Big Data Básico

Marcos Roberto Ribeiro

Formação Inicial e Continuada

INSTITUTO FEDERAL
Minas Gerais

+ IFMG
Campus Bambuí

Marcos Roberto Ribeiro

Big Data Básico

1ª Edição

Belo Horizonte

Instituto Federal de Minas Gerais

2022

Pró-reitor de Extensão	Carlos Bernardes Rosa Júnior
Diretor de Programas de Extensão	Niltom Vieira Junior
Coordenação do curso	Marcos Roberto Ribeiro
Arte gráfica	Ângela Bacon
Diagramação	Eduardo dos Santos Oliveira

FICHA CATALOGRÁFICA
Dados Internacionais de Catalogação na Publicação (CIP)

R484p Ribeiro, Marcos Roberto.
Big data básico [recurso eletrônico] / Marcos Roberto Ribeiro. – Belo Horizonte : Instituto Federal de Minas Gerais, 2022.
171p.: il. color.

E-book, no formato PDF.
Material didático para Formação Inicial e Continuada.
ISBN 978-65-5876-030-6

1. Big data. 2. NoSQL. 3. Banco de dados chave-valor. 4. Banco de dados de documentos 5. Bancos de dados de grafos. I. Título.

CDD 005.13

Catalogação: Douglas Bernardes de Castro - CRB-6/2802

2022
Direitos exclusivos cedidos ao
Instituto Federal de Minas Gerais
Avenida Mário Werneck, 2590,
CEP: 30575-180, Buritis, Belo Horizonte – MG,
Telefone: (31) 2513-5157

Sobre o material

Este curso é autoexplicativo e não possui tutoria. O material didático, incluindo suas videoaulas, foi projetado para que você consiga evoluir de forma autônoma e suficiente.

Caso opte por imprimir este *e-book*, você não perderá a possibilidade de acessar os materiais multimídia e complementares. Os *links* podem ser acessados usando o seu celular, por meio do glossário de Códigos QR disponível no fim deste livro.

Embora o material passe por revisão, somos gratos em receber suas sugestões para possíveis correções (erros ortográficos, conceituais, *links* inativos etc.). A sua participação é muito importante para a nossa constante melhoria. Acesse, a qualquer momento, o Formulário “Sugestões para Correção do Material Didático” clicando nesse *link* ou acessando o QR *Code* a seguir:

Formulário de Sugestões

Para saber mais sobre a Plataforma +IFMG acesse

http://mais.ifmg.edu.br

Palavra do autor

Uma das tecnologias mais importantes da Computação são os bancos de dados presentes em sistemas computacionais utilizados em inúmeras atividades. Mesmo com os grandes avanços já alcançados, a Computação continua em franca evolução, principalmente na área de Banco de Dados. A partir da década de 2000, com a popularização da Internet, houve um crescente aumento na produção de dados pela humanidade. Essa quantidade massiva de dados é chamada de Big Data. Para lidar como Big Data, em diversas situações, é interessante utilizar tecnologias NoSQL. Tais tecnologias passaram a ser largamente usadas no início do século XXI, principalmente, pela sua utilização em grandes sistemas Web como redes sociais. Desde então, a tecnologia NoSQL conta com diversas aplicações em uma grande variedade de sistemas. O presente curso, na forma de uma introdução a Big Data, é muito interessante para despertar e iniciar a formação de novos profissionais nessa área.

Bons estudos!

Marcos Roberto Ribeiro

Apresentação do curso

Este curso está dividido em quatro semanas, cujos objetivos de cada uma são apresentados, sucintamente, a seguir.

SEMANA 1	- Conhecer os conceitos de NoSQL e Big Data - Conhecer os básico da linguagem de programação Python; - Desenvolver códigos básicos da linguagem Python.
SEMANA 2	- Conhecer os conceitos básicos de bancos chave-valor; - Desenvolver e entender códigos usando bancos chave-valor.
SEMANA 3	- Conhecer os conceitos básicos de bancos de documentos; - Desenvolver e entender códigos usando bancos de documentos.
SEMANA 4	- Conhecer os conceitos básicos de bancos de grafos; - Desenvolver e entender códigos usando bancos de grafos.

Carga horária: 40 horas.

Estudo proposto: 2h por dia em cinco dias por semana (10 horas semanais).

Apresentação dos Ícones

Os ícones são elementos gráficos para facilitar os estudos, fique atento quando eles aparecem no texto. Veja aqui o seu significado:

Atenção: indica pontos de maior importância no texto.

Dica do professor: novas informações ou curiosidades relacionadas ao tema em estudo.

Atividade: sugestão de tarefas e atividades para o desenvolvimento da aprendizagem.

Mídia digital: sugestão de recursos audiovisuais para enriquecer a aprendizagem.

Sumário

Objetivos
- Conhecer os conceitos de NoSQL e *Big Data*
- Conhecer os básico da linguagem de programação Python;
- Desenvolver códigos básicos da linguagem Python.

Mídia digital: Antes de iniciar os estudos, vá até a sala virtual e assista ao vídeo “Apresentação do curso”.

1.1 Introdução

A partir do início do século XXI, a Computação e suas tecnologias causaram uma verdadeira revolução em diversas áreas da sociedade (MARÇULA; FILHO, 2008). Isso proporcionou grandes melhorias e mudanças nas formas de comunicação, no ambiente de trabalho, na qualidade de vida e no funcionamento de empresas e órgãos governamentais (VELLOSO, 2014). Quando falamos de computação, basicamente, estamos nos referindo a processamento e dados. O processamento são todas as operações que o computador pode realizar e os dados são todas as informações que o computador pode armazenar.

Em um computador, os dados podem ser guardados em armazenamento primário ou secundário. O armazenamento primário é a chamada memória RAM, sigla do inglês de *random access memory*, e outras memórias como o cache do processador. Já o armazenamento secundário são mídias como discos rígidos, DVDs, pendrives, etc. O armazenamento primário é mais rápido, mas, por se tratar de um armazenamento volátil, os dados permanecem guardados somente enquanto o computador estiver ligado. Por outro lado, o armazenamento secundário não é tão rápido, mas os dados ficam guardados quando o computador é desligado.

O gerenciamento do armazenamento primário é controlado pelo sistema operacional ou por programas específicos. Os dados do armazenamento secundário ficam guardados em arquivos. Dependendo do nível de complexidade de um sistema computacional o gerenciamento dos dados diretamente via arquivos pode se tornar muito complexo. Assim, em muitas situações, os programas de computador utilizam sistemas de bancos de dados para cuidar do armazenamento dos dados.

Os bancos de dados são uma das tecnologias mais importantes da Computação e estão presentes em sistemas computacionais utilizados em inúmeras áreas (RAMAKRISHNAN; GEHRKE, 2008; ELMASRI; NAVATHE, 2011). Mesmo com os grandes avanços já alcançados, a Computação continua em franca evolução, principalmente na área de Banco de Dados.

A partir da década de 2000, com a popularização da Internet, houve um crescente aumento na produção de dados pela humanidade. Essa quantidade massiva de dados é chamada de Big Data (WIKIPÉDIA, 2022a). Alguns exemplos de sistemas que envolvem Big Data são aplicações de bolsas de valores, redes sociais e científicas. Algumas dessas aplicações podem gerar terabytes de dados em um dia ou mesmo em alguns minutos (CURRY, et al. , 2022).

As principais características do Big Data são volume, variedade e velocidade. O volume diz respeito a grande quantidade de dados produzida, na ordem de terabytes diários ou maior (SOTO; LUNA; CANO, 2016).

Quanto à variedade, as fontes de dados são muito variadas como transações comerciais, redes sociais, sensores, celulares, redes sociais, etc. Além da variedade de fontes de dados, existem diferentes formatos e tipos de dados, desde dados estruturados, dados numéricos, até dados não estruturados, como textos, imagens, vídeos e áudios. Um dos maiores desafios do Big Data é gerenciar todos esses diferentes tipos de dados e fontes.

A velocidade se refere ao tempo de processamento de dados de Big Data. Devido ao grande volume e variedade de dados, todo o processamento deve ser ágil para se obter as informações necessárias.

Para lidar como Big Data, em diversas situações, é interessante utilizar tecnologias NoSQL. Tais tecnologias passaram a ser largamente utilizadas no início do século XXI, principalmente, pela sua aplicação em grandes sistemas Web como redes sociais. Desde então, a tecnologia NoSQL conta com diversas aplicações em uma grande variedade de sistemas (WIKIPÉDIA, 2022b).

1.2 Sistemas de banco de dados

Um banco de dados é uma coleção estruturada de dados usada para descrever as informações de entidades e seus relacionamentos (RAMAKRISHNAN; GEHRKE, 2008). Em muitas situações, os bancos de dados são mantidos por um sistema de gerenciamento de banco de dados (SGBD). O SGBD possibilita a criação e manutenção de um banco de dados de forma facilitada. A utilização de um SGBD implica nas seguintes vantagens:

- **Independência de dados**. Os programas aplicativos usados pelos usuários do computador não devem ser expostos aos detalhes de representação e armazenamento de dados. O ideal é que os programas se comuniquem de forma mais simples com o SGBD que, por sua vez, trata dos detalhes de armazenamento dos dados;
- **Acesso eficiente**. O SGBD utiliza várias técnicas sofisticadas para armazenar e recuperar dados eficientemente;
- **Integridade e segurança**. O SGBD pode forçar restrições de integridade, por exemplo, impedir que o mesmo cliente seja cadastrado duas vezes com o mesmo CPF. O SGBD permite também liberar ou restringir o acesso a certos dados para determinados usuários;

- **Administração de dados**. Com os dados centralizados, profissionais de banco de dados podem fazer ajustes finos para obtenção de melhorias de performance;
- **Acesso concorrente e recuperação de falhas**. O SGBD gerencia o acesso concorrente aos dados criando a ilusão de que os dados são acessado por um usuário de cada vez. Além disto, o SGBD protege os dados contra falhas do sistema;
- **Tempo reduzido de desenvolvimento de aplicativo**. Como os desenvolvedores não precisam se preocupar com detalhes de armazenamento e recuperação dos dados, o tempo que seria usado para esta tarefa pode ser gasto em outros aspectos relativos ao desenvolvimento.

Devido a suas vantagens os SGBD são largamente utilizados em muitos aplicativos comerciais, como sistemas de gestão empresarial e aplicações web. Contudo, há cenários em que a utilização do SGBD pode não ser viável como sistemas de tempo real, aplicativos para dispositivos móveis, redes sociais, etc. Dessa forma, é importante conhecer e utilizar de forma adequada as arquiteturas de banco de dados existentes. As arquiteturas de banco de dados existentes são as seguintes:

- Plataformas centralizadas;
- Pessoal;
- Cliente-servidor;
- Distribuída.

Nas plataformas centralizadas, tanto o computador quanto os aplicativos ficam hospedados em um único computador. Geralmente, para essa arquitetura, são utilizados computadores com grande capacidade de processamento, os chamados *mainframes*, possibilitando que muitos usuários manipulem grandes volumes de informações. Contudo, a aquisição desse tipo de computador envolve um alto custo financeiro.

A arquitetura pessoal de banco de dados, também chamada de *standalone*, é usada para aplicações de único usuário de forma que a aplicação e o banco de dados são hospedados no mesmo computador. A vantagem da arquitetura está em sua simplicidade.

Na arquitetura cliente servidor, há um servidor com SGBD que recebe conexões de computadores clientes. Essa é uma arquitetura intermediária entre a *standalone* e as plataformas centralizadas. Assim, tanto o servidor quando os computadores clientes devem ter certa capacidade de processamento.

A arquitetura distribuída é composta por diversos servidores onde grandes bancos de dados podem ser hospedados. Dessa maneira, essa arquitetura possui alta capacidade de processamento e de expansão. No entanto, sua utilização exige alta complexidade de implementação ou contratação do serviço em nuvem.

Além da arquitetura dos sistemas de bancos de dados, é importante considerar o modelo de dados a ser usado. Atualmente, o modelo de banco de dados relacional é o modelo dominante na maioria dos sistemas de banco de dados, independente da arquitetura adotada. Um banco de dados relacional é composto de tabelas (também

chamadas de relações). Uma tabela é um conjunto não ordenado de linhas (ou tuplas). Cada linha é composta por uma série de campos (ou valores de atributo). Esse conjunto de campos também é chamado de esquema. Cada campo é identificado por nome de campo (ou nome de atributo). O conjunto de campos das linhas de uma tabela que possuem o mesmo nome formam uma coluna. A Figura 1 mostra um exemplo de tabela de banco de dados para armazenar informações de funcionários.

id	nome	id_departamento	categoria
1	Souza	D1	C5
2	Santos	D2	C5
3	Silva	D1	
5	Soares	D1	

Figura 1 – Tabela **funcionarios** de um banco de dados
Fonte: Elaborado pelo Autor.

O conceito fundamental para estabelecer relações entre linhas de tabelas em um banco de dados relacional é a chave. Basicamente, existem os seguintes tipos de chaves:

- Chaves primárias;
- Chaves alternativas;
- Chaves estrangeiras.

Uma chave é uma coluna ou uma combinação de colunas cujos valores distinguem uma linha das demais dentro de uma tabela. A chave primária e a chave principal de uma tabela. No caso da tabela **funcionarios**, a chave primária seria a coluna **id**.

Uma chave estrangeira é uma coluna ou uma combinação de colunas, cujos valores aparecem necessariamente na chave primária de uma tabela. A chave estrangeira é o mecanismo que permite a implementação de relacionamentos em um banco de dados relacional. Supondo que tenhamos uma tabela **departamentos**, a coluna **id_departamento** da tabela **funcionarios** poderia ser uma chave estrangeira apontando para a chave primária da tabela **departamentos**.

A existência de uma chave estrangeira impõe restrições que devem ser garantidas em diversas situações de alteração do banco de dados. Quando uma linha é incluída ou alterada na tabela que contém a chave estrangeira, deve ser garantido que o valor da chave estrangeira apareça na coluna da chave primária referenciada. Em nosso exemplo, isso significa que um novo funcionário deve atuar em um departamento já existente no banco de dados.

Quando uma linha é excluída na tabela referenciada, deve ser garantido que na com a chave estrangeira não apareça o valor da chave primária que está sendo excluído. Em nosso exemplo, isso significa que um departamento não pode ser excluído, caso existam funcionário no mesmo.

A palavra chave estrangeira pode ser enganosa, levando a acreditar que a chave estrangeira sempre referencia uma chave primária de outra tabela. Entretanto, uma chave estrangeira pode referenciar a chave primária da própria tabela. Como exemplo, considere

a tabela **funcionarios** da Figura 2. Nessa tabela, fui incluído o campo **id_supervisor** na forma de uma chave estrangeira que aponta para o supervisor de cada funcionário.

id	nome	id_departamento	categoria	id_surpervisor
1	Souza	D1	C5	
2	Santos	D2	C5	1
3	Silva	D1		1
5	Soares	D1		2

Figura 2 – Tabela **funcionarios** com chave estrangeira em **id_supervisor**
Fonte: Elaborado pelo Autor.

Em alguns casos, uma tabela pode possuir mais de uma chave. Nesse caso, uma das chaves é escolhida como chave primária e as demais podem ser usadas como chaves alternativas. Assim, podemos definir como chave alterativa, as chaves de uma tabela diferentes da chave primária. Há duas diferenças básicas entre chaves primárias e alternativas. As chaves primárias podem ser usadas para a criação de chaves estrangeiras, enquanto as chaves alternativas permitem valores nulos. Como exemplo, a Figura 3 mostra a tabela **funcionarios** com o campo **cpf** sendo usado como chave alternativa.

id	nome	id_departamento	categoria	cpf
1	Souza	D1	C5	
2	Santos	D2	C5	222.222.222-22
3	Silva	D1		333.333.333-33
5	Soares	D1		555.555.555-55

Figura 3 – Tabela **funcionarios** com chave alternativa em **cpf**
Fonte: Elaborado pelo Autor.

Quando uma tabela é definida, deve ser especificado o conjunto de possíveis valores para cada uma de suas colunas. Esse conjunto de valores é chamado de domínio ou tipo. Além disso, deve ser especificado se os campos da coluna podem estar vazios ou ser nulos. Estar vazio indica que o campo não recebeu nenhum valor de seu domínio. As colunas que não permite valores vazios são chamadas de colunas obrigatórias. Já as colunas que aceitam valores nulos são chamadas de colunas opcionais. As colunas que compõem a chave primária devem ser obrigatórias.

Um dos objetivos primordiais de um sistema de banco de dados é manter a integridade de dados. Dizer que os dados estão íntegros significa dizer que eles refletem corretamente a realidade representada pelo banco de dados e que são consistentes entre si. Uma restrição de integridade é uma regra de consistência de dados que é garantida pelo sistema de banco de dados. No modelo relacional, são consideras as seguintes restrições de integridade:

- **Integridade de domínio**. A integridade de domínio especifica que o valor de um campo deve obedecer a definição de valores admitidos para a coluna (o domínio da coluna);

- **Integridade de vazio**. A integridade de vazio especifica se os campos de uma coluna podem ou não ser vazios (se a coluna é obrigatória ou opcional);
- **Integridade de chave**. A integridade de chave trata da restrição que define que os valores da chave primária e das chaves alternativas devem ser únicos;
- **Integridade referencial**. A integridade referencial define que os valores dos campos que aparecem em uma chave estrangeira devem aparecer na chave primária da tabela referenciada.

Os SGBDs relacionais, além de adotar o modelo relacionais e suas características, também fazem o gerenciamento de transações para garantir as propriedades de atomicidade, consistência, isolamento e durabilidade (ACID) sobre as operações realizadas sobre os dados. A atomicidade implica na execução de todas ou nenhuma das operações de uma transação. A consistência implica que o banco de dados permanece consistente antes e depois de uma transação. Suponha, por exemplo, a transferência de um valor de uma conta correte A para uma conta corrente B. A atomicidade e a consistência garantem que, se a transação for completada, as operações de retirada da conta A e entrada na conta B foram efetivadas. Caso ocorra algum problema, tanto a retirada quanto a entrada são canceladas.

O isolamento garante que cada usuário do banco de dados tenha a impressão de que apenas suas operações estão sendo feita sobre os dados, mesmo que existem diversas pessoas manipulado os dados simultaneamente. A durabilidade trata da persistência dos dados em armazenamento secundário. Assim, o SGBD garante que, mesmo após um desligamento acidental do computador, será possível recuperar as operações das transações finalizadas.

1.3 Tecnologias NoSQL

A partir da década de 2000, houve um crescente aumento da produção de dados pela humanidade impulsionado principalmente pela grande popularização da Internet. Estima-se que no ano de 2021 a produção diária de dados no mundo era de 2.5 exabytes. Isso equivale cinco bilhões de DVDs em um único dia (FULMAŃSKI, 2021). Alguns exemplos de aplicações são:

- redes sociais com milhões ou até bilhões de usuários;
- experimentos científicos como simulações físicas;
- sensores como GPS e dados meteorológicos;
- aplicações de astronomia.

O termo *lago de dados* foi criado para se referir aos armazenamento desses grandes volumes de dados, funcionando como um lago com várias coleções de dados que nem sempre são facilmente acessíveis ou divisíveis.

Além do grande volume de dados, as novas aplicações também precisam lidar com variação nos dados e alta velocidade. O termo *Big Data* se refere a esse grande volume de dados com características de variação e velocidade. Os sistemas de banco de dados

relacionais foram, originalmente, projetados para funcionar em um único computador. Assim, para lidar com o *Big Data* e questões de escalabilidade, seriam necessárias modificações consideráveis na arquitetura dos sistemas existentes.

1.3.1 Escalabilidade

A escalabilidade pode ser vertical com um computador mais rápido ou na horizontal dividindo o sistema em vários computadores. A escalabilidade vertical de um SGBD relacional envolve um alto custo envolvendo um computador mais rápido e, dado o volume do *Big Data*, pode não resolver o problema. Por outro lado, a implementação da escalabilidade horizontal não é simples e envolve problemas de performance pelo fato dos SGBDs terem sido projetados para o processamento central. Assim, as tecnologias NoSQL, devido a sua alta capacidade de escalabilidade, começaram a ser adotadas para lidar com o *Big Data*.

Apesar da grande aplicabilidade dos SGBDs relacionais, sua utilização para lidar com o Big Data aplica em algumas desvantagens. São elas:

- A garantia das propriedades ACID pode bloquear o sistema por um curto período de tempo prejudicado certos tipos de aplicações;
- Nem sempre é possível ter boa escalabilidade;
- Não é muito simples implementar alta variabilidade de dados em tabelas;

Uma das características que pode ser tornar negativa na abordagem do modelo relacional é sua natureza forçada pelas formas normais. O principal objetivo das formas normais é eliminar a redundância de dados. Para isso, os dados precisam ser divididos em diversas tabelas para armazenamento. Por outro lado, a leitura desses dados, frequentemente, envolve a *junção* duas ou mais tabelas. A junção é uma operação que combina linhas de tabelas de acordo com alguma condição.

Em sistemas reais a operação de junção pode ser requisitada por diversos usuários simultaneamente, enquanto vários outros usuários podem estar fazendo outras operações. Tudo isso requer acesso a disco, memória e processamento. Tentando evitar toda essa carga de trabalho as tecnologias NoSQL, em geral, armazenam os dados de forma diferente, fazendo com que seja mais fácil implementar a distributividade e escalabilidade.

1.3.2 Motivações para o NoSQL

O NoSQL não significa o abandono da linguagem SQL. Na verdade, muitas tecnologias NoSQL oferecem suporte à linguagem SQL. AS tecnologias NoSQL tratam do processamento de grandes conjuntos de dados focando em desempenho, distributividade e escalabilidade. As principais motivações consideradas no desenvolvimento das tecnologias NoSQL foram flexibilidade, disponibilidade, desempenho, custo e escalabilidade.

Uma desvantagem do modelo relacional é a necessidade de sabermos muitas, senão todas, informações antes de criarmos um banco de dados para trabalharmos.

Assim, precisamos saber quais tabelas e colunas devem existir para dar suporte a um aplicativo. Na prática, nem sempre é possível saber de antemão quais informações precisaremos. E, as vezes, incluímos uma coluna que raramente é usada. Além disso, em muitas situações, as informações podem mudar como tempo. Para lidar com esses casos, as tecnologias NoSQL oferecem a *flexibilidade* de coletar, armazenar e processar qualquer tipo de dados a qualquer momento.

A *disponibilidade* das tecnologias NoSQL, normalmente, está relacionada com o armazenamento de um mesmo dado em mais de um computador. Assim, se um computador ficar indisponível, o dado continua acessível em outro local, evitando assim que serviços da Internet fiquem "fora do ar".

Quando fazemos uma busca na Internet, em geral, recebemos milhares de resultados. Se tentarmos acessar uma página que está demorando a ser carregada, provavelmente, passaremos para o próximo resultado porque não queremos esperar nem alguns segundos. As tecnologias NoSQL buscam prover alta *velocidade* para que serviços não demorem para ser carregados.

O funcionamento de serviços na Internet dependem de um computador *servidor* ligado o tempo todo para receber a conexão dos usuários que consomem o serviço. Dessa forma, as empresas podem comprar servidores com custo elevando pensando em uma certa carga de trabalho máxima que deverá ser suportada. O problema é que esse computador pode ter um *custo* alto e ser pouco utilizado nos início da disponibilização do serviço. Além disso, em alguns anos, o computador pode não suportar a carga de trabalho, levando a mais gastos com infraestrutura. As tecnologias NoSQL possibilitam um investimento inicial de menor custo em infraestrutura. Graças, a *escalabilidade*, no futuro, é possível adicionar novos computadores de forma simples para aumentar o poder de processamento.

1.3.3 Tipos de tecnologias

Dependendo da característica desejada, um ou outro tipo de tecnologia SQL pode ser mais interessante. Os principais tipos de tecnologias NoSQL são os bancos de dados chave-valor, de documentos e de grafos.

Os banco de dados chave-valor utilizam uma estrutura de dados de matriz associativa, também chamada de dicionário ou tabela de *hash*. O termo dicionário descreve melhor as ideias por trás desse tipo de banco de dados. Em um dicionário de papel, vocês busca por uma palavra que contém uma definição. De modo geral, os banco de dados chave-valor são uma coleção de objetos ou registros, que por sua vez, são compostos por campos de dados. Tais registros são armazenados e recuperados usando uma chave que o identifica exclusivamente.

Um documento é um objeto semiestruturado, ou seja, um objeto que possui alguma estrutura não padronizada composta por uma coleção de diferentes pares de valores-chave, onde cada valor pode ser um objeto diferente. Os bancos de dados de documentos são uma evolução do banco de dados chave-valor. No armazenamento de valor-chave, não é possível pesquisar no conteúdo dos objetos. A única forma de pesquisa são as

chaves. Já nos bancos de dados de documentos, é possível buscar por objetos considerando seu conteúdo.

Os bancos de dados de grafo possuem fundamentos teóricos baseados na matemática dos grafos. Basicamente, um grafo é uma estrutura composta por nós e arestas que podem representar diversos objetos do mundo real. Alguns exemplos de grafos são:

- Mapa com cidades (nós) e estradas (arestas) ligando as mesmas;
- Rede sociais com usuários (nós) com relação de amizade (arestas) com outros usuários;
- Páginas da Internet (nós) com ligações ou *links* (arestas) para outras páginas.

Os nós e as arestas podem ter propriedades específicas como o número de habitantes de uma cidade ou a distância de uma cidade a outra. A principal utilidade desse tipo de banco de dados está nas consultas usando as características do grafo. Por exemplo, podemos buscar por as pessoas (nós) em uma rede social (grafo) que são amigas (aresta) de alguém (outro no) que mora no Brasil (propriedade).

1.4 O sistema operacional Linux

O sistema operacional Linux é um sistema operacional livre criado na década de 1990 por Linus Torvalds. Na verdade, o Linux é o chamado *kernel* do sistema operacional, ou seja, o programa principal do sistema operacional que se comunica com o hardware e gerencia todos os demais programas. Quando falamos de um sistema operacional mais completo com diversos programas utilitários, inclusive interface gráfica, nos referimos às distribuições Linux.

Atualmente existem centenas de distribuições Linux ativas. Muitas delas são de propósito geral e indicadas para instalação em computadores pessoais. Existem também muitas distribuições mais específicas e indicadas para as mais variadas finalidades como manutenção de computadores, teste de redes, recreação etc. Com essa grande variedade de distribuições o Linux vem ganhando cada vez mais espaço. Além disso, no que diz respeito à computadores servidores, o sistema operacional Linux é o mais usado atualmente, principalmente devido à sua segurança e estabilidade.

Algumas ferramentas NoSQL só funcionam no sistema operacional Linux. Assim, em muitos exemplos consideraremos o uso desse sistema operacional. Recomendamos a utilização de distribuições baseadas no Ubuntu 20.04. Hoje em dia, é relativamente simples baixar e instalar o sistema operacional. Também é possível instalar em máquinas virtuais ou ativar o *Windows Subsystem for Linux* (WSL) do sistema operacional Windows.

No caso da instalação convencional, o sistema Ubuntu 20.04 pode ser encontrado no site oficial (https://releases.ubuntu.com/20.04.5/ubuntu-20.04.5-desktop-amd64.iso). Após o download do arquivo ISO, podemos usar o aplicativo Balena Etcher para gerar uma mídia, como pendrive, para iniciar a instalação (https://www.balena.io/etcher/). Nesse caso, a mídia deve ser inserida no computador e a opção de *boot* (inicialização) por essa mídia

deve ser selecionada. A principal configuração durante a instalação é o tipo de instalação que pode ser uma das seguintes opções:

- Instalar Ubuntu ao Lado do Windows;
- Apagar o disco e instalar Ubuntu;
- Opção avançada.

Se você deseja ter apenas o Ubuntu no computador, você pode escolher a opção "Apagar o disco e instalar Ubuntu". Para manter o Windows e o Linux, a chamada instalação *Dual Boot*, você deve escolher a opção "Instalar Ubuntu ao Lado do Windows". Após a instalação, você poderá escolher qual sistema deseja usar, ao ligar o computador. A "Opção Avançada" é recomendada apenas para usuários mais experientes. O processo de instalação convencional também pode ser feito em uma máquina virtual como no Virtual Box (https://www.virtualbox.org/).

Tanto a utilização de máquinas virtuais quanto o WSL requerem que a opção de virtualização esteja ativada na BIOS do computador. Geralmente, você encontra essa opção com nomes como *Intel Virtualization Technology*, *AMD-V*, *Hyper-V*, *VT-X*, *Vanderpool* or *SVM*.

Para usuários iniciantes, a instalação WSL é a mais recomendada. Nela, você instala o Linux dentro do Windows. Esse método só está disponível a partir do Windows 10. Para iniciar a instalação, você deve abrir o utilitário de linha de comando (PowerShell ou Prompt de Comandos) clicando com o botão direito do mouse e escolher a opção de "Executar como administrador". Com o utilitário aberto, digitar o comando **wsl --install**. A Figura 4 mostra o resultado do processo de instalação. Ao final, será necessário reiniciar o computador.

Figura 4 – Processo de instalação do WSL
Fonte: Elaborado pelo Autor.

Após a reinicialização, você encontrará o Ubuntu nos programas do menu Iniciar. Na primeira execução, você deverá criar um usuário e senha como mostrado na Figura 5. É recomendável que, na primeira execução e sempre for instalar algum pacote, seja executado o comando **sudo apt update** seguido do comando **sudo apt dist-upgrade**. Esses comandos atualizam todos os pacotes instalados no sistema. Normalmente, os comandos precedidos de **sudo** solicitam a senha do usuário que você criou. No caso do **sudo apt dist-upgrade**, o sistema também confirma se você deseja proceder com a instalação.

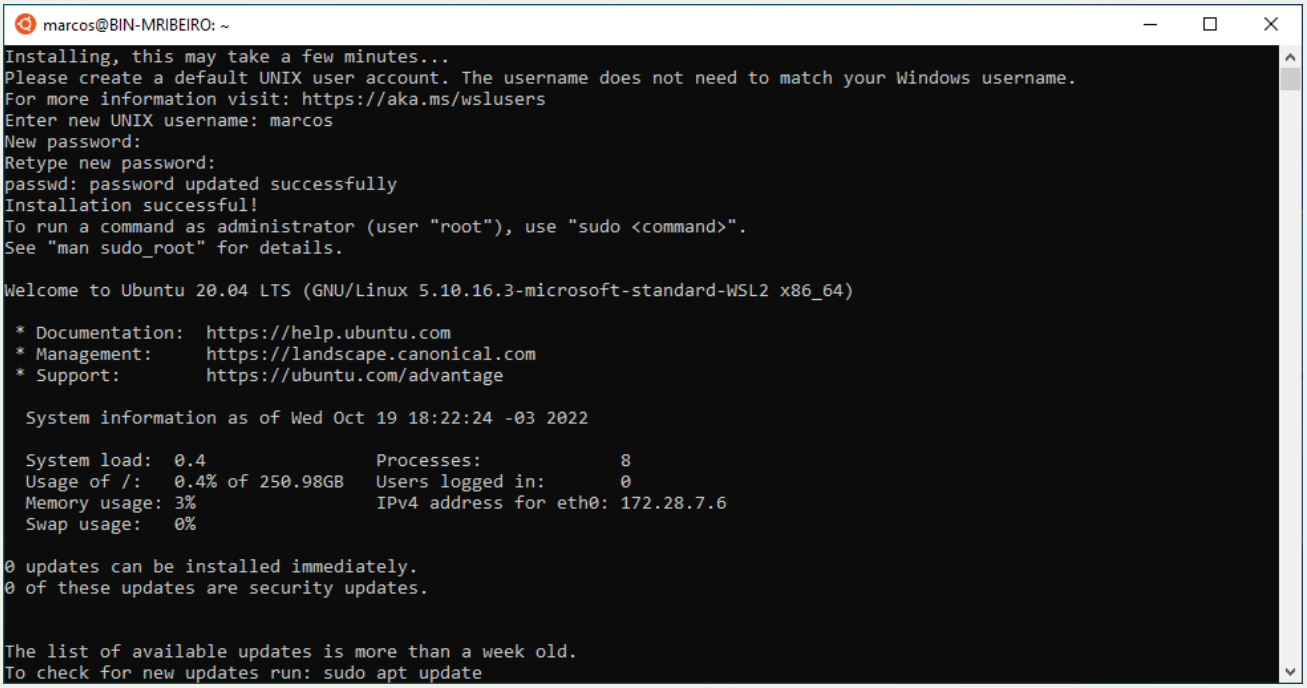

Figura 5 – Primeira execução do Ubuntu no WSL
Fonte: Elaborado pelo Autor.

1.5 A linguagem de programação Python

As linguagens de programação são ferramentas essenciais para o desenvolvimento de softwares, bem como a manipulação de dados de banco de dados NoSQL. Uma das linguagens de programação mais usadas no mundo é a linguagem Python[1] (TIOBE, 2021; PYPL, 2021). Além disso, desde sua criação no final da década de 1980, a linguagem passou por uma constante evolução até chegar à versão 3 em 2008 (WIKIPÉDIA, 2021b). A linguagem Python é uma linguagem aberta, multiplataforma e utilizada em uma grande variedade de aplicações (MENEZES, 2019; RAMALHO, 2015).

Na maioria das distribuições do sistema operacional Linux, o ambiente Python já vem instalado por padrão. Para outros sistemas operacionais, como Windows e MacOS, é preciso baixar e instalar esse ambiente. Contudo, no presente curso, utilizaremos a ferramenta Spyder[2] que já conta com o ambiente Python. No Linux, a instalação da ferramenta está disponível nos sistemas de pacotes (TAGLIAFERRI, 2018). No caso do

1 https://www.python.org/

2 http://www.spyder-ide.org

Windows e MacOS, é recomendável utilizar a instalação *Anaconda* (disponível em https://www.anaconda.com/products/distribution).

O Spyder é uma ferramenta livre e multiplataforma para desenvolvimento Python. A vantagem de utilizar essa ferramenta são os diversos recursos disponíveis na mesma como editor de código, console, explorador de variáveis, depurador e ajuda. O editor de código possui um explorador de código e autocomplemento. O console permite a execução de comandos Python. Com o explorador de variáveis é possível visualizar e modificar o conteúdo das variáveis. O depurador auxilia na execução passo a passo do código e o sistema de ajuda fornece documentação de comandos da linguagem.

Alguns recursos interessantes da ferramenta Spyder são os pontos de parada, a execução passo a passo e o explorador de variáveis. Tais recursos são muito úteis para fazer a depuração do código e encontrar possíveis erros de funcionamento.

Os pontos de paradas podem ser ativados com um clique duplo sobre o número de cada linha. Aparece um pequeno círculo vermelho antes do número da linha, se o ponto de parada foi ativado. Assim, quando o código é executado no modo depurar, o Spyder interrompe a execução no ponto de parada ativado.

A execução em modo depurar é feita pelo menu *Depurar / Depurar* ou CTRL-F5 (pelo teclado). O explorador de variáveis é o painel no lado esquerdo. Quando o código for executado, você consegue ver o valor de cada variável. A execução passo a passo, no menu *Depurar / Executar linha* ou CTRL+F10, permite executar uma linha do código de cada vez. A utilização dos recursos explicados de forma conjunta pode ser muito útil para encontrar possíveis erros no funcionamento dos códigos.

Os códigos em Python são usados para resolver problemas do mundo real e, ao serem executados, precisam trabalhar com diversos tipos de dados. A Figura 6 apresenta os tipos de dados básicos da linguagem Python (PSF, 2021). Os tipos de dados são usados principalmente para representar varáveis e literais. Os Literais são os dados informados literalmente no código. As variáveis são explicadas na próxima seção.

Tipo	Descrição
int	Números inteiros
float	Números fracionários
bool	Valores lógicos como **True** (verdadeiro) ou **False** (falso)
str	Valores textuais

Figura 6 – Tipos básicos de dados em Python
Fonte: Elaborado pelo Autor.

Uma variável representa uma posição de memória no computador cujo conteúdo pode ser lido e alterado durante a execução do código (BORGES, 2010). Em Python, deferente de outras linguagens de programação, não precisamos declarar as variáveis. As variáveis são criadas automaticamente quando ocorre uma atribuição de valor com o operador de igualdade (**=**). A atribuição de valores a variáveis pode ser realizada com literais, outras variáveis, resultados de expressões e valores retornados por funções.

A Figura 7 mostra um exemplo de código com os quatro tipos básicos de dados. No caso dos tipos textuais, é importante colocar o valor a ser atribuído entre aspas. Aqui cabe uma observação sobre a função **print()**. Podemos colocar quantos literais ou variáveis desejarmos separados por vírgula. A função irá escrever todos na tela. No caso das variáveis, a função escreve seu valor na tela (PILGRIM, 2009).

```
n1 = 10
n2 = 2.5
nome = 'José'
teste = True
print(n1, n2, nome, teste)
```

Figura 7 – Atribuição de valores a variáveis
Fonte: Elaborado pelo Autor.

Para que não ocorram erros no código, o nome de uma variável devem obedecer às seguintes regras:

- Deve obrigatoriamente começar com uma letra;
- Não deve possuir espaços em branco;
- Não pode conter símbolos especiais, exceto o sublinhado **(_)**;
- Não deve ser uma palavra reservada (uma palavra da já existente na linguagem, como **print**, por exemplo).

Dica do Professor: A linguagem Python, assim como diversas outras linguagens, é *case sensitive*. Nessas linguagens, a mudança de maiúscula para minúscula, ou vice-versa, modifica o nome da variável (ou outro elemento). Assim, os nomes **teste** e **Teste** são considerados distintos.

Um dos principais cuidados na escrita de códigos é a ter uma boa legibilidade. Um código escrito por você deve ser facilmente entendido por outra pessoa que vá estudá-lo ou usá-lo (ou por você mesmo no futuro). Alguns fatores que influenciam diretamente o entendimento de códigos são nomes sugestivos, endentação e comentários.

Os nomes sugestivos para variáveis e outras estruturas são importantes para que você identifique de forma rápida a referida variável ou estrutura. A endentação diz respeito aos espaços em branco a esquerda para alinhar e organizar os blocos de instruções. No caso do Python, a endentação é essencial porque é por meio dela que colocamos blocos de códigos dentro de outras instruções. Já os comentários textos não executáveis com a finalidade de explicar o código (SWEIGART, 2021). Os comentários começam com o caractere **#.**

```
# -*- coding: utf-8 -*-

n1 = 10         # Número inteiro
n2 = 2.5        # Número fracionário
nome = 'José' # Textual (sempre entre aspas)
teste = True  # Variável lógica
```

Figura 8 – Código com comentários
Fonte: Elaborado pelo Autor.

A Figura 8 mostra um exemplo de código com comentários em Python. Após cada atribuição, temos comentários comuns descrevendo o tipo de dado. Já na primeira linha, temos um comentário especial usado para especificar a codificação de caracteres usada no arquivo. Nesse exemplo e nos demais ao longo do livro usaremos sempre a codificação UTF8. A linguagem Python diversos outros comentários especiais usados para outras funções.

1.5.1 Entrada e saída de dados

Quando um algoritmo recebe dados do usuário, dizemos que ocorre uma entrada de dados. De forma análoga, quando o algoritmo exibe mensagens para o usuário, acontece uma saída de dados. Em Python, a entrada de dados é efetuada com a instrução **input()**. Dentro dos parênteses colocamos um texto para ser mostrado ao usuário antes da entrada dos dados. Normalmente, usamos uma variável para receber a resposta digitada. A função **input()** sempre retorna um valor textual digitado pelo usuário. Se precisarmos de outro tipo de dado, temos que realizar uma conversão[3].

