ESCOLA DE ENGENHARIA DE LORENA
EEL-USP – LORENA

CAIO PAGES CAMARGO

CRIAÇÃO DE UM *WEBSCRAPER* PARA OBTENÇÃO AUTOMÁTICA DAS GRADES CURRICULARES DOS CURSOS OFERECIDOS NA UNIDADE EEL-USP

LORENA
2020
Caio Pages Camargo

CRIAÇÃO DE UM *WEBSCRAPER* PARA OBTENÇÃO AUTOMÁTICA DAS GRADES CURRICULARES DOS CURSOS OFERECIDOS NA UNIDADE EEL-USP

Trabalho de conclusão de curso servindo como exigência parcial para obtenção do grau de bacharel em Engenharia Física na unidade Escola de Engenharia de Lorena da Universidade de São Paulo (EEL-USP) .
Orientador: Prof. Luiz Tadeu Fernandes Eleno.

Lorena
2020

Ficha catalográfica elaborada pelo Sistema Automatizado da Escola de Engenharia de Lorena,
com os dados fornecidos pelo(a) autor(a)

Camargo, Caio Pages Criação de um Webscraper para Obtenção Automática das Grades Curriculares dos Cursos Oferecidos na Unidade EEL-USP / Caio Pages Camargo; orientador Luiz Tadeu Fernandes Eleno. - Lorena, 2020.
46 p.

Monografia apresentada como requisito parcial para a conclusão de Graduação do Curso de Engenharia Física - Escola de Engenharia de Lorena da Universidade de São Paulo. 2020

1. Webscraper. 2. Python. 3. Sql. 4. Base de dados. I. Título. II. Eleno, Luiz Tadeu Fernandes, orient.

RESUMO

Jupiterweb é o nome da plataforma virtual utilizada na Universidade de São Paulo (USP) que armazena informações sobre as grades curriculares dos cursos de graduação disponibilizados na unidade Escola de Engenharia de Lorena (EEL-USP). Em algum momento durante a graduação, devido a diversos motivos, alunos encontram a necessidade de planejar mais cuidadosamente as matérias de cada um dos semestres seguintes. Com o objetivo de centralizar as informações sobre cada uma dos cursos e suas grades curriculares, este trabalho visa criar um *webscraper* utilizando a linguagem Python para obter da página *web* do Jupiterweb as informações relevantes de cada matéria de cada um dos cursos disponibilizados na unidade EEL-USP e centralizá-las em uma base de dados criada utilizando a biblioteca SQLite. A base de dados criada contem três tabelas principais: Cursos, Matérias e Requisitos, incluindo toda a informação relevante para os graduandos.

Palavras-chave: *Webscraper*; Python; SQL; Base de dados.

ABSTRACT

Jupiterweb is a virtual platform used in the University of São Paulo (USP) that contains information about the curriculum of each graduation course in the Engineering School of Lorena (EEL-USP). At some point during graduation, due to various reasons, the students experience the need to plan more cautiously the subjects they are going to take each of the remaining semesters of graduation. Pursuing the centralization of the information about each of the graduation courses and its curricula, this work aims to build a *webscraper* using the Python language to obtain from the Jupiterweb webpage the relevant information of each of the subjects in all graduation courses available at EEL-USP and gather them in an SQL database. The resulting database includes three main tables: Courses, Subjects and Requirements, including all the relevant information for the undergraduates.

Keywords: *Webscraper*; Python; SQL; database.

Sumário

1 Introdução

Jupiterweb é o nome da plataforma virtual da Universidade de São Paulo (USP) utilizada por professores e graduandos para diversas atividades que se estendem desde consulta do calendário escolar até emissão de documentos e controle de notas e histórico escolar.

Nesta plataforma também se encontram os cursos disponibilizados por cada unidade da USP, além de suas grades curriculares e informações referentes a cada uma das matérias de cada curso de graduação como ementa, bibliografia, número de créditos e docentes responsáveis.

Durante a graduação muitos graduandos, em algum momento, se deparam com a necessidade de planejar de maneira mais detalhada e cuidadosa as matérias que irão cursar ao longo de cada um dos semestres seguintes, seja devido a um atraso inesperado no curso ou a atividades extracurriculares que demandam mais tempo como uma Iniciação científica ou um estágio. Nesse momento, o acesso a informações de maneira clara, estruturada e centralizada sobre cada uma das matérias, como os requisitos, horários e oferecimento se torna de extrema importância.

Por este motivo, a idéia deste trabalho é utilizar uma técnica chamada webscraping, que utiliza uma rotina de código para parsear (do inglês *parse:* interpretar, analisar) o código HTML de uma página da internet e obter as informações desejadas, para extrair as principais informações referentes a cada matéria de cada curso oferecido pela Escola de Engenharia de Lorena da Universidade de São Paulo (EEL-USP) com o objetivo de centralizar todos esses dados em uma base de dados. Estes dados serão obtidos do portal oficial de alunos e professores da USP, Jupiterweb e, posteriormente, será criada uma base de dados relacional em SQL utilizando a biblioteca SQLite com todas as informações coletadas.

