que é `True`, então o corpo do loop é executado. Dentro do corpo do loop redefinimos o valor de `n` para que seja igual ao valor anterior de `n` menos 1, e então o novo valor de `n` é impresso.

Quando o corpo do loop termina, a execução do programa retorna ao topo do loop na linha 2 e a expressão é avaliada novamente. Ainda é verdade, então o corpo é executado novamente e 3 é impresso. Isso continua até `n` se tornar 0. Nesse ponto, quando a expressão é testada, ela é falsa (0 não é maior do que 0) e o loop termina. A execução é retomada na primeira instrução após o corpo do loop, no caso a instrução print. Observe que a expressão de controle do loop `while` é testada primeiro, antes que qualquer outra coisa aconteça.

A seguir temos um loop while que usa uma lista:

```
a = ['foo', 'bar', 'baz']
print(a)

while a:

    a.pop(-1)
    print(a)
```

```
['foo', 'bar', 'baz']
['foo', 'bar']
['foo']
[]
```

Quando uma lista é avaliada em contexto booleano, ela é verdadeira se tiver elementos nela e falsa se estiver vazia. Neste exemplo, `a` é verdadeiro desde que tenha elementos nele. Quando todos os itens forem removidos com o método `.pop()` e a lista estiver vazia, `a` será falsa e o loop terminará.

5.3.1 Formas de Interromper um Loop

Em cada exemplo que você viu até agora, todo o corpo do loop `while` é executado em cada iteração. Python fornece duas palavras-chave que encerram uma iteração de loop prematuramente:

- A instrução Python `break` imediatamente encerra um loop inteiramente. Já vimos como ela funciona no caso do `for` loop e instruções condicionais.
- A instrução `continue`, por outro lado, encerra imediatamente a iteração do loop atual. A execução salta para o topo do loop e a expressão de controle é reavaliada para determinar se o loop será executado novamente ou terminará.

Um exemplo de `break`:

```
n = 5
print(n)

while n > 0:
    n = n - 1
    if n == 2:
        break
    print(n)

print('Loop ended.')
```

```
5
4
3
Loop ended.
```

Um exemplo de `continue`:

```
n = 5

while n > 0:
    n = n - 1
```

```
    if n == 2:
        continue
    print(n)
print('Loop ended.')
```

```
4
3
1
0
Loop ended.
```

5.4 Aplicação: Teorema Central do Limite

Obs: esse exercício é o mais complexo que fizemos até agora. Nele utilizamos grande parte dos conceitos que estudamos até aqui nesse curso, desde listas a loops e também a importação de bibliotecas de comandos. Você pode pular essa parte por enquanto, mas aconselho que aos poucos você tente replicá-lo, de forma a entender todas suas etapas com o tempo.

O **Teorema Central do Limite** é um resultado super importante em estatística e com aplicações nas mais diversas áreas. Em sua formulação mais simples, o teorema diz que:

A distribuição da média amostral da variável aleatória X aproxima-se cada vez mais de uma distribuição normal conforme aumenta o tamanho da amostra, independentemente da distribuição original de X.

Suponha que estejamos amostrando de uma variável aleatória X com média finita e igual a μ e desvio padrão finito e igual a σ. Então média e desvio padrão da distribuição amostral são representados por:

$$\mu_{\bar{X}} = \mu$$

$$\sigma_{\bar{X}} = \frac{\sigma}{\sqrt{N}}$$

Vamos apresentar uma implementação desse Teorema usando Python, em várias partes. A primeira parte mostra como tirar 10 números aleatórios entre -40 e 40, usando o `NumPy`. Mais uma vez utilizaremos a função `uniform()` já que elas sorteia números reais com igual probabilidade dentro de um intervalo especificado.

```
import numpy as np

x = np.random.uniform(-40, 40, 10)
print(x)
```

```
[-36.31499031 -15.29723203  -2.37473132  36.41872434 -17.57698432
 -37.38779931  23.15343652 -37.1241972   31.48018251 -14.25971981]
```

Note que se sorteamos 10 números novamente, o resultado será diferente

```
x = np.random.uniform(-40, 40, 10)
print(x)
```

```
[-31.32779051  -5.42754595 -25.80917991   2.81536598  -6.90718383
  11.84545662 -14.70044976  -9.36390243 -32.18177847   7.11433809]
```

Para tornar o nosso exemplo mais previsível, vamos usar uma função dentro do `NumPy` que faz com o que o sorteio dos números parta do mesmo lugar e, portanto, o resultado seja o mesmo.

```
np.random.seed(1)

x = np.random.uniform(-40, 40, 10)
print(x)
```

```
[ -6.63823962  17.62595948 -39.99085001 -15.81339419 -28.25952873
 -32.61291242 -25.09918309 -12.35514184  -8.25860206   3.10533872]
```

```
np.random.seed(1)

x = np.random.uniform(-40, 40, 10)
print(x)
```

```
[ -6.63823962  17.62595948 -39.99085001 -15.81339419 -28.25952873
 -32.61291242 -25.09918309 -12.35514184  -8.25860206   3.10533872]
```

Feito o sorteio de números aleatórios, o próximo passo é tirar a média desse conjunto de números:

```
np.mean(x)
```

```
np.float64(-14.829655377323416)
```

Como vocês podem notar, a média não deu igual a zero, embora a média da distribuição uniforme com números inteiros que vai de -40 a 40 seja igual a 0. E aí surge a mágica do Teorema Central do Limite. Ainda que uma amostra não tenha a média igual à média populacional, a **média das médias** vai convergindo para zero à medida em que extraímos novas amostras e calculamos novas médias.

Agora vou fazer isso em um loop, sorteando 50 números aleatórios por amostra e primeiro com 10 amostras:

```
# number of sample
n = 10
means = []

np.random.seed(1)

for j in list(range(0,n)):

    lista_sorteio = np.random.uniform(-40, 40, 50)
    x = np.mean(lista_sorteio)
    means.append(x)

means = [np.round(elem,2) for elem in means]
print(means)
np.mean(means)
```

```
[np.float64(-2.64), np.float64(0.38), np.float64(-2.3), np.float64(-0.13), np.float64(3.95),
2.02), np.float64(2.95)]
```

```
np.float64(0.6619999999999999)
```

Agora com 100 amostras

```
# number of sample
n = 100
means = []
```

```
np.random.seed(1)

for j in list(range(0,n)):

    lista_sorteio = np.random.uniform(-40, 40, 50)
    x = np.mean(lista_sorteio)
    means.append(x)

means = [np.round(elem,2) for elem in means]
print(means)
np.mean(means)
```

```
[np.float64(-2.64), np.float64(0.38), np.float64(-2.3), np.float64(-0.13), np.float64(3.95),
2.02), np.float64(2.95), np.float64(-2.02), np.float64(-1.95), np.float64(-
1.69), np.float64(1.3), np.float64(2.67), np.float64(1.64), np.float64(-
2.17), np.float64(0.71), np.float64(-3.31), np.float64(-0.82), np.float64(-
1.1), np.float64(2.47), np.float64(-0.34), np.float64(1.6), np.float64(0.52), np.float64(-
0.78), np.float64(2.61), np.float64(2.68), np.float64(0.98), np.float64(-
2.24), np.float64(-4.57), np.float64(5.56), np.float64(2.17), np.float64(3.94), np.float64(4
7.24), np.float64(0.77), np.float64(-5.73), np.float64(0.17), np.float64(5.3), np.float64(3.9
8.14), np.float64(-4.17), np.float64(-2.82), np.float64(-1.41), np.float64(-
7.05), np.float64(1.03), np.float64(1.15), np.float64(2.35), np.float64(-
2.7), np.float64(-3.12), np.float64(0.67), np.float64(-1.7), np.float64(-
0.95), np.float64(3.93), np.float64(1.95), np.float64(2.65), np.float64(-
2.48), np.float64(-5.42), np.float64(4.43), np.float64(1.88), np.float64(-
0.04), np.float64(-1.46), np.float64(1.04), np.float64(-1.54), np.float64(-
2.82), np.float64(-2.62), np.float64(-3.85), np.float64(-0.56), np.float64(1.08), np.float64
3.13), np.float64(-1.05), np.float64(3.14), np.float64(-5.01), np.float64(-
1.15), np.float64(-4.29), np.float64(5.22), np.float64(-5.29), np.float64(1.37), np.float64(-
8.62), np.float64(-7.99), np.float64(-3.2), np.float64(3.04), np.float64(0.36), np.float64(3
1.4), np.float64(-3.84), np.float64(0.81), np.float64(7.0)]
```

```
np.float64(0.024600000000000007)
```