No caso da saída de dados, temos a instrução **print()**. Essa instrução recebe as informações a serem mostradas para o usuário separadas por vírgula. Se usarmos uma variável, será mostrado o valor dessa variável. Em diversas situações é importante compor mensagens adequadas para o usuário combinando literais e variáveis. A Figura 9 exibe um código usando as funções **input()** e **print()**. A instrução **int()** é usada para converter texto retornado pelo **input()** para um número inteiro.

```
print('Bem vindo!')
# Nome do usuário
nome = input('Informe seu nome: ')
# Idade convertida para inteiro
idade = int(input('Informe sua idade: '))
print() # Escreve uma linha em branco
# Escreve mensagem usando as variáveis
print('Olá,', nome, 'sua idade é', idade)
print('Até mais!')
```

Figura 9 – Código usando **input()** e **print()**
Fonte: Elaborado pelo Autor.

A instrução **print()**, por padrão, separa os elementos recebidos com um espaço em branco e, no final, escreve o caractere de nova linha (**'\n'**). Esse comportamento pode ser modificado através dos parâmetros **sep** (texto para separar os elementos) e **end** (texto a ser colocado no final da lina). Por exemplo, se quisermos que os elementos fiquem colados

3 Podem ocorrer erros nessa conversão, mas trataremos desse detalhe.

e a saída não passe para a próxima linha, podemos usar uma instrução no seguinte formato **print(..., sep='', end='')**. As reticências (...) devem ser substituídas pelos elementos a serem escritos na tela.

1.5.2 Operadores e expressões

Assim como a maioria das linguagens de programação, a linguagem Python possui operadores aritméticos, relacionais e lógicos (PSF, 2021). Com esses operadores é possível criar expressões que também podem ser combinadas, principalmente, para a realização de testes.

A realização de cálculos matemáticos está entre as principais tarefas feitas por algoritmos. Para executarmos tais cálculos, devemos utilizar os chamados operadores aritméticos. A Figura 10 mostra os operadores aritméticos disponíveis na linguagem Python. A ordem precedência dos operadores aritméticos é a mesma ordem precedência da matemática. Também podem ser usados parênteses para alterar a ordem de precedência. A Figura 11 mostra um exemplo simples com expressões matemáticas usando os operadores aritméticos.

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
//	Divisão inteira
%	Resto de divisão
**	Potenciação

Figura 10 – Operadores aritméticos do Python
Fonte: Elaborado pelo Autor.

```
ni = 10 // 3
print('10 // 3 =', ni)
ni = 10 % 3
print('10 % 3 =', ni)
nr = 2.34 * 3.58
print('2.34 * 3.58 =', nr)
nr = 2.99 / 4.1
print('2.99 / 4.1 =', nr)
nr = (10.5 - 7.8) * (3.2 + 200.43)
print('(10.5 - 7.8) * (3.2 + 200.43) =', nr)
```

Figura 11 – Código usando operadores aritméticos
Fonte: Elaborado pelo Autor.

Com o conteúdo visto até o momento, podemos fazer um código para simular uma calculadora simples. A ideia é obter dois números com o usuário e mostrar os resultados das possíveis operações aritméticas entre esses números. A Figura 12 mostra uma possível solução.

```
print('Informe dois números')
n1 = float(input('Primeiro número: '))
n2 = float(input('Segundo número: '))
r = n1 + n2
print(n1, '+', n2, '=', r)
r = n1 - n2
print(n1, '-', n2, '=', r)
r = n1 * n2
print(n1, '*', n2, '=', r)
r = n1 / n2
print(n1, '/', n2, '=', r)
```

Figura 12 – Código para simular calculadora simples
Fonte: Elaborado pelo Autor.

O operador **+** pode ser usado também para concatenar variáveis e literais textuais. No entanto, não é possível concatenar diretamente um elemento textual e um elemento numérico. Nesse caso, é necessário converter o valor numérico para textual antes da concatenação. O código da Figura 13 demonstra como isso pode ser feito.

```
print('Informe os dados')
nome = input('Nome: ')
sobrenome = input('Sobrenome: ')
idade = int(input('Idade: '))
mensagem = nome + ' ' + sobrenome + ', ' + str(idade)
print(mensagem)
```

Figura 13 – Código para simular calculadora simples
Fonte: Elaborado pelo Autor.

Em Python, é comum utilizar os operadores aritméticos compostos. Eles são equivalentes à aplicação do operador seguido de uma atribuição. Por exemplo, se temos uma variável **x** valendo **10** e usamos a expressão **x += 2**, estamos somando **2** à variável **x**, ou seja, é o mesmo que usar a expressão **x = x + 2**.

Os operadores relacionais são usados para comparar dois valores. Os valores a serem comparados podem ser literais, variáveis ou expressões matemáticas. O resultado dessa comparação é um valor lógico **True** (verdadeiro) ou **False** (falso). A Figura 14 apresenta os operadores relacionais disponíveis na linguagem Python.

Operador	Descrição
==	Igual
!=	Diferente
<	Menor
<=	Menor ou igual
>	Maior
>=	Maior ou igual

Figura 14 – Operadores relacionais
Fonte: Elaborado pelo Autor.

O código da Figura 15 implementa um comparador de valores que possibilita testar todos os operadores relacionais para dois números informados. Execute o referido código no Spyder e observe o resultado das comparações para diferentes valores.

```
n1 = int(input('Informe um número: '))
n2 = int(input('Informe outro número: '))
print(n1, '==', n2, '->', n1 == n2)
print(n1, '!=', n2, '->', n1 != n2)
print(n1, '<',  n2, '->', n1 < n2)
print(n1, '<=', n2, '->', n1 <= n2)
print(n1, '>',  n2, '->', n1 > n2)
print(n1, '>=', n2, '->', n1 >= n2)
```

Figura 15 – Código comparador de valores
Fonte: Elaborado pelo Autor.

Os operadores lógicos são equivalentes aos operadores da Lógica Proposicional. Em Python, temos o operador unário **not** e os operadores binários **and** e **or**. A Figura 16 mostra o funcionamento do operador **not**. Esse operador é usado para obter o oposto de um valor lógico.

Expressão	Resultado
`not True`	`False`
`not False`	`True`

Figura 16 – Operador lógico **not**
Fonte: Elaborado pelo Autor.

A Figura 17 descreve o funcionamento do operador lógico **and**. Podemos observar que esse operador retorna **True** apenas se os dois operandos forem **True**. Ou seja, se um dos operandos for **False**, o resultado do operador também será **False**.

Expressão	Resultado
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

Figura 17 – Operador lógico **and**
Fonte: Elaborado pelo Autor.

A Figura 18 exibe o funcionamento do operador lógico **or**. Esse operador retorna **False** somente se os dois operandos forem **False**. Como o resultado de um operador relacionado é um valor lógico, em muitas situações, os operadores lógicos são usados para combinar as comparações relacionais.

Expressão	Resultado
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Figura 18 – Operador lógico **or**
Fonte: Elaborado pelo Autor.

O código da Figura 19 é um exemplo de utilização dos operadores lógicos. Utilize o Spyder para executar esse código algumas vezes informando valores diferentes para as variáveis **n1** e **n2**. Observe funcionamento do código e tente calcular sozinho os valores das variáveis **p**, **q** e **r**.

```
n1 = int(input('Informe um número: '))
n2 = int(input('Informe outro número: '))
p = (n1 > n2)
q = (n1 != n2)
r = not (p or q) and (not p)
print('p =', p)
print('q =', q)
print('r =', r)
```

Figura 19 – Código com expressões lógicas e relacionais
Fonte: Elaborado pelo Autor.

1.5.3 Utilizando funções

O Python possui uma biblioteca padrão com funções que podem ser usadas diretamente no código. Além disso, podemos importar diversas outras bibliotecas que podem ser utilizadas para as mais variadas tarefas (BORGES, 2010). A Figura 20 exibe algumas funções da biblioteca padrão. Além da biblioteca padrão, abordaremos funções relacionadas ao tipo textual e também a biblioteca de funções matemáticas.

Função	Funcionamento
abs(n)	Retorna o valor absoluto de **n**
chr(n)	Retorna o caractere representado pelo número **n**
ord(c)	Retorna o código correspondente ao caractere **c**
round(n, d)	Arredonda **n** considerando **d** casas decimais
type(o)	retorna o tipo de **o**

Figura 20 – Algumas funções da biblioteca padrão do Python
Fonte: Elaborado pelo Autor.

Na linguagem Python, o tipo de dado textual (**str**) possui funções relacionadas que auxiliam na manipulação de dados desse tipo (DOWNEY, 2015). Por serem funções associadas ao tipo **str**, é preciso usá-las com o dado desse tipo. Por exemplo, se tempos uma variável **x** do tipo **str**, podem chamar uma função **lower()** com a instrução **x.lower()**.

A Figura 22 contém alguns exemplos de funções de manipulação de texto. O código da Figura 25 demostra a utilização de algumas dessas funções.

Função	Funcionamento
s.find(subtexto, ini, fim)	Procura **subtexto** em **s**, começando da posição **ini** até a posição **fim**. Retorna **−1**, se o **subtexto** não for encontrado.
s.format(x_1 , ..., x_n)	Retorna **s** com os parâmetros x_1, ..., x_n incorporados e formatados.
s.lower()	Retorna o **s** com todas as letras minúsculas.
s.replace(antigo, novo, n)	Retorna o **s** substituindo **antigo** por **novo**, nas **n** primeiras ocorrências.

Figura 21 – Algumas funções associadas ao tipo **str**
Fonte: Elaborado pelo Autor.

```
nome_completo = input('Informe seu nome completo: ')
sobrenome = input('Informe seu sobrenome: ')

pos = nome_completo.find(sobrenome)

if pos != -1:
    print('Seu sobrenome começa na posição ', pos)
else:
    print('Sobrenome não encontrado')

n = float(input('Informe um número: '))
print('{n:.8f}'.format(n=n))
```

Figura 22 – Código com manipulação de dados textuais
Fonte: Elaborado pelo Autor.

Além das funções da biblioteca padrão da linguagem, podemos importar bibliotecas adicionais. Uma dessas bibliotecas é a **math** contendo funções matemáticas (CORRÊA, 2020). A importação de bibliotecas adicionais é realizada com a instrução **import**, normalmente, incluída no início do código. A Figura 23 apresenta algumas das funções disponíveis na biblioteca **math**. Já o código da Figura 24 demostra como tais funções podem ser utilizadas.

Função	Funcionamento
ceil(x)	Retorna o teto de **x**
floor(x)	Retorna o piso de **x**
trunc(x)	Retorna a parte inteira de **x**
exp(x)	Retorna e^x
log(x, b)	Retorna o logaritmo de **x** em uma base **b**. Se a base não for especificada, retorna o logaritmo natural de **x**
sqrt(x)	Retorna a raiz quadrada de **x**
pi	Retorna o valor de π

Figura 23 – Algumas funções da biblioteca **math**
Fonte: Elaborado pelo Autor.

```
import math

n = float(input('Informe um número: '))
x = n * math.pi
print('x = n * pi = ', n)
print('Teto de x =', math.ceil(x))
print('Piso de x =', math.floor(x))
print('Log de x na base 10 =', math.log(x, 10))
print('Raiz de x =', math.sqrt(x))
```

Figura 24 – Código com funções matemáticas
Fonte: Elaborado pelo Autor.

1.5.4 Estruturas de decisão

Para resolver diversos tipos de problemas, precisamos usar as estruturas de decisão (também conhecidas como desvios condicionais ou estruturas condicionais). As estruturas de decisão são testes efetuados para decidir se uma determinada ação deve ser realizada (BORGES, 2010).

A estrutura de decisão mais simples utiliza uma única instrução **if**, seguida por uma expressão lógica e pelo bloco de instruções a ser executado se o valor da expressão lógica for verdadeiro (CEDER, 2018). Considere, por exemplo, o problema de verificar se um aluno foi aprovado. Isso é feito testando se a nota do aluno é maior ou igual a 60. A Figura 25 mostra o código para resolver esse problema.

```
nota = float(input('Informe a nota: '))
if nota >= 60:
    print('Aprovado')
print('Boas férias')
```

Figura 25 – Estrutura de decisão simples para testar aprovação de aluno
Fonte: Elaborado pelo Autor.

A endentação deve ser feita corretamente para especificar quais instruções estão dentro da estrutura condicional. Utilizamos quatro espaços para endentar um bloco de instruções. Observe que, depois da expressão lógica seguida por dois pontos (**:**), começa o

bloco de instruções do **if**. Esse bloco deve ser obrigatoriamente endentado. No código, a mensagem “Aprovado” é escrita na tela somente quando a nota é maior ou igual a 60. Por outro lado, a mensagem “Boas férias” é mostrada incondicionalmente, pois não está dentro do desvio condicional.

Na estrutura de decisão simples, executamos alguma ação somente se a expressão lógica testada for verdadeira. Por outro lado, em certas situações, também precisamos realizar alguma medida se o teste for falso. Nesse caso, precisamos utilizar uma estrutura de decisão composta acrescentando a instrução **else** após o bloco de código da instrução **if**. A instrução **else** é seguida por outro bloco de código que será executado quando o teste for falso.

Como exemplo, vamos reconsiderar o problema de aprovação do aluno e escrever uma mensagem quando o mesmo for reprovado. A Figura 26 mostra o novo código contendo a mensagem “Reprovado” quando o aluno tem nota menor que 60.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
else:
    print('Reprovado')
print('Boas férias')
```

Figura 26 – Código com estrutura de decisão composta
Fonte: Elaborado pelo Autor.

Na estrutura de decisão simples, ocorre um único teste e o fluxo de execução do algoritmo pode seguir por um ou dois caminhos. Em determinadas situações, precisamos de algoritmos com vários testes e, por consequência, vários fluxos de execução. Para fazer isso, temos que inserir uma estrutura de decisão dentro de outra estrutura de decisão. Nesse caso, dizemos que as estruturas de decisão estão aninhadas.

Como exemplo, vamos considerar mais uma vez o problema de aprovação de aluno, mas, agora, vamos tratar as seguintes situações:

- Se o aluno tiver nota maior ou igual a 60, será aprovado;
- Se a nota for menor que 40, ao aluno é reprovado;
- Por fim, se a nota for maior ou igual a 40 e menor do que sessenta o aluno está de recuperação.

A Figura 27 apresenta o código para resolver o problema considerando as novas situações. Observe que, dentro do primeiro **else**, quando o aluno não é aprovado, ocorre um segundo teste para verificar se o aluno foi reprovado ou está de recuperação. O código pode ser escrito de outras formas, mudando a ordem dos testes, e obter o mesmo resultado.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
else:
    if n >= 40:
        print('Reavaliação')
    else:
        print('Reprovado')
print('Boas férias')
```

Figura 27 – Código com estruturas de decisão aninhadas
Fonte: Elaborado pelo Autor.

Observe que, no código da Figura 27, tivemos que endentar mais o **if** mais interno. Caso tenhamos muitas estruturas de decisão aninhadas, a legibilidade pode ficar prejudicada. Assim, outra maneira de usar as estruturas de decisão aninhadas é de forma consecutiva, como mostrado na Figura 28. Nesse caso, incluímos um **if** imediatamente após o **else** para fazer o segundo teste. Também é possível juntar o **else** com o **if** formando a instrução **elif**, como é mostrado no código da Figura 29. Essa contração é especialmente útil quando temos muitos testes consecutivos a serem feitos.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
else if n >= 40:
    print('Reavaliação')
else:
    print('Reprovado')
print('Boas férias')
```

Figura 28 – Código com estruturas de decisão consecutivas
Fonte: Elaborado pelo Autor.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
elif n >= 40:
    print('Reavaliação')
else:
    print('Reprovado')
print('Boas férias')
```

Figura 29 – Código com estruturas de decisão consecutivas usando **elif**
Fonte: Elaborado pelo Autor.

Para exemplificar o uso de estruturas de decisão, consideraremos a solução de equações de segundo grau no formato $Ax^2 + Bx + C = 0$. O algoritmo para resolver esse problema deve fazer o seguinte:

1) Ler os termos A, B e C;

2) Garantir que temos uma equação de segundo grau testando se A é diferente de zero;

3) Se for uma equação de segundo grau, calculamos o delta ($\Delta = B^2 - 4 \times A \times C$);

4) Após o cálculo, temos três situações para o delta:

- Se o delta for menor que zero, a equação não possui raízes;
- Se o delta for igual a zero, então a equação possui uma única raiz;
- Por fim, se o delta é maior que zero temos duas raízes $X_1 = \frac{-B+\sqrt{\Delta}}{2 \times A}$ e $X_2 = \frac{-B-\sqrt{\Delta}}{2 \times A}$.

A Figura 30 apresenta o código para resolver equações de segundo grau.

```
import math
print('Informe os termos da equação Ax² + Bx +C')
a = float(input('A: '))
b = float(input('B: '))
c = float(input('C: '))
if a == 0:
    print('Não é uma equação de segundo grau')
else:
    delta = b**2 - 4 * a * c
    if delta < 0:
        print('A equação não tem raízes')
    elif (delta == 0):
        x1 = b * (-1) / 2 * a
        print('A equação possui a raiz:', x1, '')
    else:
        raiz_delta = math.sqrt(delta)
        x1 = (b * (-1) + raiz_delta)/ 2 * a
        x2 = (b * (-1) - raiz_delta)/ 2 * a
        print('A equação possui duas raízes:')
        print('x1 =', x1)
        print('x2 =', x2)
```

Figura 30 – Código para resolver equações de segundo grau
Fonte: Elaborado pelo Autor.

1.5.5 Estruturas de repetição

As estruturas de repetição possibilitam executar repetidamente blocos de instruções por um número definido de vezes ao até que uma dada condição seja atendida.

O laço de repetição **while** é indicado quando não sabe o número de repetições a serem executadas. Por exemplo, a listagem dos números pares menores do que um número informado pelo usuário. Nesse problema, não é possível saber o número de repetições, pois não tem como prever o número que o usuário informará.

No laço **while**, as repetições ocorrem enquanto uma determinada condição for verdadeira (CORRÊA, 2020). Assim, é importante garantir que a condição de parada realmente aconteça e o laço não fique repetindo indefinidamente. Vamos considerar novamente o problema da listagem dos números pares. Nesse caso, a condição de parada pode ser quando chegarmos a um número maior do que o número informado pelo usuário.

A Figura 31 mostra o código para listagem de números pares. Usamos a variável **atual** para armazenar o número atual, começando pelo zero. O laço **while** usa a condição

de parada **atual < n**, para verificar se o número atual ainda é menor que **n** informado pelo usuário.

```
atual = 0
n = int(input('Informe um número: '))
while atual < n:
    print(atual)
    atual += 2
```

Figura 31 – Listagem de números pares
Fonte: Elaborado pelo Autor.

A estrutura de repetição **while** possui uma condição que é testada antes mesmo de executar a primeira repetição. Enquanto essa condição for verdadeira, as repetições continuam acontecendo. Se o usuário informar zero, por exemplo, não acontece nenhuma repetição. Dentro do laço, somamos mais dois ao número atual para chegarmos ao próximo número par.

O laço de repetição **for** é indicado quando se sabe o número de repetições a serem feitas. Basicamente, o laço **for** percorre de forma automática os elementos de uma estrutura de lista. A maneira mais comum de utilizar o laço **for** é com a função **range()**. Essa função recebe um número inteiro **n** e gera um intervalo de números de 0 até **n – 1** (CORRÊA, 2020).

Para exemplificar o uso do laço **for**, vamos considerar o problema de somar 10 números informados pelo usuário. A Figura 32 mostra o código para resolver esse problema. Observe que, no laço, for usamos a função **range(10)** e a variável **cont** para percorrer os números do intervalo gerado pela função **range()**. Assim, na primeira repetição, a variável **cont** vale **0**, na segunda, vale **1**, e assim por diante até assumir o valor **9**.

```
soma = 0
for cont in range(10):
    n = float(input('Informe um número: '))
    soma += n
print(soma)
```

Figura 32 – Soma de 10 números usando o laço **for**
Fonte: Elaborado pelo Autor.

É possível notar, no código para soma dos 10 números, que a única função da variável **cont** é controlar as repetições do laço **for**. Quando temos uma variável que não é usada em nenhuma outra parte do código, podemos substituir essa variável pela variável anônima **_** (sublinhado).

Outro detalhe importante é a função **range()**. Além do fim do intervalo, podemos definir o início e a periodicidade dos números. Se usarmos, por exemplo, a instrução **range(2, 6)**, será retornado o intervalo de números "**2, 3, 4, 5**", ou seja, o primeiro número é o início e o segundo número é o fim do intervalo. Lembrando que o número do fim não é incluído no intervalo. Quando incluímos um terceiro número, definimos a periodicidade dos números. Como exemplo, se usarmos a instrução **range(3, 19, 4)** teremos a sequência "**3, 7, 11, 15**". A sequência começa em **3**, e os demais números são obtidos somando **4** ao número atual, até atingir o fim do intervalo. Além disso, no lugar dos números, podemos

usar qualquer variável ou expressão que retorne um número inteiro. Também podemos obter intervalos em ordem decrescente, por exemplo, a instrução **range(5,0,-1)** retorna a sequência "**5, 4, 3, 2, 1**".

O laço de repetição **while** precisa testar a condição de parada antes mesmo da primeira repetição. Entretanto, em diversos momentos, precisamos executar a primeira repetição antes de testar a condição de parada. Nesse caso, podemos criar um laço com a instrução **while True** e utilizar a instrução **break** para finalizar o laço de repetição.

Como exemplo, vamos considerar a soma de uma quantidade indeterminada de números informados pelo usuário. Devemos parar de somar apenas quando o usuário informar o número **0** (zero). A Figura 33 apresenta o código para resolver esse problema. É importante tomar um certo cuidado com laços **while True**, temos que garantir que a instrução **break** será executada em algum momento e o laço não fique repetido indefinidamente.

```
soma = 0
while True:
    n = float(input('Informe um número: '))
    if n == 0:
        break
    soma = soma + n
print('Soma dos números:', soma)
```

Figura 33 – Soma indefinida de números
Fonte: Elaborado pelo Autor.

Vamos considerar agora uma modificação no problema de somar números. Além do que já foi mencionado, suponha que os números negativos não devam ser somados. Diante disso, podemos usar a instrução **continue** para ignorar os números negativos e *pular* para a próxima repetição do laço (CEDER, 2018). O código da Figura 34 mostra a solução para a modificação do problema. As instruções **break** e **continue** podem ser usadas tanto no laço **while** quanto no laço **for**.

```
soma = 0
while True:
    n = float(input('Informe um número: '))
    if n < 0:
        continue
    if n == 0:
        break
    soma = soma + n
print('Soma dos números:', soma)
```

Figura 34 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

Assim como as estruturas de decisão, as estruturas de repetição também são extensamente usadas para resolver diversas categorias de problemas. Um problema interessante para ser resolvido com laço de repetição é o cálculo de máximo divisor comum (MDC) com a técnica de Euclides (WIKIPÉDIA, 2021a). Lembrando que o MDC de números é o maior número que os divide sem deixar resto. Basicamente, a técnica de Euclides consiste nos seguintes passos:

- Dividimos o maior número pelo menor e verificamos se o resto da divisão é zero;
- Em caso afirmativo, o menor número é o MDC;
- Caso contrário, substituímos o maior número pelo menor, o menor número pelo resto da divisão e repetimos o processo.

Repare que a repetição do processo no terceiro ponto caracteriza uma estrutura de repetição. Como exemplos, demonstraremos como calcular o MDC de 144 e 56. O processo é o seguinte:

- Começamos dividindo 144 por 56. Como o resto da divisão é 32, vamos considerar os números 56 e 32 e repetir o processo;
- Dividimos 56 por 32 e temos 24 como resto. Agora, consideramos 32 e 32 e dividimos novamente;
- Na divisão de 32 por 24, chegamos a 8 como resto. Assim, a próxima divisão será 24 por 8;
- Por fim, dividimos 24 por 8 e temos zero como resto. Logo, o MDC é o número 8.

O laço de repetição mais adequado para implementar o algoritmo de Euclides é o **while**, pois não tem como prever quantas divisões precisam ser feitas. A Figura 35 mostra o código do algoritmo de Euclides.

```
print('Informe dois números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))

if n1 < 1 or n2 < 1:
    print('Números inválidos para MDC')
else:
    while True:
        resto = n1 % n2
        print (n1, '/' , n2, '-> resto:', resto)
        if resto == 0:
            break
        n1 = n2
        n2 = resto
    print('O MDC é ', n2)
```

Figura 35 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

A condição de parada do laço é **True**. Todavia, dentro do laço, testamos se o resto da última divisão é zero e interrompemos o laço com a instrução **break**. Antes do laço de repetição, usamos uma estrutura de decisão para testar se os números são válidos (positivos maiores que zero). O **print()** dentro do laço não é necessário, ele foi incluído apenas para mostrar as divisões realizadas do processo.

1.5.6 Tratamento de exceções

Considere uma instrução **n = float(input('Informe um número'))**. Estamos pedindo ao usuário para digitar um número. Todavia, se for digitado qualquer texto que não possa ser convertido, para número ocorrerá um erro em tempo de execução. Assim, ainda que o código esteja correto, podem ocorrer situações que causam erros. Esses erros são chamados de exceções e podem ser contornados usando a instrução **try** da linguagem Python (PSF, 2021).

O código da Figura 36 é um exemplo simples de tratamento de exceção. Dentro da cláusula **try**, colocamos as instruções de código que deveriam ser executadas normalmente. Se ocorrer algum erro, o fluxo de execução é direcionado para a cláusula **except** (CEDER, 2018). Execute o código e informe números corretos e incorretos. Você poderá ver que, ao ocorrer um erro, será executada a instrução da cláusula **except**.

```
try:
    print('Informe dois números')
    n1 = float(input('n1: '))
    n2 = float(input('n2: '))
    r = n1 / n2
    print(n1, '/', n2, '=', r)
except:
    print('Ocorreu um erro.')
```

Figura 36 – Tratamento simples de exceção
Fonte: Elaborado pelo Autor.

No código anterior, se ocorrer um erro, a execução será finalizada. Assim, caso o usuário quiser tentar novamente, será preciso executar o código novamente. Uma solução para esse problema é colocar o tratamento de exceção dentro de um laço de repetição. Dessa maneira, se ocorrer um erro, o usuário poderá tentar novamente sem ter que executar todo o código mais uma vez. Essa solução é mostrada na Figura 37. Como foi usado um laço **while True**, temos que incluir a instrução **break** no final da cláusula **try** para, na ausência de erros, finalizar o laço. Por outro lado, se ocorrer algum erro, a cláusula **except** é disparada e, depois de sua execução, o laço continua as repetições.

```
import math
while True:
    try:
        print('Informe dois números')
        n1 = float(input('n1: '))
        n2 = float(input('n2: '))
        r = n1 / n2
        break
    except:
        print('Ocorreu um erro! Tente novamente.')
print(n1, '/', n2, '=', r)
```

Figura 37 – Tratamento de exceção com repetição
Fonte: Elaborado pelo Autor.

Em nosso código, além do erro de digitação pelo usuário, pode acontecer o erro de divisão por zero na instrução **r = n1 / n2**. A instrução **except** captura todos os tipos de erros. Se for necessário tratar erros diferentes, temos que colocar uma cláusula **except**

para cada tipo de erro. Para descobrir as classes dos erros, podemos usar a função **type()** como mostrado no código da Figura 38.

```
while True:
    try:
        print('Informe dois números')
        n1 = float(input('n1: '))
        n2 = float(input('n2: '))
        r = n1 / n2
        break
    except Exception as e:
        print('Ocorreu o seguinte erro:', type(e))
print(n1, '/', n2, '=', r)
```

Figura 38 – Descobrindo classes de erros
Fonte: Elaborado pelo Autor.

Se for digitado um número incorretamente, temos a exceção **ValueError**. Se o segundo número for zero, temos a exceção **ZeroDivisionError**. O código da Figura 39 mostra como fazer o tratamento de exceção específico para cada uma dessas classes de erro.

```
while True:
    try:
        print('Informe dois números')
        n1 = float(input('n1: '))
        n2 = float(input('n2: '))
        r = n1 / n2
        break
    except ValueError as e:
        print(e)
        print('Número inválidos! Tente novamente.')
    except ZeroDivisionError as e:
        print(e)
        print('Divisão por zero! Tente novamente.')
print(n1, '/', n2, '=', r)
```

Figura 39 – Tratamento de exceções específicas
Fonte: Elaborado pelo Autor.

A instrução **try** possui também as cláusulas **else** e **finally**. A cláusula **else** pode ser usada para realizar alguma ação quando não ocorrer erros. Já a cláusula **finally** é executa incondicionalmente, ocorrendo erros ou não. A cláusula **finally** é útil para executar ações de limpeza como fechamento de arquivos.

1.5.7 Modularização

A modularização consiste em dividir um algoritmo em partes menores chamadas sub-rotinas ou funções com o intuito de facilitar o desenvolvimento e a manutenção de código (BORGES, 2010). Considerando um problema grande a ser resolvido, as funções representam pequenas partes do problema maior com menor complexidade. Além disso, as funções podem ser trabalhadas de forma independente e a localização de erros se torna mais fácil.

Um código deve ser desenvolvido de forma modular sempre que possível. Assim, se um mesmo trecho de código é executado em pontos diferentes do programa, podemos criar uma função para executar esse código uma única vez e chamar a função quando necessário. As principais vantagens da modularização são as seguintes:

- Evita a reescrita desnecessária de códigos similares;
- Melhora legibilidade do código;
- Permite desenvolvimento em equipe (cada programador cuida de um trecho do código);
- As funções podem ser testadas isoladamente para verificação de erros;
- A manutenção se torna mais fácil, apenas algumas partes podem precisar de alteração.

Funções são trechos de código que executam determinada tarefa ao serem chamados e depois retornam o controle para o ponto em que foram chamados (CEDER, 2018). Apesar de ainda não termos criados nossas próprias funções, já escrevemos diversos códigos utilizando funções prontas como **input()** e **print()**. Quando chamamos uma função devemos informar os parâmetros necessários. Isto significa que se vamos usar uma função **teste()** que recebe um parâmetro **int** e outro **str**, então devemos usar a instrução **teste(a, b)**, onde **a** é do tipo **int** e **b** é do tipo **str**.

A criação de novas funções é feita com a instrução **def**, seguida pelo nome da função e seus parâmetros entre parênteses e dois pontos (**:**). A linha com a instrução **def** é chada de declaração ou cabeçalho da função. Tanto o nome da função e seus parâmetros devem obedecer às mesmas regras dos nomes de variáveis. Após a declaração, vem o chamado corpo da função com suas instruções endentadas. No corpo da função, quando a função precisar tiver que retornar algum valor, é usada a instrução **return**.

O código da Figura 40 contém a função **saudacao()** que não possui parâmetros e não retorna valores. É importante incluir uma linha em branco entre o corpo da função e a próxima instrução para manter uma boa legibilidade no código. É importante frisar que as instruções **print()** da função não caracterizam retorno de valor. Quando uma função retorna um valor, podemos atribuir ser resultado a uma variável. Esse é o caso da função **input()**, por exemplo.

Na última linha do código, acontece chamada à função. Essa chamada direciona o fluxo de execução do código para a primeira instrução da função. Após a execução de todas as linhas da função, o fluxo de execução retorna para a linha onde ocorreu a chamada. Além disso, as funções são executadas somente quando são chamadas. No código da Figura 40, por exemplo, a primeira instrução a ser executada é **saudacao()**.

```
def saudacao():
    print('*****************************')
    print('*         BEM VINDO         *')
    print('*****************************')

saudacao()
```

Figura 40 – Função **saudacao()**
Fonte: Elaborado pelo Autor.

No código da Figura 40 criamos apenas uma função. Porém, para a grande maioria dos problemas precisar escrever códigos com várias funções. Além disso, podemos criar uma função pode chamar outras funções. Uma função pode declarar variáveis para serem utilizadas apenas internamente. Estas variáveis, assim como os parâmetros da função, são chamadas de variáveis locais enquanto as variáveis de fora da função são as variáveis globais. É importante que as variáveis sejam locais sempre que possível.

No código anterior, podemos ver que a função **saudacao()** não recebe nenhum parâmetro e nem dados do usuário. Isso faz com que essa função sempre realize as mesmas ações. Na maioria das vezes, precisamos criar funções parametrizáveis que realizam ações específicas conforme os parâmetros recebidos. Na prática, os parâmetros são como variáveis já inicializadas recebidas pelas funções.

A Figura 41 mostra um código com a função **cubo()**. A função recebe como parâmetro o número **num** e calcula o cubo do mesmo. Contudo, essa função ainda pode ser melhorada. Observe que não foi usada a instrução **return** para retornar o resultado do cálculo. A função está simplesmente escrevendo na tela. Assim, se modificarmos a função para retornar o resultado, teremos uma função mais útil. Isso porque podemos usar a função em qualquer lugar, sem ter que escrever na tela e usar seu resultado da maneira que for mais apropriada.

```
def cubo(num):
    cubo = num * num * num
    print(num, 'ao cubo é', cubo)

n = float(input('Informe um número: '))
calcula_cubo(n)
```

Figura 41 – Função **cubo()** sem retorno
Fonte: Elaborado pelo Autor.

```
def cubo(num):
    return num * num * num

n = float(input('Informe um número: '))
print(n, 'ao cubo é', cubo(n))
print(n, 'elevado a nona é', cubo(cubo(n)))
```

Figura 42 – Função **cubo()** com retorno
Fonte: Elaborado pelo Autor.

A Figura 42 mostra a modificação da função **cubo()** com o retorno do resultado. Repare que podemos usar diretamente a função dentro do **print()**. Além disso, podemos usar o resultado da função em qualquer expressão numérica, como na instrução **cubo(cubo(n))**.

Na chamada de funções é possível, passar parâmetros em ordem diferente daquela especificada da declaração, desde que existam atribuições aos nomes dos parâmetros. Também podemos criar funções com parâmetros opcionais. Os parâmetros opcionais devem ser os últimos e devem ter um valor padrão já atribuído. Assim, na chamada da função, se o parâmetro opcional não for passado, será usado o valor padrão.

```
def juros(capital, taxa, tempo=12):
    return (capital * taxa * tempo) / 100

print('Cálculo de juros')
cap = float(input('Capital: '))
tax = float(input('Taxa: '))
tem = input('Tempo (deixe em branco para o padrão de 12): ')
if tem == '':
    jur = juros(cap, tax)
else:
    tem = float(tem)
    jur = juros(taxa=tax, capital=cap, tempo=tem)
print('O valor dos juros é', jur)
```

Figura 43 – Código com Nomes de parâmetros e parâmetros opcionais
Fonte: Elaborado pelo Autor.

A Figura 43 exibe um código com parâmetros opcionais e chamada de função usando os nomes dos parâmetros. Na função **juros()**, temos o parâmetro **tempo** como opcional, seu valor padrão é **12**. Na instrução **jur = juros(cap, tax)**, é feita uma chamada de função sem usar os nomes de parâmetros. Nesse caso, temos que manter os parâmetros na ordem correta. Primeiro, o parâmetro **capital** e, depois, o parâmetro **taxa**. Nessa linha o parâmetro opcional **tempo** não foi usado. Já na instrução **jur = juros(taxa=tax, capital=cap, tempo=tem)**, ocorre uma chamada de função usando os nomes dos parâmetros. Observe que eles não estão na mesma ordem da declaração, mas, como usamos os nomes, isso não é um problema.

A recursão ocorre quando uma função chama a si mesma. Na prática, é possível usar laços de repetição para substituir recursões. Contudo, em muitas situações, funções recursivas podem ser mais intuitivas do que laços de repetição. Uma observação importante é que precisamos ter cuidado com a condição de parada, assim como fazemos nos laços de repetição. Caso contrário, podemos cair em uma recursão infinita que a função faz chamadas a ela mesma indefinidamente.

```
def fat(num):
    if num <= 1:
        return 1
    return num * fat(num - 1)

n = int(input('Informe um número: '))
print('O fatorial de', n, 'é', fat(n))
```

Figura 44 – Função recursiva **fat()**
Fonte: Elaborado pelo Autor.

O código da Figura 44 mostra apresenta a função recursiva **fat()**. O **if** faz o teste da condição de parada. Quando o parâmetro de entrada for menor ou igual a um, seu fatorial

será um. Na última linha da função, ocorre a chamada recursiva, usando a equivalência **n! = n * (n-1)!**.

Dependendo da quantidade de código, pode ser interessante a criar módulos para agrupar as funções correlacionadas. Normalmente, os módulos possuem apenas definições de funções ou outras estruturas e o código principal controla o fluxo de execução do código importando os módulos e chamando suas funções.

Para exemplificar a criação de módulos, consideraremos o problema de calcular o cubo e o fatorial de um número. A Figura 45 mostra o módulo com as funções que realizam esses cálculos.

```
def cubo(num):
    return num * num * num

def fat(num):
    if num <= 1:
        return 1
    return num * fat(num - 1)
```

Figura 45 – Módulo **mat.py**
Fonte: Elaborado pelo Autor.

Na Figura 46 temos o código do módulo principal que controla o fluxo de execução e efetua as chamadas as funções do módulo **mat.py**. É importante que ambos módulos sejam salvos na mesma pasta ou diretório. No código, especificamos quais funções deveriam ser importados usando a instrução **from ... import ...**. Poderíamos fazer de outra maneira, fazendo somente a importação do módulo **mat** e chamando as funções no formado **mat.cubo(...)** e **mat.fat(...)**.

```
from mat import cubo, fat

n = int(input('Informe um número: '))
print(n, 'ao cubo é', cubo(n))
print('O fatorial de', n, 'é', fat(n))
```

Figura 46 – Módulo **principal.py**
Fonte: Elaborado pelo Autor.

A linguagem Python é uma linguagem interpretada, ou seja, não é preciso compilar o código-fonte para gerar arquivos executáveis. No caso do Linux, podemos criar scripts executáveis usando o comentário especial **#!/usr/bin/env python3** na primeira linha do módulo principal. Além disso, temos que dar permissão de execução ao arquivo. No caso do Windows, podemos associar a abertura de arquivos **.py** ao programa **pythonw.exe** disponível na pasta de instalação do Spyder. Assim, os scripts podem ser executados diretamente através do gerenciador de arquivos ou linha de comando. A Figura 47 mostra um script executável juntando os dois módulos do exemplo anterior.

```
#!/usr/bin/env python3

# -*- coding: utf-8 -*-

def cubo(num):
    return num * num * num

n = int(input('Informe um número: '))
print(n, 'ao cubo é', cubo(n))
```

Figura 47 – Exemplo de script executável
Fonte: Elaborado pelo Autor.

1.5.8 Decomposição de problemas

A principal vantagem da modularização é a possibilidade de usarmos decomposição de problemas para resolver problemas mais complexos. Assim, podemos quebrar um problema maior em pequenas partes. Cada uma dessas partes pode ser resolvida com uma função específica. Ao final, juntamos as funções para solucionar o problema inicial.

Para exemplificar a decomposição de problemas construiremos uma calculadora de expressões considerando os seguintes pontos:

- A calculadora consiste em um console onde o usuário digita comandos;
- O usuário pode digitar expressões aritméticas para serem calculadas, ver o histórico de expressões ou sair;
- Os comandos **h** e **s** são usados respectivamente para histórico e sair;
- Deve ser realizado tratamento de exceção no cálculo da expressão digitada para garantir que a mesma não possuir erros;
- O histórico deve guardar apenas as expressões sem erros.