2 REVISÃO BIBLIOGRÁFICA

2.1 SQL (*Structured Query Language*)

2.1.1 História do SQL

SQL (*Structured Query Language*) é uma linguagem que foi desenvolvida para consultar e manipular dados em Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDR). Sistemas Gerenciadores de Banco de Dados Relacionais ou *Relational Database Management Systems (RDBMS)*, são sistemas de gerenciamento de bases de dados baseados no modelo relacional, que é um modelo semântico pare representação de dados baseado em duas teorias: teoria dos conjuntos e lógica de primeira ordem (LPO). (Ben-Gan, 2012)

A história do SQL, inicialmente com o nome *SEQUEL (Structured English Query Language)* devido ao fato da palavra "*SEQUEL"* ser uma marca registrada pela empresa *Hawker Siddeley Dynamics Engineering* na época (Oppel, 2004), começa na IBM, onde foi desenvolvida por Donald D. Chamberlin and Raymond F. Boyce baseando-se no modelo relacional de Ted Codd (Chamberlin, 2012), durante os anos setenta.

Codd, um matemático britânco, havia desenvolvido uma linguagem baseada em notações de lógica formal e operadores matemáticos, que conseguia representar operações complexas de conjuntos de maneira sucinta. Baseando-se nessa notação de símbolos, com o objetivo de deixar essa linguagem mais acessível para aqueles que não tinham familiaridade com matemática, Boyce e Chamberlin transformaram os operadores matemáticos, os quais não eram simples de ser digitados em um teclado comum, em palavras-chave, tornando a linguagem mais acessível e clara.

Dessa maneira, Boyce e Chamberlin chegaram a uma linguagem declarativa, ou seja, uma linguagem que descreve a informação que está procurando mas não especifica um plano detalhado para como chegar à essa informação, como é o caso de uma linguagem processual (Chamberlin, 2012). O trabalho de traduzir a sequência de comandos em um plano de processamento é do compilador de otimização. Essa linguagem declarativa foi batizada como SQL.

No fim da década de setenta uma empresa americana chamada *Relational Software, Inc.* (hoje *Oracle Corporation*) desenvolveu o conceito da *SEQUEL* e criou

sua prória base de dados relacional baseada em SQL com o objetivo de vendê-la para a marinha americana e outras agências governamentais dos Estados Unidos. Em 1979 *Relational Software, Inc.* introduziu ao mundo a primeira versão de SQL disponível comercialmente, denominada *Oracle V2 (Version2)*.

Em 1986 uma definição de linguagem padrão chamada de “Linguagem de Base de Dados SQL” foi adotada pelo grupos de padrões ANSI (American National Standards Institute, organização estadunidense sem fins lucrativos que tem como objetivo facilitar padronizações, e ISO (International Organization of Standardization), entidade que congrega a padronização de 162 países (Chamberlin, 2012). Desde então, novas versões do SQL padrão foram publicada em 1996, 1999, 2003, 2006, 2008, 2011 e 2016.

Os comandos de SQL são divididos em três tipos: *Data Definition Language (DDL)*, comandos que lidam com a definição, criação e deleção de objetos no banco como *CREATE, ALTER* e *DROP*; *Data Manipulation Language (DML)*, comandos que permitem ao usuário consultar, alterar e manipular os dados no banco como *INSERT, UPDATE, JOIN, MERGE* e *DELETE*; e *Data Control Language (DCL)*, comandos que lidam com permissões como *GRANT* e *REVOKE*.

2.1.2 Teoria dos Conjuntos

A teoria de conjuntos, criada por Georg Cantor no ano de 1874 em seu artigo “A respeito de uma propriedade característica de todos os números algébricos reais” (Johnson, 1972), é um dos pilares da liguagem SQL. Cantor define um conjunto como: “Por conjunto entendemos qualquer coleção numa totalidade M de objetos distintos, produtos de nossa intuição ou pensamento.” (Ávila, 2000).

Essa definição, apesar de sucinta, contem muitas sutilezas em cada uma de suas palavras e um sentido profundo. A definição matemática de um conjunto e pertencimento a um conjunto são axiomas, portanto, não sao apoiados por provas matemáticas.

Começando pela expressão “produtos de nossa intuição ou pensamento”, podemos entender como um conjunto qualquer percepção subjetiva de um grupo de coisas. Um exemplo que clarifica esse conceito é o seguinte: se levarmos em conta uma família, podemos pensar na família como um conjunto de pessoas, mas também

podemos pensar como um conjunto de pais e um conjunto de filhos de filhos, ou até como um conjunto de homens e um conjunto de mulheres. Portanto, existe uma liberdade implícita ao definir conjuntos, que podem ser definidos de acordo com qualquer conjunto concebível.

Já a palavra “coleção” é simplesmente uma opção alternativa para a palavra “conjunto”, a expressão “numa totalidade” indica que devemos pensar em conjuntos como um grupo de elementos e não como elementos separados. Por exemplo: se pensarmos em um conjunto que representa os clientes de uma empresa, não devemos pensar nesse conjunto como clientes individuais, mas sim como um conjunto de clientes.

Em relação à palavra “objeto”, pode ser literalmente qualquer coisa, desde objetos físicos como pessoas e carros, como objetos abstratos como números ou curvas.

Por fim, a palavra “distintos” afirma que cada elemento de um conjunto deve ser único, portanto, deve ser possível identificar cada elemento de um conjunto de maneira única.

Um aspecto importante da teoria de conjuntos para a linguagem SQL é o fato dos elementos de um conjunto não serem dispostos em uma ordenação definida. De fato, a ordem dos elementos de um conjunto, além de não ter importância, para ser definido como conjunto, os elementos não podem ter ordenação. Sendo assim, os conjuntos {x,y,z} e {y,z,x} são considerados conjuntos equivalentes.