Vamos deixar essa tentativa de aproximar a média das médias de μ mais automatizada:

```
expoentes = [1,2,3,4,5,6,7,8,9,10]
N = [2**exp for exp in expoentes]

means = []
```

```
np.random.seed(1)
for n in N:

    means_atual = []
    for j in list(range(0,n)):

        lista_sorteio = np.random.uniform(-40, 40, 100)
        x = np.mean(lista_sorteio)
        means_atual.append(x)

    means.append(np.mean(means_atual))
```

```
for m in means:

    print(np.round(m,2))
```

```
-1.17
0.92
0.37
-0.25
0.31
-0.32
0.23
-0.15
-0.06
0.03
```

Olha só a convergência aí, minha gente!

Note que até aqui a gente só mostrou que a **média das médias converge para a média populacional**. O Teorema, no entanto, é mais completo do que isso. Ele diz que a distribuição da média amostral é uma Normal e, além disso, possui aquelas características em relação à distribuição original. Ainda precisamos de mais algumas aulas para ir além e olhar essas outrs questões do Teorema, mas logo logo chegamos lá.

6 Funções

In summary, this book has no content whatsoever.

6.1 O que é uma função?

No contexto das linguagens de programação, uma função é uma sequência nomeada de instruções, que executa algum tipo de operação específica mas não necessariamente numérica, como é o caso das funções matemáticas. A ideia essencial por trás de uma função é a de juntar algumas tarefas comuns ou repetidas e criar uma função para que, em vez de escrever o mesmo código várias vezes, possamos chama-la pelo nome e reutilizar o conjunto de instruções nela contida sempre que necessário.

Caso o objetivo de dividir um programa em funções ainda não tenha ficado claro, saiba que:

- Criar uma nova função dá a oportunidade de nomear um grupo de instruções, o que deixa o seu programa mais fácil de ler e de depurar
- As funções podem tornar um programa menor, eliminando o código repetitivo. Depois, caso precise fazer alguma alteração, basta fazê-la em um lugar só.
- Dividir um programa longo em funções permite depurar as partes uma de cada vez e então reuni-las em um conjunto funcional.
- As funções bem projetadas muitas vezes são úteis para muitos programas. Uma vez escritas e depuradas, você pode reutilizar as funções em programas fora daquele para o qual elas foram originalmente construídas.

Existem várias funções nativas no Python (e.g., `type()` para encontrar o tipo de um objeto) e mesmo dentro de bibliotecas com as quais já trabalhamos um pouco (e.g., `numpy.random.uniform()` para sortear números aleatórios de acordo com uma distribuição uniforme). À partir de agora veremos como podemos criar nossas próprias funções dentro da linguagem!

No Python, a sintaxe de uma função é dada por:

```
def nome_da_funcao(argumentos):
    <instruções>
```

Assim como nos *loops*, para definir uma nova função no Python é preciso começar com uma palavra-chave, nesse caso `def`. O nome que daremos à função vem logo em seguida. É através desse nome, `nome_da_funcao`, que chamaremos essa função em outras partes do nosso código. Entre parênteses definimos os `argumentos` que a função recebe para realizar o conjunto de instruções em `<instruções>`.

Note, mais uma vez, que todo o bloco de código que estiver **identado** e abaixo da linha de cabeçalho da função fará parte da função. Assim como nos *loops*, a função termina quando passarmos a primeira linha de código não-identada.

6.2 Nossa primeira função

Vamos começar criando uma função simples, que tem por objetivo printar uma das frases mais conhecidas da história do cinema:

```
def frase_cinema():
    print('Que a força esteja com você!')
```

Note que nesse caso, o parênteses logo após o nome da função está vazio. Isso quer dizer que essa função não recebe argumentos, apenas printa a frase entre aspas sempre que for chamada. Para chamá-la, basta usar o nome `frase_cinema` seguido dos parênteses vazios:

```
frase_cinema()
```

```
Que a força esteja com você!
```

6.3 Argumentos de uma função

Na função anterior, não tínhamos nenhum argumento. Ou seja, toda vez que você chamar a função `frase_cinema`, ela vai fazer a mesma coisa. Pode até ser que seja o seu objetivo fazer exatamente isso – é uma forma de economizar código.

O ponto é que a ideia de função é muito mais poderosa do que simplesmente uma "abreviação" de um monte de linhas de código. A abstração implícita no conceito de função é poderosa o suficiente para garantir que a função retorne coisas diferentes caso você altere algum **argumento**. Vamos falar disso a seguir.

6.3.1 *Keyword arguments* e *default values*

Vamos então criar uma função que tenha argumentos.

```
def my_func(name,place):
    print(f"Olá {name}! Você é de {place}?")

my_func("Emily","Paris")
```

```
Olá Emily! Você é de Paris?
```

O que acontece se você especificar o `place` primeiro e depois o `name`? Vamos descobrir.

```
my_func("Hawaii","Robert")
```

```
Olá Hawaii! Você é de Robert?
```

Meio bizarro, não? A razão é que aqui temos os chamados **argumentos posicionais**. Ou seja, a função vai assumir que o primeiro argumento é o `name` e o segundo é o `place` não importa o que tenhamos passado como argumento. Para lidar com isso, a gente pode atribuir um nome a cada um dos argumentos, ou **palavra-chave**, que aí vai ser a palavra-chave, e não a posição, que vai determinar qual o valor atribuído a cada argumento.

```
my_func(place="Hawaii",name="Robert")
```

```
Olá Robert! Você é de Hawaii?
```

Vimos aqui que a posição passou a ser irrelevante porque colocamos os nomes de cada um dos argumentos. E se quiséssemos dar mais flexibilidade ainda à função, só especificando um subconjunto dos argumentos? Para que isso funcione, nós precisamos especificar os **valores-padrão** dos argumentos caso eles não sejam fornecidos.

Aqui tem uma função que faz isso.

```
def total_calc(bill_amount,tip_perc=10):

    total = bill_amount*(1 + tip_perc/100)
    total = round(total,2)
    print(f"Please pay ${total}")
```

Nesse caso, o argumento `bill_amount` é obrigatório. Por outro lado, o argumento `tip_perc` vai assumir o valor de 10 toda vez que ele não for explicitamente fornecido na chamada da função.

6.3.2 Argumentos arbitrários

Vamos começar fazendo algumas perguntas: * E se não soubermos o número exato de argumentos de antemão? * Podemos criar funções que funcionem com um número variável de argumentos?

A resposta é *sim*! E vamos criar essa função imediatamente. Vamos criar uma função simples `my_var_sum()` que retorna a soma de todos os números passados como argumento. No entanto, o número de argumentos pode ser potencialmente diferente cada vez que chamamos a função.

```
def my_var_sum(*args):

    sum = 0
    for arg in args:
        sum += arg

    print(f"The numbers that you have add up to {sum}")
```

Observe como a definição da função agora tem `*args` em vez de apenas o nome do parâmetro. No corpo da função, fazemos um loop em `args` até usarmos todos os argumentos. A função `my_var_sum` retorna a soma de todos os números passados como argumentos. Olha só o que acontece quando chamamos a função com diferentes números de argumentos:

```
my_var_sum(99,10,54,23)
my_var_sum(9,87)
my_var_sum(5,21,36,79,45,65)
my_var_sum(1)
```

```
The numbers that you have add up to 186
The numbers that you have add up to 96
The numbers that you have add up to 251
The numbers that you have add up to 1
```

Mas e se eu quiser passar não apenas uma sequência de valores, mas uma sequência de valores *com nomes*? Não se preocupe, existe uma possibilidade de fazer isso, usando o `**kwargs` na definição dos argumentos. Qual é a diferença entre `*args` e `**kwargs`? A diferença é que você vai passar uma sequência de tamanho arbitrário de parâmetros, cada um deles nomeado.