A descrição da calculadora pode parecer muito complexa, mas vamos decompô-la em problemas menores para facilitar a implementação. Primeiro construiremos uma função que recebe o texto da expressão e calcula o resultado. Para facilitar um pouco mais a nossa vida, usaremos a função **eval()** do Python. Essa função é capaz de executar um texto como se fosse os códigos em Python. No caso de uma expressão aritmética, a função **eval()** retorna o resultado dessa expressão. Entretanto, se a expressão for possuir algum erro de sintaxe, ocorrerá um erro. Portanto, temos que fazer o tratamento de exceções ao executarmos a função **eval()**.

```
def calcula(expr):
    try:
        return eval(expr)
    except:
        print('Expressão inválida!')
        return None
```

Figura 48 – Função **calcula()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

A Figura 48 exibe o código da função **calcula()**. Usamos a instrução **try** para fazer o tratamento de exceção. No caso de algum erro, mostramos a mensagem de expressão inválida e retornamos o valor **None**. Esse valor é um valor nulo que podemos testar ao receber o resultado da função.

```
def historico(expr, res):
    global HIST
    if res is not None:
        HIST += '\n\n'+ expr
        HIST += '\n' + str(res)
```

Figura 49 – Função **historico()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

Agora criaremos uma função para atualizar o histórico da calculadora. Seu código é mostrado na Figura 49. A instrução **global HIST** é usada para podermos alterar a variável global **HIST**. Tal variável será declarada posteriormente para armazenar o histórico. Nesse problema estamos adotando maiúsculas para variáveis globais e minúsculas para variáveis locais. A função **historico()** recebe os parâmetros **expr** (texto da expressão calculada) e **res** (resultado do cálculo). O **if** faz um teste para verificar se a expressão é válida. O teste **res is not None** é usado para verificar se o resultado da expressão (**res**) é diferente de **None**. Dentro do **if** apenas adicionamos a expressão e seu resultado ao histórico. Utilizamos o **'\n'** para separar as linhas do histórico.

Depois de construirmos as duas funções auxiliares, vamos escrever a função principal da calculadora que deverá gerenciar o fluxo de execução e chamar as funções auxiliares quando necessário. A Figura 50 mostra o código dessa função.

```
def principal():
    while True:
        print('Informe a expressão matemática')
        print('(h para histórico, s para sair)')
        expr = input()
        if expr.lower() == 's':
            break
        if expr.lower() == 'h':
            print(HIST, '\n')
        else:
            res = calcula(expr)
            historico(expr, res)
            print(res, '\n')
```

Figura 50 – Função **principal()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

Temos um laço de repetição para que o usuário possa digitar quantas expressões desejar. A expressão **expr.lower() == 's'** testa se o usuário informou o comando **s** e interrompe o laço de repetição. No caso do comando **h**, o teste é feito com a expressão **expr.lower() == 'h'** e apenas mostramos o histórico de cálculos guardado na variável **HIST**. Repare que não precisamos da instrução **global HIST** nessa função porque estamos apenas lendo o conteúdo da variável global.

Se o usuário não informar os comandos **s** ou **h**, partimos para o cálculo da expressão. O resultado da expressão é guardado na variável **res**. Em seguida, chamamos a função de atualizar o histórico. Por fim, mostramos o resultado do cálculo da expressão.

As funções desenvolvidas até agora não são suficientes para que a calculadora funcione. Temos que declarar a variável global **HIST** e chamar a função **principal()**. A Figura 51 exibe o código completo da calculadora de expressões. Os comentários das duas primeiras linhas permitem que o código seja executado na forma de script. A variável global **HIST** é inicializada com um texto vazio. A instrução **if __name__ == '__main__'** utiliza a variável especial **__name__** do Python para verificar se o módulo foi executado como um script. Quando isso acontece o conteúdo dessa variável é **'__main__'**.

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  HIST = ''
5
6  def calcula(expr):
7      try:
8          return eval(expr)
9      except:
10         print('Expressão inválida!')
11         return None
12
13 def historico(expr, res):
14     global HIST
15     if res is not None:
16         HIST += '\n\n'+ expr
17         HIST += '\n' + str(res)
18
19 def principal():
20     while True:
21         print('Informe a expressão matemática')
22         print('(h para histórico, s para sair)')
23         expr = input()
24         if expr.lower() == 's':
25             break
26         if expr.lower() == 'h':
27             print(HIST, '\n')
28         else:
29             res = calcula(expr)
30             historico(expr, res)
31             print(res, '\n')
32
33 if __name__ == '__main__':
34     principal()
```

Figura 51 – Código completo da calculadora de expressões
Fonte: Elaborado pelo Autor.

1.5.9 Coleções

Os tipos de dados básicos permitem armazenar um único valor de um tipo de dado. Entretanto, em muitas situações, precisamos de estruturas de dados mais complexas,

chamadas de coleções de dados, para agrupar diversos elementos de dados. A vantagem dessas estruturas está na facilidade de acesso e manipulação desses elementos de dados. Os principais tipos de coleções de dados são listas, tuplas, dicionários e conjuntos (PSF, 2021).

Para ilustrarmos a necessidade do uso de coleções de dados, vamos considerar o código da Figura 52. O código recebe o preço de 10 produtos e calcula o preço médio dos mesmos. Agora imagine que fosse necessário listar os produtos com preço acima da média. Como fazer isto de uma forma eficiente?

```
soma = 0
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
media = soma / 10
print('O preço médio é', media)
```

Figura 52 – Cálculo do preço médio de produtos
Fonte: Elaborado pelo Autor.

O problema das preços acima da média é que usamos os preços dos produtos para calcular a média e depois precisamos novamente desses preços. Uma solução seria declarar 10 variáveis uma para cada produto. Porém, imagine uma lista de 50 produtos ou mais, a declaração de variáveis individuais não é viável na prática.

Outra solução seria pedir para o usuário para informar os preços, realizar o cálculo da média e solicitar os preços novamente. Essa solução também não é eficiente, pois o usuário precisa digitar a lista de preços duas vezes. Além da sobrecarga do usuário, pode ocorrer uma digitação errada na segunda vez. A melhor solução para o problema é a utilizar uma coleção de dados para guardar os preços de todos os produtos e, depois do cálculo da média, revisitar a coleção para buscar aqueles produtos com preço acima da média.

As listas são estruturas dinâmicas e sequenciais de elementos (CORRÊA, 2020). Por ser dinâmica, podemos alterar a lista com a inclusão ou remoção de elementos. Além disso, por sua característica sequencial, dizemos que os elementos são indexados. Assim, podemos ler ou modificar os elementos da lista usando seu índice. O índice de um elemento é a sua posição dentro da lista, com a numeração começando em 0 (zero). A Figura 53 mostra uma representação de lista de números com suas respectivas posições. O elemento **10**, por exemplo, está na posição **4**.

40	35	61	89	10	52
0	1	2	3	4	5

Figura 53 – Representação de lista de números
Fonte: Elaborado pelo Autor.

Em Python, a declaração de listas é feita colocando os elementos da lista separados por vírgula dentro de colchetes, por exemplo, **minha_lista = [1, 2, 3, 4, 5]**. Também podemos criar listas vazias simplesmente abrindo e fechando colchetes, por exemplo,

lista_vazia = []. Além disso, as listas possuem diversas funções para facilitar sua manipulação. A Figura 54 apresenta algumas dessas funções.

Função	Funcionamento
l.append(x)	Adiciona o elemento **x** no final da lista **l**
l.insert(p, x)	Insere o elemento **x** na posição **p** da lista **l**
l.pop(p)	Remove e retorna o elemento da posição **p** de **l** (se **p** não for informado, a última posição é considerada)
l.clear()	Remove todos os elementos da lista **l**

Figura 54 – Algumas funções de listas
Fonte: Elaborado pelo Autor.

O código da Figura 55 mostra uma possível solução para o problema dos produtos com preço acima da média utilizando listas. Criamos uma lista vazia para guardar os preços dos produtos (**lista_precos**). Dentro do primeiro laço de repetição, os preços são adicionados no final da lista com a função **append()**. O Segundo laço de repetição percorre os preços da lista e mostra apenas aqueles acima da média.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os preços acima da média são:')
for preco in lista_precos:
    if preco > media:
        print(preco)
```

Figura 55 – Utilização de listas para resolver o problema dos preços acima da média
Fonte: Elaborado pelo Autor.

O código anterior mostra apenas os preços acima da média, mas não exibe os produtos com esses preços. Para resolver essa questão, no segundo laço, temos que percorrer as posições da lista usando o range. Essa solução é apresentada na Figura 56. Como estamos varrendo a lista pelos índices, usamos **lista_precos[cont]** para acessar o preço na posição **cont**. O código usa a função **range()** para gerar o intervalo de índices da lista. Uma alternativa é usar a função **enumerate()** que numera os elementos da lista e os retorna junto com seus índices. Essa solução alternativa é mostrada na Figura 57.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for cont in range(10):
    if lista_precos[cont] > media:
        print('Produto', cont+1)
        print('Preço: ', lista_precos[cont])
```

Figura 56 – Solução mostrando os produtos com preços acima da média
Fonte: Elaborado pelo Autor.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for cont, preco in enumerate(lista_precos):
    if preco > media:
        print('Produto', cont+1)
        print('Preço: ', preco)
```

Figura 57 – Solução mostrando os produtos com preços acima da média usando **enumerate()**
Fonte: Elaborado pelo Autor.

Em algumas situações, precisamos inicializar listas com certos valores ou com um determinado número de posições. Nesses casos, podemos fazer isso com a função **range()** ou com o operador *. A Figura 58 mostra alguns exemplos de inicialização de listas. A função **list()** converte o intervalo gerado pelo **range(10)** para o formato de lista.

```
# Lista inicializada com números de 0 a 9
lista = list(range(10))
print(lista)

# Lista de 10 posições com valores 0
lista = [0]*10
print(lista)
```

Figura 58 – Exemplos de inicialização de listas com **range()** e *
Fonte: Elaborado pelo Autor.

Outra maneira de inicializar listas é utilizando a compressão. A compressão consiste em escrever um código entre colchetes que gera uma lista de valores. Também podemos ler um texto e quebrá-lo em lista. Nesse caso, podemos quebrar o texto com a função

split() e usar a função **len()** para obter o tamanho da lista. A Figura 59 exemplifica a compressão de listas e a criação de listas a partir de texto.

```
# Lista inicializada com números de 0 a 9
lista = [n for n in range(9)]
print(lista)
# Leitura de string e com split
texto = input('Informe números (separados com espaços): ')
lista = [int(x) for x in texto.split()]
print(lista)
print(len(lista))
```

Figura 59 – Exemplo de compressão e lista a partir de texto
Fonte: Elaborado pelo Autor.

Dica do Professor: A função **len()** é uma função da biblioteca padrão da linguagem Python que retorna a quantidade de elementos de listas ou de outros tipos de dados como texto, por exemplo.

Além da seleção de um único elemento pelo seu índice, as listas permitem diversos outros tipos de seleções. A utilização de índices negativos faz a seleção dos elementos a partir do final da lista. O índice **-1** representa o último elemento, **-2** é o penúltimo elemento, e assim por diante. Também podemos selecionar sub-listas, ou seja, pedaços da lista. No caso das sub-listas usamos a notação de colchetes após a lista indicando a posição inicial e final da sub-lista. A Figura 60 mostra alguns exemplos de sub-listas a partir de uma lista **l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**.

Para verificarmos se um elemento existe dentro de uma lista temos que varrer todos os seus elementos. Uma maneira de fazer isso escrevendo de forma mais concisa é utilizar o operador **in**. Basicamente, a instrução **x in l**, retorna **True** se o elemento **x** existir na lista **l**. De forma análoga, podemos, a expressão **x not in l**, retorna **True**, se o elemento **x** não estiver na lista **l**.

Notação	Resultado	Explicação
l[:]	**[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**	Copia a lista
l[1:]	**[1, 2, 3, 4, 5, 6, 7, 8, 9]**	Todos os elementos a partir do segundo
l[:-1]	**[0, 1, 2, 3, 4, 5, 6, 7, 8]**	Todos os elementos até o penúltimo
l[3:7]	**[3, 4, 5, 6]**	Do quarto ao sétimo elemento

Figura 60 – Seleção de sub-listas de uma lista **l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**
Fonte: Elaborado pelo Autor.

As tuplas são estruturas usadas para agrupar uma quantidade fixa de elementos. A especificação de tuplas é feita escrevendo seus elementos separados por vírgula. Se a tupla tiver um único elemento, é preciso incluir uma vírgula após esse elemento. Normalmente, por uma questão de legibilidade, colocamos essa lista de elementos entre parênteses. As tuplas são recomendadas para agrupar quantidades fixas e pequenas de elementos.

Uma aplicação interessante para tuplas é o agrupamento de dia, mês e ano em uma data. A Figura 61 ilustra essa aplicação. Observe que as tuplas de data são criadas com ano, mês e dia, nessa ordem. Isso possibilita comparar as tuplas diretamente como foi feito no **if**. A comparação é feita considerando o primeiro elemento, depois o segundo e assim por diante.

```
print('Informe as datas')
print('Primeira data')
dia = int(input('Dia: '))
mes = int(input('Mês: '))
ano = int(input('Ano: '))
data1 = (ano, mes, dia)
print('Segunda data')
dia = int(input('Dia: '))
mes = int(input('Mês: '))
ano = int(input('Ano: '))
data2 = (ano, mes, dia)
recente = data1
if data2 > data1:
    recente = data2
print('Data mais recente:', recente)
```

Figura 61 – Utilização de tuplas para representar datas
Fonte: Elaborado pelo Autor.

Cada elemento de uma tupla possui um índice, começando pelo zero (DOWNEY, 2015). Assim, podemos usar a instrução **t[n]** para acessar o elemento da tupla **t** na posição **n-1**. Os índices dos elementos das tuplas representando datas podem vistos na Figura 62.

0	1	2
ano	mes	dia

Figura 62 – Índices dos elementos de uma tupla representando data
Fonte: Elaborado pelo Autor.

Outra peculiaridade das tuplas é a possibilidade de atribuir o valor dos elementos de uma tupla a um conjunto de variáveis em uma única instrução no formato **a, ..., x = t**. Tal instrução é valida desde que o número de variáveis seja igual ao número de elementos da tupla.

Outra aplicação interessante de tuplas é no problema dos preços acima da média visto anteriormente. Além do preço, seria interessante guardar o nome do produto para mostrar quando necessário. Podemos podemos fazer isso agrupando esses dados em uma tupla. A Figura 63 demonstra como podemos implementar o código. Dentro do laço **for**, pegamos o nome e o preço do produto, respectivamente. Em seguida, juntamos os dados em uma tupla e adicionamos essa tupla à lista de produtos. No segundo laço de repetição percorremos os produtos da lista e extraímos seu nome e preço. O **if** é usado para verificar se o preço do produto está acima da média.

```
soma = 0
lista_produtos = []
print('Informe o dados dos produtos')
for cont in range(10):
    print('\nProduto ' , cont+1)
    nome = input('Nome: ')
    preco = float(input('Preço: '))
    produto = (nome, preco)
    soma += preco
    lista_produtos.append(produto)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for produto in lista_produtos:
    nome, preco = produto
    if preco > media:
        print('Produto:', nome)
        print('Preço:', preco)
```

Figura 63 – Produtos acima da média usando tuplas
Fonte: Elaborado pelo Autor.

Os conjuntos são coleções não ordenadas de elementos (CORRÊA, 2020). Eles devem ser utilizados quando a existência de um elemento na coleção é mais importante do que a ordem do elemento ou do que a quantidade de vezes que o elemento aparece. Na verdade, os conjuntos em Python, assim como na matemática, não possuem elementos repetidos.

Podemos criar um conjunto com elementos separados por vírgula entre parênteses ou usar o construtor **set()** (TAGLIAFERRI, 2018). No caso do construtor, temos que passar uma coleção de elementos, como um alista, por exemplo. Conjuntos vazios devem ser criados com o construtor **set()** sem sem nenhum parâmetro. Para exemplificar as operações sobre conjuntos, vamos considerar os conjuntos **s1 = {1, 2, 3, 4}**, **s2 = {4, 5, 6}** e **s3 = {4, 5}**. A Figura 64 mostra algumas operações sobre esses conjuntos na linguagem Python e a notação matemática correspondente. Observe que as operações mostradas podem ser feitas com operadores ou funções.

Notação Matemática	Código Python		Resultado
	Operadores	Funções	
$s_1 \cap s_2$	s1 & s2	s1.intersect(s2)	{4}
$s_1 \cup s_2$	s1 | s2	s1.union(s3)	{1, 2, 3, 4, 5, 6}
$s_1 - s_2$	s1 - s2	s1.difference(s2)	{1, 2, 3}
$s_3 \subseteq s_2$	s3 <= s2	s3.issubset(s2)	True
$s_1 \supseteq s_3$	s1 >= s3	s1.issuperset(s3)	False

Figura 64 – Operações sobre conjuntos de dados
Fonte: Elaborado pelo Autor.

Além das operações entre conjuntos podemos verificar se um elemento existe no conjunto como operador **in**. Também podemos adicionar e remover elementos com as operações **add()** e **remove()**, respectivamente.

Um dicionário é um tipo especial de coleção que faz mapeamento ou associação de chave-valor (CORRÊA, 2020). Isso significa que, para cada chave do dicionário, existe um valor associado. A chave deve ser um tipo de dado imutável, mas o valor pode ser qualquer tipo de dado. Os tipos imutáveis são os tipos de dados básicos e outros dados que não podem sofre modificações como, por exemplo, tuplas compostas de dados imutáveis.

A maneira mais comum de criar dicionários com lista de chaves e valores (chave:valor) entre chaves. Um dicionário vazio pode ser criado com **{}** (abre e fecha chaves). Considerando um dicionário **d** e uma chave **c**, podemos consultar o valor para a chave **c** com o comando **d[c]**. Já a adição ou alteração do valor pode ser feita com a instrução **d[c] = v**, onde **v** o valor a ser atribuído à chave. A Figura 65 mostra um exemplo simples de código que cria e manipula dicionários.

```
dic_vazio = {}
dic_letras = {1:'A', 2:'B', 3:'C'}
dic_letras[4] = 'E'
dic_letras[4] = 'D'
print(dic_vazio, dic_letras)
for cont in range(1, 5):
    print(cont, ':', dic_letras[cont])
```

Figura 65 – Exemplo simples com dicionários
Fonte: Elaborado pelo Autor.

Além das manipulações usando chaves e colchetes, os dicionários possuem diversas funções que facilitam seu tratamento. A Figura 66 exibe algumas dessas funções e e seu funcionamento.

Função	Funcionamento
d.clear()	Remove todos os elementos do dicionário **d**
d.copy()	Retorna uma cópia de **d**
d.items()	Retorna as chaves-valores de **d** no formato de tuplas
d.keys()	Retorna uma lista com as chaves de **d**
d.popitem()	Retorna uma tupla (chave, valor) qualquer(se **d** estiver vazio, ocorre um erro)
d.update(d2)	Atualiza os elementos de **d** utilizando os elementos de **d2**
d.values()	Retorna uma lista com os valores de **d**

Figura 66 – Algumas funções de dicionários
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

1.6 Exercícios

Escreva os códigos em Python modularizados e com tratamento de exceções para resolver os problemas a seguir:

A) Desenvolver um algoritmo que construa uma lista de tuplas com nome e salário de funcionários. Em seguida, o código deve recalcular os salários considerando os seguintes aumentos:

- 20% para salários de até R$2.000,00;
- 15% para salários entre R$2.000,00 e R$5.000,00;
- 5% para salários maiores que R$5.000,00;

Por fim, o código deve exibir o total de aumento (total dos salários novos menos o total de salários antigos e os nomes dos funcionários com salários menores que R$2.000,00.

B) Construir um simulador de urna eletrônica. Inicialmente, o simulador deve permitir o cadastro de candidatos. O usuário pode cadastrar quantos candidatos desejar. O cadastro de um candidato envolve um número e seu nome. O número deve ser armazenado em formato textual, mas deve possuir exatamente dois dígitos numéricos. A função **isdigit()** do tipo textual pode ser usada para verificar se o texto é um número válido. Por fim, os candidatos cadastrados devem ser mantidos em um dicionário (número: nome).

Após o cadastro de candidatos, o simulador deve iniciar a votação. O simulador deve permitir uma quantidade indeterminada de votos. Para votar, o usuário deve informar o número do candidato. O sistema deve mostrar o nome do candidato para o usuário confirmar. Números inválidos devem ser computados como votos nulos. Já o texto vazio deve ser contabilizado como voto em branco. A sumarização dos votos deve ser feita usando um dicionário. Ao término da votação, o simulador deve mostrar o total e a porcentagem de votos de cada candidato, nulos e brancos.

1.7 Respostas dos exercícios

Escreva os códigos em Python modularizados e com tratamento de exceções para resolver os problemas a seguir:

A) Desenvolver um algoritmo que construa uma lista de tuplas com nome e salário de funcionários. Em seguida, o código deve recalcular os salários considerando os seguintes aumentos:

- 20% para salários de até R$2.000,00;
- 15% para salários entre R$2.000,00 e R$5.000,00;
- 5% para salários maiores que R$5.000,00;

Por fim, o código deve exibir o total de aumento (total dos salários novos menos o total de salários antigos e os nomes dos funcionários com salários menores que R$2.000,00.

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  def le_salario():
5      while True:
6          try:
7              return float(input('Salário: '))
8          except:
9              print('Valor inválido! Informe novamente')
10
11 def le_funcionario():
12     nome = input('Nome: ')
13     salario = le_salario()
14     return (nome, salario)
15
16 def aumenta_salarios(lista_funcionarios):
17     total = 0
18     for cont, funcionario in enumerate(lista_funcionarios):
19         nome, salario = funcionario
20         if salario <= 2000:
21             salario *= 1.2
22         elif salario <= 5000:
23             salario *=1.15
24         else:
25             salario *=1.05
26         total += salario
27         lista_funcionarios[cont] = (nome, salario)
28     return total
29
30 def principal():
31     lista_funcionarios = []
32     total_antigo = 0
33     while True:
34         funcionario = le_funcionario()
35         lista_funcionarios.append(funcionario)
36         total_antigo += funcionario[1]
```

```
37          resp = input('Continuar (S/N): ').lower().strip()
38          if resp != 's':
39              break
40      total_novo = aumenta_salarios(lista_funcionarios)
41      total_aumento = total_novo - total_antigo
42      print('Total de aumento:', total_aumento)
43      print('Funcionários com salário abaixo de R$2.000,00:')
44      for nome, salario in lista_funcionarios:
45          if salario < 2000:
46              print(nome)
47
48  if __name__ == '__main__':
49      principal()
```

B) Construir um simulador de urna eletrônica. Inicialmente, o simulador deve permitir o cadastro de candidatos. O usuário pode cadastrar quantos candidatos desejar. O cadastro de um candidato envolve um número e seu nome. O número deve ser armazenado em formato textual, mas deve possuir exatamente dois dígitos numéricos. A função **isdigit()** do tipo textual pode ser usada para verificar se o texto é um número válido. Por fim, os candidatos cadastrados devem ser mantidos em um dicionário (número: nome).

Após o cadastro de candidatos, o simulador deve iniciar a votação. O simulador deve permitir uma quantidade indeterminada de votos. Para votar, o usuário deve informar o número do candidato. O sistema deve mostrar o nome do candidato para o usuário confirmar. Números inválidos devem ser computados como votos nulos. Já o texto vazio deve ser contabilizado como voto em branco. A sumarização dos votos deve ser feita usando um dicionário. Ao término da votação, o simulador deve mostrar o total e a porcentagem de votos de cada candidato, nulos e brancos.

```
1   #!/usr/bin/python3
2   # -*- coding: utf-8 -*-
3
4   def lista_cand(dict_cand):
5       print('\n'*100)
6       print('Candidatos:')
7       for numero, nome in dict_cand.items():
8           print(numero, '-', nome)
9
10  def adiciona_cand(dict_cand):
11      lista_cand(dict_cand)
12      print('\nInforme os dados do novo candidato')
13      num = input('Número: ')
14      nome = input('Nome: ')
15      if len(num) != 2 or not num.isdigit() or num in dict_cand:
16          print('Cadastro inválido!')
17      else:
18        dict_cand[num] = nome
19
20  def pega_voto(dict_cand, dict_votos):
21      while True:
22          lista_cand(dict_cand)
23          print('\nInforme seu voto')
24          voto = input('Número do candidato: ').strip()
```

```
25          if voto == '':
26              voto = 'Branco'
27          elif voto in dict_cand:
28              voto = voto + ' - ' + dict_cand[voto]
29          else:
30              voto = 'Nulo'
31          print('Voto:', voto)
32          resp = input('Confirma (S/N): ').lower().strip()
33          if resp == 's':
34              soma_voto(dict_votos, voto)
35              break
36
37  def soma_voto(dict_voto, voto):
38      if voto in dict_voto:
39          dict_voto[voto] += 1
40      else:
41          dict_voto[voto] = 1
42
43  def principal():
44      dict_cand = {}
45      while True:
46          adiciona_cand(dict_cand)
47          resp = input('Continuar cadastros (S/N): ').lower().strip()
48          if resp == 'n':
49              break
50      dict_voto = {}
51      total = 0
52      while True:
53          pega_voto(dict_cand, dict_voto)
54          total += 1
55          resp = input('Encerrar votação (S/N): ').lower().strip()
56          if resp == 's':
57              break
58      print('Resultado da eleição:')
59      for voto in dict_voto:
60          porcent = round(dict_voto[voto] / total * 100, 2)
61          print(voto, ': ', dict_voto[voto],
62                ' (', porcent, '%', ')', sep='')
63
64  if __name__ == '__main__':
65      principal()
```

1.8 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista ao vídeo “Revisão da Primeira Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer os conceitos básicos de bancos chave-valor;
- Desenvolver e entender códigos usando bancos chave-valor.

2.1 Introdução

O Redis é um banco de dados não relacional que armazena um mapeamento de chaves para cinco tipos diferentes de valores. No Redis, não há tabelas e não há uma maneira definida de relacionar dados. O Redis permite resolver uma ampla gama de problemas e pode ser usado como banco de dados primário ou como banco de dados auxiliar em conjunto com outros sistemas (CARLSON, 2013).

Ao usar um banco de dados na memória como o Redis, é importante ter maneiras de persistir os dados em caso de falha no servidor. O Redis tem duas formas diferentes de persistência disponíveis para gravar dados na memória e em disco em um formato compacto. Embora o Redis tenha um bom desempenho, devido ao seu design em memória, há situações em que você pode precisar de mais performance. Nesses casos, o Redis possui também recursos de replicação para aumentar seu desempenho. Ao usar o Redis em vez de um banco de dados relacional, você pode evitar gravar dados temporários desnecessários, evitar a necessidade de verificar e excluir esses dados temporários e, por fim, melhorar o desempenho.

2.2 Instalação

No Linux, o Redis pode ser instalado com o comando **sudo apt install redis-server**. Após a instalação ser concluída, podemos testar o funcionamento com o comando **redis-cli**. Esse comando se conecta como servidor local do Redis em execução. Dentro do **redis-cli**, podemos usar o comando **ping** para testar se o servidor está funcionando. A resposta deve ser a mensagem **PONG**. Conforme mostrado na Figura 67. O comando **exit** pode ser usado para sair.

```
$ redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> exit
```

Figura 67 – Teste de **ping** no Redis
Fonte: Elaborado pelo Autor.

No Windows, a instalação deve ser feita por meio do WSL usando o mesmo comando. Porém, nesse caso o Redis não é iniciado automaticamente. Antes de utilizá-lo, é preciso abrir um terminal WSL e executar o comando **redis-server**. Esse comando inicia

a execução do Redis, que pode ser finalizado com as teclas **CTRL+C**. Você deve manter o terminar aberto até que decida finalizar o servidor do Redis.

2.3 Estruturas de dados do Redis

O Redis nos permite armazenar chaves que mapeiam para cinco diferentes tipos de estrutura de dados. A Figura 68 lista tais estruturas de dados, o tipo de conteúdo e operações suportadas. Cada uma das cinco estruturas diferentes possui alguns comandos compartilhados (**DEL**, **TYPE**, **RENAME** e outros) e outros específicos. As estruturas de **STRING**, **LIST, HASH** e **SET** costumam ser familiares para a maioria dos programadores. Sua implementação e semântica são semelhantes às mesmas estruturas construídas na maioria das linguagens de programação.

Estrutura de dados	Conteúdo	Possíveis operações
STRING	Uma *string* ou valor numéricos	Manipulação de *string*, incremento e decremento de valores numéricos
LIST	Listas de strings	Inserção ou retirada de itens de ambas as extremidades, corte com base em deslocamentos, leitura de itens individuais ou múltiplos, busca e remoção itens por valor
SET	Conjuntos não ordenados de strings únicas	Inserção, busca ou remoção de itens individuais, verificação de associação, interseção, união, diferença
HASH	Tabelas hashes com chaves para valores	Inserção, busca ou remoção itens individuais, busca do *hash*
ZSET	Mapeamentos de strings para valores numéricos, ordenado pelos valores numéricos	Inserção, busca ou remoção de valores individuais, busca de itens com base em intervalos de pontuação ou valor do membro

Figura 68 – Estruturas de dados suportadas pelo Redis
Fonte: Elaborado pelo Autor.

2.3.1 Trabalhando com STRINGs

No Redis, **STRINGs** as strings podem ser usadas para guardar valores textuais, inteiros e números de ponto flutuante (números com valores fracionários). Os valores numéricos podem ser incrementados ou decrementados (os inteiros podem ser convertidos em flutuantes se necessários). As operações básicas com **STRINGs** são as seguintes:

- **GET:** Retorna o valor armazenado para uma dada chave;
- **SET:** Armazena um valor para uma dada chave;
- **DEL:** Apaga o valor armazenado para uma chave.

A instrução **DEL** funciona para todos os tipos de estruturas de dados. A Figura 69 mostra alguns exemplos de operações com **STRINGs** usando o **redis-cli**. A instrução **set**

hello world solicita o armazenamento do valor **'world'** associado à chave **'hello',** recebemos um **'OK'** dizendo que chave-valor foi armazenada. Em seguida, o comando **get hello** busca o valor para a chave **'hello'** e recebemos a resposta que é **'world'**. A instrução **del hello** pede a deleção da chave **'hello'**. Recebemos a resposta **'(integer) 1'** indicando que um valor foi apagado. Por fim, buscamos novamente pela chave chave **'hello'** e recebemos a resposta **'(nil)'**, informando que a chave não foi encontrada.

```
$ redis-cli
127.0.0.1:6379> set hello world
OK
127.0.0.1:6379> get hello
"world"
127.0.0.1:6379> del hello
(integer) 1
127.0.0.1:6379> get hello
(nil)
127.0.0.1:6379>
```

Figura 69 – Exemplo básico com **STRING**
Fonte: Elaborado pelo Autor.

Além das operações básicas também podemos fazer as seguintes operações:

- **INCR:** Incrementa o valor armazenado na chave em 1;
- **DECR:** Decrementa o valor armazenado na chave em 1;
- **INCRBY:** Incrementa o valor armazenado na chave pelo valor inteiro fornecido;
- **DECRBY:** Decrementa o valor armazenado na chave pelo valor inteiro fornecido;
- **INCRBYFLOAT:** Incrementa o valor armazenado na chave pelo valor flutuante fornecido.

Na definição da **STRING**, sempre que possível, o redis converte o valor para inteiro ou flutuante. Se você tentar incrementar ou decrementar uma chave que não existe ou uma **STRING** vazia, o Redis trata tal chave como se tivesse o valor zero. Se o valor não for numérico, ocorre um erro.

Depois de vemos algumas instruções podemos também trabalhar com o Redis em um terminal Python. Nesse caso, será necessário instalar a biblioteca **redis**, necessária para conectar com o banco de dados do Redis. No Linux, isso pode ser feito com o comando **sudo apt install python3-redis**. No Windows, devemos usar um terminal Anaconda e informar o comando **pip install redis**.

Você pode abrir um terminal Python em qualquer terminal do Linux usando o comando **pyhton3**. A Figura 70 mostra alguns exemplos de comandos em um terminal Python. O primeiro passo é a criação da variável **conn** que representa a conexão como banco de dados do Redis. O parâmetro **decode_responses** é usado para fazer a decodificação automática dos dados. A instrução **conn.get('k')** solicita o valor da chave **'k'**, como ela não existe a instrução retorna **None** (valor nulo em Python). Em seguida, usamos o comando **conn.incr('k')** para aumentar o valor da chave **'k'**. Novamente, a chave ainda não existe, então o redis a cria como valor zero e depois a incrementa.

Depois da chave já criada, usamos as instruções **conn.incr('k', 15)** e **conn.decr('k', 5)** para aumentar o valor da chave em 15 e depois decrementar em 5. O terminal mostra os valores da chave após as alterações (16 e 11). O comando **conn.get('k')** retorna o valor da chave e o comando **conn.set('k', '13')** atribui o valor 13 à chave. Por último, usamos a instrução **conn.incr('k')** para incrementar o valor da chave em 1.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.get('k')
>>> conn.incr('k')
1
>>> conn.incr('k', 15)
16
>>> conn.decr('k', 5)
11
>>> conn.get('k')
'11'
>>> conn.set('k', '13')
True
>>> conn.incr('k')
14
```

Figura 70 – Exemplo de comandos com **STRING** em um terminal Python
Fonte: Elaborado pelo Autor.

O Redis suporta também comandos de manipulação de partes da **STRING** como os seguintes:

- **APPEND:** Concatena um valor fornecido na **STRING**;
- **GETRANGE:** Busca uma parte da **STRING**;
- **SETRANGE:** Modifica uma parte da **STRING**.

A Figura 71 mostra alguns exemplos de uso desses comandos.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.append('oi', 'ola ')
4
>>> conn.append('oi', 'mundo!')
10
>>> conn.getrange('oi', 3, 7)
'la mu'
>>> conn.setrange('oi', 0, 'O')
10
 >>> conn.setrange('oi', 4, 'M')
10
 >>> conn.get('oi')
'Ola Mundo!'
>>> conn.append('oi', ' Como vai voce?')
25
>>> conn.get('oi')
'Ola mundo! Como vai voce?'
```

Figura 71 – Exemplo de comandos para manipular partes de **STRING**
Fonte: Elaborado pelo Autor.

O primeiro comando, **conn.append('oi', 'ola ')**, concatena o texto **'ola '** ao conteúdo da chave **'oi'**. Como a chave ainda não existe, o Redis cria uma **STRING** vazia e concatena o texto a mesma. A segunda concatenação ocorre entre o texto já existente com o texto **'mundo !'**. A instrução **conn.getrange('oi', 3, 7)** retorna a parte da **STRING** da posição 3 até a posição 7. Lembrando que a numeração das posições começa em zero. Depois, usamos a função **setrange()** para para alterar as posições 0 e 4. Os três últimos comandos mostram o conteúdo da **STRING** e concatenam mais um texto.

2.3.2 Trabalhando com LISTs

As **LISTs** do Redis armazenam uma sequência ordenada de **STRINGs**. As operações que podem ser executadas em **LISTs** são similares ao que encontramos em quase todas as linguagens de programação. Basicamente, podemos trabalhar com as seguintes instruções:

- **LPUSH / RPUSH:** Empurra um valor para a extremidade esquerda ou direita da lista;
- **LRANGE:** Lista um intervalo de valores da lista;
- **LINDEX:** Busca um item em uma determinada posição na lista;
- **LPOP / RPOP:** Desempilha um valor da extremidade esquerda ou direita da lista e o retorna;
- **LTRIM:** Corta a lista.

A Figura 72 mostra alguns exemplos de operações com **LIST**.

```
127.0.0.1:6379> rpush L b
(integer) 1
127.0.0.1:6379> rpush L c d
(integer) 3
127.0.0.1:6379> lrange L 0 -1
1) "b"
2) "c"
3) "d"
127.0.0.1:6379> lpush L a
(integer) 4
127.0.0.1:6379> lrange L 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
127.0.0.1:6379> rpop L
"d"
127.0.0.1:6379> lindex L -1
"c"
127.0.0.1:6379> lpop L
"a"
127.0.0.1:6379> lrange L 0 -1
1) "b"
2) "c"
```

Figura 72 – Exemplo básico com **LIST**
Fonte: Elaborado pelo Autor.

A instrução **rpush L b** cria a lista **'L'** já com o item **'b'** adicionado a direita. Já a instrução **rpush L c d** adiciona a direita da lista **L** os itens **'c'** e **'d'**. O comando **lrange L 0 -1** é usado para listar os todos itens da lista. A primeira posição da lista é sempre **0** (zero) e a última pode ser referenciada como **-1** (menos um). A instrução **lpush L a** insere o item **'a'** na extremidade esquerda de **'L'**. Mostramos novamente a lista completa e, depois, usamos o comando **rpop L** para remover e recuperar o item **'d'** da extremidade direita da lista. Mostramos a lista mais uma vez e removemos o item **'a'** com a instrução **lpop L**. No final, a lista **L** contém apenas os itens **'b'** e **'c'**.

2.3.3 Trabalhando com SETs

Os **SETs,** assim como os **LISTs**, guardam vários strings, mas usam tabelas *hash* para manter todas as strings exclusivas (sem repetições). Como os **SETs** não são ordenados, não podemos empurrar e remover itens das extremidades como fizemos com **LISTs**. Em vez disso, adicionamos e removemos itens por valor com os comandos **SADD** e **SREM**. Também podemos descobrir se um item está no **SET** com **SISMEMBER**, ou buscar o conjunto inteiro com **SMEMBERS**. Uma observação importante é que o comando **SMEMBERS** pode ser lento para **SETs** grandes. A Figura 73 apresenta um exemplo com **SET**.

```
127.0.0.1:6379> sadd S a
(integer) 1
127.0.0.1:6379> sadd S b c
(integer) 2
127.0.0.1:6379> sadd S b
(integer) 0
127.0.0.1:6379> smembers S
1) "c"
2) "b"
3) "a"
127.0.0.1:6379> sismember S d
(integer) 0
127.0.0.1:6379> sismember S a
(integer) 1
127.0.0.1:6379> srem S c
(integer) 1
127.0.0.1:6379> smembers S
1) "b"
2) "a"
```

Figura 73 – Exemplo básico com **SET**
Fonte: Elaborado pelo Autor.

Começamos criando um conjunto **'S'** contendo o item **'a'**, usando o comando **sadd S a**. Em seguida, a instrução **sadd S b c** adiciona os itens **'b'** e **'c'** ao mesmo conjunto. Para o comando **sadd S b**, recebemos a resposta **'(integer) 0'** porque o item **'b'** já está no conjunto **'S'**. A instrução **smembers S** mostra todos os itens de **'S'**. Usamos **sismember S d** para testar se **'d'** está em **'S'** e recebemos a resposta negativa **'(integer) 0'**. No caso, do comando **sismember S a**, recebemos a resposta positiva porque o item **'a'** existe em **'S'**. Por fim, removemos o item **'c'** do conjunto e mostramos seus membros mais uma vez.

Além dos comandos anteriores, os SETs suportam as seguintes operações:

- **SCARD:** Retorna o número de itens no SET.
- **SRANDMEMBER:** Retorna itens aleatórios. Se for solicitada uma quantidade negativa, podem ocorrer repetições;
- **SPOP:** Remove e retorna um item aleatório;
- **SMOVE:** Se o item estiver na origem, remove o item da origem e adiciona-o ao destino, retornando o item foi movido.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.sadd('s1', 'a', 'b', 'c')
3
>>> conn.srem('s1', 'c', 'd')
1
>>> conn.srem('s1', 'c', 'd')
0
>>> conn.scard('s1')
2
>>> conn.smembers('s1')
{'a', 'b'}
>>> conn.smove('s1', 's2', 'a')
True
>>> conn.smove('s1', 's2', 'c')
False
>>> conn.smembers('s2')
{'a'}
```

Figura 74 – Exemplo de comandos para manipular partes de **SETs**
Fonte: Elaborado pelo Autor.