2.1.3 Lógica de Primeira Ordem (LPO) na Linguagem SQL

Ao conceber a linguagem SQL, Boyce e Chamberlin usaram se basearam em uma notação criada pelo matemático britânico Codd, que era capaz de representar operações complexas entre conjuntos de maneira sucinta. Esta notação de Codd usava de lógica de primeira ordem ou predicate logic, um tipo de lógica usado nos campos de matemática, linguística, filosofia e ciência da computação, para fazer operações entre conjuntos.

Predicados podem ser definidos como expressões que são válidas ou não, ou seja, expressões booleanas que são avaliadas como *true* (verdadeiras) ou *false* (falsas). Apesar dos predicados serem implementados na maior parte das linguagens

de programação como uma lógica de dois valores, *true* ou *false*, na linguagem SQL é implementada uma lógica de três valores: *true*, *false* e *unknown* (desconhecido). (Ben-Gan, 2012)

A avaliação de predicados como true e false são bem intuitivas: se uma expressão avaliada é válida, como a expressão 1 > 0, retorna o valor *true*, enquanto, se uma expressão não é válida, como a expressão 1 < 0, retorna o valor *false*. Para o caso de uma expressão avaliada como unknown, a explicação já não é muito intuitiva. Na linguagem SQL, quando um dado é faltante, ou seja, a informação não foi inserida, é dado a esse valor o marcador “NULL” (nulo) e quando um dado qualquer é comparado com um dado “NULL”, a avaliação resulta em unknown. Isto se dá pois a linguagem SQL trata valores nulos como algo desconhecido, logo, quando algo é comparado a um dado desconhecido, não podemos dizer com certeza se a comparação é verdadeira ou falsa. Devido à falta de intuitividade, esse comportamento pode gerar um alto grau de complexidade e confusão na hora de tratar os dados, principalmente para aqueles que já tem um passado com outras linguagens de programação, onde é usada a lógica de dois valores.

Por exemplo: na lógica de dois valores, tudo que não é verdadeiro necessariamente será falso, enquanto na linguagem SQL, devido à possibilidade de uma expressão ser avaliada como *unknown*, uma expressão que não é verdadeira, não necessariamente será falsa pois pode ser avaliada como *unknown*. Portanto, quando definimos que queremos filtrar apenas os casos cujos predicados não são verdadeiros, serão filtrados todos casos que são avaliados como falsos e *unknown*. O mesmo vale para quando queremos filtrar todos casos cujos predicados não sçao avaliados como falsos: serão retornados todos casos que são verdadeiros e *unknown*.

2.1.4 Normalização de Base de Dados

Normalização é um processo matemático formal que garante que cada entidade será representada por uma única relação (Ben-Gan, 2012). Uma entidade é um objeto na base de dados que pode representrar qualquer coisa, desde departamentos de uma empresa até posições de jogadores de futebol. Esse processo evita redundância enquanto mantem a completidão dos dados, além de evitar anomalias na modificação dos mesmos. A implementação desse processo consiste em seguir três regras de normalização criadas por Codd, que também são chamadas de formas normais (FN).

1ª FN: todos atributos (colunas) de uma tabela devem ser atômicos, ou seja, a tabela não deve conter grupos repetidos ou atributos com mais de um valor.

Esta regra implica que todas tabelas devem ter um atributo ou combinação de atributos que representa um registro único na tabela. Isso pode ser atingido criando que é chamado de *primary key*, ou chave primária. Uma chave primária é um atributo ou combinação de atributos que formam o elemento único de uma tabela e são usadas para relacionar-se com outras tabelas no banco. Alguns exemplos do nosso dia a dia de possíveis atributos que poderiam ser usados como chaves primárias de uma tabela no banco são placas de carros, RGs e CPFs. Todos esses exemplos têm em comum o fato de serem elementos únicos e não se repetirem. Se fosse criada uma tabela para armazenar dados de pessoas físicas e seus nomes, por exemplo, o atributo de nome não seria uma boa escolha de chave primária, por exemplo, devido a possibilidade de existir mais de uma pessoas com o mesmo nome, o que violaria o conceito da 1ª FN.

2ª FN: primeiramente, para satisfazer a 2ª FN, é necessário que a 1ª FN também seja satisfeita. Segundamente, todos os atributos não chaves da tabela devem depender unicamente da chave primária, não podendo depender apenas de parte dela.

Tendo em mente o conceito de chave primária, todas as outras colunas da tabela que não são a(s) coluna(s) de chave primária, são colunas não chave. Todas essas colunas não chave devem depender apenas da(s) coluna(s) chave, portanto, qualquer coluna que dependa de uma outra coluna, senão a(s) coluna(s) de chave primária, devem ser retiradas da tabela e ser usadas em uma outra tabela se relacionando com a coluna da qual ela depende.