O Python vai entender o `**kwargs` como um dicionário. Cada elemento passado é um par chave-valor, e dentro da função você tem que desempacotar o valor da chave. Vamos seguir alguns exemplos.

```
def myFun(**kwargs):

    for key, value in kwargs.items():
        print("%s == %s" % (key, value))


myFun(first='Geeks', mid='for', last='Geeks')
```

```
first == Geeks
mid == for
last == Geeks
```

Podemos misturar os tipos de argumentos (posicionais versus `*args`/`*kwargs`)? Sim, como podemos ver no exemplo seguinte:

```
def myFun(arg1, **kwargs):

    for key, value in kwargs.items():
        print(arg1+" %s == %s" % (key, value))


myFun("Hi - ", first='Geeks', mid='for', last='Geeks')
```

```
Hi -  first == Geeks
Hi -  mid == for
Hi -  last == Geeks
```

6.3.3 Elementos dentro da função: execução condicional

As funções podem retornar objetos booleanos também, o que pode ser conveniente para esconder testes complicados dentro de funções. Por exemplo:

```
def is_divisible(x, y):

    if x % y == 0:
        print(True)
    else:
        print(False)

is_divisible(6, 4)
```

```
False
```

6.3.4 Aplicação: Teorema Central do Limite - Parte 1

O primeiro passo para replicarmos, na forma de uma função, a aplicação do Teorema Central do Limite da aula anterior é definir uma função que recebe como argumentos a distribuição da varíavel aleatória X, o intervalo no qual a variável está definida e o número de sorteios que faremos em cada amostra. Com o que aprendemos até agora podemos definir a função `func_tcl` abaixo:

```
def func_tcl(dist=None,intervalo=(0,1),n=100):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X! Volte 2 c
    else:
        x = dist(intervalo[0],intervalo[1],n)
        print(x)
```

Para todos os 3 argumentos definimos valores *default* que, caso não sejam alterados a função vai utilizá-los em suas operações. Vamos rodar a função com todos os valores *default* e ver o que acontece:

```
func_tcl()
```

```
Você esqueceu de carregar uma função que defina a distribuição de X! Volte 2 casas.
```

Nesse caso, a execução condicional `if` dentro da função acende já que não passamos nenhuma distribuição como default, apenas o valor `None`. Vamos então repetir o que fizemos na aula passada e utilizar a distribuição uniforme, no intervalo $[-40, 40]$ e sorteando apenas 10 números desta distribuição:

```
import numpy as np

func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=10)
```

```
[ 30.88040807  29.12899108  -6.15811447 -10.65267205  12.5144459
 -30.44578837 -37.0344741  -20.28896821  23.97685334 -11.88096816]
```

6.3.5 Elementos dentro da função: iteração

Da mesma forma que podemos ter condições lógicas dentro da função, podemos ter blocos de repetição (`for` e `while`) dentro da função, assim como no caso de `*args`. Vamos olhar para um outro exemplo em que a função recebe uma lista de strings como argumento e faz operações em cada um dos elementos, um a um:

```
states = [' Alabama ','Georgia!','Georgia','georgia','FlOrIda','south carolina##','West virgi

def clean_strings(lista_strings):
    result=[]

    for value in lista_strings:
        value = value.strip()
        value = value.title()
        value = value.replace('#','')
        value = value.replace('?','')
        value = value.replace('!','')
        result.append(value)

    print(result)

print(states)
print()
clean_strings(states)
```

```
[' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda', 'south carolina##', 'West virginia

['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida', 'South Carolina', 'West Virginia']
```

6.3.6 Aplicação: Teorema Central do Limite - Parte 2

Agora podemos dar mais um passo e incorporar o loop dentro da função, passando como novo argumento da função o número de amostras com o qual queremos trabalhar:

```
def func_tcl(dist=None,intervalo=(0,1),n=100, samples=10):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X!')
    else:
        means = []
```

```
        for j in range(0,samples):
            x_func_tcl = dist(intervalo[0],intervalo[1],n)
            mean_x = sum(x_func_tcl)/len(x_func_tcl)

            means.append(mean_x)

        print('Essa é a lista de médias:')
        print(means)
        print('\nE essa é a média das médias:')
        print(sum(means)/len(means))
```

```
# utilizamos a função seed mais uma vez para tornar os resultados previsíveis
np.random.seed(1)

func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=50,samples=10)
```

```
Essa é a lista de médias:
[np.float64(-2.6419520291841945), np.float64(0.382420445207494), np.float64(-
2.2971869003899186), np.float64(-0.12856511296217676), np.float64(3.945331018819081), np.floa
2.0177969126899638), np.float64(2.952952578448611)]

E essa é a média das médias:
0.662169619365357
```

Que bacana!!!

Mas note que o valor da média que nos interessa nesse caso só aparece printado quando chamamos a função. O que podemos fazer para conseguir guardá-lo como uma nova variável, por exemplo?

6.4 Valor de retorno de uma função

O comando `return` é usado para a saída de uma função de volta ao lugar no código de onde ela foi chamada. A sintaxe desse comando é simples:

```
return <expression_list>
```

Essa declaração pode ter um comando que é executado e o valor resultante devolvido. Se esse comando não for anexado ou não tiver nada além do `return`, a função é devolvida com o valor de `None`. Por exemplo:

```
def my_func(name,place):

    print(f"Olá {name}! Você é de {place}?")
    return

print(my_func("Jane","Paris"))
```

```
Olá Jane! Você é de Paris?
None
```

Funções que não contenham o comando `return` sempre retornarão um valor vazio. Quando queremos utilizar uma função com o objetivo de realizar algum tipo de operação e **guardar** o valor dessa operação, não podemos esquecer do `return`. Vejamos o exemplo da função que calcula a raiz quadrada de um número:

```
def raiz(x):

    fx = x**0.5

y = raiz(100)
print(y)
```

```
None
```

```
def raiz(x):

    fx = x**0.5
    return fx

y = raiz(100)
print(y)
```

```
10.0
```

6.4.1 Expectativa de vida de variáveis dentro da função

Quando você cria uma variável **dentro** de uma função ela é local, ou seja, só existe dentro da própria função. Por exemplo:

```
def concat_strings(str1,str2):

    texto_concatenado = str1 + ' ' + str2
    print(texto_concatenado)


texto1 = 'Que a força esteja com você,'
texto2 = 'jovem Padawan.'

concat_strings(texto1,texto2)
```

```
Que a força esteja com você, jovem Padawan.
```

Essa função recebe dois argumentos, concatena-os e exibe o resultado em uma única linha de texto. No entanto, assim que a função é encerrada, a variável `texto_concatenado` é deletada. O que acontece se tentarmos acessá-la?

```
print(texto_concatenado)
```

```
NameError: name 'texto_concatenado' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[22], line 1
----> 1 print(texto_concatenado)
NameError: name 'texto_concatenado' is not defined
```