A Figura 74 mostra mais um exemplo usando esses comandos. O primeiro comando cria o conjunto **s1** com os itens **'a'**, **'b'** e **'c'**. A segunda instrução solicita a remoção dos itens **'c'** e **'d'**. Como o item **'d'** não existe no conjunto, apenas o item **'c'** é removido. A mesma instrução de remoção é usada novamente, mas nenhum item é removido. O comando **conn.scard('s1')**, mostra a quantidade de itens de **s1**. Em seguida, usamos a função **smove()** para mover os itens **'a'** e **'c'** do conjunto **s1** para os conjunto **s2**. O item **'c'** não é movido porque não existe mais em **s1**. Por fim, o comando **conn.smembers('s2')** mostra os itens do conjunto **s2**.

Usando os comandos já mostrados, podemos acompanhar eventos e itens exclusivos. Além disso, os **SETs** possuem instruções interessantes para operações de conjuntos. Algumas dessas instruções são as seguintes:

- **SDIFF:** Retorna os itens que estão no primeiro e não estão nos demais conjuntos (diferença de conjuntos);
- **SDIFFSTORE:** Parecido com o **SDIFF**, mas cria um novo conjunto com o resultado da operação;
- **SINTER:** Retorna os itens que estão em todos os conjuntos (interseção de conjuntos);
- **SINTERSTORE:** Semelhante ao **SINTER**, mas cria um novo conjunto como resultado da interseção;
- **SUNION:** Retorna os itens que estão em qualquer um dos conjuntos (união de conjuntos);
- **SUNIONSTORE:** Similar a operação **SUNION**, mas cria um novo conjunto como resultado da operação.

A Figura 75 mostra mais um exemplo com operações de conjuntos usado o Redis.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.sadd('set1', 'a', 'b', 'c', 'd')
4
>>> conn.sadd('set2', 'c', 'd', 'e', 'f')
4
>>> conn.sdiff('set1', 'set2')
{'a', 'b'}
>>> conn.sinter('set1', 'set2')
{'c', 'd'}
>>> conn.sunion('set1', 'set2')
{'a', 'b', 'c', 'd', 'e', 'f'}
```

Figura 75 – Exemplo de operações de conjuntos
Fonte: Elaborado pelo Autor.

2.3.4 Trabalhando com HASHes

Enquanto **LISTs** e **SETs** armazenam conjuntos de itens, os **HASHes** armazenam um mapeamento de chaves para valores. Os valores que podem ser armazenados em **HASHes** são os mesmos que podem ser armazenados como **STRINGs**. Podemos dizer que os **HASHes** são como miniaturas do Redis dentro do próprio Redis. Do ponto de vista de um banco de dados relacional, podemos considerar um **HASH** como um linha de uma tabela. Os principais comandos para lidar com **HASHes** são os seguintes:

- **HSET**: Armazena o valor na chave no *hash*;
- **HGET**: Busca o valor na chave de *hash* fornecida;
- **HMSET**: Armazena múltiplos pares chave-valor no *hash*;
- **HMGET**: Busca múltiplas chaves em um *hash*;
- **HGETALL**: Busca todo o *hash*;
- **HDEL**: Remove uma chave do *hash*, se existir;
- **HLEN**: Retorna o número de pares chave-valor em um *hash*.

A Figura 76 mostra um exemplo com **HASH**. A instrução **hset H a 1** cria o **HASH** com o mapeamento **'a: 1'**. O comando **hset H b 2 c 3** adiciona também os mapeamentos **'b:2'** e **'c:3'** ao **HASH**. Depois, mostramos o **HASH** completo com a instrução **hgetall H** e apagamos o mapeamento com a chave **'b'**. Por último, mostramos o **HASH** completo mais uma vez e obtemos o valor para a chave **'c'** de dentro do **HASH**.

```
127.0.0.1:6379> hset H a 1
(integer) 1
127.0.0.1:6379> hset H b 2 c 3
(integer) 2
127.0.0.1:6379> hgetall H
1) "a"
2) "1"
3) "b"
4) "2"
5) "c"
6) "3"
127.0.0.1:6379> hdel H b
(integer) 1
127.0.0.1:6379> hgetall H
1) "a"
2) "1"
3) "c"
4) "3"
127.0.0.1:6379> hget H c
"3"
```

Figura 76 – Exemplo básico com **HASH**
Fonte: Elaborado pelo Autor.

A Figura 77 mostra um exemplo de código em Python manipulando **HASHes**. A função **hmset()** demonstra como podemos adicionar vários itens ao *hash* com uma única instrução. A função **hmget()**, analogamente, mostra como buscar mais de uma chave no *hash* ao mesmo tempo. A função **hlen()** normalmente é usada para depurar *hashes* muito grandes. A função **hdel()** suporta vários argumentos e retorna **True** se algum campo for removido.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.hmset('h', {'k1':'v1', 'k2':'v2', 'k3':'v3'})
True
>>> conn.hmget('h', ['k2', 'k3'])
['v2', 'v3']
>>> conn.hlen('h')
3
>>> conn.hdel('h', 'k1', 'k3')
True
```

Figura 77 – Exemplo com **HASH** em Python
Fonte: Elaborado pelo Autor.

Para a manipulação das chaves e valores internos de um **HASH** podem ser usados os seguintes comandos:

- **HEXISTS**: Verifica se a chave fornecida existe no *hash*;
- **HKEYS**: Busca as chaves no *hash*;
- **HVALS**: Busca os valores no *hash*;
- **HINCRBY**: Incrementa o valor armazenado na chave dada pelo incremento inteiro;

- **HINCRBYFLOAT**: Incrementa o valor armazenado na chave dada pelo incremento flutuante.

Os comandos **HINCRBY** e **HINCRBYFLOAT** são semelhantes aos comandos **INCRBY** e **INCRBYFLOAT** usados com **STRINGs**. A XXXX mostra outro exemplo de HASH usando esses comandos.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.hmset('h2', {'a':'Ola', 'b':'A'*1000})
True
>>> conn.hkeys('h2')
['a', 'b']
>>> conn.hexists('h2', 'n')
False
>>> conn.hincrby('h2', 'n')
1
>>> conn.hexists('h2', 'n')
True
```

Figura 78 – Exemplo com **HASH** e incremento de chaves em Python
Fonte: Elaborado pelo Autor.

2.3.5 Trabalhando com ZSETs

Assim como os **HASHes**, os **ZSETs** também possuem um tipo de chave e valor. As chaves (chamadas de membros) são exclusivas e os valores (chamados de pontuação) são numéricos. Os **ZSETs** têm a propriedade única no Redis de poder ser acessado por membro (como um **HASH**), mas os itens também podem ser acessados pela ordem de classificação e valores das pontuações. Os principais comandos para lidar com **ZSETs** são os seguintes:

- **ZADD**: Adiciona o membro com a pontuação dada ao **ZSET**;
- **ZRANGE**: Busca os itens no **ZSET** de suas posições de forma ordenada;
- **ZRANGEBYSCORE**: Busca itens no **ZSET** com base em uma série de pontuações;
- **ZREM**: Remove o item do **ZSET**, se existir;
- **ZCARD**: Retorna o número de membros;
- **ZINCRBY**: Incrementa o membro;
- **ZCOUNT**: Retorna o número de membros com pontuações entre o mínimo e o máximo fornecidos;
- **ZRANK**: Retorna a posição do membro;
- **ZSCORE**: Retorna a pontuação do membro.

A Figura 79 apresenta um exemplo com **ZSET**. A instrução **zadd Z 10 a**, inicializa o **ZSET** com o mapeamento **'a:10'**. Em seguida, o comando **zadd Z 10 b**, adiciona o mapeamento **'b:10'**. A consulta **zrange Z 0 100** busca os membros com pontuação entre 0

e 100, retornado **'a'** e **'b'**. A instrução **zadd Z 11 a**, muda a pontuação do membro **'a'** para **'11'**. Por fim, são feitas mais algumas consultas pela pontuação dos membros.

```
127.0.0.1:6379> zadd Z 10 a
(integer) 1
127.0.0.1:6379> zadd Z 10 b
(integer) 1
127.0.0.1:6379> zrange Z 0 100
1) "a"
2) "b"
127.0.0.1:6379> zadd Z 11 a
(integer) 0
127.0.0.1:6379> zrangebyscore Z 10 10
1) "b"
127.0.0.1:6379> zrangebyscore Z 10 11
1) "b"
2) "a"
127.0.0.1:6379> zrangebyscore Z 11 11
1) "a"
```

Figura 79 – Exemplo básico com **ZSET**
Fonte: Elaborado pelo Autor.

Outro exemplo, dessa vez na linguagem Python, pode ser visto na Figura 80. A função **zadd()** demonstra como adicionar vários membros e pontuações com uma única instrução no **ZSET**. Observe que, no Python, a ordem dos membros e pontuações é o oposto do que é feito no **redis-cli**. A função **zcard()** mostra a quantidade de membros e a instrução **conn.zincrby('z', 3, 'c')**, incrementa a pontuação do membro **'c'** em três. Depois, usamos as funções **zscore()** e **zrank()** para obter a pontuação de **'b'** e posição de **'c'**, respectivamente.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.zadd('z', {'a': 3, 'b': 2, 'c': 1})
3
>>> conn.zcard('z')
3
>>> conn.zincrby('z', 3, 'c')
4.0
>>> conn.zscore('z', 'b')
2.0
>>> conn.zrank('z', 'c')
2
>>> conn.zcount('z', 0, 3)
2
>>> conn.zrem('z', 'b')
True
>>> conn.zrange('z', 0, -1, withscores=True)
[('a', 3.0), ('c', 4.0)]
```

Figura 80 – Exemplo em Python com **ZSET**
Fonte: Elaborado pelo Autor.

A instrução **conn.zcount('zset-key', 0, 3)** conta o número de itens com pontuações entre zero e três. A função **zem()** é usada para remover o membro 'b'. Por fim, a função

zrange() lista os membros da primeira até a última posição (**-1**) mostrando também suas pontuações (**withscores=True**).

Além dos comandos principais já mostrados, o Redis também possui comandos interessantes para operações como interseção e união de **ZSETs**. Podemos destacar os seguintes comandos:

- **ZREVRANK**: Retorna a posição do membro (considerando a ordenação inversa);
- **ZREVRANGE**: Busca os membros por pontuação (considerando a ordenação inversa);
- **ZREVRANGEBYSCORE**: Busca os membros entre considerando uma faixa de pontuação (ordenação inversa);
- **ZREMRANGEBYRANK**: Remove os membros considerando uma faixa de posições;
- **ZREMRANGEBYSCORE**: Remove os membros considerando uma faixa de pontuações;
- **ZINTERSTORE**: Realiza interseção entre **ZSETs**;
- **ZUNIONSTORE**: Realiza união entre **ZSETs**.

```
>>> import redis
>>> conn = redis.Redis(decode_responses=True)
>>> conn.zadd('z1', 'a', 1, 'b', 2, 'c', 3)
3
>>> conn.zadd('z2', 'b', 4, 'c', 1, 'd', 0)
3
>>> conn.zinterstore('zi', ['z1', 'z2'])
2
>>> conn.zrange('zi', 0, -1, withscores=True)
[('c', 4.0), ('b', 6.0)]
>>> conn.zunionstore('zu', ['z1', 'z2'], aggregate='min')
4
>>> conn.zrange('zu', 0, -1, withscores=True)
[('d', 0.0), ('a', 1.0), ('c', 1.0), ('b', 2.0)]
>>> conn.sadd('s1', 'a', 'd')
2
>>> conn.zunionstore('zu2', ['z1', 'z2', 's1'])
4
>>> conn.zrange('zu2', 0, -1, withscores=True)
[('d', 1.0), ('a', 2.0), ('c', 4.0), ('b', 6.0)]
```

Figura 81 – Exemplo em Python com operações de conjunto sobre **ZSETs**
Fonte: Elaborado pelo Autor.

A Figura 81 mostra um exemplo, em Python, usando os comandos explicados para operações de conjunto sobre **ZSETs**. As duas primeiras funções **zadd()** criam os **ZSETs z1** e **z2** com alguns membros e pontuações. A função **zinterstore()** faz a interseção de **z1** e **z2** armazenando o resultado no **ZSET zi**. Observe que o parâmetro **aggregate** não foi informado. Nesse caso, a interseção é feita com os membros e a pontuação é somada como mostrado na Figura 82. O resultado da operação são os membros **b** e **c** são comuns

em ambos conjuntos. A pontuação de **b** é 6.0 (2.0 + 4.0) e a pontuação de **c** é 4.0 (3.0 + 1.0). Esse resultado é mostrado com a função **zrange()**.

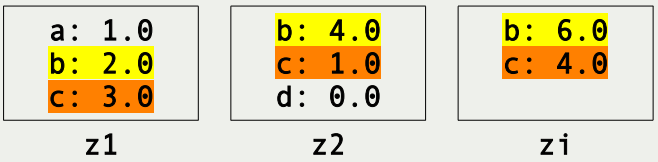

Figura 82 – Operação de interseção entre **ZSETs**
Fonte: Elaborado pelo Autor.

Depois da operação de interseção, fazemos uma operação de união com função **zunionstore()**. Dessa vez, atribuímos o valor **'min'** ao parâmetro **aggregate**. Isso faz com que ocorra a união entre os membros e a pontuação resultante seja a menor para cada membro. A Figura 83 mostra como ocorre essa operação.

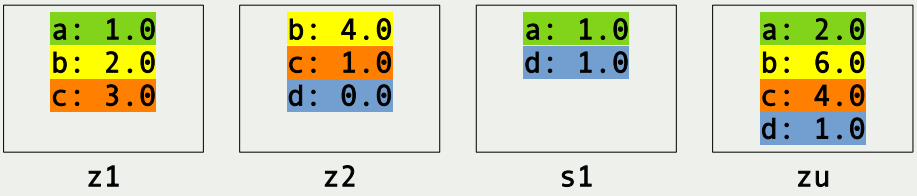

Figura 83 – Operação de interseção entre **ZSETs**
Fonte: Elaborado pelo Autor.

Por último, criamos um **SET** e demonstramos como podemos fazer a união entre **ZSETs** e **SETs**. Nesse caso, a pontuação dos elementos do **SET** é considerada um. O resultado dessa operação é mostrado na Figura 83.

2.4 Expirando chaves

Em algumas situações, os dados gravados no Redis podem não ser mais necessários. Podemos apagar tais dados explicitamente com o comando **DEL**, ou podemos fazer com que o item expire depois de um certo tempo. Quando dizemos que uma chave expirará significa que o Redis irá excluir automaticamente tal chave quando o tempo de expiração chegar. A expiração de chaves pode ser útil para lidar com a limpeza de dados armazenados em cache. Uma observação importante é que só podemos expirar chaves inteiras e não itens individuais. Os principais comandos para lidar com expiração de chaves são os seguintes:

- **PERSIST**: Remove a expiração de uma chave;
- **TTL**: Retorna a quantidade de tempo restante para a expiração;
- **EXPIRE**: Define a expiração de uma chave em um determinado número de segundos;
- **EXPIREAT**: Define o momento de expiração de uma chave (esse momento é representado como um *timestamp* Unix);
- **PTTL**: Retorna o número de milissegundos restantes para a expiração;

- **PEXPIRE**: Define expiração de uma chave em um determinado número de milissegundos;
- **PEXPIREAT**: Define o momento de expiração especificado em *timestamp* Unix de milissegundos.

A Figura 84 mostra um exemplo com comandos de expiração em Python.

```
>>> import redis
>>> import time
>>> conn = redis.Redis(decode_responses=True)
>>> conn.set('chave', 'valor')
True
>>> conn.get('chave')
'value'
>>> conn.expire('chave', 2)
True
>>> time.sleep(2)
>>> conn.get('chave')
>>> conn.set('chave', 'valor')
True
>>> conn.expire('chave', 100); conn.ttl('chave')
True
100
```

Figura 84 – Exemplo em Python com operações de conjunto sobre **ZSETs**
Fonte: Elaborado pelo Autor.

2.5 Exemplo de controle de curtidas

Uma característica comum á várias redes sociais é a possibilidade dos usuários "curtirem" postagens. Assim, tais postagens podem ser classificados de acordo com o número de votos que receberam em um determinado período. Nesta seção, vamos construir um protótipo de *back-end* baseado em Redis para controlar "curtidas" sobre links. Para simplificar, a popularidade dos links, vai levar em consideração apenas as curtidas ocorridas nas primeiras 24 horas.

Para começar, temos que definir quais informações são necessárias para um link e como vamos armazená-las. Para esse exemplo, vamos armazenar os links como HASHes contendo título, link, usuário dono da postagem, hora da postagem e número de curtidas. A Figura 85 mostra um exemplo desse HASH.

```
dono: 'usuario:bengal',
link: 'surely.openly.pet.osprey',
titulo: 'truly subtly glad hawk',
tempo: 1665321108.8354921,
curtidas: 17
```

Figura 85 – HASH de link
Fonte: Elaborado pelo Autor.

Como vamos implementar um protótipo em Python, será necessário instalar a biblioteca **petname**. No Linux, usamos o comando **sudo apt install python3-petname**. No terminal anaconda do Windows, podemos usar o comando **pip install petname**.

A biblioteca **petname** é usada apenas para gerar nomes aleatórios na criação do banco de dados. Inicialmente, vamos importar as bibliotecas definir a constante **EXPIRE** e conectar com o banco de dados, conforme mostrado na Figura 86. Além das bibliotecas **redis** e **petname**, usamos a biblioteca **time** para obter a hora atual do sistema e a biblioteca **pprint** para fazer impressões melhor formatadas na tela.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import redis
import time
import petname
from pprint import pprint

# Links expiram em um dia
EXPIRE = 24 * 60 * 60

# Conecta com banco de dados número 1
CONN = redis.Redis(db=1, decode_responses=True)
```

Figura 86 – Início do código de controle de curtidas
Fonte: Elaborado pelo Autor.

A constante **EXPIRE** define o tempo de 24 horas para contabilização das curtidas e a variável **CONN** faz a conexão com o banco de dados 1 do **Redis**. Se o parâmetro **db** não for informado, o banco de dados 0 é usado como padrão.

Após a conexão com o banco de dados, podemos criar uma função para incluir cadastrar novos links. Isso é implementado no código da Figura 87.

```
def novo_link(usuario, titulo, link):
    # Obtém id do link
    link_num = str(CONN.incr('link:num'))
    # Identificador (link:?)
    link_id = 'link:' + link_num
    # Cria conjunto de curtidas para o link
    curtiu = 'curtiu:' + link_num
    # Adiciona a curtida do usuário
    CONN.sadd(curtiu, usuario)
    # Define tempo de expiração das curtidas
    CONN.expire(curtiu, EXPIRE)
    # Pega tempo atual do sistema
    agora = time.time()
    CONN.hmset(link_id,
               {'titulo': titulo,
                'link': link,
                'dono': 'usuario:' + usuario,
                'tempo': agora,
                'curtidas': 0,
                })
    # Cria ZSET para contabilizar curtidas (inicialmente 0)
    CONN.zadd('curtidas', {link_id: 0})
    return link_id
```

Figura 87 – Função **novo_link()** do código de controle de curtidas
Fonte: Elaborado pelo Autor.

Usamos a STRING **link:num** para numerar automaticamente os links, com a função **incr()** incrementamos o número para cada novo link inserido. Os links são identificados com base em um número **n**. Assim, **'link:n'** é a chave para o HASH completo do link. De forma similar **'curtiu:n'** é a chave para o conjunto de usuários que curtiram o link.

A função **sadd()** é utilizada para criar o conjunto **SET** contendo os usuários que curtiram o link. Esse conjunto é necessário para evitar que um mesmo usuário curta várias vezes o mesmo link. Além disso, usamos a função **expire()** para definir o tempo de expiração do **SET**. Assim, depois de 24 horas, o link não poderá mais receber curtidas.

Usamos a função **time()** para pegar a hora atual do sistema e criamos o **HASH** com todos os dados do link usando a função **hmset()**. Por fim, inserimos no **ZSET curtidas**, o número de curtidas do link (inicialmente zero).

Após a inserção dos links, estamos prontas para criar uma função **curtir()** mostrada na Figura 88. A função recebe o usuário e o número do link a ser curtido. Em primeiro lugar, a função testa se a chave **'curtiu:n'** existe. Lembrando que as chaves de curtidas expiram depois de 24 horas. Em caso afirmativo, adicionamos o usuário ao **SET** dessa chave. A função **sadd()** retorna verdadeira apenas se o usuário for adicionado ao conjunto. Nesse caso, usamos a função **zincrby()** para aumentar o número de curtidas no **ZSET curtidas** para o elemento **'link:n'**. Além disso, incrementamos também o número de curtidas do link dentro do **HASH 'link:n'**.

```
def curtir(usuario, link_num):
    # Id do link
    link_id = 'link:' + str(link_num)
    # Verifica se as curtidas não expiraram
    if CONN.exists('curtiu:' + link_num):
        # Adiciona usuários à lista de curtidas do link
        # Se o mesmo ainda não tiver curtido
        if CONN.sadd('curtiu:' + link_num, usuario):
            # Incrementa nota do link no ZSET
            CONN.zincrby('curtidas', 1, link_id)
            # Incrementa número de curtidas do link
            CONN.hincrby(link_id, 'curtidas', 1)
```

Figura 88 – Função **curtir()** do código de controle de curtidas
Fonte: Elaborado pelo Autor.

```
def top10():
    # Seleciona os 10 links com maior nota em ordem decrescente
    link_ids = CONN.zrevrange('curtidas', 1, 10)
    link_list = []
    for link_id in link_ids:
        link_data = CONN.hgetall(link_id)
        link_data['id'] = link_id
        link_list.append(link_data)
    return link_list
```

Figura 89 – Função **top10()** do código de controle de curtidas
Fonte: Elaborado pelo Autor.

Após a inserção dos links e curtidas, podemos criar a função **top10()** para retornar os dez links com maior número de curtidas. Como usamos um **ZSET** para armazenar os

elementos com suas curtidas, podemos usar a função **zrevrange()** para pegar os dez mais curtidos. Para finalizar, como estamos trabalhando com um protótipo, vamos criar funções para gerar dados aleatórios e inserir do Redis como mostrado na Figura 90.

```
def gen_link():
    # Gera um link aleatório
    link_words = petname.random.randint(3, 5)
    link = petname.generate(link_words, '.')
    return link


def generate():
    # Gera um banco fictício no Redis
    num_links = 50
    num_curtidas = num_links * 10
    # Limpa o banco de dados atual
    CONN.flushdb()
    print('Gerando links')
    for _ in range(num_links):
        link = gen_link()
        usuario = petname.generate(1)
        titulo_size = petname.random.randint(3, 6)
        titulo = petname.generate(titulo_size, ' ')
        novo_link(usuario, titulo, link)
    print('Gerando curtidas')
    for _ in range(num_curtidas):
        usuario = petname.generate(1)
        link_num = str(petname.random.randint(1, num_links))
        curtir(usuario, link_num)


if __name__ == '__main__':
    generate()
    top_links = top10()
    for num, link in enumerate(top_links):
        print('-'*50)
        print(num+1)
        print('-'*50)
        pprint(link)
        print()
```

Figura 90 – Funções **gen_link()**, **generate()** e **main** do código de controle de curtidas
Fonte: Elaborado pelo Autor.

A função **gera_link()** gera um link aleatório, enquanto a função **generate()** insere 50 links do banco de dados e dá 500 curtidas aleatórios nos links inseridos. A função **main** apenas chama a geração de links e mostra os 10 links mais curtidos no final. Após a execução do código, você poderá acessar o redis com o comando **redis-cli -n 1** e pesquisar pelos dados inseridos.

A Figura 91 mostra alguns exemplos de comandos executados após a geração dos links. É claro que o conteúdo pode ser diferente porque estamos usando a biblioteca **petname** para gerar aleatoriamente os dados. O comando **KEYS *** mostra todas as chaves do banco (a figura omite da linha 2 a 99 por questão de espaço).

```
127.0.0.1:6379[1]> keys *
  1) "curtiu:45"
...
100) "link:17"
101) "curtiu:14"
102) "curtiu:39"
127.0.0.1:6379[1]> hgetall link:17
 1) "titulo"
 2) "kindly rapid fox"
 3) "link"
 4) "neatly.vast.hare"
 5) "dono"
 6) "usuario:impala"
 7) "tempo"
 8) "1665336735.554103"
 9) "curtidas"
10) "7"
127.0.0.1:6379[1]> smembers curtiu:14
1) "aphid"
2) "maggot"
3) "guppy"
4) "falcon"
5) "marten"
6) "joey"
```

Figura 91 – Consulta ao banco de dados de links após a geração dos dados aleatórios
Fonte: Elaborado pelo Autor.

2.6 Aplicações Web com Redis

O Redis possui particularidades específicas interessantes para alguns contextos de aplicações Web. Geralmente, um serviço Web responde pelo protocolo HTTP a solicitações feitas por navegadores. As etapas típicas pelas quais um servidor da Web passa para responder a uma solicitação são as seguintes:

1) O servidor recebe uma solicitação;

2) A solicitação é encaminhada para um manipulador;

3) O manipulador pode fazer solicitações para buscar dados de um banco de dados;

4) Com os dados recuperados, o manipulador retorna uma resposta, que é passada de volta ao cliente.

As solicitações da Web nesse tipo de situação são consideradas *sem estado*, pois os próprios servidores da Web não armazenam informações sobre solicitações anteriores, na tentativa de permitir a substituição fácil de servidores com falha. Muitas vezes, ao reduzirmos a carga do banco de dados tradicional, transferindo parte do processamento e armazenamento para o Redis, as páginas da Web podem ser carregadas mais rapidamente com menos recursos.

2.6.1 Login e cache de *cookies*

Sempre que fazemos login em serviços na Internet, como contas bancárias ou e-mail, esses serviços usam *cookies* para se lembrar de usuários anteriores. *Cookies* são pequenos pedaços de dados que os sites pedem aos nossos navegadores para armazenar e reenviar a cada solicitação a esse serviço.

Para a memorização de login, pode ser usado um *cookie* assinado ou um *cookie* de *token*. Os *cookies* assinados normalmente armazenam o nome do usuário, e outras informações como último login e outros dados que o serviço considerar útil. Junto com essas informações específicas do usuário, o *cookie* também inclui uma assinatura que permite ao servidor verificar se as informações que o navegador enviou não foram adulteradas (como trocar o nome de login, por exemplo).

Os *cookies* de *token* usam uma série de bytes aleatórios como dados no *cookie*. No servidor, o *token* é usado como chave para pesquisar o usuário que possui esse *token* consultando um banco de dados de algum tipo. Com o tempo, *tokens* antigos podem ser excluídos para dar espaço para novos *tokens*. A Figura 92 mostra as vantagens e desvantagens dos dois tipos de *cookie*.

	Vantagens	Desvantagem
Cookie assinado	- Todo informação necessária para verificação está no cookie. - Informações adicionais podem ser assinadas facilmente.	- O manuseio correto de assinaturas é difícil. - Fácil esquecer de assinar e verificar dados, levando a vulnerabilidades de segurança.
Cookie de token	- Mais fácil de adicionar informações. - Tamanho pequeno, para que clientes móveis e lentos possam enviar solicitações mais rapidamente.	- Mais informações para armazenar no servidor. - No uso de banco de dados relacional, o acesso aos *cookies* pode ser mais lento.

Figura 92 – Tipos de *cookie*, vantagens e desvantagens
Fonte: Elaborado pelo Autor.

Como exemplo vamos considerar uma loja na internet com milhões de acessos diários. Essa loja optou por usar um *cookie* de *token* para fazer referência a uma entrada em uma tabela de banco de dados relacional, que armazena informações de login do usuário.

Ao armazenar essas informações no banco de dados, a loja também pode armazenar informações como há quanto tempo o usuário navega ou quantos itens ele visualizou e, posteriormente, entregar propaganda direcionada ao usuário.

Os usuários, geralmente, passam por muitos itens diferentes antes de escolher algum para comprar. Registrar informações sobre todos os diferentes itens vistos pelo usuário, pode resultar em várias gravações de banco de dados. Atualmente, devido à carga relativamente grande ao longo do dia, a loja teve que configurar 10 servidores de banco de dados relacionais para lidar com a carga durante os horários de pico.

Uma alternativa para isso é substituir os bancos de dados relacionais pelo Redis para lidar com os cookies. Para começar, usaremos um HASH para armazenar nosso mapeamento de *tokens* de *cookie* de login para o usuário conectado. Para verificar o login, precisamos buscar o usuário com base no *token* e devolvê-lo, se estiver disponível. O código da Figura 93 mostra como isso pode ser feito.

```
def check_token(conn, token):
    return conn.hget('login:', token)
```

Figura 93 – Função **check_token()**
Fonte: Elaborado pelo Autor.

A parte mais complexa é a atualização do *cookie* que ocorre quando o usuário visita a página. Nesse momento, podemos registrar o *timestamp* (data e hora atual) atual e possíveis produtos visualizados pelo usuário, como mostrado no código da Figura 94. Além do **STRING 'login:token'**, o código atualiza o **ZSET 'recent:token'** usando o *timestamp* como pontuação. Para o **ZSET 'viewed:token'**, adicionamos novos produtos visualizados e mantemos apenas os 25 mais recentes usando a função **zremrangebyrank()**.

```
def atualiza_token(conn, token, usuario, produto=None):
    timestamp = time.time()
    conn.hset('login:', token, usuario)
    conn.zadd('recente:', token, timestamp)
    if produto:
        conn.zadd('produtos:' + token, produto, timestamp)
        conn.zremrangebyrank('produtos:' + token, 0, -26)
```

Figura 94 – Função **atualiza_token()**
Fonte: Elaborado pelo Autor.

Em um servidor moderno é possível registrar essas informações para pelo menos 20.000 visualizações de produtos a cada segundo. Isso representa uma considerável melhora de desempenho se comparado ao mesmo cenário usando um banco de dados relacional.

2.6.2 Carrinhos de compras

Outra funcionalidade comumente implementada por meio de *cookies* são os carrinhos de compras de lojas na Internet. A grande vantagem dessa funcionalidade é que não é necessário armazenar o carrinho de compras em um banco de dados. Por outro lado, é preciso analisar o *cookie* para garantir que os itens do carrinho são válidos. Além disso, os *cookies* são transmitidos a cada solicitação e grandes *cookies* podem causar atrasos na comunicação.

No Redis, podemos utilizar o mesmo identificador de cookie de sessão para armazenarmos o carrinho de compras. Basicamente, o carrinho de compras é um HASH que mapeia cada código de produto para a quantidade a ser comprada pelo cliente. Assim, a atualização desses dados é necessária apenas quando o usuário modifica o carrinho de compra. A Figura 95 exibe o código da função **adiciona_carrinho()** que adiciona produtos ao carrinho de compras.

```
def adiciona_carrinho(conn, usuario, produto, quantidade):
    if quantidade <= 0:
        conn.hrem('carrinho:' + usuario, produto)
    else:
        conn.hset('carrinho:' + usuario, produto, quantidade)
```

Figura 95 – Função **adiciona_carrinho()**
Fonte: Elaborado pelo Autor.

Os *cookies* de carrinho e de sessão ajudam a reduzir o tamanho da solicitação, além de permitir a realização de cálculos estatísticos sobre os visitantes. Esses dados estatísticos são muito usados por varejistas para analisar quais produtos um cliente viu e quais realmente comprou. Com tais informações a loja pode fazer propaganda direcionada para clientes. Quando cliente acessar um determinado produto, a loja pode mostrar outros produtos relacionados que foram comprados por outros clientes.

2.6.3 Cache de páginas

A grande maioria das páginas da Internet que acessamos são geradas dinamicamente por uma linguagem de programação executada em um servidor Web. As páginas Web modernas são geradas a partir de modelos de página com cabeçalhos, rodapés, menus, barras de ferramentas e até JavaScript.

Apesar de ser possível gerar conteúdo dinamicamente, a maioria das páginas de varejistas na Internet não muda muito regularmente. Alguns produtos são adicionados, produtos antigos são removidos e, às vezes, há algumas promoções. Mas, apenas algumas informações de conta, pedidos e carrinho de compras têm conteúdo que precisa ser gerado em cada solicitação.

Supondo que um site de uma loja tem 90% de páginas que não mudam de um dia para o outro, podemos usar o Redis para evitar a geração dinâmica desnecessária dessas páginas. Ao reduzir a quantidade de tempo gasto na geração de conteúdo, podemos reduzir a carga de trabalho dos servidores e disponibilizar as páginas mais rapidamente para os usuários.

Os *frameworks* de desenvolvimento web Python costumam oferecer capacidade de adicionar camadas que podem preprocessar solicitações à medida que são requisitadas. Assim, podemos criar a função **cache_requisicao()**, mostrada na Figura 96, em uma camada para ser chamada antes da geração da página.

```
def cache_requisicao(conn, requisicao, callback):
    if not pode_cache(conn, requisicao):
        return callback(requisicao)
    pagina = 'cache:' + hash_requisicao(requisicao)
    conteudo = conn.get(pagina)
    if not conteudo:
        conteudo = callback(requisicao)
        conn.setex(pagina, conteudo, 300)
    return conteudo
```

Figura 96 – Função **cache_requisicao()**
Fonte: Elaborado pelo Autor.

A função **cache_requisicao()** verifica se uma determinada página solicitada já está em cache e entrega a mesma sem a necessidade da geração. Se, por outro lado, a página não estiver em cache, a função faz a geração e já armazena em cache para solicitações futuras. A rotina **callback()** é usada para fazer a geração da página solicitada. O código elimina a necessidade de gerar dinamicamente as páginas visualizadas por 5 minutos. Dependendo da complexidade do conteúdo, essa alteração pode reduzir a latência de uma página com muitos dados em alguns milissegundos.

2.6.4 Produtos mais vistos

A navegação de clientes em páginas de lojas na Internet pode nos dar informações valiosas. Assim, as páginas mais visualizadas podem receber uma atenção maior e serem personalizadas para melhorar a experiência do usuário e afetar o comportamento de compra.

Na seção anterior implementamos o cache de páginas, mas não nos preocupamos com a quantidade de páginas em cache. Se a quantidade de páginas em cache for muito grande, podemos ter problemas com a quantidade de memória disponível no servidor do Redis. Dessa forma, pode ser mais interessante limitar o número máximo de páginas em cache. Em uma loja na Internet, a maioria das páginas são páginas de produtos podemos implementar essa limitação no número de produtos.

Na atualização de *token* implementada anteriormente mantivemos uma referência para cada item que foi visitado. Essas informações poderiam ser usadas para decidir quais páginas armazenar em cache, mas isso poderia levar um tempo considerável para ser calculado. Uma maneira mais eficiente é modificar a função **atualiza_token()** para registrar todos os produtos que são visualizados como mostrado na Figura 97.

```
def atualiza_token(conn, token, usuario, produto=None):
    timestamp = time.time()
    conn.hset('login:', token, usuario)
    conn.zadd('recente:', token, timestamp)
    if produto:
        conn.zadd('produtos:' + token, produto, timestamp)
        conn.zremrangebyrank('produtos:' + token, 0, -26)
        conn.zincrby('vistos:', item, 1)
```

Figura 97 – Função **atualiza_token()** modificada
Fonte: Elaborado pelo Autor.

A última linha adicionada à função atualiza o **ZSET** de produtos vistos pelos clientes. Com o tempo, alguns produtos serão mais visualizados e outros menos. Os produtos mais vistos tem a pontuação decrementada e tendem a ficar no início da lista. Logo, somente os produtos do início da lista precisam ficam em cache porque são mais visualizados.

Para manter nossa lista de principais páginas atualizada, precisamos cortar nossa lista de itens visualizados e, ao mesmo tempo, ajustar a pontuação para permitir que novos itens se tornem populares. O reescalonamento de um **ZSET** pode ser feito com a função **zinterstoe()**, que nos permite combinar um ou mais **ZSETs** e multiplicar cada pontuação por um determinado número. A Figura 98 apresenta a função **reescalona()** usada para atualizar o **ZSET** de produtos mais vistos.

```
def reecalona(conn):
    while True:
        conn.zremrangebyrank('vistos:', 20000, -1)
        conn.zinterstore('vistos:', {'vistos:': .5})
        time.sleep(300)
```

Figura 98 – Função **atualiza_token()** modificada
Fonte: Elaborado pelo Autor.

A função **reescalona()** possui um laço infinito para atualização do **ZSET** de produtos vistos a cada cinco minutos. A função **zremrangebyrank()** mantém apenas os 20.000 produtos mais visto no **ZSET** e a função **zinterstore()** redimensiona todas as contagens em 50%. Agora podemos modificar também a função **cache_requisicao()** para fazer o cache apenas das páginas dos 10.000 produtos mais visualizados, como mostrado na Figura 99.