Utilizando o exemplo deste trabalho: dados de todas as matérias das grades curriculares dos cursos e seus respectivos cursos foram obtidos. Esses dados incluem tanto informações referentes apenas às matérias, como créditos de aula e código da matéria, como informações referentes apenas aos cursos, como créditos totais e duração ideal do curso, como mostrado na tabela 1:

Tabela 1 - Exemplo de dados não normalizados

Código Matéria	Nome Matéria	Créditos Matéria	Curso	Duração Ideal Curso
LOB1003	Cálculo I	4	Engenharia Física	10 semestres
LOB1018	Física I	4	Engenharia Física	10 semestres
LOB1205	Ecologia Básica	2	Engenharia Ambiental	10 semestres
LOT2045	Biologia	4	Engenharia Ambiental	10 semestres

Fonte: O autor

Nesse caso a chave primária é a combinação dos atributos “Código Matéria” e “Curso”, pois dois cursos diferentes podem ter a mesma matéria. É possível observar nessa tabela que os dados relacionados apenas ao curso se repetem para cada linha das matérias referentes àquele curso. Como a duração ideal do curso depende apenas do atributo “Curso”, que não é apenas parte da chave primária e não ela inteira, este atributo deve ser jogado para outra tabela que deve conter apenas o atributo “Curso” e as informações relacionadas apenas ao atributo “Curso”. Neste caso a normalização ficaria de acordo com o seguinte:

Tabela 2 - Tabela "Matérias" com dados normalizados

Código Matéria	Nome Matéria	Créditos Matéria	Curso
LOB1003	Cálculo I	4	Engenharia Física
LOB1018	Física I	4	Engenharia Física
LOB1205	Ecologia Básica	2	Engenharia Ambiental
LOT2045	Biologia	4	Engenharia Ambiental

Fonte: O autor

Tabela 3 - Tabela "Cursos" com dados normalizados

Curso	Duração Ideal Curso
Engenharia Física	10 semestres
Engenharia Química (noturno)	12 semestres
Engenharia Ambiental	10 semestres
Engenharia de Materiais	10 semestres

Fonte: O autor

3ª FN: primeiramente, para satisfazer a 3ª FN, é necessário que a 2ª FN também seja satisfeita. Segundamente, os atributos não chaves da tabela devem ser mutuamente independentes e ser unica e exclusivamente dependentes da chave primária, ou seja, um atributo não chave não pode depender de outro atributo não chave. Um atributo X só é dependente de um atributo Y se para cada valor de Y existir apenas um valor de X.

O exemplo usado para explicação da 2ª FN também explica a 3ª FN. O atributo “Duração Ideal Curso” dependia apenas de um outro atributo não chave, portanto, é necessário movê-lo para uma outra tabela.

2.2 Python

Python é uma linguagem de programação interpretada, de alto nível e orientada a objetos criada em 1990 por Guido van Rossum, um matemático e programador Holandês, com o objetivo de criar uma linguagem versátil e com sintaxe simples e de fácil leitura para a otimização de tempo de desenvolvimento. (Tuya, Suárez-Cabal, & Riva, 2010)

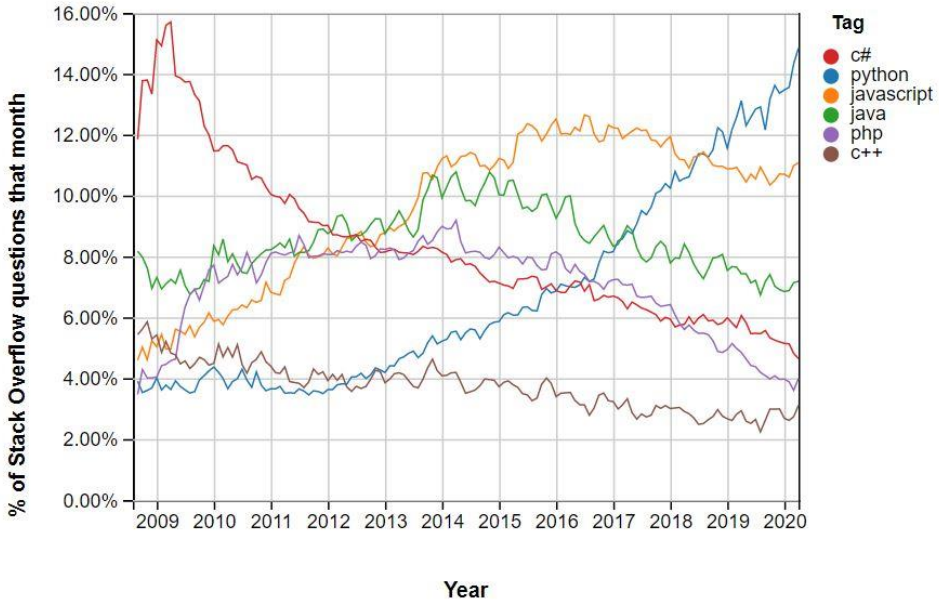
Por ser uma linguagem interpretada e ter uma sintaxe simples, Python tem como uma de suas grandes qualidades a facilidade e velocidade ao escrever um programa. Como o interpretador gerencia detalhes que em outras linguagens de programação têm que ser escritos no código explicitamente, como tipo das variáveis e suas alocações de memória, o códigos em Python tendem a ser mais limpos e curtos.

Outra grande vantagem do uso de Python é a sua integração com diversas ferramentas e sistemas. Com Python é possível gerenciar e executar scripts de outras

linguagens de programação como Java e C++ e, portanto, gerenciar a interface entre várias ferramentas e linguagens diferentes.

Devido às diferentes vantagens citadas acima e sua curva de aprendizagem rápida, Python vem sendo cada vez mais utilizado para as mais diversas aplicações e atualmente já tomou o primeiro lugar de javascript como a linguagem que é mais procurada no *StackOverflow*, principal fórum de perguntas e resposta de programação na internet, como evidencia a figura 1:

Figura 1 - Gráfico mostrando o crescimento de perguntas relacionadas às principais linguagens de programação



Fonte: StackOverflow

2.2.1 Webscrapers

Web scraping, web harvesting ou web data extraction são nomes dados à atividade de extração de dados de uma página da internet. Essa informação pode ser posteriormente exportada e manipulada para os mais diversos fins.