6.4.2 Aplicação: Teorema Central do Limite - Parte 3

Retomando a função da parte 2:

```
def func_tcl(dist=None,intervalo=(0,1),n=100, samples=10):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X!')
    else:
        means = []
        for j in range(0,samples):
            x_func_tcl = dist(intervalo[0],intervalo[1],n)
            mean_x = sum(x_func_tcl)/len(x_func_tcl)
```

```
            means.append(mean_x)
            mean_of_means = sum(means)/len(means)

        print('Essa é a lista de médias:')
        print(means)
        print('\nE essa é a média das médias:')
        print(mean_of_means)
```

```
np.random.seed(1)

func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=50,samples=10)
```

```
Essa é a lista de médias:
[np.float64(-2.6419520291841945), np.float64(0.382420445207494), np.float64(-
2.2971869003899186), np.float64(-0.12856511296217676), np.float64(3.945331018819081), np.flo
2.0177969126899638), np.float64(2.952952578448611)]

E essa é a média das médias:
0.662169619365357
```

Como acabamos de ver, se quisermos acessar a variável `mean_of_means` o Python retornará um erro. As variáveis definidas dentro de uma função tem vida curta: elas existem apenas dentro da função!

```
print(mean_of_means)
```

```
NameError: name 'mean_of_means' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[25], line 1
----> 1 print(mean_of_means)
NameError: name 'mean_of_means' is not defined
```

No entanto, agora já sabemos o que a palavra-chave `return` faz dentro de uma função. Vamos utilizá-la!

```
def func_tcl(dist=None,intervalo=(0,1),n=100, samples=10):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X!')
    else:
```

```
        means = []
        for j in range(0,samples):
            x_func_tcl = dist(intervalo[0],intervalo[1],n)
            mean_x = sum(x_func_tcl)/len(x_func_tcl)

            means.append(mean_x)
            mean_of_means = sum(means)/len(means)

        return mean_of_means
```

```
np.random.seed(1)

func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=50,samples=10)
```

```
np.float64(0.662169619365357)
```

Maravilha, agora a função nos retorna apenas o que é do nosso interesse: a média das médias. Para chegar, finalmente, no mesmo resultado da aula passada basta colocar essa função dentro de um loop e tacar-lhe pau nesse carrinho, Marcos!

```
expoentes = [1,2,3,4,5,6,7,8,9,10]
Y = [2**exp for exp in expoentes]

means_of_means = []

np.random.seed(1)
for y in Y:

    mean_of_means = func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=100,samples=y)
    means_of_means.append(mean_of_means)
```

```
for m in means_of_means:

    print(np.round(m,2))
```

```
-1.17
0.92
0.37
-0.25
0.31
```

```
-0.32
0.23
-0.15
-0.06
0.03
```

Voilá!

Como exercício para casa, tente fazer alterações nessa função de modo que ela receba não um valor com o número de amostras, mas uma lista de números de amostras. A função deve cuspir como resultado a lista `means_of_means` e não apenas o valor `mean_of_means`. Pratique!

6.5 Documentação

Uma **docstring** é uma string no início de uma função, definida pelo usuário, que serve como documentação do que a função faz. A docstring vem logo depois da primeira linha que define a função e é delimitada por aspas triplas, o que permite que a string se estenda por várias linhas, como vocês devem se lembrar.

Vamos criar como exemplo uma função que printa o tipo do objeto que é passado como argumento:

```
def imprime_tipo(x):

    '''
    Função criada para a matéria EAE1106 - Métodos Computacionais para Economia
    Objetivo: função simples que imprime o tipo do objeto recebido como argumento.
    '''

    print(type(x))
```

Embora opcional, a documentação é uma **boa prática de programação**. A menos que você consiga se lembrar qual foi o cardápio do bandejão na semana passada, sempre documente seu código. Podemos acessar a documentação de determinada função utilizando o atributo `__doc__`.

```
print(imprime_tipo.__doc__)
```

```
Função criada para a matéria EAE1106 - Métodos Computacionais para Economia
Objetivo: função simples que imprime o tipo do objeto recebido como argumento.
```

Isso vale também para funções nativas e definidas em outros pacotes do Python. Por exemplo,

```
# documentação da função nativa len()
print(len.__doc__)
```

```
Return the number of items in a container.
```

```
import time

# documentação da função time() dentro do pacote time
print(time.time.__doc__)
```

```
time() -> floating-point number

Return the current time in seconds since the Epoch.
Fractions of a second may be present if the system clock provides them.
```

```
import numpy as np

# documentação da função uniform() dentro do pacote NumPy
print(np.random.uniform.__doc__)
```

```
        uniform(low=0.0, high=1.0, size=None)

        Draw samples from a uniform distribution.

        Samples are uniformly distributed over the half-open interval
        ``[low, high)`` (includes low, but excludes high).  In other words,
        any value within the given interval is equally likely to be drawn
        by `uniform`.

        .. note::
            New code should use the `~numpy.random.Generator.uniform`
            method of a `~numpy.random.Generator` instance instead;
            please see the :ref:`random-quick-start`.

        Parameters
        ----------
        low : float or array_like of floats, optional
```

```
    Lower boundary of the output interval.  All values generated will be
    greater than or equal to low.  The default value is 0.
high : float or array_like of floats
    Upper boundary of the output interval.  All values generated will be
    less than or equal to high.  The high limit may be included in the
    returned array of floats due to floating-point rounding in the
    equation ``low + (high-low) * random_sample()``.  The default value
    is 1.0.
size : int or tuple of ints, optional
    Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
    ``m * n * k`` samples are drawn.  If size is ``None`` (default),
    a single value is returned if ``low`` and ``high`` are both scalars.
    Otherwise, ``np.broadcast(low, high).size`` samples are drawn.

Returns
-------
out : ndarray or scalar
    Drawn samples from the parameterized uniform distribution.

See Also
--------
randint : Discrete uniform distribution, yielding integers.
random_integers : Discrete uniform distribution over the closed
                  interval ``[low, high]``.
random_sample : Floats uniformly distributed over ``[0, 1)``.
random : Alias for `random_sample`.
rand : Convenience function that accepts dimensions as input, e.g.,
       ``rand(2,2)`` would generate a 2-by-2 array of floats,
       uniformly distributed over ``[0, 1)``.
random.Generator.uniform: which should be used for new code.

Notes
-----
The probability density function of the uniform distribution is

.. math:: p(x) = \frac{1}{b - a}

anywhere within the interval ``[a, b)``, and zero elsewhere.

When ``high`` == ``low``, values of ``low`` will be returned.
If ``high`` < ``low``, the results are officially undefined
and may eventually raise an error, i.e. do not rely on this
function to behave when passed arguments satisfying that
```

```
inequality condition. The ``high`` limit may be included in the
returned array of floats due to floating-point rounding in the
equation ``low + (high-low) * random_sample()``. For example:

>>> x = np.float32(5*0.99999999)
>>> x
np.float32(5.0)


Examples
--------
Draw samples from the distribution:

>>> s = np.random.uniform(-1,0,1000)

All values are within the given interval:

>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True

Display the histogram of the samples, along with the
probability density function:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

Note do exemplo acima que uma docstring pode conter uma descrição detalhada do funcionamento de uma função, inclusive com exemplos de aplicação. Legal, né? Esse tipo de documentação também está disponível para as bibliotecas como um todo. Isso pode nos ajudar, por exemplo, a conhecer o conteúdo de uma determinada biblioteca.

```python
import numpy as np

# documentação do NumPy
print(np.__doc__)
```

```
NumPy
=====

Provides
  1. An array object of arbitrary homogeneous items
  2. Fast mathematical operations over arrays
  3. Linear Algebra, Fourier Transforms, Random Number Generation

How to use the documentation
----------------------------
Documentation is available in two forms: docstrings provided
with the code, and a loose standing reference guide, available from
`the NumPy homepage <https://numpy.org>`_.

We recommend exploring the docstrings using
`IPython <https://ipython.org>`_, an advanced Python shell with
TAB-completion and introspection capabilities.  See below for further
instructions.

The docstring examples assume that `numpy` has been imported as ``np``::

  >>> import numpy as np

Code snippets are indicated by three greater-than signs::

  >>> x = 42
  >>> x = x + 1

Use the built-in ``help`` function to view a function's docstring::

  >>> help(np.sort)
  ... # doctest: +SKIP

For some objects, ``np.info(obj)`` may provide additional help.  This is
particularly true if you see the line "Help on ufunc object:" at the top
of the help() page.  Ufuncs are implemented in C, not Python, for speed.
The native Python help() does not know how to view their help, but our
np.info() function does.