```
def cache_requisicao(conn, requisicao):
    produto = extract_produto(requisicao)
    if not produto or is_dynamic(requisicao):
        return False
    rank = conn.zrank('viewed:', produto)
    return rank is not None and rank < 10000
```

Figura 99 – Função **cache_requisicao()** modificada
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

2.7 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Jogo de bingo:

- Defina variáveis textuais para serem usadas como nome de chaves no Redis. É interessante termos chaves para guardarmos a lista de jogadores, o conjunto de vencedores do jogo, o conjunto de números do jogo, a lista de números já sorteados no jogo, os conjuntos com os números de cada jogador e os acertos de cada jogador;

- Crie a função **cria_jogador()** para criar os jogadores. A função deve receber o nome e a lista de números com todos os números do jogo. Use a função **shuffle()** da biblioteca **random** para embaralhar a lista de números. Por fim, crie um conjunto no Redis com 25% dos números da lista para serem os números do jogador durante o jogo;

- Implemente a função **marca()** para marcar um número de um jogador. A função deve verificar se o jogador possui esse número e adicioná-lo no conjunto de números acertados pelo jogador;

- Escreva a função **faltantes()** para retornar os números que faltam para o jogador acertar. A função pode usar a operação de diferença de conjuntos entre o conjunto de números do jogador e os números que ele já acertou;

- Faça a função **sorteia()** para sortear os número do jogo. Você pode usar a função **srandmember()** para retornar um número aleatório do conjunto de números do jogo. O número deve ser removido do conjunto de números e adicionado à lista de números sorteados. É interessante usar uma lista de números sorteados para mostrar a ordem na qual eles foram sorteados. A função **sorteia()** deve também marcar o número sorteado para cada jogador. Os jogadores que acertarem todos os seus números devem ser adicionados ao conjunto de vencedores;

- Crie a função **imprime()** para mostrar o estado do jogo. Essa função deve mostrar a lista de números já sorteados. E uma lista de números faltantes de cada jogador. É interessante ordenar essa lista com a função **sort()** para que os números faltantes fiquem sempre ordenados na tela;

- Implemente a função **bingo()** para controlar o jogo. A função deve gerar uma lista com todos os números e usá-la para chamar a função **cria_jogador()** para inicializar os números de cada jogador. Essa mesma lista deve ser usada para criar um conjunto com todos os números do jogo. Feito isso, a função deve iniciar um laço de repetição que de ser executado até que hajam vencedores no jogo. Dentro do laço devemos chamar a função **imprime()**, solicitar que o usuário pressione uma tecla e chamar a função **sorteia()**.

Após o laço, mostramos o estado final do jogo chamando a função **imprime()** e mostramos a lista de vencedores do jogo.

- Por fim, escreva a função **principal()** que deve limpar o banco de dados usando a função **flushdb()**. É importante conectar em um banco de dados que não tenha outras informações para não perder nenhum dado. A limpeza é necessária porque a cada jogo devemos criar novamente os jogadores com seus números. Em seguida, a função deve solicitar o nome dos jogadores e adicioná-los a lista de jogadores. Por fim, deve ser chamada a função **bingo()** para iniciar o jogo.

B) Jogo da forca:

- O jogo deve ter a possibilidade de cadastrar palavras e dicas, além do jogo propriamente dito. Para isso vamos usar o Redis para guardar o conjunto de palavras e um HASH com a dica de cada palavra;

- Comece criando a função **lista_palavras()**, que deve listar todas as palavras já cadastradas e suas respectivas dicas. É interessante mostrar as palavras em ordem alfabética;

- Implemente a função **palavra_valida()** para verificar se uma palavra é válida. Para simplificar, vamos considerar que a palavra é válida somente se seus caracteres são letras de A a Z (sem acentos ou cedilha);

- Faça a função **incluir()** para solicitar a palavra e a dica ao usuário e cadastrar no banco de dados. Antes de cadastrar, chame a função **palavra_valida()** para testar se a palavra é válida;

- Escreva a função **excluir()** para excluir do banco de dados uma palavra informada pelo usuário. Antes de excluir, é interessante verificar se a palavra existe. O código deve excluir a palavra e sua dica;

- Cria a função **cadastro()** para exibir as opções ao usuário (incluir, excluir ou sair) e chamar as funções apropriadas;

- Implemente a função **tem_letra()** para verificar se uma letra exite na palavra. Em caso afirmativo, mostre essa letra da representação da palavra na tela. Essa representação deve, inicialmente, possuir um traço para cada letra. A medida que o usuário for acertando as letras, o jogo de substituir os traços pela letras acertadas;

- Faça também a função **eh_letra_valida()** para testar se uma letra é válida. Uma letra é válida se está entre as letras A e Z e não foi digitada no jogo ainda;

- Escreva a função **mostrar()** para exibir o estado atual do jogo na tela. A função deve mostrar a representação da palavra com traços e letras já acertadas, a dica da palavra, as letras já digitadas e o número de erros;

- Crie a função **forca()** para controlar o jogo. A função deve receber uma palavra sorteada e sua dica e, inicialmente, inicializar a representação da palavra na tela com um traço para cada letra. Código deve inicializar também o número de erros com zero e as letras já digitadas como uma *string* vazia. Depois disso, a função deve iniciar um laço de repetição que deve ser executado até o jogador acertar a palavra ou atingir cinco erros. Dentro do

laço, a função deve pegar uma letra digitada pelo usuário, testar se a mesma é válida e adicioná-la às letras digitadas. Em seguida, a função deve verificar se a letra existe na palavra e contabilizar o erro, se for o caso. O laço de repetição deve também mostrar o estado do jogo a cada iteração;

- Por fim, implemente a função **principal()** para mostrar um menu de opções ao usuário (jogar, cadastrar palavra ou sair). Se o usuário escolher jogar, a função deve sortear a palavra, obter sua dica e chamar a função **forca()** para iniciar o jogo.

2.8 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Jogo de bingo

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from random import shuffle
import redis

# Nomes de chaves
JOGADORES = 'jogadores'
VENCEDORES = 'vencedores'
NUMEROS = 'numeros'
SORTEADOS = 'sorteados'
ACERTOS = 'acertos'
# Total de números do bingo
TOTAL_NUM = 80
# Total de números do jogador
JOGADOR_NUM = int(TOTAL_NUM * 0.25)
# Conecta com banco de dados número 2
CONN = redis.Redis(db=2, decode_responses=True)

def cria_jogador(jogador, lista_numeros):
    # Chave para os números do jogador
    jogador_numeros = jogador + ':' + NUMEROS
    # Embaralha lista de números
    shuffle(lista_numeros)
    # Cria o conjunto de números do jogador
    CONN.sadd(jogador_numeros, *lista_numeros[:JOGADOR_NUM])

def marca(jogador, numero):
    # Chave para os números do jogador
    jogador_numeros = jogador + ':' + NUMEROS
    # Verifica se o jogador possui o número sorteado
    if CONN.sismember(jogador_numeros, numero):
        # Chave para os números que o jogador acertou
        jogador_acertos = jogador + ':' + ACERTOS
        # Adiciona o número sorteado aos números do jogador
        CONN.sadd(jogador_acertos, numero)

def faltantes(jogador):
    # Chave para os números do jogador
    jogador_numeros = jogador + ':' + NUMEROS
    # Chave para os números acertados pelo jogador
    jogador_acertos = jogador + ':' + ACERTOS
    # Retorna os números faltantes (ainda não acertados)
    return CONN.sdiff(jogador_numeros, jogador_acertos)

def sorteia():
    # Sorteia um número
    numero = CONN.srandmember(NUMEROS)
```

```
49      # Remove o número sorteado da lista de números
50      CONN.srem(NUMEROS, numero)
51      # Adiciona aos números sorteados (mantendo a ordem do sorteio)
52      CONN.rpush(SORTEADOS, numero)
53      # Percorre os jogadores
54      for jogador in CONN.lrange(JOGADORES, 0, -1):
55          # Marca o número sorteado para o jogador
56          marca(jogador, numero)
57          # Obtém os números faltantes para o jogador
58          jogador_faltantes = faltantes(jogador)
59          # Verifica se o jogador já acertou todos os números
60          if len(jogador_faltantes) == 0:
61              # Adiciona o jogador ao conjunto de vencedores
62              CONN.sadd(VENCEDORES, jogador)
63
64  def imprime():
65      # Cinquenta linhas em bracno para mostrar novo resultado
66      print('\n'*50)
67      print('BINGO\n')
68      # Lista de números sorteados
69      lista_sorteados = [str(n) for n in
70                         CONN.lrange(SORTEADOS, 0, -1)]
71      print('Sorteados:', ' '.join(lista_sorteados), '\n')
72      # Percorre jogadores
73      for jogador in CONN.lrange(JOGADORES, 0, -1):
74          # Números faltantes de cada jogador
75          faltantes_jogador = list(faltantes(jogador))
76          faltantes_jogador.sort()
77          faltantes_str = [str(n) for n in faltantes_jogador]
78          print(jogador + ':', ' '.join(faltantes_str))
79
80  def bingo():
81      # Lista com todos os números
82      lista_numeros = list(range(1, TOTAL_NUM+1))
83      # Inicializa jogadores
84      for jogador in CONN.lrange(JOGADORES, 0, -1):
85          cria_jogador(jogador, lista_numeros)
86      # Conjunto com todos os números
87      CONN.sadd(NUMEROS, *lista_numeros)
88      # Enquanto não houver vencedores
89      while CONN.scard(VENCEDORES) == 0:
90          # Mostra estado do jogo
91          imprime()
92          # Sorteia um número
93          input('Pressione ENTER para sortear um número.')
94          sorteia()
95      # Mostra o estado final do jogo
96      imprime()
97      # Mostra a lista de vencedores
98      print('\nVencedor(res):')
99      for jogador in CONN.smembers(VENCEDORES):
100         print(jogador)
101
102 def principal():
103     print('Iniciando o jogo')
```

```
104     # Limpa p banco de dados
105     CONN.flushdb()
106     # Pega o nome dos jogadores
107     print('Informe os nomes dos jogadores (vazio para parar)')
108     while True:
109         mensagem = 'Jogador ' + str(CONN.llen(JOGADORES)+1) + ': '
110         jogador = input(mensagem)
111         if jogador.strip() == '':
112             break
113         CONN.rpush(JOGADORES, jogador.strip())
114     # Testa se ao menos dois jogadores foram incluídos
115     if CONN.llen(JOGADORES) > 2:
116         # Inicia o bingo
117         bingo()
118
119 if __name__ == '__main__':
120     principal()
```

B) Jogo da forca

```
 1  #!/usr/bin/env python3
 2  # -*- coding: utf-8 -*-
 3
 4  import redis
 5
 6  # Chave para conjunto de palavras
 7  PALAVRAS = 'palavras'
 8  # Chave para hash de dicas
 9  DICAS = 'dicas'
10  # Conecta com banco de dados número 3
11  CONN = redis.Redis(db=3, decode_responses=True)
12
13  def lista_palavras():
14      print('-'*50)
15      print('Lista de palavras')
16      print('-'*50)
17      # Verifica se existem palavras cadastradas
18      if CONN.scard(PALAVRAS) == 0:
19          print('Nenhuma palavra cadastrada!')
20      else:
21          # Lista de palavras a partir do conjunto de palavras
22          lista_palavras = list(CONN.smembers(PALAVRAS))
23          # Ordena as palavras
24          lista_palavras.sort()
25          # Lista as palavras
26          for palavra in lista_palavras:
27              # Pega a dica da palavra
28              dica = CONN.hget(DICAS, palavra)
29              # Mostra a palavra e sua dica
30              print(palavra, ' (', dica, ')', sep='')
31      print('-'*50)
32
33  def palavra_valida(palavra):
34      # Para cada letra da palavra
35      for letra in palavra:
```

```
36          # Verifica se a letra está entre A e Z
37          if letra < 'A' or letra > 'Z':
38              # Se não estiver, retorna False
39              return False
40      # Retorna True, se a palavra for válida
41      return True
42
43  def incluir():
44      print('Inclusão de palavra')
45      # Recebe a palavra e a dica do usuário
46      palavra = input('Palavra: ').strip().upper()
47      dica = input('Dica: ')
48      # Verifica se a palavra é valida
49      if not palavra_valida(palavra):
50          print('Palavra inválida!')
51      # Verifica se a palavra já existe no dicionário
52      elif CONN.sismember(PALAVRAS, palavra):
53          print('Palavra já cadastrada!')
54      else:
55          CONN.sadd(PALAVRAS, palavra)
56          CONN.hset(DICAS, palavra, dica)
57
58  def excluir():
59      # Recebe a palavra do usuário
60      palavra = input('Palavra a ser excluída: ')
61      palavra = palavra.strip().upper()
62      # Verifica se a palavra não existe
63      if not CONN.sismember(PALAVRAS, palavra):
64          print('Palavra não encontrada')
65      else:
66          # Remove a palavra e a dica
67          CONN.srem(PALAVRAS, palavra)
68          CONN.hdel(DICAS, palavra)
69
70  def cadastro():
71      while True:
72          lista_palavras()
73          print('Informe a opção desejada:')
74          resp = input('(I)ncluir (E)xcluir (S)air ')
75          resp = resp.lower().strip()
76          if resp == 's':
77              break
78          elif resp == 'i':
79              incluir()
80          elif resp == 'e':
81              excluir()
82
83  def tem_letra(palavra, letra, palavra_tela):
84      # Supões que não tem a letra
85      encontrada = False
86      # Percorre os caracteres da palavra
87      for cont, carac in enumerate(palavra):
88          # Verifica se a letra é igual ao caractere atual
89          if carac == letra:
90              # Letra encontrada
```

```
 91                 encontrada = True
 92                 # Mostra a letra na representação da tela
 93                 palavra_tela[cont] = letra
 94         return encontrada
 95
 96     def eh_letra_valida(letra, digitadas):
 97         # A letra deve ser um único caracter
 98         if len(letra) != 1:
 99             return False
100         # Dever ser entre A e Z e não ter sido digitada antes
101         if letra < 'A' or letra > 'Z' or letra in digitadas:
102             return False
103         return True
104
105     def mostrar(palavra_tela, dica, digitadas, erros):
106         # 100 linhas em branco para o efeito de limpar tela
107         print('\n'*100)
108         # Representação de tela da palavra
109         print(''.join(palavra_tela))
110         if len(dica) > 0:
111             print('\n\nDICA: ' + dica)
112         # Letras digitadas e erros
113         print('Letras digitadas:', digitadas)
114         print('Erros:', erros)
115
116     def forca(palavra, dica):
117         # Palavra na tela (nenhuma letra descoberta ainda)
118         palavra_tela = ['_'] * len(palavra)
119         # Erros do jogador
120         erros = 0
121         # Letras já digitadas
122         digitadas = ''
123         # Laço que repete até o Forca terminar
124         while True:
125             # Mostra o estado do Forca
126             mostrar(palavra_tela, dica, digitadas, erros)
127             # Pega uma letra
128             letra = input('Informe uma letra: ').upper().strip()
129             # Se a letra não for válida, pega novamente
130             if not eh_letra_valida(letra, digitadas):
131                 continue
132             # Adiciona letra às digitadas
133             digitadas += letra
134             # Verifica se a letra não existe na palavra
135             if not tem_letra(palavra, letra, palavra_tela):
136                 # Contabiliza o erro
137                 erros += 1
138                 # Testa se já tem 5 erros
139                 if erros == 5:
140                     # O jogador perde e o Forca termina
141                     mostrar(palavra_tela, dica, digitadas, erros)
142                     print('Você perdeu!')
143                     break
144             # Verifica se a palavra foi compeltada
145             if '_' not in palavra_tela:
```

```
146             # O jogador vernce e o Forca termina
147             mostrar(palavra_tela, dica, digitadas, erros)
148             print('Você acertou!')
149             break
150
151 def principal():
152     # Menu principal
153     while True:
154         print('\n'*100)
155         print('-'*50)
156         print('Jogo da forca')
157         print('-'*50)
158         print('(J)ogar')
159         print('(C)adastro de palavras')
160         print('(S)air')
161         resp = input('Informe a opção desejada: ').strip().lower()
162         if resp == 'j':
163             # Não é possível jogar com menos de 10 palavras
164             if CONN.scard(PALAVRAS) < 10:
165                 print('Poucas palavras cadastradas!')
166                 input()
167             else:
168                 # Sorteia um apalavra do conjunto
169                 palavra = CONN.srandmember(PALAVRAS)
170                 # Pega a dica da palavra
171                 dica = CONN.hget(DICAS, palavra)
172                 # Incia o jogo
173                 forca(palavra, dica)
174                 # Espere ENTER ao terminar o jogo
175                 input()
176         elif resp == 'c':
177             cadastro()
178         elif resp == 's':
179             break
180
181 if __name__ == '__main__':
182     principal()
```

2.9 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista ao vídeo “Revisão da Segunda Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer os conceitos básicos de bancos de documentos;
- Desenvolver e entender códigos usando bancos de documentos.

3.1 Introdução

Os bancos de dados de documentos são muito utilizados em sistemas Web porque os documentos, em geral, usam o mesmo formato de dados desses sistemas. Os documentos são flexíveis e semiestruturados, permitindo alterações conforme a necessidade do sistema. Alguns exemplos de aplicações são catálogos, perfis de usuários e sistemas de postagens como blogs.

O MongoDB é um sistema de banco de dados de documentos disponibilizado na forma de software livre que oferece recursos de alta disponibilidade e suporte a transações ACID a partir da versão 4.0. No MongoDB, os documentos são estruturas de dados compostas de chaves (ou campos) e valores. Os valores das chaves podem incluir valores simples ou mesmo outros documentos. Os documentos, por sua vez, são organizados em coleções (GOALKICKER.COM, 2018).

As principais vantagens do MongoDB são sua simplicidade de uso, capacidade para lidar com altas cargas de trabalho e alta disponibilidade. Por outro lado, os relacionamentos entre dados precisam ser tratados via aplicação, pois o MongoDB não possui suporte a chaves estrangeiras.

3.2 Instalação

No Linux, o MongoDB pode ser instalado com o comando **sudo apt install mongodb**. No Ubuntu 20.04, por padrão, é instalada a versão 3.6.8. É obter versões mais recentes no site oficial do projeto (https://www.mongodb.com/try/download/community), mas a versão 3.6.8 é suficiente para desenvolver os exemplos que veremos.

As instalações para Windows também podem ser obtidas no site do projeto. Recomendamos a utilização da versão 3.6.23 por ser a versão mais próxima da versão do Linux. Após a instalação ser concluída, podemos testar o funcionamento com o comando **mongo** que abre o cliente de linha de comando Mongo Shell. Esse comando se conecta como servidor do MongoDB em execução, conforme mostrado na Figura 100. O comando **exit** pode ser usado para sair.

```
$ mongo
MongoDB shell version v3.6.8
connecting to: mongodb://127.0.0.1:27017
Implicit session: session { "id" : UUID("63e74ba5-6998-42ac-9b2b-...") }
MongoDB server version: 3.6.8
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        http://docs.mongodb.org/
Questions? Try the support group
        http://groups.google.com/group/mongodb-user
Server has startup warnings:
2022-10-28T07:15:00.968-0300 I STORAGE  [initandlisten]
2022-10-28T07:15:01.549-0300 I CONTROL  [initandlisten]
> exit
bye
```

Figura 100 – Execução do comando **mongo** para conexão com o MongoDB
Fonte: Elaborado pelo Autor.

3.3 Estruturas de dados no MongoDB

No que diz respeito à estrutura dos dados, o MongoDB tem duas principais diferenças em relação a bandos de dados relacionais. Primeiro, as entradas de dados não possuem, necessariamente, o mesmo esquema (mesmos atributos ou colunas). Além disso, é possível ter entradas aninhadas dentro de outras entradas. Contudo, podemos fazer uma analogia entre os elementos de um banco de dados orientado a documento e um banco de dados relacional, como mostrado na Figura 101. Assim como nas tabelas de bancos de dados relacionais, as coleções do MongoDB podem ter índices para melhorar o desempenho de pesquisas.

Relacional	Orientado a documento
Tabela	Coleção
Linha de tabela	Documento de coleção
Coluna de tabela	Propriedade de documento

Figura 101 – Analogia entre estruturas de bancos de dados relacionais e orientados a documentos
Fonte: Elaborado pelo Autor.

Os documentos do MongoDB obedecem ao formato JavaScript Object Notation (JSON). O JSON é um formato de dados muito utilizando, principalmente, na integração de sistemas. O formato foi criado em no ano 2000, com o objetivo de ser um compacto, aberto e com utilização de texto legível. A API do MongoDB é baseada na linguagem JavaScript. Assim, ao utilizar o cliente padrão **mongo**, podemos usar códigos em JavaScript e até declarar variáveis.

Na prática, os dados no formato JSON são muito parecidos com o resultado de um **print()** de dicionários e listas. A Figura 102 exibe um exemplo de dados no formato JSON. Observe que, para cada chave, existe um valor. Mais precisamente, o MongoDB utiliza o formato Binary JSON (BSON) que é semelhando ao JSON, mas suporta mais tipos de dados. Os principais tipos de dados suportados não os seguintes:

1) **double**: valores numéricos com casas decimais;

2) **string**: valores textuais;

3) **object**: objetos JavaScript;

4) **array**: listas;

5) **binData**: dados binários (como imagens);

6) **undefined**: indefinido (não é mais usado na prática);

7) **objectId**: identificador de objetos;

8) **bool**: valores lógicos (true / false);

9) **date**: datas;

10) **null**: Valores nulos.

```
{
    "nome": "José",
    "sobrenome": "Silva",
    "nascimento": "1984-04-12",
    "emails": [
        "josesilva@exemplo.com",
        "jsilva@exemplo.com"
    ],
    "endereco": {
        "cidade": "Bambuí",
        "uf": "MG"
    }
}
```

Figura 102 – Exemplo de dados no formato JSON
Fonte: Elaborado pelo Autor.

3.4 Trabalhando com o cliente mongo

Já usamos o cliente de linha de comando Mongo Shell para testar a conexão com o MongoDB após a instalação, mas apenas entramos e saímos como comando **exit**. Caso não seja especificado, o comando **mongo** se conecta ao banco de dados *test*. Além disso, a autenticação não vem habilitada por padrão, ou seja, o sistema não pede usuário e senha. Em sistemas reais é necessário configurar a autenticação para não termos problemas de segurança.

O mongo conta diversos comandos para manipular e consultar os dados. Vamos começar falando dos seguintes:

- **help**: Exibe a ajuda para os principais comandos do Mongo Shell;
- **show dbs**: Lista todos os bancos de dados existentes;
- **db**: Mostra o nome do banco de dados atual;
- **show collections**: Lista todas as coleções de um banco de dados;
- **use <banco>**: Conecta ao banco de dados informado.

O comando **help** é muito útil para tirar dúvidas sobre comandos. Além disso, ele está presente nos seguintes níveis:

- **db.help()**: Lista os comandos aplicáveis a bancos de dados;
- **db.<coleção>.help()**: Lista os comandos aplicados a coleções;
- **db.<coleção>.<comando>().help()**: Mostra informações sobre um comando.

O MongoDB não cria efetivamente um banco de dados até que alguma informação seja gravada nele. Isso ocorre quando criamos uma coleção ou documento. A criação de novos documentos pode ser feita como comando **db.<coleção>.insertOne(doc)**, onde **doc** pode ser um dicionário de dados. Para listar todos os documentos de uma coleção pode ser usado o comando **db.<coleção>.find()**. A Figura 103 mostra um exemplo com alguns dos comandos explicados até o momento.

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> use agenda
switched to db agenda
> db.contatos.insertOne(
... {
...     "nome": "José",
...     "sobrenome": "Silva",
...     "nascimento": "1984-04-12",
...     "emails": ["josesilva@exemplo.com","jsilva@exemplo.com"],
...     "endereco": {
...         "cidade": "Bambuí",
...         "uf": "MG"
...     }
... })
WriteResult({ "nInserted" : 1 })
> db.contatos.find().pretty()
{
        "_id" : ObjectId("635d548e0171b3fe3946daa6"),
        "nome" : "José",
        "sobrenome" : "Silva",
        "nascimento" : "1984-04-12",
        "emails" : [
                "josesilva@exemplo.com",
                "jsilva@exemplo.com"
        ],
        "endereco" : {
                "cidade" : "Bambuí",
                "uf" : "MG"
        }
}
> show dbs
admin   0.000GB
agenda  0.000GB
config  0.000GB
local   0.000GB
```

Figura 103 – Execução do comando **mongo** para conexão com o MongoDB
Fonte: Elaborado pelo Autor.

Primeiro, mostramos o bancos de dados existentes como comando show dbs. Podemos ver que existem apenas os bancos **admin**, **config** e **local**. Esses bancos criados pelo MongoDB para controlar informações de sistemas. O ideal é criamos outros bancos de dados e não alterarmos tais informações. O comando **use agenda** muda para o banco **agenda**. Em seguida, por meio do comando **db.contatos.insertOne()** criamos a coleção **contatos** contendo um documento (as aspas nos nomes das chaves podem ser omitidas).

O comando **find()** é usado para listar os documentos de uma coleção. Em nosso exemplo, aplicamos o comando **pretty()** sobre o comando **find()** para mostrar os dados de forma apresentável na tela. Sem o **pretty()** todas as informações do documento seriam impressas uma após a outra. Podemos observar que o documento exibido possui a chave **_id** que não especificamos na criação. O MongoDB cria inclui tal chave para todos os documentos. A chave **_id** funciona como uma chave primária, ou seja, cada documento de uma coleção deve possuir um valor único para essa chave.

No final, usamos o comando **show dbs** novamente. Nesse momento, podemos ver que o banco **agenda** já foi criado. Em nosso exemplo, criamos uma coleção ao criarmos o primeiro documento. Se for necessário, podemos criar uma coleção vazia como comando **db.createCollection("<coleção>")**. Já a exclusão de uma coleção pode ser feita com o comando **db.<coleção>.drop()**. Para excluir um banco de dados, devemos acessá-lo com o comando use e depois usar o comando **db.dropDatabase()**.

Dica do Professor: Tome cuidado com os comandos de exclusão. Use apenas se tiver certeza que deseja excluir e certifique-se de que está no banco de dados correto.

3.5 Nomes de bancos, coleções e chaves

No MongoDB, nomes de bancos, de coleções e de chaves possuem algumas restrições em relação a alguns caracteres. Os nomes de bancos devem obedecer às seguintes restrições:

- Não possuir os caracteres **/\. "$*<>:|?**;
- Não possuir caracteres nulos ou ser vazio;
- Não ter mais de 64 caracteres.

Os nomes de coleções não podem conter o caractere **$**, não podem ser vazios e não podem começar com o prefixo **system.** (reservado para o sistema). Os nomes de chaves não podem conter pontos, não podem ser vazios e não pode começar com **$**.

3.6 Manipulando documentos

As operações básicas para manipulação de documentos podem ser feitas com as seguinte funções:

- **insertOne()**: Insere novos documentos em uma coleção;

- **updateOne()**: Atualiza documentos em uma coleção;
- **deleteOne()**: Remove documentos de uma coleção;
- **find()**: Procura por documentos em uma coleção.

3.6.1 Inserindo documentos

No caso da inserção, podemos criar um objeto e, depois, inseri-lo com a instrução **insertOne()**. Outra função de pose de usada para inserir documentos é a **insertMany()**. Essa função recebe uma lista de documentos a serem inseridos na coleção. A na Figura 104 demostra a inserção de mais alguns documentos da coleção **contatos**. Observe que inserimos documentos com chaves diferentes para mostrar a flexibilidade de esquema do MongoDB.

```
> joao = {
...     nome: "João",
...     sobrenome: "Silva",
...     nascimento: "1988-05-11",
...     emails: [
...         "joaosilva@exemplo.com"
...     ],
... }
{
        "nome" : "João",
        "sobrenome" : "Silva",
        "nascimento" : "1988-05-11",
        "emails" : [
                "joaosilva@exemplo.com"
        ]
}
> db.contatos.insertOne(joao)
WriteResult({ "nInserted" : 1 })
> db.contatos.insertMany([
... {nome: "Maria", sobrenome: "Silva", emails: ["mariasilva@exemplo.com"]},
... {nome: "Ana", sobrenome: "Lima", nascimento: "1990-02-15"}
... ])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("635d6459a72ae2c3ee405a49"),
                ObjectId("635d6459a72ae2c3ee405a4a")
        ]
}
> db.contatos.find()
{ "_id" : ObjectId("635d548e0171b3fe3946daa6"), "nome" : "José", "sobrenome" :
"Silva", "nascimento" : "1984-04-12", "emails" : [ "josesilva@exemplo.com",
"jsilva@exemplo.com" ], "endereco" : { "cidade" : "Bambuí", "uf" : "MG" } }
{ "_id" : ObjectId("635d5a15a72ae2c3ee405a48"), "nome" : "João", "sobrenome" :
"Silva", "nascimento" : "1988-05-11", "emails" : [ "joaosilva@exemplo.com" ] }
{ "_id" : ObjectId("635d6459a72ae2c3ee405a49"), "nome" : "Maria", "sobrenome" :
"Silva", "emails" : [ "mariasilva@exemplo.com" ] }
{ "_id" : ObjectId("635d6459a72ae2c3ee405a4a"), "nome" : "Ana", "sobrenome" :
"Lima", "nascimento" : "1990-02-15" }
```

Figura 104 – Exemplo de inserção de documento como objeto
Fonte: Elaborado pelo Autor.

3.6.2 Consultando documentos

Até o momento, usamos a função **find()** sem passar nenhum parâmetro. Quando chamamos a função dessa forma, todos os documentos da coleção são retornados. Contudo, podemos incluir como parâmetro um dicionário representando um filtro para ser usado na busca dos documentos. Assim, somente os documentos que atenderem ao filtro serão retornados.

O tipo de filtro mais simples que pode ser usado é um dicionário do tipo chave-valor. Além do filtro, a função **find()** pode receber um segundo parâmetro para especificar quais chaves devem ser retornadas para os documentos encontrados. Esse parâmetro é chamado de *projeção*. Basicamente, a projeção deve associar **1** às chaves a serem exibidas e **0** às chaves a serem suprimidas. A chave **_id** sempre é exibida por padrão. A Figura 105 mostra alguns exemplos de consultas sobre os documentos já inseridos na coleção **agenda**. Se desejarmos procurar considerando documentos aninhados, podemos incluir uma condição do tipo **{"chave.subchave": valor}**.

```
> db.contatos.find({nome:"José"})
{ "_id" : ObjectId("635d548e0171b3fe3946daa6"), "nome" : "José",
"sobrenome" : "Silva", "nascimento" : "1984-04-12", "emails" :
[ "josesilva@exemplo.com", "jsilva@exemplo.com" ], "endereco" :
{ "cidade" : "Bambuí", "uf" : "MG" } }
> db.contatos.find({nome:"José"}, {nome:1, sobrenome:1})
{ "_id" : ObjectId("635d548e0171b3fe3946daa6"), "nome" : "José",
"sobrenome" : "Silva" }
> db.contatos.find({nome:"José"}, {_id:0, nome:1, sobrenome:1})
{ "nome" : "José", "sobrenome" : "Silva" }
> db.contatos.find({sobrenome:"Silva"}, {_id:0, nome:1, sobrenome:1,
nascimento:1})
{ "nome" : "José", "sobrenome" : "Silva", "nascimento" : "1984-04-12" }
{ "nome" : "João", "sobrenome" : "Silva", "nascimento" : "1988-05-11" }
{ "nome" : "Maria", "sobrenome" : "Silva" }
> db.contatos.find({"endereco.cidade":"Bambuí"}, {nome:1, sobrenome:1,
nascimento:1})
{ "_id" : ObjectId("635d548e0171b3fe3946daa6"), "nome" : "José",
"sobrenome" : "Silva", "nascimento" : "1984-04-12" }
```

Figura 105 – Exemplo de consultas com a instrução **find()**
Fonte: Elaborado pelo Autor.

A função **find()** permite também limitar o número de documentos com o método **limit()** a serem retornados e pular os **n** primeiros documentos usando o método **skip(n)**. Por exemplo, se quisermos mostrar apenas o terceiro e o quarto documento da coleção contatos, podemos usar a instrução **db.contatos.find().skip(2).limit(2)**. Também podemos ordenar o resultado com o método sort(). Se quisermos ordenar os contatos pelo sobrenome, podemos usar comando **db.contatos.find().sort({sobrenome:1})**, por exemplo.

As condições da consulta podem ser incrementadas como uso de operadores de consulta. As principais categorias desses operadores são as seguintes:

- Operadores de comparação (Figura 106);

- Operadores lógicos (Figura 107);
- Operadores de elementos (Figura 108);
- Operadores de avaliação (Figura 109);
- Operadores de *array* (Figura 110).

Operador	Descrição
$eq	=
$gt	>
$gte	>=
$in	∈ (pertence ao conjunto)
$lt	<
$lte	<=
$ne	≠
$nin	∉ (não pertence ao conjunto)

Figura 106 – Operadores de comparação em condições de buscas
Fonte: Elaborado pelo Autor.

Operador	Descrição
$and	AND (E lógico)
$not	NOT (NÃO lógico)
$or	OR (OU lógico)

Figura 107 – Operadores de lógicos em condições de buscas
Fonte: Elaborado pelo Autor.

Operador	Descrição
$exists	Testa se o documento possui uma chave
$type	Testa se uma chave é de um determinado tipo

Figura 108 – Operadores de elementos em condições de buscas
Fonte: Elaborado pelo Autor.

Operador	Descrição
$expr	Avaliar expressões com outros operadores
$regex	Avalia expressões regulares
$where	Avaliar os documentos com códigos em JavaScript

Figura 109 – Operadores de avaliação em condições de buscas
Fonte: Elaborado pelo Autor.

Operador	Descrição
$all	Verdadeiro se o array possui todos os elementos especificados na condição
$elemMatch	Seleciona documentos com elementos de *array* que atendam às condições do operador
$size	Seleciona documentos com o *array* no tamanho especificado

Figura 110 – Operadores de *array* em condições de buscas
Fonte: Elaborado pelo Autor.

```
> db.contatos.find({nascimento: {$gte: "1990"}})
{ "_id" : ObjectId("635d6459a72ae2c3ee405a4a"), "nome" : "Ana",
"sobrenome" : "Lima", "nascimento" : "1990-02-16" }
> db.contatos.insertOne({nome:"Joaquim", sobrenome:"Ribeiro", endereco:
{cidade:"Medeiros", uf:"MG"}})
WriteResult({ "nInserted" : 1 })
> db.contatos.find({"endereco.cidade":{$in: ["Bambuí", "Medeiros"]}})
{ "_id" : ObjectId("635d548e0171b3fe3946daa6"), "nome" : "José",
"sobrenome" : "Silva", "nascimento" : "1984-04-12", "emails" :
[ "josesilva@exemplo.com", "jsilva@exemplo.com" ], "endereco" :
{ "cidade" : "Bambuí", "uf" : "MG" } }
{ "_id" : ObjectId("635d9353bd71205caa332be3"), "nome" : "Joaquim",
"sobrenome" : "Ribeiro", "endereco" : { "cidade" : "Medeiros", "uf" :
"MG" } }
> db.contatos.find({"sobrenome":{$ne : "Silva"}})
{ "_id" : ObjectId("635d6459a72ae2c3ee405a4a"), "nome" : "Ana",
"sobrenome" : "Lima", "nascimento" : "1990-02-16" }
{ "_id" : ObjectId("635d9353bd71205caa332be3"), "nome" : "Joaquim",
"sobrenome" : "Ribeiro", "endereco" : { "cidade" : "Medeiros", "uf" :
"MG" } }
> db.contatos.find({sobrenome:{$nin: ["Silva", "Ribeiro"]}})
{ "_id" : ObjectId("635d6459a72ae2c3ee405a4a"), "nome" : "Ana",
"sobrenome" : "Lima", "nascimento" : "1990-02-16" }
> db.contatos.find({endereco:{$exists:true}})
{ "_id" : ObjectId("635d548e0171b3fe3946daa6"), "nome" : "José",
"sobrenome" : "Silva", "nascimento" : "1984-04-12", "emails" :
[ "josesilva@exemplo.com", "jsilva@exemplo.com" ], "endereco" :
{ "cidade" : "Bambuí", "uf" : "MG" } }
{ "_id" : ObjectId("635d9353bd71205caa332be3"), "nome" : "Joaquim",
"sobrenome" : "Ribeiro", "endereco" : { "cidade" : "Medeiros", "uf" :
"MG" } }
> db.contatos.find({$and : [{endereco:{$exists:true}}, {emails:
{$size:2}}] })
{ "_id" : ObjectId("635d548e0171b3fe3946daa6"), "nome" : "José",
"sobrenome" : "Silva", "nascimento" : "1984-04-12", "emails" :
[ "josesilva@exemplo.com", "jsilva@exemplo.com" ], "endereco" :
{ "cidade" : "Bambuí", "uf" : "MG" } }
```

Figura 111 – Exemplo de consultas com operadores de comparação e de elementos
Fonte: Elaborado pelo Autor.

O operador **$eq** não é muito utilizado porque é mais fácil escrever condições no formato **{chave: valor}**. A Figura 111 mostra alguns exemplos usando operadores de consulta. As explicações para as consultas são as seguintes:

- **find({nascimento: {$gte: "1990"}}):** Busca os contatos com data de nascimento a partir de 1990;
- **find({"endereco.cidade":{$in: ["Bambuí", "Medeiros"]}}):** Busca contatos com endereço nas cidades de Bambuí ou Medeiros;
- **find({"sobrenome":{$ne : "Silva"}}):** Busca por contatos com sobrenome diferente de Silva;
- **find({sobrenome:{$nin: ["Silva", "Ribeiro"]}}):** Busca contatos com sobrenome diferente de Silva e de Ribeiro;
- **find({endereco:{$exists:true}}):** Busca por contatos que tenha endereço cadastrado;
- **find({$and : [{endereco:{$exists:true}}, {emails:{$size:2}}]}):** Busca contatos com endereço e dois e-mails cadastrados.

3.6.3 Alterando documentos

No MongoDB, podemos atualizar um documento completo ou de forma parcial. A chave**_id** é a única que não pode ser sobrescrita. A função **updateOne()** altera apenas o primeiro documento encontrado. Contudo, podemos também alterar diversos documentos que satisfaçam uma dada condição usando a função **updateMany()**. Existe também a função **replaceOne()** que substitui o documento inteiro.

```
> db.contatos.find({nome:"Ana"})
{ "_id" : ObjectId("635d6459a72ae2c3ee405a4a"), "nome" : "Ana",
"sobrenome" : "Lima", "nascimento" : "1990-02-16" }>
db.contatos.replaceOne({nome:"Ana"}, {nascimento:"1990-02-16"})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.contatos.find({nome:"Ana"})
> db.contatos.find({nascimento:"1990-02-16"})
{ "_id" : ObjectId("635d6459a72ae2c3ee405a4a"), "nascimento" : "1990-02-
16" }
> db.contatos.replaceOne ({nascimento:"1990-02-16"}, {nome:"Ana",
sobrenome:"Lima", nascimento:"1990-02-16"})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.contatos.find({nascimento:"1990-02-16"})
{ "_id" : ObjectId("635d6459a72ae2c3ee405a4a"), "nome" : "Ana",
"sobrenome" : "Lima", "nascimento" : "1990-02-16" }
```

Figura 112 – Exemplo de modificação com substituição de documento
Fonte: Elaborado pelo Autor.

A Figura 112 mostra uma exemplo de modificação com substituição de documento. Na primeira chamada, de **find()** mostramos o documento original. Em seguida, usamos a função **replaceOne()**. Ocorre a substituição do documento e somente a chave **nascimento** é gravada. Com isso, após a modificação, o documento com nome igual a "Ana" não é

mais encontrado. Para restaurar os dados originais, fazemos outra modificação com substituição do documento, dessa vez, informando todas as chaves.

Uma modificação parcial pode ser feita com o uso de um objeto associado ao documento. Nesse caso, precisamos fazer uma busca, associar o documento ao objeto, alterar o objeto e gravar a modificando usando o objeto alterado. Outra maneira, mais intuitiva inclusive, é usar operadores de atualização. As principais categorias de operadores de atualização são operadores de atualização de chaves e os operadores de atualização de *arrays*.

Os principais operadores de atualização de chave são os seguintes:

- **$inc**: Incrementa o valor de uma chave com outro valor;
- **$min**: atualiza a chave se o valor especificado for menor do que o valor existente;
- **$max**: atualiza a chave se o valor especificado for maior do que o valor existente;
- **$mul**: Multiplica o valor da chave por outro valor;
- **$rename**: Renomeia uma chave;
- **$set**: Modifica ou cria uma nova chave;
- **$unset**: Remove uma chave.

A Figura 113 apresenta alguns exemplos de utilização dos operadores de atualização de chaves.