Apesar da extração de dados ser feita manualmente a maior parte do tempo, afinal, sempre que se está navegando na internet e obtendo determinados dados à partir dessa navegação também se está fazendo um tipo de *web scraping* manual. Em diversos casos, quando existe uma quantidade considerável de informação a ser extraida, o processo manual não seria o mais indicado e muito menos o mais eficiente.

Por este motivo existem vários tipos de *web scrapers* automatizados, que buscam a informação desejada em várias páginas diferentes da internet de maneira automática.

A maior parte dos *web scrapers* tem uma estrutura de funcionamento semelhante:

1. Fazer uma solicitação HTTP do URL (Uniform Resource Locator) da página da internet da qual se deseja obter informações;
2. Baixar todo o código HTML referente ao URL desejado. Uma observação importante é que hoje em dia boa parte das páginas da internet não são criadas usando apenas HTML, mas também têm elementos dinâmicos que não são possíveis criar usando apenas HTML e, portanto, usam outras linguagens como JavaScrip, CSS e outros;
3. Localizar dado desejado no HTML baixado;
4. Gerar um resultado com os dados obtidos em um formato mais adequado para o usuário final.

3 METODOLOGIA

3.1 Bibliotecas de Python

Python é uma linguagem de programação com uma quantidade e variedade muito ampla de bibliotecas. A seguir estão as bibliotecas usadas neste trabalho:

3.1.1 *Beautiful Soup*

Beautiful Soup é uma biblioteca de Python criada para parsear conteúdos de páginas da web de maneira fácil e intuitiva. Esta biblioteca é construída baseando-se em outros parsers de Python, como lxml e html5lib.

3.1.2 Requests

Requests é uma biblioteca de Python criada para facilitar a criação e manuseamento de *requests* ou solicitações de HTTP (HyperText Transfer Protocol). HTTP é um protocolo de aplicação usado para estruturar solicitações e respostas de servidores na internet e é basicamente o que gerencia a comunicação entre as páginas de internet e os usuários.

3.1.3 Pandas

Pandas é uma bilioteca extremamente poderosa e de fácil aprendizado de Python usada para análise e manipulação de dados. É uma das bilbiotecas de Python mais conhecidas e indicada para casos em que irá ser feita algum tipo de análise de dados tabulares como planilhas e bases de dados.

Algumas das suas principais funcionalidades são:

- Conversão de diferentes tipos de fonte de dados para o formato desejado, sejam eles CSV (Comma-Separated Values), JSON (JavaScript Object Notation) ou muitos outros;
- Fácil manipulação da codificação dos dados;

- Fácil remoção ou manipulação de dados faltantes;
- Fácil ordenação e agregação de dados de múltiplas fontes diverentes.

3.2 HTML

HyperText Markup Language (HTML) não é uma linguagem de programação pois não permite criar funcionalidades dinâmicas, mas sim uma sintaxe para formatação e organização de documentos. O HTML usa o que são chamadas de *tags*, marcadores com diferentes funcionalidades no código que permitem o usuário formatar o visual da página. Alguns exemplos de *tags* são: “<p>”, que cria um parágrafo, “<table>”, que cria uma tabela e “<b>”, que deixa o texto em negrito.

Todas páginas que são acessadas usando o os navegadores como Google Chrome e Mozilla Firefox contêm HTML em parte delas, pois o que é observado na nossa tela de computador quando se acessa uma página da *web* é a interpretação que os navegadores fazem de um código HTML. Apesar disso, como mencionado anteriormente, todo o *background* da página como processamento e autenticação, são feitos por outras linguagens de programação como java e php.

4 RESULTADOS E DISCUSSÃO

4.1 Explicação do código

Primeiramente é necessário baixar todo o código HTML da página da qual deseja-se obter as informações. Para isso é usada a biblioteca *requests*, que faz uma requisição HTTP para o endereço URL do site e obtém todo o código HTML dessa página usando o método ".get()", que recebe o link da página como parâmetro. Além disso o método '.raise_for_status()" gera um erro caso ocorra algum erro ao tentar baixar o código HTML da página:

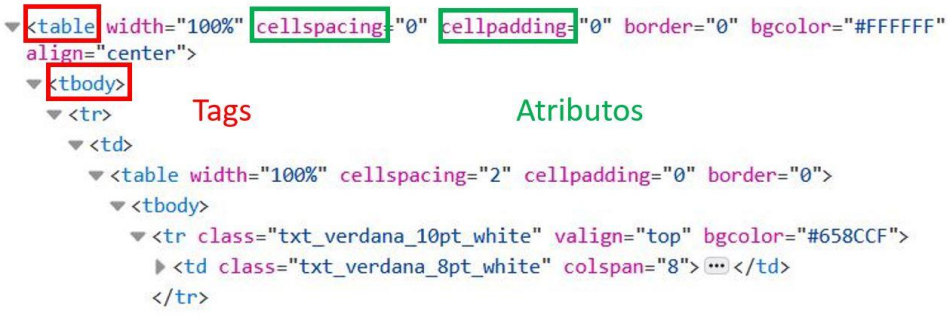
```
#Baixa html inteiro do link provido
    html = requests.get(ScrapeLink)
    html.raise_for_status()

    #Variável com todo o código html da página incial
    html_all = bs4.BeautifulSoup(html.text, 'html.parser')
```

O código acima também usa a biblioteca *BeautifulSoup* (bs4) para parsear o código HTML da página.