Available subpackages
---------------------
lib
    Basic functions used by several sub-packages.
```

```
random
    Core Random Tools
linalg
    Core Linear Algebra Tools
fft
    Core FFT routines
polynomial
    Polynomial tools
testing
    NumPy testing tools
distutils
    Enhancements to distutils with support for
    Fortran compilers support and more (for Python <= 3.11)

Utilities
---------
test
    Run numpy unittests
show_config
    Show numpy build configuration
__version__
    NumPy version string

Viewing documentation using IPython
-----------------------------------

Start IPython and import `numpy` usually under the alias ``np``: `import
numpy as np`.  Then, directly past or use the ``%cpaste`` magic to paste
examples into the shell.  To see which functions are available in `numpy`,
type ``np.<TAB>`` (where ``<TAB>`` refers to the TAB key), or use
``np.*cos*?<ENTER>`` (where ``<ENTER>`` refers to the ENTER key) to narrow
down the list.  To view the docstring for a function, use
``np.cos?<ENTER>`` (to view the docstring) and ``np.cos??<ENTER>`` (to view
the source code).

Copies vs. in-place operation
-----------------------------
Most of the functions in `numpy` return a copy of the array argument
(e.g., `np.sort`).  In-place versions of these functions are often
available as array methods, i.e. ``x = np.array([1,2,3]); x.sort()``.
Exceptions to this rule are documented.
```

Podemos usar sempre esse atalho caso desejemos conhecer as funcionalidades que uma biblioteca guarda por trás de suas cortinas!

6.6 Funções anônimas

Em Python, uma função anônima é uma função definida sem nome. Quem poderia imaginar, não é mesmo?

Enquanto as funções normais são definidas usando a palavra-chave `def` em Python, as funções anônimas são definidas usando a palavra-chave `lambda`. Portanto, funções anônimas também são chamadas de **funções lambda**. A estrutura usual de uma função lambda é a seguinte:

```
lambda <argumentos>: <expressão>
```

Usualmente a gente precisa de uma função lambda porque precisamos de uma função rápida por um período de tempo e/ou quando a gente usa técnicas mais poderosas que possuem funções como argumento, como `filter` e `map`. Vou fazer um exemplo com cada uma delas.

Usando filter

A função `filter()` em Python recebe uma função e uma lista como argumentos. A função é chamada com todos os itens da lista e uma nova lista é retornada contendo itens para os quais a função avalia `True`. Aqui está um exemplo de uso da função para filtrar apenas números pares de uma lista.

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

```
[4, 6, 8, 12]
```

Usando map

A função `map()` em Python recebe uma função e uma lista. A função é chamada com todos os itens da lista e uma nova lista é retornada contendo os itens retornados por essa função para cada item. Aqui está um exemplo de uso da função `map()` para dobrar todos os itens em uma lista.

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Parte II

Computação numérica, análise de dados e visualização

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

7 Arrays, matrizes e álgebra linear

Uma das principais aplicações para o Python bla bla bla

7.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental para computação científica em Python. Essa biblioteca é peça fundamental em outras bibliotecas igualmente importantes, como o Pandas e o Matplotlib. É uma biblioteca Python que tem como principal objeto o `ndarray`, um array multidimensional que guarda bastante semelhança com a ideia de vetores e matrizes, embora seja um objeto específico dentro da linguagem, com suas características e métodos próprios. O pacote contém também uma variedade de rotinas para operações rápidas em arrays, incluindo matemática, lógica, álgebra linear básica, operações estatísticas básicas e muito mais.

Mas o que é de fato um `ndarray`? É um objeto multidimensional que nos permite armazenar dados de forma sequencial e que podem ser acessados via indexação. Ué, mas isso é muito parecido com uma lista (ou um conjunto de listas). Qual a diferença então?

- NumPy arrays têm um tamanho fixo na criação, ao contrário das listas, que podem crescer. Alterar o tamanho de um ndarray criará um novo array e excluirá o original.
- Todos os elementos em um array devem ser do mesmo tipo de dados, diferentemente de listas, que são objetos mais genéricos. Isso facilita a gestão de memória e torna operações com esse tipo de objeto ordens de magnitude mais rápidas do que se utilizássemos listas.
- A maior velocidade e eficiência de armazenamento fazem do NumPy uma das bibliotecas mais utilizadas em aplicações matemáticas e científicas. Saber apenas as ferramentas nativas do Python, como listas, hoje já não é mais suficiente.

São muitas as qualidades do `NumPy` que fazem dele a melhor escolha quanto o assunto é lidar com objetos sequenciais, multidimensionais, e com os quais queremos operar tal qual vetores e matrizes. Mas chega de lenga lenga, vamos ao trabalho!

7.2 Elementos básicos do NumPy

Antes de tudo, é preciso importar o `NumPy`, já que se trata de uma biblioteca não nativa do Python.

```
import numpy as np
```

7.2.1 Arrays unidimensionais

Comecemos criando um `numpy.ndarray` do zero, contendo os números 1, 2 e 3. Podemos fazê-lo da seguinte forma:

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

```
[1 2 3]
<class 'numpy.ndarray'>
```

A sintaxe é essa mesma: um par de colchetes dentro dos parênteses. Se tentarmos passar sem os colchetes, o Python retornará um erro.

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

```
TypeError: array() takes from 1 to 2 positional arguments but 3 were given
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 a = np.array(1, 2, 3)
TypeError: array() takes from 1 to 2 positional arguments but 3 were given
```

Da mesma forma que uma sequência de 3 números inteiros, podemos criar um `numpy.ndarray` que repete 3 vezes o número 0 ou 3 vezes o número 1.

```
a = np.zeros(3)
b = np.ones(3)

print(a)
print(b)
```

```
[0. 0. 0.]
[1. 1. 1.]
```

Note que em ambos os casos os números aparecem com o ponto da casa decimal, o que é um indicativo de que estão armazenados como valores do tipo `float`. E se quiséssemos criar esses mesmos arrays, mas especificando que os números são inteiros, i.e., do tipo `int`?

```
a = np.zeros(3, dtype=int)
b = np.ones(3, dtype=int)

print(a)
print(b)
```

```
[0 0 0]
[1 1 1]
```

A função `numpy.linspace(x,y,z)` nos permite criar um array que vai de `x` até `y`, com `z` elementos igualmente espaçados.

```
a = np.linspace(0,8,5, dtype=int)

print(a)
```

```
[0 2 4 6 8]
```

Podemos acessar os elementos de um array qualquer utilizando a mesma ideia de indexação de listas:

```
print('Array a =',a)
print('\nPrimeiro elemento de a = ',a[0])
print('Segundo elemento de a = ',a[1])
print('Último elemento de a = ',a[-1])
print('Dois primeiros elementos de a = ',a[0:2])
```

```
Array a = [0 2 4 6 8]

Primeiro elemento de a =  0
Segundo elemento de a =  2
Último elemento de a =  8
Dois primeiros elementos de a =  [0 2]
```

7.2.2 Arrays multidimensionais

Como falamos anteriormente, um objeto do tipo `numpy.ndarray` é um array *n-dimensional* (por isso o `nd` em `ndarray`). Até agora trabalhamos apenas com uma dimensão, mas para as nossas aplicações é particularmente interessante o caso em que o número de dimensões é igual a 2, i.e., para o caso em que o array assume a forma de uma **matriz**.

Podemos criar esse tipo de array usando a mesma lógica de antes, com pequenas alterações na sintaxe:

```
A_22 = np.array([[1, 2], [3, 4]], dtype=int)

print('Matriz A =\n',A_22)
```

```
Matriz A =
 [[1 2]
 [3 4]]
```

O NumPy nos oferece algumas funções interessantes para criarmos matrizes específicas:

```
print('Matriz identidade 2x2 =\n',np.eye(2,dtype=int))
print('\nMatriz diagonal 3x3 =\n',np.diag([1,2,3]))
```

```
Matriz identidade 2x2 =
 [[1 0]
 [0 1]]

Matriz diagonal 3x3 =
 [[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Podemos criar a matriz **transposta** de uma dada matriz utilizando a função `np.transpose()` (ou apenas o método `.T`):

```
print('Matriz A =\n',A_22)
print('\nTransposta da matriz A =\n',np.transpose(A_22))
print('\nTransposta da matriz A =\n',A_22.T)
```

```
Matriz A =
 [[1 2]
 [3 4]]

Transposta da matriz A =
 [[1 3]
 [2 4]]

Transposta da matriz A =
 [[1 3]
 [2 4]]
```

Assim como em arrays unidimensionais, podemos acessar os elementos de uma matriz utilizando indexação, mas agora em 2 dimensões:

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

print('Matriz A =\n',A)
print('\nElemento 11 de A = ',A[0,0])
print('Elemento 23 de A = ',A[1,2])
print('Primeira linha de A = ',A[0,:])
print('Segunda coluna de A = ',A[:,1])
print('\nSubmatriz de A delimitada pelos elementos 22 e 33 =\n',A[1:,1:])
```

```
Matriz A =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Elemento 11 de A =  1
Elemento 23 de A =  6
Primeira linha de A =  [1 2 3]
Segunda coluna de A =  [2 5 8]

Submatriz de A delimitada pelos elementos 22 e 33 =
 [[5 6]
 [8 9]]
```

7.2.3 Propriedades de arrays

O NumPy fornece alguns métodos úteis que, apesar de não receberem nenhum argumento, nos permitem acessar algumas das características dos arrays que criamos.

- `ndim` retorna o número de dimensões do array;
- `shape` retorna o tamanho do array em cada uma de suas dimensões;
- `dtype` retorna o tipo de dado contido no array;
- `size` retorna o número total de elementos contidos no array.

```
X1 = np.array([[1,2,3], [4,5,6], [7,8,9]])
X2 = X1.flatten() # O método flatten reduz um array de n-dimensões em um array de uma única d

print('Array X1 =\n',X1)
print('\nDimensões de X1 = ', X1.ndim)
print('Shape de X1 = ', X1.shape)
print('Tipo de dado em X1 = ', X1.dtype)
print('Número de elementos em X1 = ', X1.size)

print('\n\nArray X2 =\n',X2)
print('\nDimensões de X2 = ', X2.ndim)
print('Shape de X2 = ', X2.shape)
print('Tipo de dado em X2 = ', X2.dtype)
print('Número de elementos em X2 = ', X2.size)
```

```
Array X1 =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Dimensões de X1 =  2
Shape de X1 =  (3, 3)
Tipo de dado em X1 =  int64
Número de elementos em X1 =  9


Array X2 =
 [1 2 3 4 5 6 7 8 9]

Dimensões de X2 =  1
Shape de X2 =  (9,)
Tipo de dado em X2 =  int64
Número de elementos em X2 =  9
```

Um método interessante de arrays é o `reshape(x,y)` que reorganiza um array existente de acordo com os argumentos `x` e `y`:

```
X = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])

print('Array X1 4x4 =\n',X)
print('\n X1 reorganizado em 2x8 =\n',X.reshape(2,8))
print('\n X1 reorganizado em 8x2 =\n',X.reshape(8,2))
```

```
Array X1 4x4 =
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

 X1 reorganizado em 2x8 =
 [[ 1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16]]

 X1 reorganizado em 8x2 =
 [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]
 [13 14]
 [15 16]]
```

7.3 Operações básicas com arrays

7.3.1 Operações aritméticas simples

Dois tipos de operações que serão úteis para arrays de qualquer dimensão são:

1. Operações entre um array e um único número.
2. Operações entre dois arrays da mesma forma.

Quando realizamos operações em um array usando um único número, simplesmente aplicamos essa operação a cada elemento do array. Isso vale tanto para arrays unidimensionais (vetores) quanto multidimensionais (por ex., matrizes).

```
x = np.array([1,2,3], dtype=int)
print("x =\n", x)
print("\n2 + x =\n", 2 + x)
print("\n2 - x =\n", 2 - x)
print("\n2 * x =\n", 2 * x)
print("\nx / 2 =\n", x / 2)
```

```
x =
 [1 2 3]

2 + x =
 [3 4 5]

2 - x =
 [ 1  0 -1]

2 * x =
 [2 4 6]

x / 2 =
 [0.5 1.  1.5]
```

```
X = np.ones((2, 2), dtype=int)

print("X =\n", X)
print("\n2 + X =\n", 2 + X)
print("\n2 - X =\n", 2 - X)
print("\n2 * X =\n", 2 * X)
print("\nX / 2 =\n", X / 2)
```

```
X =
 [[1 1]
 [1 1]]

2 + X =
 [[3 3]
 [3 3]]

2 - X =
 [[1 1]
 [1 1]]
```

```
2 * X =
 [[2 2]
 [2 2]]

X / 2 =
 [[0.5 0.5]
 [0.5 0.5]]
```

Para operações entre dois arrays de mesmo tamanho, basta aplicar a operação elemento a elemento (*elementwise*, em inglês) entre os arrays.

```
x = np.array([2, 4, 6], dtype=int)
y = np.array([2, 2, 1], dtype=int)

print("x =\n", x)
print("\ny =\n", y)
print("\nx + y =\n", x + y)
print("\nx - y =\n", x - y)
```

```
x =
 [2 4 6]

y =
 [2 2 1]

x + y =
 [4 6 7]

x - y =
 [0 2 5]
```

```
X = np.array([[2, 4], [6, 8]], dtype=int)
Y = np.array([[2, 2], [2, 2]], dtype=int)

print("X =\n", X)
print("\nY =\n", Y)
print("\nX + Y =\n", X + Y)
print("\nX - Y =\n", X - Y)
```

```
X =
```

```
 [[2 4]
 [6 8]]

Y =
 [[2 2]
 [2 2]]

X + Y =
 [[ 4  6]
 [ 8 10]]

X - Y =
 [[0 2]
 [4 6]]
```

As operações de multiplicação e divisão entre arrays são um pouco diferentes. São duas as possibilidades:

- Multiplicação e divisão elemento a elemento.
- Multiplicação entre arrays usando a lógica de matriz e "divisão" usando a lógica de matriz inversa.

Para realizar as operações *elementwise* basta utilizar os sinais usuais de `*` e `/`, independentemente do número de dimesões do array.

```
print('Vetores x e y:')
print('x =',x)
print('y =',y)
print("\n x * y =\n", x * y)
print("\n x / y =\n", x / y)

print('\nMatrizes X e Y:')
print('X =\n',X)
print('Y =\n',Y)
print("\n X * Y =\n", X * Y)
print("\n X / Y =\n", X / Y)
```

```
Vetores x e y:
x = [2 4 6]
y = [2 2 1]

 x * y =
```

```
 [4 8 6]

 x / y =
 [1. 2. 6.]