```
> db.contatos.find({nome: "Joaquim"})
{ "nome" : "Joaquim", "sobrenome" : "Ribeiro", "endereco" : { "cidade" :
"Medeiros", "uf" : "MG" } }
> db.contatos.updateMany({nome: "Joaquim"}, {$set: {"endereco.cidade":
"Belo Horizonte"}, {idade: 40}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.contatos.find({nome: "Joaquim"})
{ "nome" : "Joaquim", "sobrenome" : "Ribeiro", "endereco" : { "cidade" :
"Belo Horizonte", "uf" : "MG" }, "idade" : 40 }
> db.contatos.updateMany ({nome: "Joaquim"}, {$inc: {idade: 1}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.contatos.find({nome: "Joaquim"})
{ "nome" : "Joaquim", "sobrenome" : "Ribeiro", "endereco" : { "cidade" :
"Belo Horizonte", "uf" : "MG" }, "idade" : 41 }
> db.contatos.updateMany ({nome: "Joaquim"}, {$unset: {idade: ""}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.contatos.find({nome: "Joaquim"})
{ "nome" : "Joaquim", "sobrenome" : "Ribeiro", "endereco" : { "cidade" :
"Belo Horizonte", "uf" : "MG" } }
> db.contatos.updateMany ({endereco: {$exists:false}}, {$set: {endereco:
{cidade:"Medeiros", uf:"MG"}}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
> db.contatos.find({})
{ "nome" : "José", "sobrenome" : "Silva", "nascimento" : "1984-04-12",
"emails" : [ "josesilva@exemplo.com", "jsilva@exemplo.com" ], "endereco"
: { "cidade" : "Bambuí", "uf" : "MG" } }
{ "nome" : "João", "sobrenome" : "Silva", "nascimento" : "1988-05-11",
"emails" : [ "joaosilva@exemplo.com" ], "endereco" : { "cidade" :
"Medeiros", "uf" : "MG" } }
{ "nome" : "Maria", "sobrenome" : "Silva", "emails" :
[ "mariasilva@exemplo.com" ], "endereco" : { "cidade" : "Medeiros", "uf"
: "MG" } }
{ "nome" : "Ana", "sobrenome" : "Lima", "nascimento" : "1990-02-16",
"endereco" : { "cidade" : "Medeiros", "uf" : "MG" } }
{ "nome" : "Joaquim", "sobrenome" : "Ribeiro", "endereco" : { "cidade" :
"Belo Horizonte", "uf" : "MG" } }
```

Figura 113 – Exemplos de utilização dos operadores de atualização de chaves
Fonte: Elaborado pelo Autor.

Observe que, na última chamada da função **updateMany(),** todos os documentos que atendem à condição de busca são alterados alterados. Em nosso exemplo, todos os contatos sem endereço receberam o endereço de Medeiros-MG.

Os operadores de atualização de *arrays* são os seguintes:

- **$addToSet**: Adiciona um elemento a um *array* se esse elemento não existir;
- **$pop**: Remove o primeiro ou último item de um *array*;
- **$pull**: Remove os elementos de um *array* que atendem a um filtro;
- **$pullAll**: Remove os elementos informados de um *array*;
- **$push**: Adiciona um elemento a um *array* (mesmo mesmo que esse elemento já exista)

Além dos operadores de atualização de array, podemos modificar uma posição de um array com uma operação do tipo **{$set: {<array>.<n>: valor}}**. Essa operação atribui um **valor** a posição **n** do *array*. A Figura 114 mostra alguns exemplos de modificação de *arrays*.

Observe os exemplos e repare no uso do modificador **$each**. Ele é usado quando o operador deve receber um único elemento para a operação, como no caso do **$addToSet**. Se não for feito dessa forma, o operador incluiria um *array* como elemento do outro *array* (o resultado seria um *array* como ["A", "B", ["B", "C", "D"]]).

```
> db.produtos.insertOne({_id: 1, nome: "Telefone XYZ", tags: ["A",
"B"]})
WriteResult({ "nInserted" : 1 })
> db.produtos.updateOne({_id: 1}, {$addToSet: {tags: "C"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.updateOne ({_id: 1}, {$addToSet: {tags: "B"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.produtos.updateOne ({_id: 1}, {$addToSet: {tags: {$each: ["B", "C",
"D"]}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.find()
{ "_id": 1, "descricao": "Telefone XYZ", "tags": [ "A", "B", "C", "D"] }
> db.produtos.updateOne ({_id: 1}, {$pop: {tags: 1}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.find()
{ "_id" : 1, "descricao" : "Telefone XYZ", "tags" : [ "A", "B", "C" ] }
> db.produtos.updateOne ({_id: 1}, {$pop: {tags: -1}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.find()
{ "_id" : 1, "descricao" : "Telefone XYZ", "tags" : [ "B", "C" ] }
> db.produtos.updateOne ({_id: 1}, {$push: {tags: {$each: ["A", "F",
"G"]}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.find()
{ "_id" : 1, "descricao" : "Telefone XYZ", "tags" : [ "B", "C", "A",
"F", "G" ] }
> db.produtos.updateOne ({_id: 1}, {$pull: {tags: {$in: ["C", "F"]}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.find()
{ "_id" : 1, "descricao" : "Telefone XYZ", "tags" : [ "B", "A", "G" ] }
> db.produtos.updateOne ({_id: 1}, {$pull: {tags: {$gt: "F"}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.find()
{ "_id" : 1, "descricao" : "Telefone XYZ", "tags" : [ "B", "A" ] }
> db.produtos.updateOne ({_id: 1}, {$pullAll: {tags: ["B", "C"]}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.produtos.find()
{ "_id" : 1, "descricao" : "Telefone XYZ", "tags" : [ "A" ] }
```

Figura 114 – Exemplos de modificação de *arrays*
Fonte: Elaborado pelo Autor.

3.6.4 Removendo documentos

A remoção de documentos é uma operação relativamente simples feita com as funções **deleteOne()** e **deleteMany()**. Basicamente, elas recebem como parâmetro um objeto com as condições de uma consulta semelhante às consultas feitas com a operação **find()**. Assim, é importante consultar os documentos e ter certeza de que a condição de consulta está correta e não apagarmos documentos errados.

3.7 Replicação e Fragmentação

A utilização do MongoDB para Big Data, normalmente, envolve os conceitos de replicação e fragmentação. A replicação consiste na execução do MongoDB em um grupo de vários computadores para garantir alta disponibilidade. Esse conjunto é chamado de *cluster* e seus computadores são chamados de nós. Na replicação, um nó é chamado de primário e os demais são chamados de secundários. O nó primário sincroniza os dados com os demais para garantir que todos tenham os mesmos dados.

A cada determinado período de tempo, chamado de *heartbeat*, os nós se comunicam para verificar se estão ativos. Caso o nó primário fique indisponível, um nó secundário é eleito para se tornar o novo primário. Além disso, novos nós podem ser adicionados ao *cluster* sem causar interrupções no sistema.

Quando utilizamos um *cluster* podemos utilizar a opção **writeConcern** na atualização para descrever o nível de garantia desejado que que a operação de atualização seja considerada efetivada. Podemos especificar, por exemplo, o número de nós que devem gravar os dados e o tempo limite (*timeout*) para a efetivação da operação.

A fragmentação, também chamada de particionamento, é usada quando o banco de dados atinge um tamanho inviável de ser mantido em um único computador. A fragmentação permite que uma coleção do banco de dados seja quebrada em várias partes. Além disso, cada uma dessas partes pode ficar em um único computador um em um *cluster* com vários nós.

3.8 MongoDB e Python

A utilização do MongoDB na prática envolve o desenvolvimento de aplicações que se conectam ao banco de dados. O MongoDB possui suporte a diversas linguagens de programação. No caso da linguagem Python, podemos usar a biblioteca **pymongo** que pode ser instalada no Linux com comando sudo **apt install python3-pymongo**. No Windows, devemos usar um terminal Anaconda e informar o comando **pip install pymongo**.

Uma característica interessante da biblioteca **pymongo** é a reconexão automática em caso de falha quando estamos trabalhando com replicação. Isso ajuda a resolver muitos problemas práticos em um sistema de produção.

O primeiro passo para trabalharmos com MongoDB no Python é fazer a conexão com o banco de dados através da classe **MongoClient**. Por padrão, o servidor (IP ou

endereço do servidor) é o computador local (**localhost**) e a porta é a **27017** (porta padrão do MongoDB). A Figura 115 mostra um exemplo de conexão que lista os bancos de dados disponíveis com o método **list_database_names()**, acessa o banco de dados **agenda** e lista as coleções disponíveis com o método **list_collection_names()**.

```
from pymongo import MongoClient

def conecta():
    '''Conexão'''
    cliente = MongoClient()
    print('Conectado!')
    print('Bancos de dados disponíveis:')
    print(cliente.list_database_names())
    banco = cliente['agenda']
    print('Coleções do banco agenda:')
    print(banco.list_collection_names())

if __name__ == "__main__":
    conecta()
```

Figura 115 – Código para conexão com MongoDB
Fonte: Elaborado pelo Autor.

A partir de um banco de dados, podemos obter uma coleção através do método **get_collection()** ou simplesmente usando o nome da coleção como uma chave no objeto de dicionário. Com a coleção, podemos realizar todas as operações de inserção, modificação, remoção e consulta já estudados. A Figura 116 mostra um exemplo que lista todos os contados da coleção contatos. A Figura 117 mostra o resultado da execução do código.

```
from pymongo import MongoClient
import pprint

def conecta(nome_banco, host='localhost'):
    '''Conexão'''
    cliente = MongoClient()
    return cliente[nome_banco]

def main():
    '''Função principal'''
    agenda = conecta('agenda')
    contatos = agenda['contatos']
    for contato in contatos.find():
        pprint.pprint(contato)

if __name__ == '__main__':
    main()
```

Figura 116 – Listagem de documentos no MongoDB com Python
Fonte: Elaborado pelo Autor.

```
{'_id': ObjectId('635d548e0171b3fe3946daa6'),
 'emails': ['josesilva@exemplo.com', 'jsilva@exemplo.com'],
 'endereco': {'cidade': 'Bambuí', 'uf': 'MG'},
 'nascimento': '1984-04-12',
 'nome': 'José',
 'sobrenome': 'Silva'}
{'_id': ObjectId('635d5a15a72ae2c3ee405a48'),
 'endereco': {'cidade': 'Medeiros', 'uf': 'MG'},
 'emails': ['joaosilva@exemplo.com'],
 'nascimento': '1988-05-11',
 'nome': 'João',
 'sobrenome': 'Silva'}
{'_id': ObjectId('635d6459a72ae2c3ee405a49'),
 'endereco': {'cidade': 'Medeiros', 'uf': 'MG'},
 'emails': ['mariasilva@exemplo.com'],
 'nome': 'Maria',
 'sobrenome': 'Silva'}
{'_id': ObjectId('635d6459a72ae2c3ee405a4a'),
 'endereco': {'cidade': 'Medeiros', 'uf': 'MG'},
 'nascimento': '1990-02-16',
 'nome': 'Ana',
 'sobrenome': 'Lima'}
{'_id': ObjectId('635d9353bd71205caa332be3'),
 'endereco': {'cidade': 'Belo Horizonte', 'uf': 'MG'},
 'nome': 'Joaquim',
 'sobrenome': 'Ribeiro'}
```

Figura 117 – Resultado do código para listagem de documentos no MongoDB com Python
Fonte: Elaborado pelo Autor.

Os objetos de coleções da biblioteca **pymongo** possuem, basicamente, métodos equivalentes a todas as operações de manipulação de documentos do Mongo Shell. No caso da inserção, podemos usar os métodos **insert_one()** e **insert_many()** como mostrado na Figura 118.

```
from pymongo import MongoClient

def conecta(nome_banco, host='localhost'):
    '''Conexão'''
    cliente = MongoClient()
    return cliente[nome_banco]

def main():
    '''Função principal'''
    agenda = conecta('agenda')
    contatos = agenda['contatos']
    roberto = {
            'nome': 'Roberto',
            'sobrenome': 'Nascimento',
            'emails': ['rcn@exemplo.com']}
    contatos.insert_one(roberto)
    contatos.insert_many([
            {'nome': 'Carlos', 'sobrenome': 'Souza'},
            {'nome': 'Davi', 'sobrenome': 'Araújo'}
            ])
    for contato in contatos.find():
        print(contato)

if __name__ == '__main__':
    main()
```

Figura 118 – Inserção de documentos no MongoDB com Python
Fonte: Elaborado pelo Autor.

A consulta de documentos é feita como método **find()** da coleção análogo à função **find()** do Mongo Shell. A Figura 119 mostra um exemplo de código fazendo consulta e ordenação de documentos.

```
from pymongo import MongoClient

def conecta(nome_banco, host='localhost'):
    '''Conexão'''
    cliente = MongoClient()
    return cliente[nome_banco]

def main():
    '''Função principal'''
    agenda = conecta('agenda')
    contatos = agenda['contatos']
    cursor = contatos.find({'sobrenome': 'Silva'},
                           {'_id': 0}).sort('nome')
    for contato in cursor:
        print(contato)

if __name__ == '__main__':
    main()
```

Figura 119 – Consulta por documentos no MongoDB com Python
Fonte: Elaborado pelo Autor.

As operações de modificação podem ser feitas com os métodos **update_one()** e **update_many()** análogos às funções **updateOne()** e **updateMany()** do Mongo Shell. Já as

operações de remoção podem ser realizadas com os métodos **delete_one()** e **delete_many()** equivalentes as funções **deleteOne()** e **deleteMany()** do mongo Shell. A Figura 120 mostra um exemplo com operações de modificação e remoção.

```
from pymongo import MongoClient

def conecta(nome_banco, host='localhost'):
    '''Conexão'''
    cliente = MongoClient()
    return cliente[nome_banco]

def main():
    '''Função principal'''
    agenda = conecta('agenda')
    contatos = agenda['contatos']
    print('Contatos Originais')
    for contato in contatos.find():
        print(contato)
    print('Contato Davi modificado')
    contatos.update_one({'nome': 'Davi'},
                        {'$set':
                            {'emails': ['da@exemplo.com']}})
    print('Contato Carlos excluído')
    contatos.delete_one({'nome': 'Carlos'})
    for contato in contatos.find():
        print(contato)

if __name__ == '__main__':
    main()
```

Figura 120 – Modificação e remoção de documentos no MongoDB com Python
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

3.9 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Campeonato de futebol

- Desenvolva um sistema para controlar um campeonato de futebol. Considere a utilização de uma coleção para os times e outra para os jogos. Os times devem possuir nome e o total de pontos no campeonato (inicialmente zero). Os jogos são compostos por data, times e gols. Utilize apenas o **_id** dos times nos jogos;

- Comece criando variáveis globais para a conexão com o banco, os nomes das coleções e suas chaves. Essas variáveis ajudam a evitar erros de digitação de nomes que poderiam acontecer usando litareis em várias partes do código;

- Crie uma função para incluir novos times. A função deve receber o nome do time, verificar se ele já existe e, em caso, negativo criar um novo documento com o nome informado;

- Implemente a função **listar_times()** para listar todos os times em ordem decrescente pelos pontos;

- Desenvolva a função **soma_pontos()** para somar pontos aos times de acordo com o resultado de um jogo. O método recebe como parâmetro os _id dos times mandante e visitante bem como os gols desses times no jogo. O time vencedor ganha três pontos. Em caso de empate, ambos times ganham um ponto. Os pontos ganhos devem ser somados ao valor do campo de pontos do dicionário de cada time;

- Escreva a função **incluir_jogo()** que deve receber a data do jogo, os nomes dos times e os gols. A função deve verificar se os nomes de times existem e são diferente. Se essa verificação proceder, o jogo deve ser inserido e a função **soma_pontos()** deve ser chamada. Na inserção do jogo, guarde apenas o **_id** dos times convertendo para **str**;

- Desenvolva a função **listar_jogos()** para listar os jogos ordenados pela data. Quando obter o **_id** dos times no documento do jogo, faça uma conversão usando a classe **ObjectId()** da biblioteca **bson.objectid**. Essa conversão é necessária para buscar os dados dos times pelo **_id**;

- Crie a função **menu()** para exibir o menu de opções para o usuário e obter a resposta selecionada. As opções são incluir time, incluir, jogo, listar jogos e sair.

- Implemente também a função **principal()** com um laço de repetição para listar os jogos, exibir o menu de opções e executar a opção selecionada pelo usuário. O laço de repetição é interrompido quando o usuário escolhe a opção sair.

B) Agenda de contatos

- Desenvolva uma aplicação de agenda de contatos. Os contatos devem possuir, obrigatoriamente, **_id** e **nome**. Além disso, podem ser incluídos outros campos opcionais de acordo com a necessidade do usuário;

- Comece criando variáveis globais para a conexão com o banco, o nome da coleção e suas chaves. Essas variáveis ajudam a evitar erros de digitação de nomes que poderiam acontecer usando litareis em várias partes do código;

- Crie a função **dados_contato()** para receber os dados de um contato informados pelo usuário. A função deve receber uma mensagem a ser exibida na tela, pegar o nome do contato e outros campos opcionais. A função pode usar um laço de repetição para pegar quantos campos opcionais o usuário desejar. Os valores de campos com múltiplos itens separados por vírgula devem ser convertidos em listas;

- Implemente a função **inserir()** para cadastrar novos contatos no banco de dados. A função **dados_contato()** pode ser usada para pegar os dados do contato. O **_id** do contato deve ser um valor inteiro calculado como o número total de contatos mais um. Após pegar os dados e calcular o **_id**, a função deve gravar o contato no banco de dados;

- Escreva a função **mostra()** que recebe um contato e o mostra na tela;

- Desenvolva a função **alterar()** para modificar os dados de um contato. A função deve mostrar os dados atuais e pegar os novos dados com a função **dados_contato()**. Em seguida, a função deve substituir os dados antigos pelos novos no banco de dados;

- Crie a função **dados_pesquisa()** para pegar os dados a serem pesquisados pelo usuário. A função deve pegar pares de campos e valores com o usuário. O usuário pode informar quantos pares de campos e valores desejar. No final, a função deve retornar um dicionário com os pares recebidos;

- Faça a função **pesquisar()** para realizar buscas no banco de dados. Primeiro, pegue os dados de pesquisa chamando a função **dados_pesquisa()**. Em seguida, faça a busca no banco de dados e mostre cada contato encontrado;

- Implemente a função **listar()** para mostrar todos os contatos cadastrados em ordem crescente pelo nome;

- Escreva a função **excluir()** para apagar um contado do banco de dados. A função deve perguntar o **_id** do contato a ser removido ao usuário e proceder com a exclusão no banco de dados;

- Crie a função **menu()** para exibir o menu de opções para o usuário e obter a resposta selecionada. As opções são incluir, excluir, alterar, pesquisar e sair.

- Implemente também a função **principal()** com um laço de repetição para listar os contatos, exibir o menu de opções e executar a opção selecionada pelo usuário. O laço de repetição é interrompido quando o usuário escolhe a opção sair.

3.10 Respostas dos exercícios

A) Campeonato de futebol

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from pymongo import MongoClient
from bson.objectid import ObjectId

# Banco de dados
BANCO = MongoClient()['campeonato']
# Coleção de times
TIMES = 'times'
# Coleção de jogos
JOGOS = 'jogos'
# Chaves de times e jogos
ID = '_id'
NOME = 'nome'
PONTOS = 'pontos'
MANDANTE = 'mandante'
VISITANTE = 'visitante'
DATA = 'data'
GOLS = 'gols'

# Escreve linha na tela
def linha():
    print('-'*50)

# Incluir time
def incluir_time(nome_time):
    times = BANCO[TIMES]
    # Verifica se o time não existe
    busca = times.find_one({NOME: nome_time})
    if busca is None:
        # Insere o time
        times.insert_one({NOME: nome_time, PONTOS: 0})

# Lista times
def listar_times():
    times = BANCO[TIMES]
    # Verifica se não há times cadastrados
    if times.count_documents({}) == 0:
        print('Nenhum time cadastrado!')
        linha()
    else:
        # Busca times ordenando decrescente pelos pontos
        busca = times.find().sort(PONTOS, -1)
        for posicao, time in enumerate(busca):
            print(posicao+1, ') ', time[NOME], ': ',
                        time[PONTOS], sep='')

# Soma pontos de acordo com o resultado do jogo
def soma_pontos(id_mandante, id_visitante, gols_mandante,
```

```
51                  gols_visitante):
52      times = BANCO[TIMES]
53      # Testa se número de gols do mandante é maior que visitante
54      if gols_mandante > gols_visitante:
55          # Mais três pontos para o mandante
56          times.update_one({ID: id_mandante}, {'$inc': {PONTOS: 3}})
57      # Testa se número de gols do visitante é maior que mandante
58      elif gols_visitante > gols_mandante:
59          # Mais três pontos para o visitante
60          times.update_one({ID: id_visitante}, {'$inc': {PONTOS: 3}})
61      else:
62          # Em caso de empante, mais um ponto para cada
63          times.update_one({ID: id_mandante}, {'$inc': {PONTOS: 1}})
64          times.update_one({ID: id_visitante}, {'$inc': {PONTOS: 1}})
65
66  # Incluir um jogo
67  def incluir_jogo(data, nome_mandante, nome_visitante,
68                   gols_mandante, gols_visitante):
69      times = BANCO[TIMES]
70      jogos = BANCO[JOGOS]
71      # Busca pelos times
72      mandante = times.find_one({NOME: nome_mandante})
73      visitante = times.find_one({NOME: nome_visitante})
74      # Testa se os times existem e não são o mesmo time
75      if mandante is not None and visitante is not None and \
76              mandante[ID] != visitante[ID]:
77          # Cria dicionário do jogo
78          jogo = {DATA: data,
79                  MANDANTE: str(mandante[ID]),
80                  VISITANTE: str(visitante[ID]),
81                  GOLS: [gols_mandante, gols_visitante]}
82          # Insere o jogo
83          soma_pontos(mandante[ID], visitante[ID],
84                      gols_mandante, gols_visitante)
85          jogos.insert_one(jogo)
86
87  def listar_jogos():
88      # Busca jogos
89      jogos = BANCO[JOGOS]
90      times = BANCO[TIMES]
91      # Verifica se não existem jogos
92      if jogos.count_documents({}) == 0:
93          print('Nenhum jogo cadastrado!')
94          linha()
95      else:
96          linha()
97          print('Jogos do campeonato')
98          # Busca jogos ordenando pela data
99          busca_jogos = jogos.find().sort(DATA)
100         for jogo in busca_jogos:
101             # Id dos times
102             id_mandante = ObjectId(jogo[MANDANTE])
103             id_visitante = ObjectId(jogo[VISITANTE])
104             data = jogo[DATA]
105             # Obtém o documento do time com base no ID
```

```
106                 mandante = times.find_one({ID: id_mandante})
107                 visitante = times.find_one({ID: id_visitante})
108                 gols_mandante, gols_visitante = tuple(jogo[GOLS])
109                 print(data, ': ',
110                       mandante[NOME], ' [', gols_mandante, '] x [',
111                       gols_visitante, '] ', visitante[NOME], sep='')
112             linha()
113         input()
114
115     # Menu de opções
116     def menu():
117         linha()
118         print('Campeonato')
119         linha()
120         listar_times()
121         print('+(T)ime', '+(J)ogo', '(L)istar jogos',
122               ' (S)air', sep=' | ')
123         return input('Informe a opção desejada: ').strip().lower()
124
125     # Função principal
126     def principal():
127         while True:
128             resp = menu()
129             if resp == 't':
130                 time = input('Informe o time: ').strip()
131                 incluir_time(time)
132             elif resp == 'j':
133                 print('Informe os dados do jogo')
134                 data = input('Data: ').strip()
135                 time_m = input('Mandante: ').strip()
136                 time_v = input('Visitante: ').strip()
137                 gols_m = int(input('Gols mandante: ').strip())
138                 gols_v = int(input('Gols visitante: ').strip())
139                 incluir_jogo(data, time_m, time_v, gols_m, gols_v)
140             elif resp == 'l':
141                 listar_jogos()
142             elif resp == 's':
143                 break
144
145     if __name__ == '__main__':
146         principal()
```

B) Agenda de contatos

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from pymongo import MongoClient

# Banco de dados
BANCO = MongoClient()['agenda']
# Coleção de contatos
CONTATOS = 'contatos'
# Chaves obrigatórias para contatos
ID = '_id'
NOME = 'nome'

# Escreve linha na tela
def linha(caractere):
    print(caractere * 50)

# Pega os dados de um contato
def dados_contato(mensagem):
    print(mensagem)
    # O nome é obrigatório
    nome = input('NOME: ').strip()
    contato = {}
    contato[NOME] = nome
    while True:
        # Demais campos são opcionais
        resp = input('Adicionar mais campos? (S/N)')
        if resp.strip().lower() == 'n':
            break
        campo = input('Informe o campo: ').strip().upper()
        print('Informe o valor do campo',
              '(múltiplos valores separados por vírgula)')
        valor = input(campo + ': ').strip()
        contato[campo] = valor
    return contato

# Insere um novo contato
def inserir():
    linha('-')
    print('Inserção de contato:')
    linha('-')
    # Pega os dados do contato
    novo_contato = dados_contato('Informe os dados do novo
contato')
    contatos = BANCO[CONTATOS]
    # Calcula o ID (incremental pelo número de contatos)
    contato_id = contatos.count_documents({}) + 1
    novo_contato[ID] = contato_id
    # Insere o contato no banco de dados
    contatos.insert_one(novo_contato)
    print('\n Contato inserido.')
    input()

```

```
 53  # Mostra contato
 54  def mostra(contato):
 55      # Para cada chave do contato
 56      for chave in contato:
 57          # Pega valor da chave
 58          valor = contato[chave]
 59          print(chave.upper(), ':', sep='', end=' ')
 60          # Testa se o valor é uma lista
 61          if isinstance(valor, list):
 62              # Concatena os itens da lista
 63              valor = ', '.join(valor)
 64          print(valor)
 65      linha('-')
 66
 67  # Altera um contato
 68  def alterar():
 69      linha('-')
 70      print('Alteração de contato:')
 71      linha('-')
 72      # Pega o id a ser excluído
 73      contato_id = int(input('_ID do contato: '))
 74      contatos = BANCO[CONTATOS]
 75      # Busca o contato pelo id
 76      contato = contatos.find_one({ID: contato_id})
 77      # Testa se o contato não foi encontrado
 78      if contato is None:
 79          print('_ID não encontrado!')
 80          linha('-')
 81          return
 82      linha('-')
 83      print('Dados originais:')
 84      linha('-')
 85      # Mostra os dados originais do contato
 86      mostra(contato)
 87      # Pega os novos dados
 88      alterado = dados_contato('Informe os novos dados para o
contato')
 89      # Substitui os dados antigos pelos novos
 90      contatos.replace_one(contato, alterado)
 91      print('\n Contato alterado.')
 92      input()
 93
 94  # Pega dados para pesquisa
 95  def dados_pesquisa():
 96      print('Informe os dados para pesquisa:')
 97      contato = {}
 98      while True:
 99          # Pega pelo menos um campo para pesquisa
100          campo = input('Campo: ').strip().upper()
101          valor = input('Valor para ' + campo + ': ').strip()
102          contato[campo] = valor
103          resp = input('Informar mais campos? (S/N) ')
104          if resp.strip().lower() == 'n':
105              break
106      return contato
```

```
107
108 # Pesquisa por contatos
109 def pesquisar():
110     linha('-')
111     print('Pesquisa de contato:')
112     linha('-')
113     # Pega campos para pesquisa
114     filtro = dados_pesquisa()
115     contatos = BANCO[CONTATOS]
116     # Verifica se não existem contatos retornados pela pesquisa
117     if contatos.count_documents(filtro) == 0:
118         print('Nenhum contato encontrado.')
119         input()
120         return
121     # Mostra cada contato encontrado
122     for contato in contatos.find(filtro).sort(NOME):
123         mostra(contato)
124     input()
125
126 # Lista todos os contatos
127 def listar():
128     contatos = BANCO[CONTATOS]
129     # Testa se não existem contatos cadastrados
130     if contatos.count_documents({}) == 0:
131         print('Nenhum contato encontrado.')
132         linha('-')
133         return
134     # Mostra todos os contatos ordenados pelo nome
135     for contato in contatos.find().sort(NOME):
136         mostra(contato)
137
138 # Exlui um contato
139 def excluir():
140     linha('-')
141     print('Exclusão de contato:')
142     linha('-')
143     # Pega o id do contato
144     contato_id = int(input('_ID do contato: '))
145     contatos = BANCO[CONTATOS]
146     # Busca pelao contato
147     contato = contatos.find_one({ID: contato_id})
148     # Testa se o contato não existe
149     if contato is None:
150         print('_ID não encontrado!')
151         linha('-')
152         return
153     # Deleta o contato
154     contatos.delete_one({ID: contato_id})
155     print('\n Contato excluído.')
156     input()
157
158 def menu():
159     # Menu que lista e recebe a opção do usuário
160     linha('=')
161     print('Agenda de contatos')
```

```
162       linha('=')
163       listar()
164       print('(I)ncluir | (E)xcluir | (A)lterar | (P)esquisar |
(S)air')
165       return input('Informe a opção desejada: ').strip().lower()
166
167   def executar():
168       while True:
169           resp = menu()
170           if resp == 'i':
171               inserir()
172           elif resp == 'e':
173               excluir()
174           elif resp == 'a':
175               alterar()
176           elif resp == 'p':
177               pesquisar()
178           elif resp == 's':
179               break
180
181   if __name__ == '__main__':
182       executar()
```

3.11 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista ao vídeo “Revisão da Terceira Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer os conceitos básicos de interface gráfica;
- Entender o funcionamento de eventos relacionados com componentes;
- Desenvolver códigos com interfaces gráficas.

4.1 Introdução

Um grafo é uma estrutura composta por uma série de nós que podem ser conectados por arestas. Os grafos são estruturas muito úteis em diversas áreas, mas, principalmente, na Matemática e na Computação. A Internet, por exemplo, pode ser visualizada como um grafo, com diferentes páginas da web como nós e hiperlinks entre elas como arestas. Outros exemplos são pessoas conectadas em redes sociais, árvores genealógicas e mapas de cidades. A Figura 121 mostra um exemplo de grafo representando um grupo de pessoas de uma rede social. Nesse exemplo, os nós são as pessoas e as arestas as ligações entre elas.

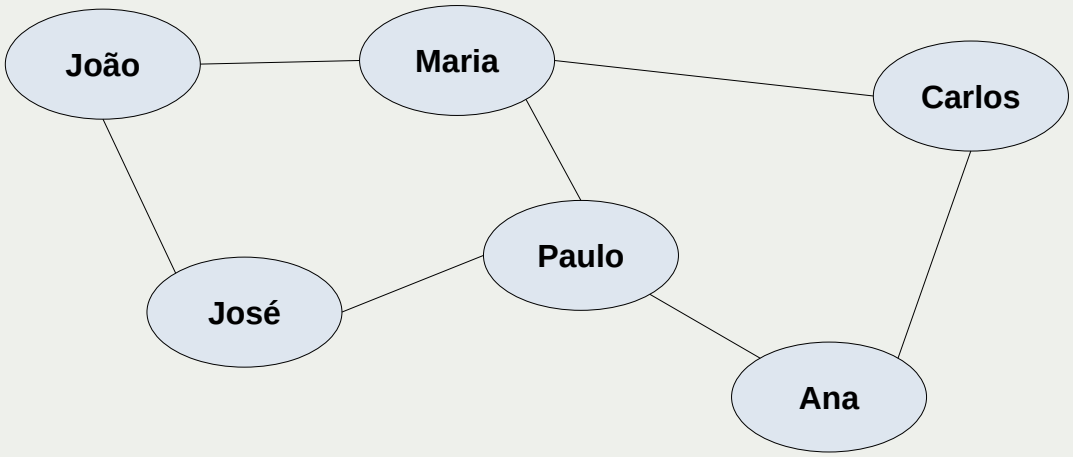

Figura 121 – Grafo de grupo de pessoas de uma rede social
Fonte: Elaborado pelo Autor.

Os bancos de dados de grafos armazenam dados na forma de nós (ou vértices), arestas e propriedades. Tanto os nós como as arestas podem receber propriedades específicas. Como exemplo, considere um mapa contendo cidades. Podemos representar tal mapa com um grafo onde os nós são as cidades e as arestas são as estradas. Os nós das cidades podem ter propriedades como nome e população. As arestas das estradas podem ter a distância como propriedade (ROBINSON, ; WEBBER; EIFREM, 2015).

Embora um banco de dados de grafos possa ter apenas nós, na maioria das vezes ele terá arestas também. Apesar de ser possível representar grafos em outros tipos de bancos de dados, com o uso dos bancos de dados de grafo, essa representação se torna mais simples. Além disso, os sistemas de banco de dados de grafos, em geral, possuem desempenho melhor para lidar com esse tipo de estrutura.

Os sistemas de banco de dados de grafos são sistemas concebidos para lidar com bases de dados em formato de grafos. Sua principal vantagem é a linguagem de consulta com diversas operações sobre grafos já implementadas. O Neo4j é um dos sistema de banco de dados de grafos de código aberto, implementado na linguagem Java com as características que mencionamos. Ele está disponível para os sistemas operacionais Linux e Windows.

O Neo4j pode pode ser usado sob uma licença corporativa ou uma licença comunitária. A licença comunitária permite o uso gratuito do sistema mesmo que o software que com acesso ao sistema seja fechado. A licença corporativa, por sua vez, foi projetada para implementações comerciais onde escala e disponibilidade são importantes. Os requisitos mínimos para execução do Neo4j são 2GB de memória e 10GB de espaço livre em disco.

4.2 Instalação

O Neo4j possui diversas versões, apesar de já existirem versões mais recentes, neste livro, vamos adotar a versão 4.1. Usaremos a licença comunitária que pode ser baixada de forma gratuita na página oficial do sistema (https://neo4j.com/download-center/#community).

Os pacotes de instalação do Neo4j não estão disponíveis por padrão no Linux. Contudo, é possível fazer a instalação manualmente ou usar os pacotes do repositório oficial. A Figura 122 mostra os comandos necessários para fazer a instalação usando o repositório oficial.

```
wget -O - https://debian.neo4j.com/neotechnology.gpg.key | sudo apt-key add -
sudo add-apt-repository "deb https://debian.neo4j.com stable 4.1"
sudo apt install neo4j
```

Figura 122 – Código do arquivo **calculadora.py**
Fonte: Elaborado pelo Autor.

A instalação manual funciona para Linux e Windows desde que você possua uma máquina virtual Java (https://www.java.com/download/). O procedimento envolve baixar a um arquivo compactado da página oficial e extraí-lo. Para iniciar o sistema, basta abrir a pasta **bin** em um terminal e executar o comando **neo4j start**. Para finalizar o servidor, você pode usar o comando **neo4j stop**.

Após a inicialização, podemos acessar o sistema em um navegador através do endereço **http://localhost:7474/**. A Figura 123 mostra a tela de primeira acesso ao sistema. No primeiro acesso ao sistema, você deve informar **neo4j** para usuário e senha. O Neo4j vai solicitar que você altere a senha de acesso que deve ser diferente da anterior.

Depois do primeiro acesso, você pode alterar a senha novamente usando o comando **:SERVER change-password** no navegador conforme mostrado na Figura 124. Em nosso caso, mudamos a senha novamente para **neo4j**. Assim, nos demais exemplos, quando for necessário vamos considerar usuário e senha como **neoj4**.

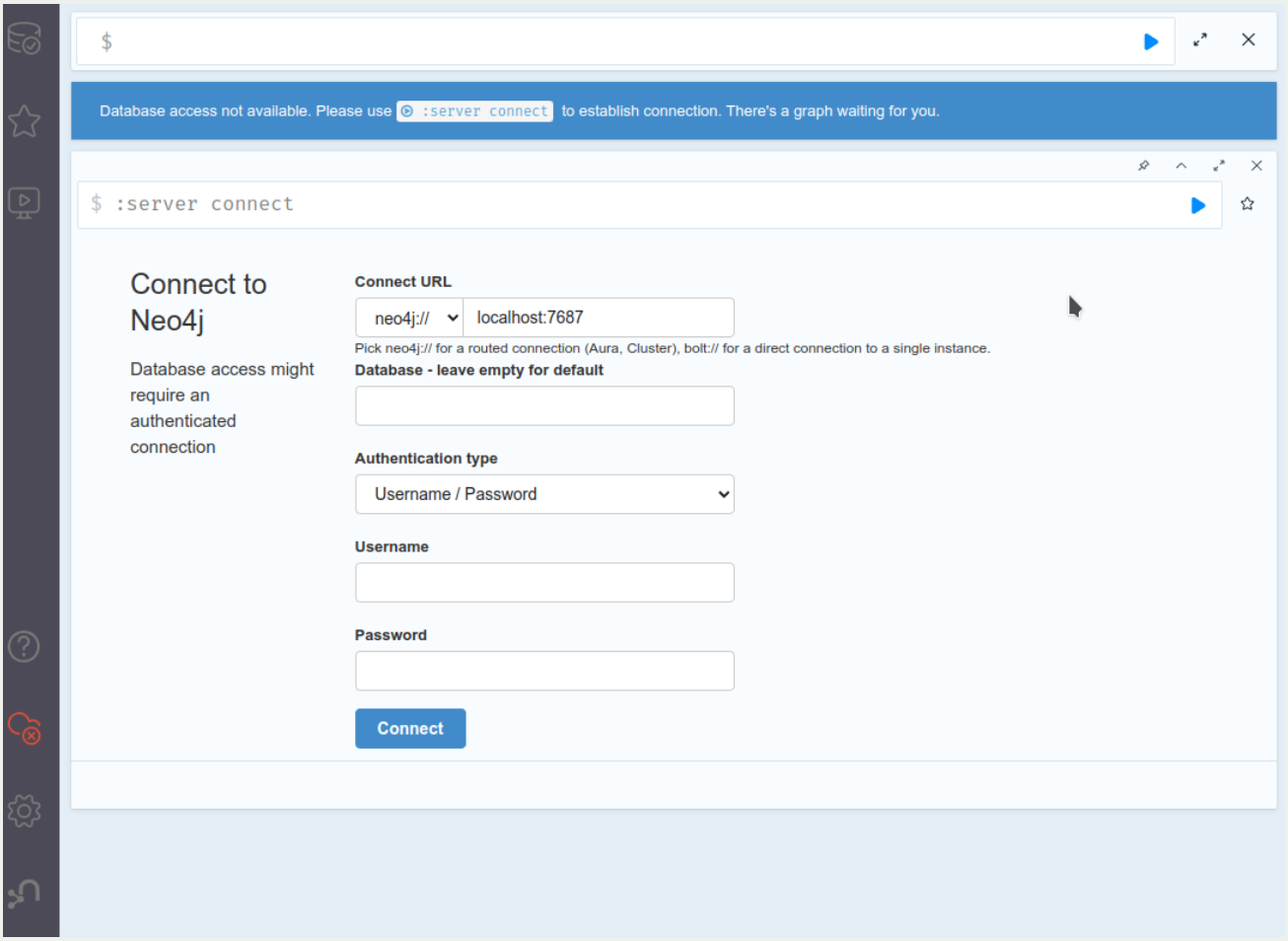

Figura 123 – Tela de primeira acesso do Neo4j
Fonte: Elaborado pelo Autor.

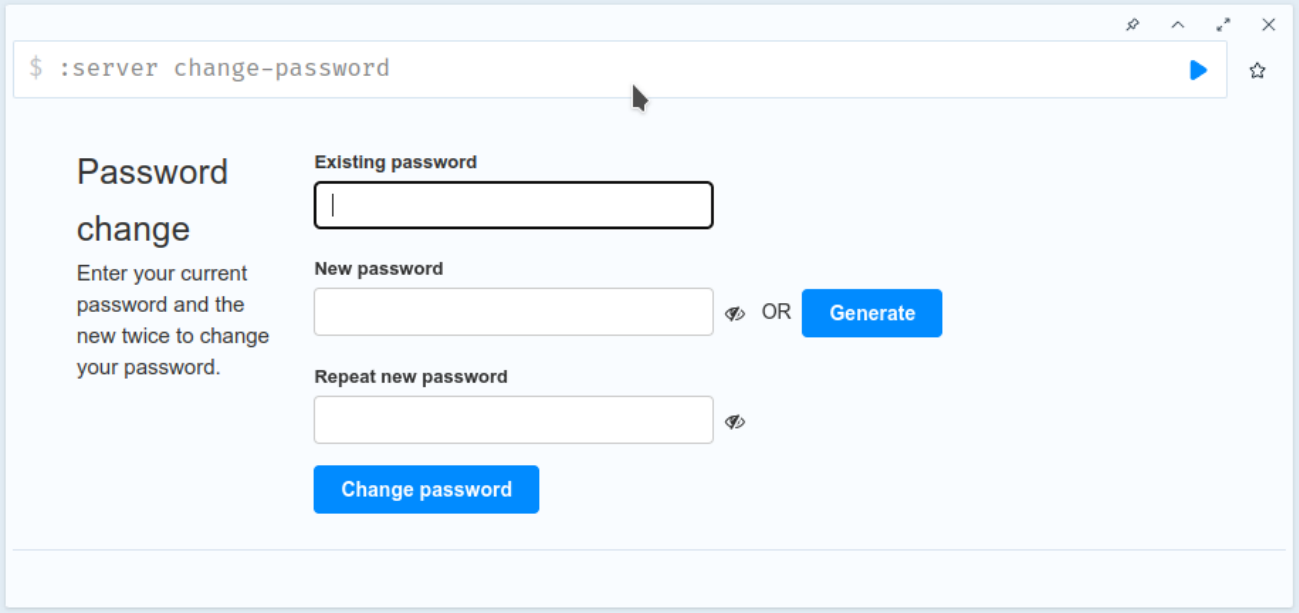

Figura 124 – Resultado de uma consulta no Neo4j visualizado no navegador
Fonte: Elaborado pelo Autor.

4.3 A linguagem Cypher

O Neo4j utiliza a uma linguagem de consulta declarativa chamada Cypher. De certa forma, a Cypher possui alguma semelhança com a linguagem SQL, o que facilita para aqueles acostumados a lidar com bancos de dados relacionais. A sintaxe básica do Cypher é utiliza as seguintes notações:

- Nós: representados por parênteses **( )**;
- Propriedades: representados por chaves **{ }**;
- Relacionamentos: representados por colchetes **[ ]**.

Essas notações podem ser combinadas de várias maneiras na realização das consultas. A especificação de propriedades e relacionamentos pode se usada para filtrar o conjunto de resultados. Um tipo de instrução simples que pode ser muito utilizado é a consulta **MATCH (n) RETURN n;** (todas as consultas terminam com ponto e vírgula).

A instrução **MATCH...RETURN...** é usada buscar nós e relacionamentos no banco de dados. A cláusula **MATCH** é usada para especificar as condições da busca e, na cláusula **RETURN**, especificamos quais informações devem ser retornadas.

Na consulta **MATCH (n) RETURN n;** não informamos nenhum filtro. O **n** é um nome genérico que damos ao nó. Dessa forma, a consulta retorna todos os nós do banco de dados. Não é aconselhável executar essa consulta em ambientes de produção que podem possuir milhões de nós, levando a um longo tempo de processamento. O uso de restrições é recomendado em bancos de dados grandes para tornar o conjunto a ser retornado menor e a consulta ser respondida mais rapidamente.

A linguagem Cypher tenta manter simples a forma de especificar relacionamentos de forma descritiva. Considere a consulta **MATCH (a)-->(b) RETURN a, b;**, como exemplo que retorna pares de nós conectados. Observe que, na cláusula **MATCH**, informamos a direção da aresta (**-->**) de **a** para **b**. Essa consulta pode também, dependendo do banco de dados, retornar uma resposta muito grande.

Quando você executamos uma consulta Cypher, especificamos um padrão que é usado para buscar os dados correspondentes. Basicamente, as consultas mais complexas combinam diversos padrão para chegar a resultados mais específicos. As cláusulas da consulta podem ser minúsculas ou maiúsculas. Neste livro, escrevemos as cláusulas sempre em maiúsculas para facilitar a leitura. Os valores, por outro lado, diferenciam maiúsculas de minúsculas.

O Neo4j utiliza rótulos para diferenciar tipos diferentes de nós. Um rótulo é representado por dois pontos seguidos do nome do rótulo. Por exemplo, podemos usar a consulta **MATCH (n:Pessoa) RETURN n;** para buscar os nós que representam pessoas. Se for necessário usar vários rótulos podemos colocar um após o outro separados por dois pontos. Por exemplo, a consulta **MATCH (n:Pessoa:Brasil) RETURN n;** busca pelas pessoas que moram no Brasil.

Sempre que incluímos algum identificador entre parênteses em uma consulta estamos nos referindo a um nó. Em alguns casos, podemos ter nós sem identificadores,

mas, em geral, eles possuem identificadores para serem usados em outros locais da consulta. Os identificadores são formados por letras, números e sublinhados e devem começar com uma letra. As letras maiúsculas e minúsculas fazem diferença nos identificadores.

As propriedades são aplicadas sobre nós e relacionamentos. As propriedades são representadas usando chaves, mas podem estar dentro dos parênteses de um nó. Na realização de consultas, as propriedades atuam como filtros. Quando criamos ou modificamos nós as propriedades são incorporadas ao nó.

Um exemplo típico do uso de propriedades é **(n:Pessoa {nome: "Maria"})**. Em uma consulta, esse exemplo indica que estamos procurando por uma pessoa com nome de Maria. Os identificadores de propriedades obedecem às mesmas regras dos identificadores de nós. Os tipos de dados suportados em propriedades são os seguintes:

- Valores numéricos (inteiros e flutuantes);
- Textuais (String);
- Valores lógicos ou booleanos (**true** ou **false**);
- *Arrays* (não vazios e com itens do mesmo tipo).

Os relacionamentos são recursos poderosos de todo banco de dados de grafos que representam uma ligação entre dois nós. A informação mínima de um relacionamento é a direção, por exemplo, **(a)-->(b)** especifica que **a** está relacionado com **b**.

Um exemplo mais complexo de um relacionamento seria **(a:Pessoa {nome: "Maria"})-[r:CONHECE]->(b:Pessoa {nome: "Ana"})**. Os nós são duas pessoas com nomes de Maria e Ana. Dentro dos colchetes, podemos usar um identificador para o relacionamento e também um tipo. Os tipos são semelhantes aos rótulos, mas cada relacionamento só pode ter um tipo. Em nosso exemplo, o relacionamento está especificado como **-[r:CONHECE]->**, onde **r** é o identificador **CONHECE** é o tipo. Além do tipo, os relacionamento também podem ter propriedades.

Quando existe uma ligação (direta ou indireta) entre dois nós, dizemos que existe um caminho entre esses dois nós. Um dos pontos fortes dos bancos de dados de grafos e do Neo4j são as consultas envolvendo caminhos.

A instalação do Neo4j inclui o cliente de linha de comando **cypher-shell** localizado na pasta **bin**. Ao executá-lo será solicitado o usuário e a senha como mostrado na Figura 125.