Tendo todo o código HTML da página em mãos, é necessário parsear novamente o código para encontrar e obter apenas as informações que são desejadas. Para isso é necessário navegar pelo que são chamadas de *tags* do HTML. *Tags* são elementos da linguagem HTML que definem a estrutura das informações na página. Cada *tag* pode ter um ou mais atributos que personalizam essa *tag*.

Figura 2 - Código HTML e seus elementos destacados

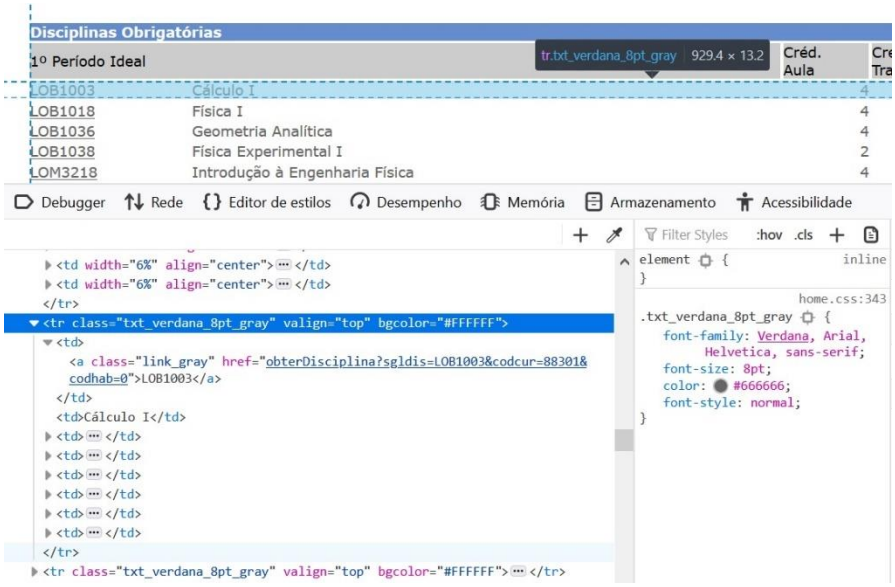


Fonte: Jupiterweb

Na imagem acima é mostrado um exemplo do código HTML da página onde foi usado o *webscraper* produzido neste trabalho. Em vermelho estão destacadas as *tags* e em verde os atributos de uma *tag*. É possível observar que existe uma estrutura em níveis onde dentro de uma *tag*, existem várias outras *tags* que, em si, também têm outras *tags*. Neste caso “<table” é uma *tag*, que tem a *tag* “<body>” dentro de si, enquanto a *tag* “<body>” tem a *tag* “<tr>” e por assim em diante. Este é apenas uma pequena fração do código inteiro e, como tem apenas uma *tag* “<td>”, é responsável por apenas uma linha da página.

As *tags* “<tr>” e “<td>” são importantes na compreensão de estruturas tabulares em HTML pois são elas quem definem linhas e as colunas, respectivamente, como demonstrado na figura 3:

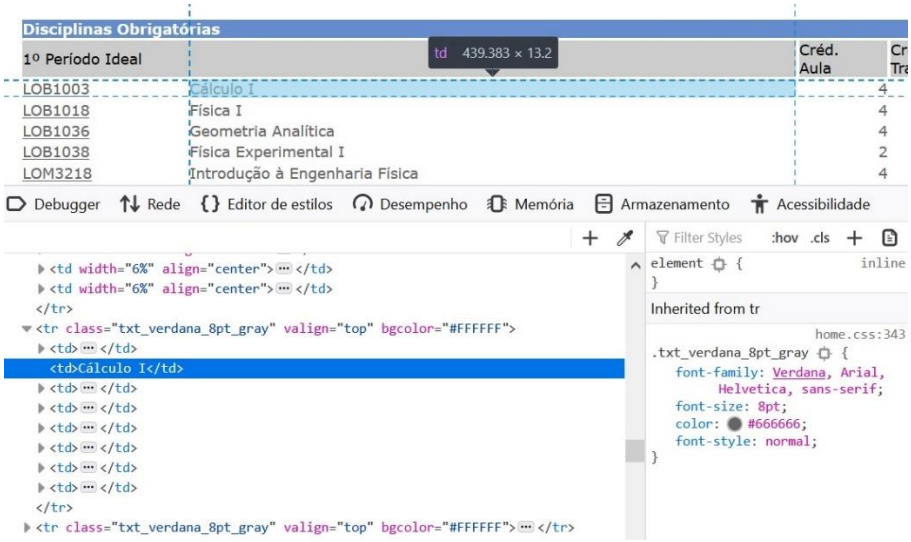
Figura 3 - Código HTML e sua representação gráfica



Fonte: Jupiterweb

Ao selecionar o a linha de código do HTML, área da página referente a essa linha de código é demarcada. Nesse caso, quando a foi selecionada a linha com uma *tag* “<tr>”, a linha inteira da estrutura tabular da página ficou destacada, o que significa que dentro dessa *tag* está todo o código que forma essa linha da tabela.

Figura 4 - Código HTML e sua representação gráfica



Fonte: Jupiterweb

Da mesma forma, a Figura 4 mostra que ao clicar na linha com a *tag* “<td>” que contem o texto “Cálculo I”, a célula com o mesmo texto é destacada, demonstrando que a *tag* “<td>” representa uma coluna dentro de uma linha, ou seja, uma célula da estrutura tabular contida na página.