Matrizes X e Y:
X =
 [[2 4]
 [6 8]]
Y =
 [[2 2]
 [2 2]]

 X * Y =
 [[ 4  8]
 [12 16]]

 X / Y =
 [[1. 2.]
 [3. 4.]]
```

A **multiplicação** de matriz com matriz, do jeito que a gente conhece do Ensino Médio é feita usando o símbolo `@` (ou através da função `np.dot()`):

```
X= np.array([[1, 2], [3, 4]], dtype=int)
Y= np.array([[10, 20], [30, 40]], dtype=int)

print('X =\n',X)
print('Y =\n',Y)
print('\nX * Y =\n', X @ Y)
print('\nX * Y =\n', np.dot(X,Y))
```

```
X =
 [[1 2]
 [3 4]]
Y =
 [[10 20]
 [30 40]]

X * Y =
 [[ 70 100]
 [150 220]]
```

```
X * Y =
 [[ 70 100]
 [150 220]]
```

Para calcular a inversa de uma matriz utilizando o NumPy é preciso um pouco mais de estrutura e de conhecimento acerca do subpacote `linalg` que nos traz funções para realizar operações de álgebra linear. Falaremos disso já já!

7.3.2 Funções universais

As funções universais no Numpy são funções matemáticas simples. É apenas um termo que demos às funções matemáticas na biblioteca Numpy, que cobrem uma ampla variedade de operações. Essas funções incluem funções trigonométricas padrão, funções para operações aritméticas, manipulação de números complexos, funções estatísticas, etc. Essas funções possuem como características principais:

- Elas executam operações de array elemento por elemento.
- Elas suportam vários recursos, como conversão de tipos.
- As funções universais são objetos que pertencem à classe `numpy.ufunc`.
- As funções do Python também podem ser criadas como uma função universal usando a função da biblioteca `frompyfunc`.
- Algumas funções universais são chamadas automaticamente quando o operador aritmético correspondente é usado em arrays. Por exemplo, quando a adição de dois arrays é executada em elementos usando o operador '+', então `np.add()` é chamado internamente.

Dentre as principais funções universais matemáticas estão:

- `sin`, `cos`, `tan`: calcular seno, cosseno e tangente de ângulos.
- `hypot`: calcule a hipotenusa do triângulo retângulo dado.
- `arcsinh`, `arcosh`, `arctanh`: calcular seno hiperbólico inverso, cosseno e tangente.
- `deg2rad`: converter grau em radianos.
- `rad2deg`: converter radianos em graus.

Dentre as principais funções universais estatísticas estão:

- `amin`, `amax`: retorna o mínimo ou máximo de um array ou ao longo de um eixo.
- `ptp`: retorna o intervalo de valores (máximo-mínimo) de um array ou ao longo de um eixo.
- `sum`: retorna a soma de valores de um array ao longo de um eixo.
- `percentile(a, p, eixo)`: calcular o p-ésimo percentil da matriz ou ao longo do eixo especificado.
- `median`: calcular a mediana dos dados ao longo do eixo especificado.

- `mean`: calcular a média dos dados ao longo do eixo especificado.
- `var`: calcular a variância de dados ao longo do eixo especificado.
- `log`: calcular o log dos dados ao longo do eixo especificado.

Alguns exemplos:

```
angulos_notaveis_deg = np.array([30,45,60])
angulos_notaveis_rad = np.deg2rad(angulos_notaveis_deg)

seno_notaveis = [np.round(elem,2) for elem in np.sin(angulos_notaveis_rad)]
cosseno_notaveis = [np.round(elem,2) for elem in np.cos(angulos_notaveis_rad)]
tangente_notaveis = [np.round(elem,2) for elem in np.tan(angulos_notaveis_rad)]

print('Seno, cosseno e tangente de 30 graus: ',seno_notaveis[0],', ',cosseno_notaveis[0],' e
print('Seno, cosseno e tangente de 45 graus: ',seno_notaveis[1],', ',cosseno_notaveis[1],' e
print('Seno, cosseno e tangente de 60 graus: ',seno_notaveis[2],', ',cosseno_notaveis[2],' e
```

```
Seno, cosseno e tangente de 30 graus:  0.5 ,  0.87  e  0.58
Seno, cosseno e tangente de 45 graus:  0.71 ,  0.71  e  1.0
Seno, cosseno e tangente de 60 graus:  0.87 ,  0.5  e  1.73
```

```
x = np.array([1,2,3,4,5])

print('Array x = ',x)
print('\nMínimo de x = ',np.amin(x))
print('Máximo de x = ',np.amax(x))
print('Intervalo de x = ',np.ptp(x))
print('Soma de x = ',np.sum(x))
print('Média de x = ',np.mean(x))
print('Log de x = ',[np.round(elem,2) for elem in np.log(x)])
```

```
Array x =  [1 2 3 4 5]

Mínimo de x =  1
Máximo de x =  5
Intervalo de x =  4
Soma de x =  15
Média de x =  3.0
Log de x =  [np.float64(0.0), np.float64(0.69), np.float64(1.1), np.float64(1.39), np.float6
```

7.3.3 Arrays e listas

Agora que já vimos um pouco de operações básicas, podemos entender um pouco melhor a diferença de desempenho entre arrays e listas. Considere um array de dez milhões de números inteiros e uma lista equivalente:

```
my_array = np.arange(10000000)
my_list = list(range(10000000))
```

Agora vamos multiplicar, elemento a elemento, todos os números por 2 e salvar o resultado correspondente. Utilizaremos a função `time` do pacote `time`para fazer a medição do tempo utilizado pelos dois métodos.

```
import time

# arrays
start_array = time.time()
my_array2 = my_array * 2
end_array   = time.time()


# listas
start_lista = time.time()
my_list2 = [x * 2 for x in my_list]
end_lista   = time.time()

ratio = (end_lista - start_lista) / (end_array - start_array)
```

Qual abordagem será que levou menos tempo?

```
print('Tempo necessário para a realização dos cálculos utilizando arrays: {:.4f} segundos'.fo
print('Tempo necessário para a realização dos cálculos utilizando listas: {:.4f} segundos'.fo
print('\nA abordagem de listas demorou {:.0f}x mais tempo! Esqueça listas e use arrays ;)'.fo
```

```
Tempo necessário para a realização dos cálculos utilizando arrays: 0.0120 segundos
Tempo necessário para a realização dos cálculos utilizando listas: 0.2840 segundos

A abordagem de listas demorou 23x mais tempo! Esqueça listas e use arrays ;)
```

7.4 Aplicação: solução de sistema de equações lineares

Chega de exemplos vazios, vamos usar o NumPy para resolver um problema concreto e que nos é muito familiar: a solução de sistemas de equações lineares. Considere o sistema linear de equações dado por

$$A \cdot \vec{x} = \vec{b}$$

tal que A é a matriz de coeficientes, $\vec{x}$ é o vetor de incógnitas e $\vec{b}$ o vetor de constantes.

7.4.1 Algoritmo de eliminação de Gauss-Jordan

Segundo Hubbard e Hubbard (2015), uma matriz de coeficientes A é representada em sua forma escalonada reduzida por linhas (*reduced row echelon form*, em inglês) se

- Em toda e qualquer linha, a primeira entrada diferente de zero é igual a 1 (1 *pivotal*).
- O 1 *pivotal* de uma linha mais abaixo está sempre à direita de um outro 1 *pivotal* de alguma linha acima.
- Em toda e qualquer coluna que contém um 1 *pivotal*, todas as outras entradas são iguais a zero.
- Toda linha contendo apenas zeros está no fim da matriz.

À partir dessa definição, é possível mostrar que para qualquer matriz A, existe uma matriz $\tilde{A}$ na forma escalonada reduzida por linhas que pode ser obtida à partir de operações elementares nas linhas de A. Além disso, é possível mostrar também que $\tilde{A}$ é única. Ao algoritmo utilizado para encontrar $\tilde{A}$ é dado o nome de **Algoritmo de Eliminação de Gauss-Jordan**.

ALGORITMO DE ELIMINAÇÃO DE GAUSS-JORDAN

Para levar uma matriz A a sua forma escalonada reduzida por linhas $\tilde{A}$ devemos seguir os seguintes passos:

1. Encontre a primeira coluna que não é composta apenas de zeros, chame isso de primeira coluna pivotal e chame sua primeira entrada diferente de zero de pivô. Se o pivô não for na primeira linha, mova a linha que a contém para o topo da matriz.

2. Divida a primeira linha inteira pelo pivô, de modo que a primeira entrada da primeira coluna pivotal seja igual a 1.

3. Adicione múltiplos apropriados da primeira linha às outras linhas para garantir que todas as outras entradas da primeira coluna pivotal sejam iguais a 0. O 1 na primeira coluna é agora um pivô 1.

4. Escolha a próxima coluna que contém pelo menos uma entrada diferente de zero abaixo da primeira linha e coloque a linha que contém o novo pivô na posição da segunda linha. Faça do pivô um pivô 1: divida a linha inteira pelo pivô e adicione múltiplos apropriados desta linha às outras linhas abaixo, para tornar todas as outras entradas desta coluna iguais a 0.

5. Repita o processo até que a matriz esteja em sua forma escalonada reduzida por linhas.

Assuma o caso em que a matriz de coeficiente A e o vetor de constantes b são tais que a matriz ampliada é dada por:

$$\left[\begin{array}{ccc|c} 2 & 2 & 1 & 1 \\ 1 & 3 & 1 & 2 \\ 1 & 2 & 2 & -1 \end{array}\right]$$

- Passo 1: Defina a matriz M