```
$ bin/cypher-shell
username: neo4j
password: *****
Connected to Neo4j 4.1.0 at neo4j://localhost:7687 as user neo4j.
Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
neo4j@neo4j>
```

Figura 125 – Execução do cliente de linha de comando **cypher-shell**
Fonte: Elaborado pelo Autor.

A execução mostrada se conecta, por padrão, ao servidor local. O utilitário possui diversos parâmetros, como o endereço do servidor, que podem ser informado na execução. Para ver mais detalhes desses parâmetros, você pode usar o comando **cypher-shell --help**.

O **cypher-shell** é um cliente mais leve e prático. Entretanto, quando estamos começando a trabalhar com banco de dados de grafos, pode ser interessante utilizar o cliente do navegador para termos uma representação visual do gráfico.

4.3.1 Inserção de dados

Para estudarmos a linguagem Cypher, vamos usar como exemplo uma rede de amigos. Vamos começar criando os nós para representar as pessoas como mostrado na Figura 126. Para visualizar os nós criados podemos usar a instrução **'MATCH (n) RETURN n;'**.

```
// Cria um único nó
CREATE (:Pessoa {Nome: 'Raquel'});
// Cria vários nós
CREATE (:Pessoa {Nome: 'Mônica'}),
       (:Pessoa {Nome: 'Felícia'});
// Não cria Raquel que já existe
CREATE (:Pessoa {Nome: 'Raquel'});
// Cria demais nós
MERGE (:Pessoa {Nome: 'Roberto'});
MERGE (:Pessoa {Nome: 'Carlos'});
MERGE (:Pessoa {Nome: 'José'});
```

Figura 126 – Criação de nós para representar pessoas
Fonte: Elaborado pelo Autor.

Após a criação dos nós, podemos criar relacionamentos para especificar a amizade entre as pessoas. Para isso vamos usar um a instrução **MATCH...MERGE...;**. Na cláusula **MATCH**, identificamos os nós participantes e na cláusula **MERGE** criamos o relacionamento. A Figura 127 mostra a criação dos relacionamentos em nosso banco de dados.

```
MATCH (a:Pessoa {Nome: 'Raquel'}), (b:Pessoa {Nome: 'Mônica'})
MERGE (a) -[:AMIGO]- (b);
MATCH (a:Pessoa {Nome: 'Raquel'}), (b:Pessoa {Nome: 'Roberto'})
MERGE (a) -[:AMIGO]- (b);
MATCH (a:Pessoa {Nome: 'Raquel'}), (b:Pessoa {Nome: 'Felícia'})
MERGE (a) -[:AMIGO]- (b);
MATCH (a:Pessoa {Nome: 'Mônica'}), (b:Pessoa {Nome: 'José'})
MERGE (a) -[:AMIGO]- (b);
MATCH (a:Pessoa {Nome: 'Mônica'}), (b:Pessoa {Nome: 'Carlos'})
MERGE (a) -[:AMIGO]- (b);
MATCH (a:Pessoa {Nome: 'José'}), (b:Pessoa {Nome: 'Carlos'})
MERGE (a) -[:AMIGO]- (b);
MATCH (a:Pessoa {Nome: 'José'}), (b:Pessoa {Nome: 'Felícia'})
MERGE (a) -[:AMIGO]- (b);
```

Figura 127 – Criação dos relacionamentos de amizade entre as pessoas
Fonte: Elaborado pelo Autor.

Em nosso exemplo, especificamos o relacionamento com a notação **-[:AMIGO]-** que não expressa a direção da aresta. Fazemos isso porque consideramos que a relação de amizade é recíproca. No entanto, no Neo4j, todas as arestas são direcionais. Quando não especificamos, a aresta é criada em uma direção arbitrária. Isso não chega a ser um problema porque podemos consultar essas arestas sem considerar a direção como se o grafo fosse não direcionado.

Com os nós e arestas já criados, podemos fazer uma consulta e observar o resultado. A Figura 128 mostra a execução dessa consulta no navegador, onde podemos visualizar o grafo.

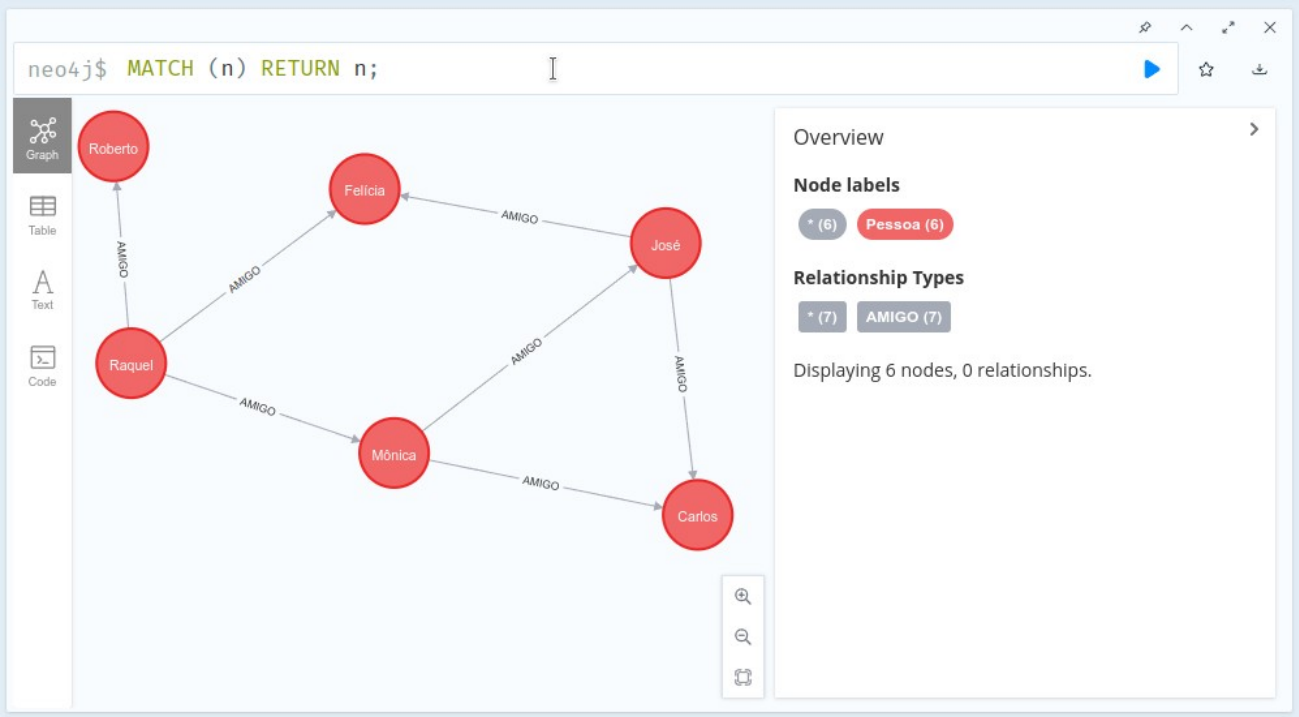

Figura 128 – Visualização do grafo de amizade no navegador
Fonte: Elaborado pelo Autor.

Para enriquecer um pouco mais nosso grafo, vamos acrescentar as preferências das pessoas. A Figura 129 mostra as instruções para inserirmos alguns exemplos de preferências.

```
MERGE (:Comida {Nome: 'Pizza'});
MERGE (:Esporte {Nome: 'Corrida'});
MERGE (:Esporte {Nome: 'Futebol'});
MATCH (a:Pessoa {Nome: 'José'}), (b:Comida {Nome: 'Pizza'})
MERGE (a) -[:GOSTA]-> (b);
MATCH (a:Pessoa {Nome: 'Carlos'}), (b:Comida {Nome: 'Pizza'})
MERGE (a) -[:GOSTA]-> (b);
MATCH (a:Pessoa {Nome: 'Mônica'}), (b:Esporte {Nome: 'Corrida'})
MERGE (a) -[:GOSTA]-> (b);
MATCH (a:Pessoa {Nome: 'Raquel'}), (b:Esporte {Nome: 'Futebol'})
MERGE (a) -[:GOSTA]-> (b);
```

Figura 129 – Criação dos relacionamentos de amizade entre as pessoas
Fonte: Elaborado pelo Autor.

4.3.2 Alteração e remoção de dados

A alteração de dados pode ser feita com as instruções **MATCH...SET...** e **MATCH...REMOVE...**, enquanto a remoção é feita com a instrução **MATCH...DELETE...**. Para demonstrarmos tais operações, vamos incluir mais algumas informações no banco de dados como mostrado na Figura 130. As informações são mais uma pessoa, uma comida e mais alguns relacionamentos. A Figura 131 mostra o grafo após a inclusão dos novos dados.

```
MERGE (:Pessoa {Nome: 'Adriano'});
MERGE (:Comida {Nome: 'Lasanha'});
MATCH (a:Pessoa {Nome: 'Adriano'}), (b:Pessoa {Nome: 'José'})
MERGE (a) -[:AMIGO]- (b);
MATCH (a:Pessoa {Nome: 'Adriano'}), (b:Comida {Nome: 'Lasanha'})
MERGE (a) -[:GOSTA]-> (b);
MATCH (a:Pessoa {Nome: 'José'}), (b:Comida {Nome: 'Lasanha'})
MERGE (a) -[:GOSTA]-> (b);
```

Figura 130 – Inclusão de mais informações no banco de dados de amigos
Fonte: Elaborado pelo Autor.

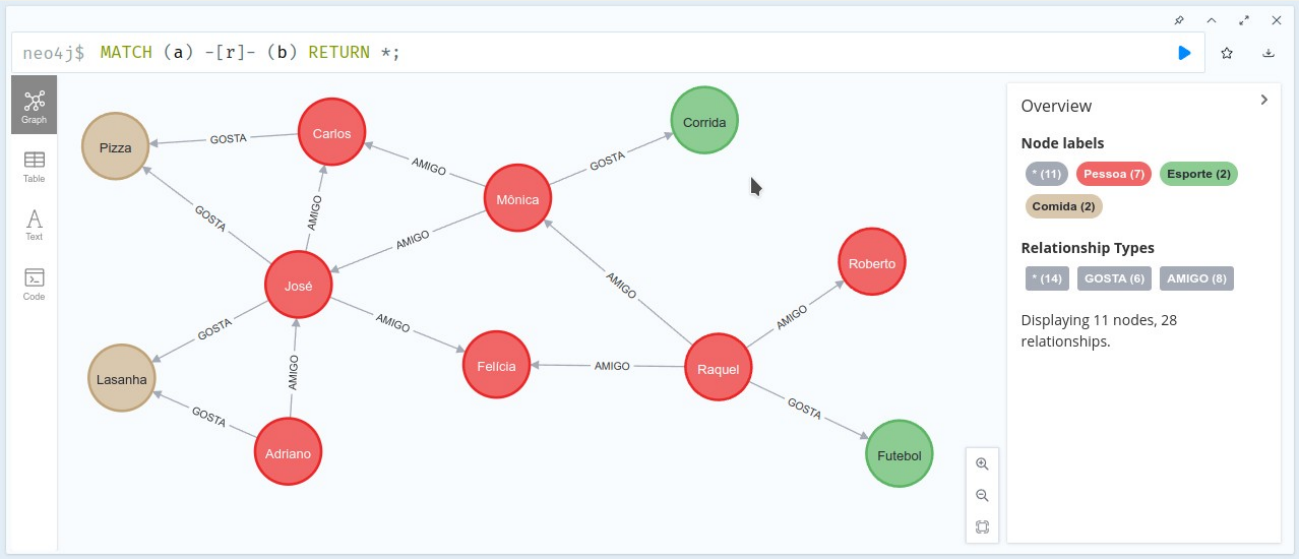

Figura 131 – Grafo de amigos após a inclusão de mais informações
Fonte: Elaborado pelo Autor.

Com o banco de dados atual, vamos mostrar como modificar dados. Vamos tomar como exemplo o nó que representa a pessoa Adriano e acrescentar a propriedade de sobrenome. A Figura 132 mostra como essa operação pode ser feita no **cypher-shell**.

No exemplo, utilizamos uma instrução com as cláusulas **MATCH**, **SET** e **RETURN**. A cláusula **MATCH** é usada para encontrar o nó existente que representa a pessoa Adriano. A cláusula **SET** é usada para acrescentar ou modificar uma propriedade. Em nosso exemplo, como o sobrenome não estava cadastrado, foi incluída uma nova propriedade com essa informação. A cláusula **RETURN** mostra o nó após a alteração.

Se usarmos uma igualdade com um dicionário na cláusula **SET**, o Neo4j vai substituir os dados do nó pelos dados do dicionário. A Figura 133 apresenta um exemplo de instrução que faz isso.

```
neo4j@neo4j> MATCH (n:Pessoa {Nome: 'Adriano'})
             SET n.Sobrenome = 'Ribeiro'
             RETURN n;
+--------------------------------------------------+
| n                                                |
+--------------------------------------------------+
| (:Pessoa {Nome: "Adriano", Sobrenome: "Ribeiro"}) |
+--------------------------------------------------+

1 row available after 12 ms, consumed after another 16 ms
Set 1 properties
```

Figura 132 – Inclusão de sobrenome no nó *Adriano*
Fonte: Elaborado pelo Autor.

```
neo4j@neo4j> MATCH (n:Pessoa {Nome: 'Adriano'})
             SET n = {Nome: 'Adriano Ribeiro', Nascimento: 1990}
             RETURN n;
+------------------------------------------------------+
| n                                                    |
+------------------------------------------------------+
| (:Pessoa {Nascimento: 1990, Nome: "Adriano Ribeiro"}) |
+------------------------------------------------------+

1 row available after 15 ms, consumed after another 19 ms
Set 3 properties
```

Figura 133 – Alteração de nó substituindo os dados
Fonte: Elaborado pelo Autor.

A cláusula **SET** permite também a adição e alteração de vários propriedades em uma única instrução. Isso pode ser feito como uma atribuição de **+=** e um dicionário ou informando atribuições para cada propriedade separadas por vírgula. A Figura 134 demostra como esses dois tipos de alterações podem ser feitos.

```
neo4j@neo4j> MATCH (n:Pessoa {Nome: 'Adriano Ribeiro'})
             SET n += {Nascimento: 1995, Idade: 27};
0 rows available after 38 ms, consumed after another 0 ms
Set 2 properties
neo4j@neo4j> MATCH (n:Pessoa {Nome: 'Adriano Ribeiro'})
             SET n.Nome = 'Adriano', n.Profissao = 'Engenheiro'
             RETURN n;
+-------------------------------------------------------------------------------+
| n                                                                             |
+-------------------------------------------------------------------------------+
|(:Pessoa {Nascimento:1995, Idade:27, Nome:"Adriano", Profissao:"Engenheiro"}) |
+-------------------------------------------------------------------------------+

1 row available after 17 ms, consumed after another 21 ms
Set 2 properties
```

Figura 134 – Alteração de múltiplas propriedades
Fonte: Elaborado pelo Autor.

Outro tipo de alteração possível com a cláusula **SET** é adicionar mais rótulos aos nós. A Figura 135 exibe um exemplo mostrando como adicionar o rótulo *Massa* à comida *Lasanha*.

```
neo4j@neo4j> MATCH (n:Comida {Nome: 'Lasanha'})
             SET n:Massa
             RETURN n;
+-----------------------------------+
| n                                 |
+-----------------------------------+
| (:Comida:Massa {Nome: "Lasanha"}) |
+-----------------------------------+

1 row available after 16 ms, consumed after another 0 ms
Added 1 labels
```

Figura 135 – Adição do rótulo *Massa* na comida *Lasanha*
Fonte: Elaborado pelo Autor.

A remoção de propriedades e rótulos é feita com a instrução **MATCH...REMOVE...**. Na cláusula **MATCH**, incluímos o filtro para buscar o nó e, na cláusula **REMOVE** especificamos o que deve ser removido. A Figura 136 demostra como podemos remover a profissão do nó *Adriano* e o rótulo *Massa* do nó *Lasanha*.

```
neo4j@neo4j> MATCH (n {Nome: 'Adriano'})
             REMOVE n.Profissao
             RETURN n;
+-------------------------------------------------------------+
| n                                                           |
+-------------------------------------------------------------+
| (:Pessoa {Nascimento: 1995, Idade: 27, Nome: "Adriano"}) |
+-------------------------------------------------------------+

1 row available after 9 ms, consumed after another 1 ms
Set 1 properties
neo4j@neo4j> MATCH (n:Comida {Nome: 'Lasanha'})
             REMOVE n:Massa
             RETURN n;
+-----------------------------+
| n                           |
+-----------------------------+
| (:Comida {Nome: "Lasanha"}) |
+-----------------------------+

1 row available after 1 ms, consumed after another 0 ms
```

Figura 136 – Remoção da profissão do nó *Adriano* e o rótulo *Massa* do nó *Lasanha*
Fonte: Elaborado pelo Autor.

Além de incluir e alterar dados, podemos também apagar nós e relacionamentos utilizando a instrução **MATCH...DELETE...**. A cláusula **MATCH**, como de costume, localiza o nó ou relacionamento a ser excluído e a cláusula **DELETE** efetiva a deleção. O Neo4j não permite a exclusão de nós com relacionamentos como mostrado na Figura 137.

```
neo4j@neo4j> MATCH (n {Nome: 'Lasanha'})
             DELETE n;
Cannot delete node<229>, because it still has relationships. To delete
this node, you must first delete its relationships.
```

Figura 137 – Erro ao tentar apagar nó com relacionamentos
Fonte: Elaborado pelo Autor.

Quando o nó possui relacionamentos, podemos remover tais relacionamentos antes de tentar apagar o nó ou usar a instrução **MATCH...DETACH DELETE...** que apaga o nó localizado e todos os seus relacionamentos. A Figura 138 mostra como podemos fazer isso. No caso do nó Lasanha, primeiro removemos os relacionamentos para, depois, apagar o nó. Já para o nó Adriano, apagamos o nó juntamento com seus relacionamentos.

```
neo4j@neo4j> MATCH ()-[r:GOSTA]->(n {Nome: 'Lasanha'})
             DELETE r;
2 rows available after 15 ms, consumed after another 1 ms
Deleted 2 relationships
neo4j@neo4j> MATCH (n {Nome: 'Lasanha'})
             DELETE n;
0 rows available after 1 ms, consumed after another 0 ms
Deleted 1 nodes
neo4j@neo4j> MATCH (n {Nome: 'Adriano'})
             DETACH DELETE n;
1 row available after 15 ms, consumed after another 16 ms
Deleted 1 nodes, Deleted 1 relationships
```

Figura 138 – Remoção de nós e seus relacionamentos
Fonte: Elaborado pelo Autor.

4.3.3 Consultas simples

Com esse simples banco de dados já possível fazer algumas consultas interessantes. Um exemplo de consulta é buscar pelos amigos da Raquel com mostrado na Figura 139.

```
neo4j@neo4j> MATCH (a:Pessoa {Nome: 'Raquel'}) -[:AMIGO]- (b)
             RETURN (b);
+------------------------------+
| b                            |
+------------------------------+
| (:Pessoa {Nome: "Felícia"}) |
| (:Pessoa {Nome: "Roberto"}) |
| (:Pessoa {Nome: "Mônica"})  |
+------------------------------+

3 rows available after 2 ms, consumed after another 0 ms
```

Figura 139 – Consulta pelos amigos de Raquel no **cypher-shell**
Fonte: Elaborado pelo Autor.

Considerando as preferências, podemos pesquisar pelas pessoas que gostam de pizza ou pelas pessoas que gostam de esporte. A Figura 140 mostra a execução da consulta por quem gosta de pizza e outra por quem gosta de esportes.

```
neo4j@neo4j> MATCH (a:Pessoa) -[:GOSTA]-> (b:Comida {Nome: 'Pizza'})
             RETURN (a);
+----------------------------+
| a                          |
+----------------------------+
| (:Pessoa {Nome: "Carlos"}) |
| (:Pessoa {Nome: "José"})   |
+----------------------------+

2 rows available after 19 ms, consumed after another 1 ms
neo4j@neo4j> MATCH (a:Pessoa) -[:GOSTA]-> (b:Esporte)
             RETURN (a);
+----------------------------+
| a                          |
+----------------------------+
| (:Pessoa {Nome: "Mônica"}) |
| (:Pessoa {Nome: "Raquel"}) |
+----------------------------+

2 rows available after 16 ms, consumed after another 0 ms
```

Figura 140 – Consultas por quem gosta de pizza e por quem gosta de esportes
Fonte: Elaborado pelo Autor.

4.3.4 O banco de dados movies

Para continuarmos estudando a linguagem Cypher, vamos utilizar o banco de dados **movies** que é disponibilizado junto com o Neo4j. Esse banco de dados representa um grafo de filmes e pessoas como atores, diretores, etc. Como estamos usando a versão comunitária, não é possível criar múltiplos banco de dados. Dessa forma, antes de criamos o banco de dados **movies**, vamos apagar as informações que já inserimos no banco de dados. A Figura 141 demonstra como podemos fazer isso com as cláusulas **MATCH** e **DELETE** combinadas.

```
neo4j@neo4j> MATCH (a) -[r]- (b) DELETE r;
0 rows available after 8 ms, consumed after another 0 ms
Deleted 11 relationships
neo4j@neo4j> MATCH (n) DELETE n;
0 rows available after 2 ms, consumed after another 0 ms
Deleted 9 nodes
```

Figura 141 – Instruções para apagar todos os dados do banco de dados
Fonte: Elaborado pelo Autor.

A criação do banco de dados **movies** pode ser feita pelo navegador usando a instrução **:play movies** como mostrado na Figura 142. Essa instrução abre um tutorial que apresenta instruções para criação do banco de dados e e diversas consultas.

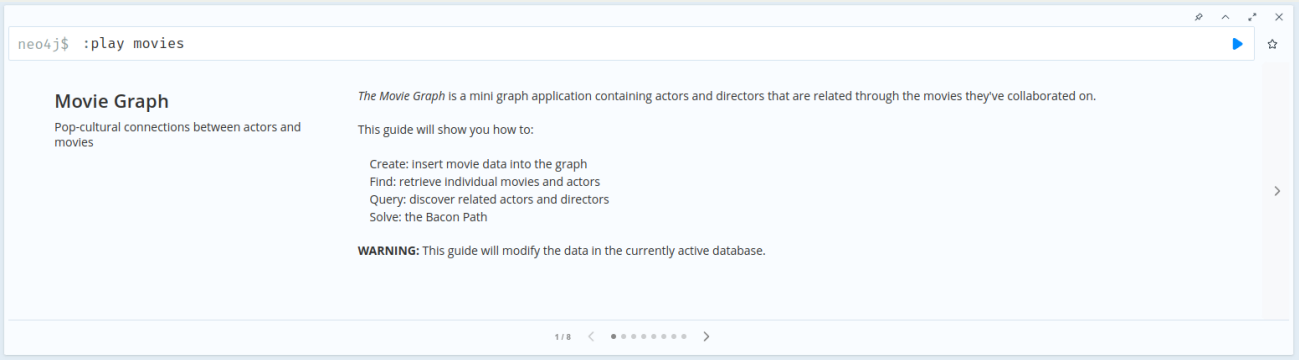

Figura 142 – Execução da instrução **:play movies** no navegador
Fonte: Elaborado pelo Autor.

Ao abrir o tutorial, mude para a segunda página que contém as instruções de criação do banco de dados. Na segunda página, mostrada na Figura 143, clique sobre o botão de *play* para executar as instruções de execução.

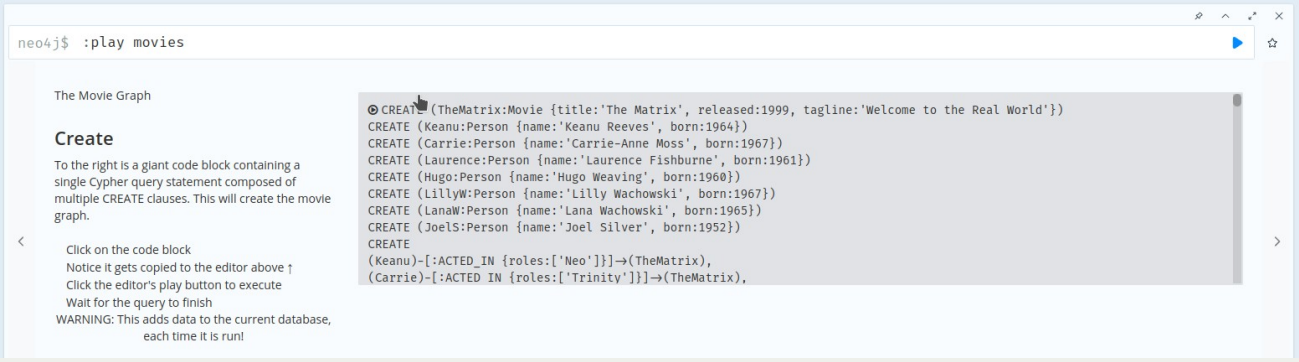

Figura 143 – Página com as instruções de criação do banco de dados **movies**
Fonte: Elaborado pelo Autor.

A Figura 144 mostra o resultado da execução das instruções de criação. Além das instruções de criação, a última instrução executada é uma consulta que mostra sobre filmes estrelados por Tom Hanks.

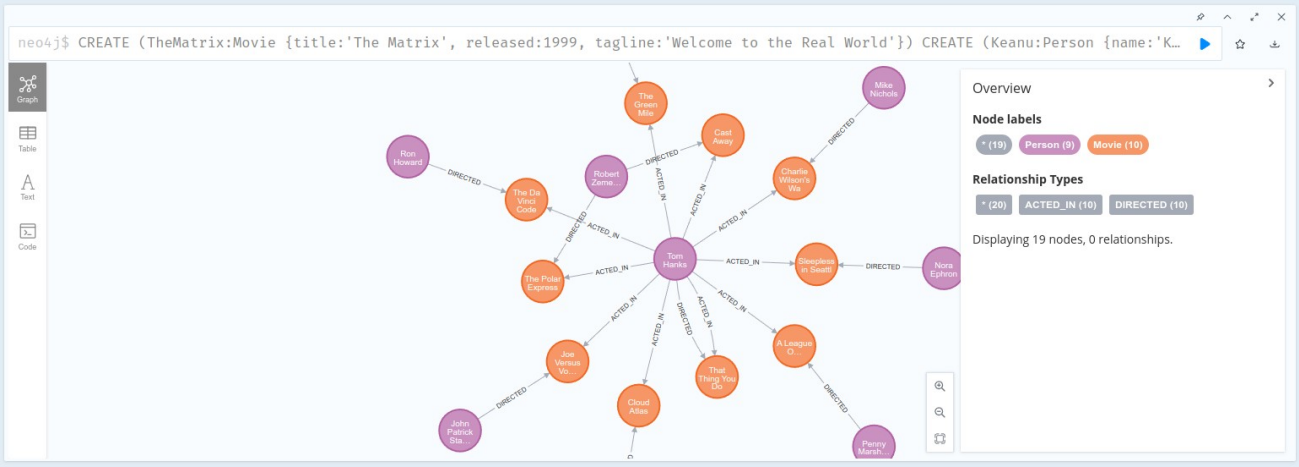

Figura 144 – Criação do banco de dados **movies**
Fonte: Elaborado pelo Autor.

O navegador mostra também o grafo relacionado da última consulta realizada e algumas informações no painel *Overview*. Podemos observar que os nós são rotulados como **Person** (pessoas) e **Movies** (filmes). Quanto as relacionamentos, aparecem os tipos

ACTED_IN (ligação entre atores e filmes que atuaram) e **DIRECTED** (ligação entre uma pessoa e o filme que ela dirigiu).

4.3.5 Consultas no banco de dados movies

Com a criação do banco de dados **movies** podemos explorar um pouco mais as possibilidades de consultas em bancos de dados de grafos. Como não é aconselhável executarmos consultas que retornem todos os nós de um banco de dados, podemos usar a cláusula **LIMIT** para limitar o número de resultados a serem retornados.

```
neo4j@neo4j> MATCH (n:Person) RETURN n LIMIT 5;
+--------------------------------------------------+
| n                                                |
+--------------------------------------------------+
| (:Person {name: "Richard Harris", born: 1930})   |
| (:Person {name: "Clint Eastwood", born: 1930})   |
| (:Person {name: "Takeshi Kitano", born: 1947})   |
| (:Person {name: "Dina Meyer", born: 1968})       |
| (:Person {name: "Ice-T", born: 1958})            |
+--------------------------------------------------+
```

Figura 145 – Consulta com limite no número de resultados a serem retornados
Fonte: Elaborado pelo Autor.

A cláusula **LIMIT** pode ser mais útil ainda quando estamos trabalhando com relacionamentos. Na prática, quantidade de relacionamento tende a ser maior do que a quantidade de nós. A Figura 146 mostra um exemplo de consulta com relacionamentos e limite no número de resultados. Outro detalhe que podemos observar nessa consulta é função **TYPE()** usada para mostrar o tipo de relacionamento entre os nós.

```
neo4j@neo4j> MATCH (p:Person)-[r]->(m:Movie)
             RETURN p.name, TYPE(r), m.title  LIMIT 10;
+-----------------------------------------------------+
| p.name             | TYPE(r)    | m.title           |
+-----------------------------------------------------+
| "Richard Harris"   | "ACTED_IN" | "Unforgiven"      |
| "Clint Eastwood"   | "DIRECTED" | "Unforgiven"      |
| "Jessica Thompson" | "REVIEWED" | "Unforgiven"      |
| "Clint Eastwood"   | "ACTED_IN" | "Unforgiven"      |
| "Gene Hackman"     | "ACTED_IN" | "Unforgiven"      |
| "Keanu Reeves"     | "ACTED_IN" | "Johnny Mnemonic" |
| "Takeshi Kitano"   | "ACTED_IN" | "Johnny Mnemonic" |
| "Ice-T"            | "ACTED_IN" | "Johnny Mnemonic" |
| "Robert Longo"     | "DIRECTED" | "Johnny Mnemonic" |
| "Dina Meyer"       | "ACTED_IN" | "Johnny Mnemonic" |
+-----------------------------------------------------+
```

Figura 146 – Consulta com relacionamentos e limite no número de resultados
Fonte: Elaborado pelo Autor.

Considere agora uma consulta para buscar os filmes nos quais Tom Hanks participou. A Figura 147 mostra algumas consultas relacionadas a esses filmes. Na primeira delas, tentamos relacionar o nó de Tom Hanks aos filmes. Contudo, alguns filmes são mostrados duas vezes. Isso acontece porque Tom Hanks atuou como diretor e ator no

filme *That Thing You Do*. Para resolver esse problema, repetimos a consulta incluindo a palavra **DISTINCT** na cláusula **RETURN** para eliminar as duplicatas do resultado. Na segunda consulta movemos o filtro no nome Tom Hanks para a cláusula **WHERE** apenas para mostrar que a consulta pode ser feita de outra forma.

```
neo4j@neo4j> MATCH (:Person {name: 'Tom Hanks'})--(movie:Movie)
             RETURN movie.title;
+--------------------------+
| movie.title              |
+--------------------------+
| "You've Got Mail"        |
| "The Polar Express"      |
| "A League of Their Own"  |
| "The Da Vinci Code"      |
| "Apollo 13"              |
| "Joe Versus the Volcano" |
| "The Green Mile"         |
| "Charlie Wilson's War"   |
| "Cloud Atlas"            |
| "Cast Away"              |
| "That Thing You Do"      |
| "That Thing You Do"      |
| "Sleepless in Seattle"   |
+--------------------------+
neo4j@neo4j> MATCH (p:Person)--(movie:Movie)
             WHERE p.name = 'Tom Hanks'
             RETURN DISTINCT movie.title;
+--------------------------+
| movie.title              |
+--------------------------+
| "You've Got Mail"        |
| "The Polar Express"      |
| "A League of Their Own"  |
| "The Da Vinci Code"      |
| "Apollo 13"              |
| "Joe Versus the Volcano" |
| "The Green Mile"         |
| "Charlie Wilson's War"   |
| "Cloud Atlas"            |
| "Cast Away"              |
| "That Thing You Do"      |
| "Sleepless in Seattle"   |
+--------------------------+
```

Figura 147 – Consulta com filmes nos quais Tom Hanks participou
Fonte: Elaborado pelo Autor.

Outra consulta relacionada com Tom Hanks participou seria mostrar os tipos e quantidades de participações do artista em filmes. A Figura 148 apresenta uma consulta para buscas essas informações. Observe que, agora, utilizamos a função **TYPE()** para obter o tipo de participação e a função **COUNT()** para contar a quantidade de cada participação.

```
neo4j@neo4j> MATCH (p:Person)-[r]-()
             WHERE p.name = 'Tom Hanks'
             RETURN TYPE(r), COUNT(*);
+-----------------------+
| TYPE(r)    | COUNT(*) |
+-----------------------+
| "ACTED_IN" | 12       |
| "DIRECTED" | 1        |
+-----------------------+
```

Figura 148 – Consulta com a quantidade de tipos de participações de Tom Hanks em filmes
Fonte: Elaborado pelo Autor.

A função **COUNT()** é uma função de agregação. As funções de agregação são úteis para computar operações para cada valor de campo. Em nosso exemplo, contamos quantos filmes para cada tipo de relacionamento como nó que representa Tom Hanks. O Neo4j possui diversas outras funções de agregação interessantes. As principais delas são as seguintes:

- **AVG()**: média de valores numéricos;
- **COUNT()**: Contagem de valores;
- **MAX()**: Valor máximo;
- **MIN():** Valor mínimo;
- **SUM()**: Soma valores numéricos.

Uma exemplo interessante de aplicação de funções de agregação seria sobre a avaliação de usuários sobre filmes. Essas avaliações ficam armazenadas na propriedade **rating** dos relacionamento do tipo **REVIEWED**. A Figura 149 mostra como podemos usar essa dado para calcular as notas média, máxima e mínima dos filmes avaliados. Na cláusula **RETURN** renomeamos os campos retornados com a palavra chave **AS** para que o resultado fique com melhor apresentação.