Ao identificar dentro de quais *tags* as informações desejadas se encontram, é possível usar o método “.findAll()” ou “.find_all()” para encontrar todas as ocorrências da *tag* especificada no argumento do método. Um exemplo do uso desse método é o seguinte:

```
html_links_table = html_all.find_all("table",{"width":"400","border":"0"
, "align":"center"})
```

Nesse caso, o código procura no HTML todas as ocorrências uma *tag* chamada “*table”* que contem os atributos “*width”* com valor de “400”, *“border”* com o valor “0” e *“align”* com o valor “*center”* e retorna uma lista com os códigos contidos dentro de cada uma dessas ocorrências da *tag*.

```
html_aux = str(html_all).split('''<table border="0" cellpadding="0" cell
spacing="2" width="100%">''',1)[1]
html_to_scrape = bs4.BeautifulSoup(html_aux, 'html.parser')
html_to_scrape.findAll("tr")
```

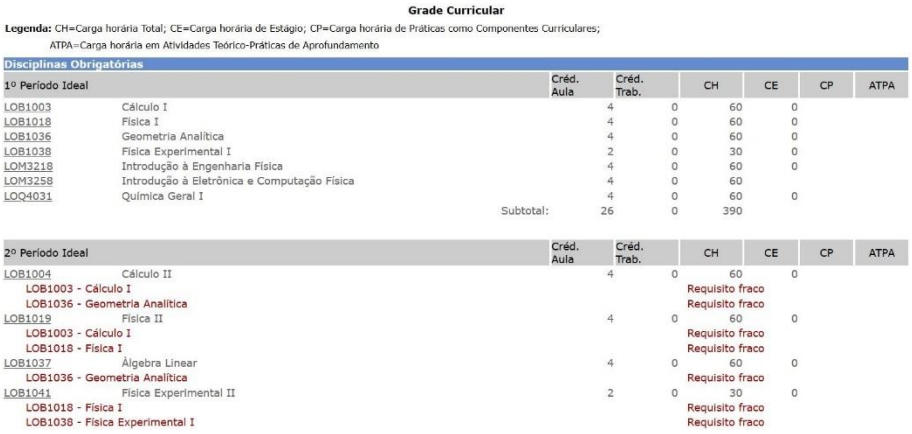
Nas linhas de código acima, o método “.split()” de *string* nativo do Python é usado para retirar todo o pedaço anterior à *string* especificada no argumento, ou seja,

todo o código HTML que estivesse antes do texto “<table border="0" cellpadding="0" cellspacing="2" width="100%">” foi descartado. Isso foi feito para que todas as ocorrências da *tag* “<tr>” encontradas pelo método método “.findAll(“tr”)” fossem úteis, evitando assim uma cadeia de condições *if* e *loops for* para que apenas as ocorrências úteis para o trabalho fossem filtradas.

No caso deste trabalho, o objetivo é navegar pela pagina do Júpiter, o ambiente virtual para alunos e professores da Universidade de São Paulo e obter as informações das grades curriculares de cada um dos cursos da unidade da Escola de Engenharia de Lorena (EEL), bem como as informações sobre cada um dos cursos oferecidos na unidade.

As informações sobre as matérias da grade curricular de cada um dos cursos está disposta da seguinte maneira na página:

Figura 5 - Grade curricular do curso de engenharia física



Grade Curricular

Legenda: CH=Carga horária Total; CE=Carga horária de Estágio; CP=Carga horária de Práticas como Componentes Curriculares; ATPA=Carga horária em Atividades Teórico-Práticas de Aprofundamento

Disciplinas Obrigatórias

1º Período Ideal		Créd. Aula	Créd. Trab.	CH	CE	CP	ATPA
LOB1003	Cálculo I	4	0	60	0		
LOB1018	Física I	4	0	60	0		
LOB1036	Geometria Analítica	4	0	60	0		
LOB1038	Física Experimental I	2	0	30	0		
LOM3218	Introdução à Engenharia Física	4	0	60	0		
LOM3258	Introdução à Eletrônica e Computação Física	4	0	60			
LOQ4031	Química Geral I	4	0	60	0		
	Subtotal:	26	0	390			

2º Período Ideal		Créd. Aula	Créd. Trab.	CH	CE	CP	ATPA
LOB1004	Cálculo II	4	0	60	0		
LOB1003 - Cálculo I				Requisito fraco			
LOB1036 - Geometria Analítica				Requisito fraco			
LOB1019	Física II	4	0	60	0		
LOB1003 - Cálculo I				Requisito fraco			
LOB1018 - Física I				Requisito fraco			
LOB1037	Álgebra Linear	4	0	60	0		
LOB1036 - Geometria Analítica				Requisito fraco			
LOB1041	Física Experimental II	2	0	30	0		
LOB1018 - Física I				Requisito fraco			
LOB1038 - Física Experimental I				Requisito fraco			

Fonte: Jupiterweb

Como mencionado anteriormente, as *tags* “<tr> e “<td>” definem a estrutura tabular das informações, portanto, elas serão as principais *tags* pelas quais o código navegará para obter todas as informações desejadas.

No código HTML da página, cada um dos períodos está contido em uma *tag* chamada “<table>”. Cada uma das linhas com informações dentro dessas tabelas, incluindo o cabeçalho, é delimitado por uma *tag* “<tr>” que, em si, contem várias *tags* “<td>”, cada uma com uma das informações apresentadas na linha. Portanto, tomando como exemplo a primeira linha abaixo do cabeçalho “1º Período Ideal”, existe uma *tag* “<tr>” com várias *tags* “<td>”, entre as quais a primeira contem a informação “LOB1003”, a segunda contem a informação “Cálculo I” e assim em diante.