```
M = np.array([(2, 2, 1, 1),
              (1, 3, 1, 2),
              (1, 2, 2, -1)],dtype=float)
print('Matriz ampliada =\n',M)
```

```
Matriz ampliada =
 [[ 2.  2.  1.  1.]
 [ 1.  3.  1.  2.]
 [ 1.  2.  2. -1.]]
```

- Passo 2: Divida a linha 1 pelo primeiro elemento da primeira linha

```
M[0,:] = M[0,:] / M[0,0]

print(M)
```

```
[[ 1.   1.   0.5  0.5]
 [ 1.   3.   1.   2. ]
 [ 1.   2.   2.  -1. ]]
```

- Passo 3: Subtraia a linha 1 da linha 2 e da linha 3

```
M[1,:] = M[1,:] - M[0,:]
M[2,:] = M[2,:] - M[0,:]

print(M)
```

```
[[ 1.   1.   0.5  0.5]
 [ 0.   2.   0.5  1.5]
 [ 0.   1.   1.5 -1.5]]
```

- Passo 4: Divida a linha 2 pelo segundo elemento da segunda linha

```
M[1,:] = M[1,:] / M[1,1]

print(M)
```

```
[[ 1.    1.    0.5   0.5 ]
 [ 0.    1.    0.25  0.75]
 [ 0.    1.    1.5  -1.5 ]]
```

- Passo 5: Subtraia a linha 2 da linha 3

```
M[2,:] = M[2,:] - M[1,:]

print(M)
```

```
[[ 1.    1.    0.5   0.5 ]
 [ 0.    1.    0.25  0.75]
 [ 0.    0.    1.25 -2.25]]
```

- Passo 6: Divida a linha 3 pelo terceiro elemento da terceira linha

```
M[2,:] = M[2,:] / M[2,2]

print(M)
```

```
[[ 1.    1.    0.5   0.5 ]
 [ 0.    1.    0.25  0.75]
 [ 0.    0.    1.   -1.8 ]]
```

- Passo 7: Multiplique a linha 3 pelo terceiro elemento da segunda linha e subtraia da linha 2

```
M[1,:] = M[1,:] - M[1,2] * M[2,:]

print(M)
```

```
[[ 1.   1.   0.5  0.5]
 [ 0.   1.   0.   1.2]
 [ 0.   0.   1.  -1.8]]
```

- Passo 8: Multiplique a linha 3 pelo terceiro elemento da primeira linha e subtraia da linha 1

```
M[0,:] = M[0,:] - M[0,2] * M[2,:]

print(M)
```

```
[[ 1.   1.   0.   1.4]
 [ 0.   1.   0.   1.2]
 [ 0.   0.   1.  -1.8]]
```

- Passo 9: Subtraia a linha 2 da linha 1

```
M[0,:] = M[0,:] - M[1,:]

print(M)
```

```
[[ 1.   0.   0.   0.2]
 [ 0.   1.   0.   1.2]
 [ 0.   0.   1.  -1.8]]
```

Temos a nossa matriz em sua forma escalonada reduzida por linhas! A solução do sistema é tal que:

```
print('A solução de x1 é: {:.2f}'.format(M[0,3]))
print('A solução de x2 é: {:.2f}'.format(M[1,3]))
print('A solução de x3 é: {:.2f}'.format(M[2,3]))
```

```
A solução de x1 é: 0.20
A solução de x2 é: 1.20
A solução de x3 é: -1.80
```

Uma forma mais simples de chegar na forma escalonada reduzida por linhas é através do pacote `SymPy` e das funções `Matrix` e `rref`:

```
import sympy

M_sympy = sympy.Matrix([(2, 2, 1, 1),
                        (1, 3, 1, 2),
                        (1, 2, 2, -1)])

M_sympy.rref()[0]
```

$$\begin{bmatrix} 1 & 0 & 0 & \frac{1}{5} \\ 0 & 1 & 0 & \frac{6}{5} \\ 0 & 0 & 1 & -\frac{9}{5} \end{bmatrix}$$

7.4.2 Solução de sistemas exatamente identificados

Uma outra forma de resolver sistemas de equações exatamente identificados, quando o número de incógnitas é igual ao número de equações (i.e., matriz A é quadrada) é através da matriz inversa de A. Para o caso em que A^{-1} existe, o sistema é tal que

$$\vec{x} = A^{-1} \cdot \vec{b}$$

Para que seja possível fazer dessa forma, A deve ser uma matriz quadrada e seu determinante ser diferente de 0. Mais uma vez assuma o caso em que A e $\vec{b}$ são dados por:

$$A = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 3 & 1 \\ 1 & 2 & 2 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

O primeiro passo é conferir se a matrix A é quadrada. Podemos fazer isso usando a função `numpy.shape()`

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

np.shape(A)[0] == np.shape(A)[1]
```

```
True
```

O próximo passo é ver se o determinante da matriz é igual a 0 e para isso usamos um função do subpacote de álgebra linera do numpy, `numpy.linalg.det()`:

```
print('A =')
print(A)

print('\nDeterminante = ',np.round(np.linalg.det(A)))
```

```
A =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Determinante =  5.0
```

Como ambas as condições são satisfeitas, por fim basta calcular a inversa da matriz A com `numpy.linalg.inv()` e multiplicar por $\vec{b}$ para chegar nos valores de `x`, `y` e `z` que solucionam o sistema de equações.

```
print('A =')
print(A)

print("\nInversa de A = ")
print(np.linalg.inv(A))
```

```
A =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Inversa de A =
[[ 0.8 -0.4 -0.2]
 [-0.2  0.6 -0.2]
 [-0.2 -0.4  0.8]]
```

```
solution = np.linalg.inv(A) @ b
solution
```

```
array([ 0.2,  1.2, -1.8])
```

```
print('A solução de x1 é: {:.2f}'.format(solution[0]))
print('A solução de x2 é: {:.2f}'.format(solution[1]))
print('A solução de x3 é: {:.2f}'.format(solution[2]))
```

```
A solução de x1 é: 0.20
A solução de x2 é: 1.20
A solução de x3 é: -1.80
```

Uma última alternativa para resolver o sistema $A \cdot \vec{x} = \vec{b}$, é utilizar a função `solve` do subpacote de álgebra linear do `NumPy`. Essa função faz o processo de resolução do sistema de forma direta e nos cospe o vetor de resultado.

```
np.linalg.solve(A,b)
```

```
array([ 0.2,  1.2, -1.8])
```

8 Gestão e análise de dados

Uma das principais aplicações para o Python bla bla bla

8.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental

9 Visualização

Uma das principais aplicações para o Python bla bla bla

9.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental

10 Análise empírica integrada

Uma das principais aplicações para o Python bla bla bla

10.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental

Parte III

Temas complementares

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

11 Introdução à programação orientada a objetos (OOP)

In summary, this book has no content whatsoever.

12 Introdução ao R

In summary, this book has no content whatsoever.

References