```
neo4j@neo4j> MATCH () -[r:REVIEWED]- (m:Movie)
             RETURN m.title AS filme, AVG(r.rating) AS media,
                    MIN(r.rating) AS minima, MAX(r.rating) AS maxima;
+-----------------------------------------------------------------+
| filme                | media              | minima | maxima |
+-----------------------------------------------------------------+
| "Unforgiven"         | 85.0               | 85     | 85     |
| "Cloud Atlas"        | 95.0               | 95     | 95     |
| "The Da Vinci Code"  | 66.5               | 65     | 68     |
| "Jerry Maguire"      | 92.0               | 92     | 92     |
| "The Replacements"   | 75.66666666666667  | 62     | 100    |
| "The Birdcage"       | 45.0               | 45     | 45     |
+-----------------------------------------------------------------+
```

Figura 149 – Consulta com avaliação média, mínima e máxima de filmes
Fonte: Elaborado pelo Autor.

Na cláusula **WHERE**, podemos ter várias condições combinadas com os operadores lógicos **AND**, **OR** e **NOT**. Como exemplo, vamos considerar a consulta da Figura 150 que busca pelo nome e ano de nascimento de diretores que nasceram a partir de 1960 e

dirigiram filmes nos quais Tom Hanks atuou. Nesse caso, como desejamos que a consulta atenda a duas restrições, utilizamos o operador **AND**.

```
neo4j@neo4j> MATCH (th) -[:ACTED_IN]-> (m) <-[:DIRECTED]- (d)
             WHERE th.name = 'Tom Hanks' AND d.born >= 1960
             RETURN DISTINCT d.name, d.born;
+---------------------------+
| d.name            | d.born |
+---------------------------+
| "Tom Tykwer"      | 1965   |
| "Lana Wachowski"  | 1965   |
| "Lilly Wachowski" | 1967   |
+---------------------------+
```

Figura 150 – Consulta com diretores que nasceram desde 1960 e dirigiram filmes atuados por Tom Hanks
Fonte: Elaborado pelo Autor.

Outro exemplo com os operadores lógicos pode ser visto na primeira consulta da Figura 151. Essa consulta busca pessoas que atuaram como ator ou diretor no filme *That Thing You Do*. Observe que foi necessário usar parênteses para testar se a pessoa atuou como ator ou diretor. A segunda consulta também obtém o mesmo resultado, mas o filtro de atuar como ator ou diretor foi movido para a cláusula **MATCH**. Nesse caso, o símbolo **|** funciona como o operador **OR** na cláusula **MATCH**.

```
neo4j@neo4j> MATCH (m)<-[r]-(p)
             WHERE m.title = 'That Thing You Do'
               AND (TYPE(r) = 'ACTED_IN' OR TYPE(r) = 'DIRECTED')
             RETURN p.name, TYPE(r);
+--------------------------------+
| p.name            | TYPE(r)    |
+--------------------------------+
| "Liv Tyler"       | "ACTED_IN" |
| "Tom Hanks"       | "DIRECTED" |
| "Charlize Theron" | "ACTED_IN" |
| "Tom Hanks"       | "ACTED_IN" |
+--------------------------------+

neo4j@neo4j> MATCH (m)<-[r:ACTED_IN|DIRECTED]-(p)
             WHERE m.title = 'That Thing You Do'
             RETURN p.name, TYPE(r);
+--------------------------------+
| p.name            | TYPE(r)    |
+--------------------------------+
| "Liv Tyler"       | "ACTED_IN" |
| "Tom Hanks"       | "DIRECTED" |
| "Charlize Theron" | "ACTED_IN" |
| "Tom Hanks"       | "ACTED_IN" |
+--------------------------------+
```

Figura 151 – Consulta por pessoas que atuaram como ator ou diretor em *That Thing You Do*
Fonte: Elaborado pelo Autor.

Na cláusula **WHERE** nos podemos especificar também condições usando padrões assim como é feito na cláusula **MATCH**. Considere como exemplo uma consulta para buscar pelas pessoas que não atuaram nos mesmos filmes em que *Tom Hanks* atuou. A

Figura 152 mostra como essa consulta pode ser feita. Incluímos a cláusula **LIMIT** na consulta porque o número de pessoas no resultado é muito grande.

```
neo4j@neo4j> MATCH (t:Person) -[:ACTED_IN]-> (m:Movie), (p:Person)
             WHERE t.name='Tom Hanks'
               AND NOT (p) -[:ACTED_IN]-> (m)
             RETURN p
             LIMIT 10;
+-----------------------------------------------+
| p                                             |
+-----------------------------------------------+
| (:Person {name: "Richard Harris", born: 1930}) |
| (:Person {name: "Clint Eastwood", born: 1930}) |
| (:Person {name: "Takeshi Kitano", born: 1947}) |
| (:Person {name: "Dina Meyer", born: 1968})     |
| (:Person {name: "Ice-T", born: 1958})          |
| (:Person {name: "Robert Longo", born: 1953})   |
| (:Person {name: "Halle Berry", born: 1966})    |
| (:Person {name: "Jim Broadbent", born: 1949})  |
| (:Person {name: "Tom Tykwer", born: 1965})     |
| (:Person {name: "David Mitchell", born: 1969}) |
+-----------------------------------------------+
```

Figura 152 – Pessoas que não atuaram nos mesmos filmes em que *Tom Hanks* atuou
Fonte: Elaborado pelo Autor.

Em muitas consultas pode ser necessário ordenar o resultado. Nessas situações, podemos usar a cláusula **ORDER BY** após a cláusula **RETURN**. Na cláusula **ORDER BY**, informamos quais campos dever se considerados na ordenação do resultado e também se a ordenação deve ser decrescente (**DESC**). A Figura 153 mostra um exemplo de consulta onde buscamos pelo nome e quantidade de vezes que uma pessoa dirigiu um filmes. Os resultados são ordenados em ordem decrescente pela quantidade de vezes e depois, em ordem crescente, pelo nome da pessoa.

```
neo4j@neo4j> MATCH (p) -[r:DIRECTED]-> (m)
             RETURN p.name, COUNT(*)
             ORDER BY COUNT(*) DESC, p.name
             LIMIT 10;
+-----------------------------------+
| p.name                 | COUNT(*) |
+-----------------------------------+
| "Lana Wachowski"       | 5        |
| "Lilly Wachowski"      | 5        |
| "Rob Reiner"           | 3        |
| "Ron Howard"           | 3        |
| "James Marshall"       | 2        |
| "Mike Nichols"         | 2        |
| "Nora Ephron"          | 2        |
| "Robert Zemeckis"      | 2        |
| "Cameron Crowe"        | 1        |
| "Chris Columbus"       | 1        |
+-----------------------------------+
```

Figura 153 – Top 10 diretores
Fonte: Elaborado pelo Autor.

Até o momento não incluímos filtros sobre as funções de agregação. Quando isso for necessário, precisamos usar a cláusula **WITH** para renomear o campo com a função de agregação e depois filtrar na cláusula **WHERE**. Como exemplo considere a consulta para buscar as pessoas que dirigiram dois ou mais filmes mostrada na Figura 154.

```
neo4j@neo4j> MATCH (p) -[:DIRECTED]-> (m)
             WITH p, COUNT(*) AS quant
             WHERE quant >= 2
             RETURN p.name, quant;
+---------------------------+
| p.name             | quant |
+---------------------------+
| "Lana Wachowski"   | 5     |
| "Lilly Wachowski"  | 5     |
| "Ron Howard"       | 3     |
| "James Marshall"   | 2     |
| "Robert Zemeckis"  | 2     |
| "Mike Nichols"     | 2     |
| "Rob Reiner"       | 3     |
| "Nora Ephron"      | 2     |
+---------------------------+
```

Figura 154 – Pessoas que dirigiram dois ou mais filmes
Fonte: Elaborado pelo Autor.

4.4 Neo4j e Python

A biblioteca oficial para acessar o Neo4j, denominada **neo4j**, oferece recursos básicos de conexão e execução de instruções na linguagem Cypher. Existem também algumas bibliotecas livres desenvolvidas pela comunidade de desenvolvedores que possuem algumas funcionalidades interessantes que podem auxiliar o desenvolvimento. Dentre elas, destacamos a **py2neo** que pode ser instalada pelo terminal com o comando **pip install py2neo**.

Com a biblioteca instalada, podemos importar a classe **Graph** e criar a conexão como Neo4j informando a senha. A partir de uma conexão, podemos fazer consultas com os objetos atributos **nodes** e **relationships**. Também podemos fazer consultas personalizadas com o método **run()**. A Figura 155 mostra um exemplo de código que conecta co nosso banco de dados de filmes e executa algumas consultas por nós. O resultado da execução do código é mostrado na Figura 156.

As consultas foram feitas usando o método **match()** do objeto **nodes**. Além disso, como elas retornariam vários resultados, utilizamos o método **limit()** para limitar o número de nós a serem exibidos. Observe que, na primeira consulta, imprimimos todo o nó na tela. Nas demais consultas, mostramos apenas as propriedades **name** e **title** dos nós de pessoas e filmes, respectivamente.

```
from py2neo import Graph

graph = Graph(password='neo4j')

print('Pessoas:')
node_list = graph.nodes.match('Person').limit(5)
for node in node_list:
    print('-', node)

print('\nTítulos de filmes:')
node_list = graph.nodes.match('Movie').limit(5)
for node in node_list:
    print('-', node['title'])

print('\nPessoas ordenadas pelo nome:')
node_list = graph.nodes.match('Person').order_by('_.name').limit(5)
for node in node_list:
    print('-', node['name'])
```

Figura 155 – Consultas por nós no banco de dados de filmes em Python
Fonte: Elaborado pelo Autor.

```
Pessoas:
- (_1:Person {born: 1930, name: 'Richard Harris'})
- (_2:Person {born: 1930, name: 'Clint Eastwood'})
- (_4:Person {born: 1947, name: 'Takeshi Kitano'})
- (_5:Person {born: 1968, name: 'Dina Meyer'})
- (_6:Person {born: 1958, name: 'Ice-T'})

Títulos de filmes:
- Unforgiven
- Johnny Mnemonic
- Cloud Atlas
- The Da Vinci Code
- V for Vendetta

Pessoas ordenadas pelo nome:
- Aaron Sorkin
- Al Pacino
- Angela Scope
- Annabella Sciorra
- Anthony Edwards
```

Figura 156 – Resultado das consultas por nós no banco de dados de filmes em Python
Fonte: Elaborado pelo Autor.

As consultas usando o objeto **relationships** precisam partir de algum nó específico. Uma possível exemplo seria buscar os relacionamentos do nó que representa *Tom Hanks*. Primeiro, buscamos pelo nó e depois buscamos pelos relacionamentos do nó, como mostrado no código da Figura 157. A Figura 158 mostra o resultado da execução do código.

O nó **tom** obtido pelo **match()** do objeto **nodes** é usado na consulta com o método **match()** do objeto **relationships**. O resultado da consulta é uma lista de relacionamentos. Usamos o laço de repetição para percorrer os relacionamentos retornados e mostrar o **name** do nó inicial (**start_node**), o tipo do relacionamento e o **title** do nó final (**end_node**).

```
from py2neo import Graph

graph = Graph(password='neo4j')

print('\nRelacionamentos de Tom Hanks algum outro nó:')
tom = graph.nodes.match(name='Tom Hanks').first()
for rel in graph.relationships.match((tom, None)):
    print(rel.start_node['name'], tuple(rel.types())[0],
rel.end_node['title'])
```

Figura 157 – Consulta por relacionamentos do nó *Tom Hanks* em Python
Fonte: Elaborado pelo Autor.

```
Relacionamentos de Tom Hanks algum outro nó:
Tom Hanks ACTED_IN You've Got Mail
Tom Hanks ACTED_IN The Polar Express
Tom Hanks ACTED_IN A League of Their Own
Tom Hanks ACTED_IN The Da Vinci Code
Tom Hanks ACTED_IN Apollo 13
Tom Hanks ACTED_IN Joe Versus the Volcano
Tom Hanks ACTED_IN The Green Mile
Tom Hanks ACTED_IN Charlie Wilson's War
Tom Hanks ACTED_IN Cloud Atlas
Tom Hanks ACTED_IN Cast Away
Tom Hanks DIRECTED That Thing You Do
Tom Hanks ACTED_IN That Thing You Do
Tom Hanks ACTED_IN Sleepless in Seattle
```

Figura 158 – Resultado da consulta por relacionamentos do nó *Tom Hanks* em Python
Fonte: Elaborado pelo Autor.

As consultas mais complexas precisam ser executadas pelo método **run()**. Como exemplo, vamos considerar uma consulta para buscar os 10 primeiros relacionamentos entre pessoas e filmes e outra para listar os filmes com participação de Tom Hanks, sem repetições. A Figura 159 mostra tais consultas podem ser realizadas em Python.

```
from py2neo import Graph

graph = Graph(password='neo4j')

print('\nPessoas relacionadas com filmes:')
consulta = '''MATCH (p:Person)-[r]->(m:Movie)
    RETURN p.name, TYPE(r), m.title
    LIMIT 10;'''
r_list = graph.run(consulta)
for record in r_list:
    print('-', record['p.name'], record['TYPE(r)'], record['m.title'])

print('\nFilmes com algum tipo de participação do Tom Hanks:')
consulta = '''MATCH (:Person {name: 'Tom Hanks'})--(movie:Movie)
    RETURN DISTINCT movie.title;'''
r_list = graph.run(consulta)
for record in r_list:
    print(record)
```

Figura 159 – Consultas mais complexas em Python usando o método **run()**
Fonte: Elaborado pelo Autor.

4.4.1 Aplicativo de rede de amigos

Para entendermos melhor como manipular dados no Neo4j, vamos criar uma aplicação simples sobre um banco de dados de uma rede de amigos. Inicialmente, vamos importar as classes necessários, criar uma variável global para conexão como banco de dados e definir algumas configurações para o banco de dados. A Figura 160 exibe esses passos iniciais e a função **linha()** para escrever linhas na tela.

```
# -*- coding: utf-8 -*-

'''Rede de amigos em Python com Neo4j'''

from py2neo import Graph, Node, Relationship

NEO = Graph(password='neo4j')

# Rótulos de nós
PESSOA = 'Pessoa'
# Tipo de relacionamento
AMIZADE = 'AMIZADE'
# Propriedade de nós
NOME = 'Nome'

def linha(caractere='-'):
    '''Escreve linha na tela'''
    print(caractere * 80)
```

Figura 160 – Início do projeto de rede de amigos
Fonte: Elaborado pelo Autor.

```
def busca_no(nome):
    '''Verifica se um nó já existe'''
    return NEO.nodes.match(PESSOA, Nome=nome).first()

def cria_no(nome):
    '''Cria um nó'''
    nodo = Node(PESSOA, Nome=nome)
    NEO.create(nodo)
    return nodo

def exclui_no(no_pessoa):
    '''Deleta nó e seus relacionamentos'''
    # Apaga relacionamentos
    for link in NEO.relationships.match((no_pessoa, None)):
        NEO.separate(link)
    # Apaga nó
    NEO.delete(no_pessoa)
```

Figura 161 – Funções para busca, criação e exclusão de nós na rede de amigos
Fonte: Elaborado pelo Autor.

A partir da variável **NEO** (conexão com o banco de dados), podemos criar diversas funções para consultar e manipular os dados, como mostrado na Figura 161. A função **busca_no()** procura por um nó que representa uma pessoa a partir de seu nome. Já a função **cria_no()** usa a classe **Node** para criar um novo nó de uma pessoa e retorná-lo.

Observe que o nó instanciado precisa ser gravado no banco de dados com o método **create()**. Por fim, a função **exclui_no()** serve para apagar um nó do banco de dados. Antes de apagar o nó, a função remove todos os seus relacionamentos usando o método **separate()**. Se isso não for feito, ocorre um erro ao tentarmos remover um nó que possua relacionamentos.

Considerando que já temos como criar pessoas para nossa rede, vamos implementar as funções para gerenciar as amizades. As amizades serão representadas por relacionamentos entre os nós. Basicamente, vamos escrever uma função para criar e outra para excluir amizades. A Figura 162 apresenta essas duas funções.

```
def cria_amizade(no_pessoa, nome_amigo):
    '''Cria um relacionamento'''
    # Cria o nó amigo
    no_amigo = busca_no(nome_amigo)
    if no_amigo is None:
        no_amigo = cria_no(nome_amigo)
    # Busca pelo relacionamento (nos dois sentidos)
    link1 = NEO.match((no_pessoa, no_amigo), AMIZADE).first()
    link2 = NEO.match((no_amigo, no_pessoa), AMIZADE).first()
    # Verifica se os relacionamentos não existem
    if link1 is None and link2 is None:
        # Cria o relacionamento
        link = Relationship(no_pessoa, AMIZADE, no_amigo)
        NEO.create(link)

def exclui_amizade(no_pessoa, nome_amigo):
    '''Deleta um relacionamento'''
    no_amigo = busca_no(nome_amigo)
    if no_amigo is not None:
        # Busca e exclui amizades de ida e de volta
        link = NEO.match((no_pessoa, no_amigo), AMIZADE).first()
        if link is not None:
            NEO.separate(link)
        link = NEO.match((no_amigo, no_pessoa), AMIZADE).first()
        if link is not None:
            NEO.separate(link)
```

Figura 162 – Funções para criar e excluir amizades na rede de amigos
Fonte: Elaborado pelo Autor.

A função **cria_amizade()** recebe um nó de pessoa e o nome do novo amigo. Primeiro, verificamos se o nó que representa o amigo já existe usando a função **busca_no()**. Caso não exista, criamos um novo nó para representar o amigo. Em seguida, verificamos se já existe a relação de amizade. Observe que estamos considerando amizade recíproca. Como no Neo4j, os relacionamentos são todos direcionados, temos que buscar pela amizade nos dois sentidos. Por fim, se a amizade não for encontrada, criamos um novo relacionamento entre os nós.

A função **exclui_amizade()** recebe também o nó da pessoa e nome do amigo a ser excluído. Inicialmente, buscamos pelo nó do amigo. Se encontramos, buscamos pelo relacionamento de amizade nos dois sentidos e os apagamos.

Agora precisamos de uma função que busque a liste de amigos de uma pessoa e conte quantos aqui essa pessoa tem. A Figura 163 traz a função **mostra_amigos()** responsável por essa tarefa. Preferimos utilizar uma consulta Cypher porque já pegamos as amizades de ida e volta e ordenamos os amigos pelo nome.

```
def mostra_amigos(no_pessoa):
    '''Mostra amigos de um nó'''
    # Cria uma lista com nomes
    consulta = '''MATCH (a:Pessoa)-[:AMIZADE]-(b:Pessoa)
        WHERE a.Nome = '{nome}'
        RETURN b.Nome
        ORDER BY b.Nome'''.format(nome=no_pessoa[NOME])
    lista_rec = NEO.run(consulta)
    lista_nomes = [rec['b.Nome'] for rec in lista_rec]
    # Exibe a lista
    print(len(lista_nomes), 'amigos:')
    linha()
    print(', '.join(lista_nomes))
```

Figura 163 – Função para mostrar os amigos de uma pessoa na rede de amigos
Fonte: Elaborado pelo Autor.

Com as funções de manipulação e consulta de dados prontas, podemos passar para as funções de interação com o usuário. Vamos implementar as funções **menu_incluir()** e **menu_excluir()** para incluir e excluir pessoas, respectivamente. A Figura 164 mostra o código dessas funções.

```
def menu_incluir(nome):
    '''Inclui pessoa'''
    print('Pessoa não encontrada!')
    resp = input('Deseja incluir (S/N) ').strip().lower()
    if resp == 's':
        cria_no(nome)

def menu_excluir(no_pessoa):
    '''Exluir nó pessoa'''
    resp = input('Deseja realmente excluir (S/N) ').strip().lower()
    if resp == 's':
        # Exclui e confirma exclusão
        exclui_no(no_pessoa)
        return True
    # Não excluiu
    return False
```

Figura 164 – Funções para os menus de incluir e excluir pessoas na rede de amigos
Fonte: Elaborado pelo Autor.

A função **menu_incluir()** recebe um nome de pessoa e confirma se o usuário deseja incluir. Em caso afirmativo, a função **cria_no()** é chamada para incluir a pessoa. A função **menu_excluir()** recebe um nó de uma pessoa, pergunta se o usuário deseja realmente excluir e chama a função **exclui_no()** para efetivar a remoção. A função **menu_excluir()** retorna **True** se o usuário confirmar a exclusão e **False**, em caso afirmativo.

Agora podemos criar uma função para mostrar os dados de uma pessoa, exibir um menu de opções para o usuário e chamar as funções apropriadas de acordo com a escolha feita pelo usuário. Tal função terá o nome de **menu_pessoa()** e seu código é mostrado na Figura 165.

```
def menu_pessoa(no_pessoa):
    '''Menu de amigos'''
    while True:
        linha('=')
        print('Rede de amigos')
        linha('=')
        print(no_pessoa[NOME])
        linha()
        mostra_amigos(no_pessoa)
        linha()
        print('(+|-) Amigo | (E)xcluir | (S)air')
        resp = input('Opção desejada: ').strip()
        if resp.lower() == 's':
            break
        if resp == 'e':
            if menu_excluir(no_pessoa):
                break
        nome_amigo = resp[1:]
        if resp[0] == '+':
            cria_amizade(no_pessoa, nome_amigo)
        if resp[0] == '-':
            exclui_amizade(no_pessoa, nome_amigo)
```

Figura 165 – Função com menu de opções para uma pessoa na rede de amigos
Fonte: Elaborado pelo Autor.

A função possui um laço infinito que se repete até que o usuário deseje sair do menu de opções da pessoa ou exclua a pessoa atual. A cada iteração, o laço exibe o nome da pessoa, mostra seus amigos e exibe o menu com as opções de incluir ou excluir amigo, excluir a pessoa atual ou sair. A incluir ou exclusão de amigos pelo usuário é feita com o símbolo **+** ou **-**, respectivamente, seguido pelo nome do amigo.

Para finalizamos o nosso projeto, vamos criar uma rotina de menu principal e chamá-la no **main** do arquivo. A Figura 166 exibe o referido código. A função **menu()** possui um laço infinito para mostrar a lista de pessoas da rede de amigos. O usuário deve digitar o nome de uma pessoa ou pressionar **S** para interromper o laço e encerrar a execução.

Quando um nome de pessoa é digitado, nós buscamos peno nó dessa pessoa usando a função **busca_no()**. Se a pessoa for encontrada, nós chamamos a função **menu_pessoa()** para mostrar as opções para a pessoa informada. Se a pessoa não existir, nós executamos a função **menu_incluir()** para perguntar se o usuário deseja incluir uma nova pessoa.

```
def menu():
    '''Menu principal'''
    while True:
        linha('=')
        print('Rede de amigos')
        linha('=')
        mostra_pessoas()
        linha('=')
        resp = input('(S)air ou o nome de uma pessoa: ').strip()
        if resp.lower() == 's':
            break
        no_pessoa = busca_no(resp)
        if no_pessoa is None:
            menu_incluir(resp)
        else:
            menu_pessoa(no_pessoa)

if __name__ == '__main__':
    menu()
```

Figura 166 – Função com menu principal e **main** da rede de amigos
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

4.5 Exercício

Escreva os códigos em Python para implementar uma aplicação de controle de árvore genealógica com o banco de dados armazenado no Neo4j. Considere os seguintes pontos:

- Faça as importações necessárias para trabalhar com o Neo4j e declare variáveis globais para serem usadas como rótulo dos nós (Pessoa), tipo de relacionamento (PAI) e propriedade dos nós (Nome);

- Implemente as funções **busca_no(nome)** e **cria_no(nome)** para buscar e criar um novo nó de pessoa, respectivamente. A rotina **cria_no()** deve retornar o nó criado;

- Escreva a função **exclui_no(no_pessoa)** para remover um nó de pessoa. Não esqueça excluir os relacionamentos do nó antes de removê-lo;

- Crie as funções **insere_filho(no_pessoa, nome_filho)** e **remove_filho(no_pessoa, nome_filho)** para inserir ou excluir o filho de uma pessoa. Na inserção do filho, verifique se já existe um nó com o nome do filho. Se não houver, crie esse nosso. Depois insira um relacionamento de PAI do **no_pessoa** para o nó do filho. Na remoção, verifique se o nó correspondente ao nome existe e está como filho do no_pessoa. Em caso afirmativo, remova o relacionamento entre eles;

- Elabore as funções **busca_filhos(no_pessoa)**, para buscar os filhos de **no_pessoa**, e **busca_pais(no_pessoa)** para retornar os mais de **no_pessoa**. Utilize os relacionamentos do tipo PAI para fazer as buscas;

- Faça as funções **busca_ancestrais(no_pessoa)** e **busca_descendentes(no_pessoa)** para buscar os ancestrais e descendente de **no_pessoa**, respectivamente. Utilize recursão e as funções já criadas para buscar pais e filhos para realizar essas tarefas;

- Implemente a função **mostra_pessoa(no_pessoa)** para mostrar o nome, ancestrais e descendentes de uma pessoa. Implemente também a função **menu_incluir(nome)** para confirmar com o usuário e inclui um novo nó de pessoa a partir do **nome**;

- Escreva a função **menu_pessoa(no_pessoa)** para receber nó de uma pessoa, mostrar suas informações e exibir um menu de opções para essa pessoa. As informações da pessoa podem ser mostrada chamando a função **mostra_pessoa()**. O menu deve incluir as opções de incluir e excluir filhos, excluir a pessoa e sair do menu. Após o usuário escolher a opção desejada, chame a função apropriada;

- Por fim, crie a função **menu()** com o menu principal do programa e o chame no **main** do arquivo. O menu principal deve mostra a lista de pessoas já cadastradas, e pedir para o

usuário informar o nome de uma pessoa ou escolher a opção de sair. Quando o usuário entrar com um nome, a função deve verificar se o nó correspondente exite. Em caso afirmativo, deve ser chamada a função **menu_pessoa()** para esse nó. Em caso negativo, a função **menu_incluir()** deve ser invocada para perguntar se o usuário deseja incluir uma pessoa com o nome informado.

4.6 Resposta dos exercício

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'''Árvore genealógica com Neo4j e Python'''

from py2neo import Graph, Node, Relationship

NEO = Graph(password='neo4j')

# Rótulos de nós
PESSOA = 'Pessoa'
# Tipos de relacionamentos
PAI = 'PAI'
# Propriedade de nós
NOME = 'Nome'
# NASCIMENTO = 'Nascimento'


def linha(caractere='-'):
    '''Escreve linha na tela'''
    print(caractere * 50)

def busca_no(nome):
    '''Verifica se um nó já existe'''
    return NEO.nodes.match(PESSOA, Nome=nome).first()

def cria_no(nome):
    '''Cria um nó'''
    if len(nome) == 0:
        return None
    nodo = Node(PESSOA, Nome=nome)
    NEO.create(nodo)
    return nodo

def exclui_no(no_pessoa):
    '''Deleta nó e seus relacionamentos'''
    # Apaga relacionamentos
    for link in NEO.relationships.match((no_pessoa, None)):
        NEO.separate(link)
    # Apaga nó
    NEO.delete(no_pessoa)

def insere_filho(no_pessoa, nome_filho):
    '''Insere ligações entre pai e filho'''
    no_filho = NEO.nodes.match(PESSOA, Nome=nome_filho).first()
    if no_filho is None:
        no_filho = cria_no(nome_filho)
    link = Relationship(no_pessoa, PAI, no_filho)
    NEO.merge(link)

def remove_filho(no_pessoa, nome_filho):
```

```
52      '''Remove ligações entre pai e filho'''
53      no_filho = NEO.nodes.match(PESSOA, Nome=nome_filho).first()
54      if no_filho is not None:
55          link = NEO.relationships.match((no_pessoa, no_filho),
56                                                  PAI).first()
57          if link is not None:
58              NEO.separate(link)
59
60  def busca_filhos(no_pai):
61      '''Busca nós filhos'''
62      lista_filhos = []
63      for link in NEO.relationships.match((no_pai, None), PAI):
64          lista_filhos.append(link.end_node)
65      return lista_filhos
66
67  def busca_pais(no_filho):
68      '''Busca nós pais'''
69      lista_pais = []
70      for link in NEO.relationships.match((None, no_filho), PAI):
71          lista_pais.append(link.start_node)
72      return lista_pais
73
74  def mostra_lista_pessoas():
75      '''Lista pessoas'''
76      for no_pessoa in NEO.nodes.match(PESSOA).order_by('_.Nome'):
77          print(no_pessoa[NOME])
78
79  def mostra_ancestrais(no_pessoa, nivel=0):
80      '''Lista ancestrais'''
81      # Busca pessoa
82      if no_pessoa is not None:
83          # Imprime o nome da pessoa
84          print('  '*nivel, no_pessoa[NOME], sep='')
85          lista_pais = busca_pais(no_pessoa)
86          # Busca resursiva pelos ancestrais
87          for no_pai in lista_pais:
88              mostra_ancestrais(no_pai, nivel+1)
89
90  def mostra_descendentes(no_pessoa, nivel=0):
91      '''Lista ancestrais'''
92      # Busca pessoa
93      if no_pessoa is not None:
94          # Imprime o nome da pessoa
95          print('  '*nivel, no_pessoa[NOME], sep='')
96          lista_filhos = busca_filhos(no_pessoa)
97          # Busca resursiva pelos ancestrais
98          for no_filho in lista_filhos:
99              mostra_descendentes(no_filho, nivel+1)
100
101 def mostra_pessoa(no_pessoa):
102     '''Mostra informações de nó pessoa'''
103     linha('=')
104     print(no_pessoa[NOME])
105     linha('=')
106     print('Ancestrais:')
```

```
107         linha('-')
108         mostra_ancestrais(no_pessoa)
109         linha('-')
110         print('Descendentes:')
111         linha('-')
112         mostra_descendentes(no_pessoa)
113         linha('-')
114
115     def menu_incluir(nome):
116         '''Menu incluir'''
117         print('Pessoa não encontrada!')
118         resp = input('Deseja incluir (S/N) ').strip().lower()
119         if resp == 's':
120             cria_no(nome)
121
122     def menu_pessoa(no_pessoa):
123         '''Menu pessoa'''
124         while True:
125             linha()
126             mostra_pessoa(no_pessoa)
127             linha()
128             print('(+|-) Filho | (E)xcluir | (S)air')
129             resp = input('Opção desejada: ').strip()
130             if resp.lower() == 's':
131                 break
132             if resp.lower() == 'e':
133                 confirma = input('Confirma a exclusão? (S/N): ')
134                 if confirma.strip().lower() == 's':
135                     exclui_no(no_pessoa)
136                 break
137             if resp[0] == '+':
138                 insere_filho(no_pessoa, resp[1:].strip())
139             elif resp[0] == '-':
140                 remove_filho(no_pessoa, resp[1:].strip())
141
142     def menu():
143         '''Menu principal'''
144         while True:
145             linha('=')
146             print('Árvore genealógica')
147             linha('=')
148             mostra_lista_pessoas()
149             linha('=')
150             resp = input('(S)air ou ' +
151                          'o nome de uma pessoa: ').strip()
152             if resp.lower() == 's':
153                 break
154             no_pessoa = busca_no(resp)
155             if no_pessoa is not None:
156                 menu_pessoa(no_pessoa)
157             else:
158                 menu_incluir(resp)
159
160     if __name__ == '__main__':
161         menu()
```

4.7 Revisão

Antes de finalizarmos o curso, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de finalizarmos, vá até a sala virtual e assista ao vídeo “Revisão da Quarta Semana” para recapitular tudo que aprendemos.

Bons estudos!

Finalizando o curso

A prática é uma atividade fundamental para um bom aprendizado da lógica de programação e de qualquer linguagem de programação. Uma boa tarefa prática interessante é revisitar os problemas e exercícios estudados e tentar resolvê-los por conta própria. Além disso, você pode se aprofundar mais em diversos conteúdos usando as referencias citadas no livro.

Recomendamos também que você sempre busque pelo constante aprimoramento profissional. Você pode encontrar diversos conteúdos interessantes na Plataforma +IFMG (https://mais.ifmg.edu.br).

Atividade final

Atividade: Para concluir o curso e gerar o seu certificado, vá até a sala virtual e responda ao Questionário “Avaliação final”. Este teste é constituído por 10 perguntas de múltipla escolha, que se baseiam em todo o conteúdo estudado.

Referências

BORGES, L. E. **Python para Desenvolvedores**. 2. ed. Rio de Janeiro: Edição do Autor, 2010. Disponível em: https://ricardoduarte.github.io/python-para-desenvolvedores/

CARLSON, J. **Redis in Action**. 2013. Disponível em: https://redis.com/ebook/redis-in-action/

CEDER, N. The **Quick Python Book**. 3. ed. Shelter Island: Manning Publications, 2018. Disponível em: https://livebook.manning.com/book/the-quick-python-book-third-edition/

CORRÊA, E. **Meu primeiro livro de Python**. 2. ed. Rio de Janeiro: Edubd, 2020. Disponível em: https://github.com/edubd/meu_primeiro_livro_de_python

CURRY, E. et al. (Ed.). **Technologies and Applications for Big Data Value**. Gewerbestrasse: Springer Nature, 2022. Disponível em: https://doi.org/10.1007/978-3-030-78307-5

DOWNEY, A. B. T**hink Python**: How to Think Like a Computer Scientist. 2. ed. Needham: Green Tea Press, 2015. Disponível em: https://greenteapress.com/wp/think-python-2e/

ELMASRI, R.; NAVATHE, S. B. **Sistemas de banco de dados**. 6. ed. São Paulo: Pearson, 2011.

FULMAŃSKI, P. (Ed.). **NoSQL**: Theory and examples. 2021. Disponível em: https://fulmanski.pl/books/doc/nosql_theory_and_examples_excerpt.pdf

GOALKICKER.COM. **MongoDB**: Notes for Professionals. 2018. Disponível em: https://goalkicker.com/MongoDBBook

MARÇULA, M.; FILHO, P. A. B. **Informática**: Conceitos e Aplicações. 3. ed. São Paulo: Érica, 2008.

MENEZES, N. N. C. **Introdução à programação com Python**: algoritmos e lógica de programação para iniciantes. 3. ed. São Paulo: Novatec, 2019.

PILGRIM, M. **Dive Into Python 3**. New York: Apress, 2009. Disponível em: https://diveintopython3.net/

PYPL. **PYPL**: PopularitY of Programming Language. 2021. Disponível em: https://pypl.github.io/PYPL.html

PYTHON SOFTWARE FOUNDATION (PSF). **Python 3.10.1 documentation**. 2021. Disponível em: https://docs.python.org/

RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de banco de dados**. 3. ed. São Paulo: McGrawHill, 2008.

RAMALHO, L. **Python fluente**: programação clara, concisa e eficaz. São Paulo: Novatec, 2015.

ROBINSON, I.; WEBBER, J.; EIFREM, E. **Graph Databases**. Sebastopol: O'Reilly Media, Inc., 2015. Disponível em: https://neo4j.com/graph-databases-book/

SOTO, S. V.; LUNA, J. M.; CANO, A. (Ed.). **Big Data on Real-World Applications**. London: IntechOpen, 2016. Disponível em: https://doi.org/10.5772/61396

SWEIGART, A. **Beyond the basic stuff with python**: best practices for writing clean code. San Francisco: No Starch Press, 2021. Disponível em: https://inventwithpython.com/beyond/

TAGLIAFERRI, L. **How To Code in Python 3**. New York: DigitalOcean, 2018. Disponível em: https://assets.digitalocean.com/books/python/how-to-code-in-python.pdf

TIOBE. **TIOBE Index for December 2021**. 2021. Disponível em: https://www.tiobe.com/tiobe-index/

VELLOSO, F. **Informática**: conceitos básicos. 9. ed. Rio de Janeiro: Elsevier, 2014.

WIKIPÉDIA. **Algoritmo de Euclides**. 2021. Disponível em: https://pt.wikipedia.org/wiki/Python

WIKIPÉDIA. **Python**. 2021. Disponível em: https://pt.wikipedia.org/wiki/Python

Currículo do autor

Marcos Roberto Ribeiro

O autor possui graduação em Ciência da Computação pelo Centro Universitário de Formiga (2005), mestrado em Ciência da Computação pela Universidade Federal de Uberlândia (2008) e doutorado em Ciência da Computação pela Universidade Federal de Uberlândia (2018). Atua como professor em disciplinas de cursos técnicos e superiores do Instituto Federal Minas Gerais - Campus Bambuí desde 2010. As disciplinas ministradas estão relacionadas principalmente com Programação, Banco de Dados e Desenvolvimento de Sistemas.

Currículo Lattes: http://lattes.cnpq.br/8439091552425995

Glossário de códigos QR (Quick Response)

Mídia digital
Apresentação do curso

Mídia digital
Revisão da primeira semana

Mídia digital
Revisão da segunda semana

Mídia digital
Revisão da terceira semana

Mídia digital
Revisão da quarta semana

Plataforma +IFMG

Formação Inicial e Continuada EaD

A Pró-Reitoria de Extensão (Proex), desde o ano de 2020, concentrou seus esforços na criação do Programa +IFMG. Esta iniciativa consiste em uma plataforma de cursos *online*, cujo objetivo, além de multiplicar o conhecimento institucional em Educação à Distância (EaD), é aumentar a abrangência social do IFMG, incentivando a qualificação profissional. Assim, o programa contribui para o IFMG cumprir seu papel na oferta de uma educação pública, de qualidade e cada vez mais acessível.

Para essa realização, a Proex constituiu uma equipe multidisciplinar, contando com especialistas em educação, *web design*, *design* instrucional, programação, revisão de texto, locução, produção e edição de vídeos e muito mais. Além disso, contamos com o apoio sinérgico de diversos setores institucionais e também com a imprescindível contribuição de muitos servidores (professores e técnico-administrativos) que trabalharam como autores dos materiais didáticos, compartilhando conhecimento em suas áreas de atuação.

A fim de assegurar a mais alta qualidade na produção destes cursos, a Proex adquiriu estúdios de EaD, equipados com câmeras de vídeo, microfones, sistemas de iluminação e isolação acústica, para todos os 18 *campi* do IFMG.

Somando à nossa plataforma de cursos *online*, o Programa +IFMG disponibilizará também, para toda a comunidade, uma Rádio *Web* Educativa, um aplicativo móvel para Android e iOS, um canal no Youtube com a finalidade de promover a divulgação cultural e científica e cursos preparatórios para nosso processo seletivo, bem como para o Enem, considerando os saberes contemplados por todos os nossos cursos.

Parafraseando Freire, acreditamos que a educação muda as pessoas e estas, por sua vez, transformam o mundo. Foi assim que o +IFMG foi criado.

O +IFMG significa um IFMG cada vez mais perto de você!

Professor Carlos Bernardes Rosa Jr.
Pró-Reitor de Extensão do IFMG

Características deste livro:
Formato: A4
Tipologia: Arial e Capriola.
E-book:
1ª. Edição
Formato digital