Como o HTML segue uma estrutura “hierárquica” onde várias *tags* se encontram dentro de uma outra *tag*, primeiro deve-se encontrar as *tags* que externas, que aparecem primeiro no código, sendo ela neste caso a *tag* “<table>”, o que é feito executando o código abaixo:

```
html_aux = str(html_all).split('''<table border="0" cellpadding="0" cell
spacing="2" width="100%">''',1)[1]
html_to_scrape = bs4.BeautifulSoup(html_aux, 'html.parser')
```

Já dentro da *tag* “<table>”, a próxima *tag* a ser identificada é a *tag* “<tr>”, que delimita cada uma das linhas dentro da tabela. Com o código abaixo isso é feito:

```
for row in html_to_scrape.findAll("tr"):
```

Lembrando que o método “findAll()” retorna uma lista com todas as ocorrências da *tag* especificada no argumento, portanto, para acessar cada uma dessas ocorrências é necessário utilizar um *for loop*.

Dentro de cada ocorrência da *tag* “<tr>”, deseja-se buscar cada uma das ocorrências da *tag* “<td>”, onde estão grande parte das informações a serem obtidas. O código abaixo executa essa ação:

```
cells = row.findAll("td")
```

Além das informações desejadas, as tags “<td>” também contêm informações e tags adicionais que não são relevantes para a aquisição dos dados em questão. Levando isso em conta, é importante salientar que ao longo do código HTML da página existe uma grande quantidade de código “excedente”, que contêm apenas informações de formatação da página, informações não desejadas e informações cujas únicas funções são a configuração visual da página, o que faz necessário um trabalho de “filtragem” apenas das ocorrências das tags que contêm informações úteis para trabalho. Essa filtragem é feita usando cláusulas if, que analisam a condição imposta nessa cláusula e filtram apenas os elementos que satisfazem essa condição. Um exemplo de cláusula if usada no código é a seguinte:

```
if cell.a is not None and cell.a.has_attr("href"):
```

Essa cláusula if analisa duas condições e so retorna os resultados que satisfazem ambas condições. Primeiramente é analisado se o resultado retornado pelo método “.a” aplicado ao objeto cell não é None, ou seja, verifica se esse esse método retorna algum resultado. Segundamente, se a primeira condição for satisfeita,

é analisado se o objeto cell, que representa uma tag “<td>”, tem algum atributo chamado “href”, ou seja, analisa-se se existe um endereço de alguma página web dentro da tag “<td>”.

Portanto, já dentro da tag “<td>” a maior parte do código se concentra em filtrar apenas as informações e tags relevantes para a aquisição dos dados desejados usando uma série de cláusulas if.

Quando as informações desejadas são filtradas, estas são armazenadas em uma lista de *Python*, que é exatamente o que o nome diz: uma sequência ou coleção ordenada de valores que podem ser identificados por um índice.

As listas foram criadas de acordo com as necessidades ou praticidade em relação a estrutura do código HTML. Por exemplo: foi criada uma lista chamada “Codigos_Materias” para dados da primeira coluna da tabela mostrada na figura 5 (“LOB1003”, “LOB1018”, ...) e uma lista chamada “Nomes_Materias” para os dados da segunda coluna (“Cálculo 1”, “Física I”, ...) pois essas informações estão localizadas diretamente dentro da *tag* “<td>” em duas ocorrências diferentes desta *tag*. Enquanto isso, as informações sobre o restante das colunas, como “Créd. Aula”, “Créd. Trab.” e “CH”, estão contidas sentro de *tags* “ ” dentro das *tags* “<td>”. Por esse motivo foi mais conveniente criar uma lista separada chamada “Dados_Materia” para armazenar todas essas informações contidas nas *tags* “ ”. Ao fim de cada linha da tabela, ou seja, antes do código analisar a próxima ocorrência da *tag* “<tr>”, essa lista com os dados de uma matéria é adicionada a uma lista chamada “Dados_Materias”, que armazenará esses dados para todas as matérias formando uma lista de listas, obtendo um formato matricial.

Exemplificando: olhando para a figura 5, podemos ver que na primeira linha da tabela, iniciando na coluna “Créd. Aula”, temos os valores “4”,”0”,”60”,”0”,vazio e vazio. Quando o código iniciar a leitura dessa linha, ele armazenará essas informações na lista “Dado_Materia” da seguinte forma:

```
['4','0','60','0','','']
```

Antes da segunda linha ser lida, essa lista será armazenada como um único elemento na lista “Dados_Materias” da seguinte forma:

```
[['4','0','60','0','','']]
```

Ao fazer a leitura da segunda linha, que também contem os valores “4”,”0”,”60”,’0’,vazio e vazio, a lista “Dado_Materia” será igual à da primeira linha e será adicionada à lista “Dados_Materias” como um único elemento, deixando a mesma da seguinte forma:

```
[['4','0','60','0','',''],
 ['4','0','60','0','','']]
```

Adicionalmente, depois da quarta linha da tabela ser lida, a lista “Dados_Materias” terá a seguinte forma e terá esse mesmo comportamento até que as tabelas de todos os períodos ideais sejam lidas:

```
[['4','0','60','0','',''],
 ['4','0','60','0','',''],
 ['4','0','60','0','',''],
 ['2','0','30','0','','']]
```

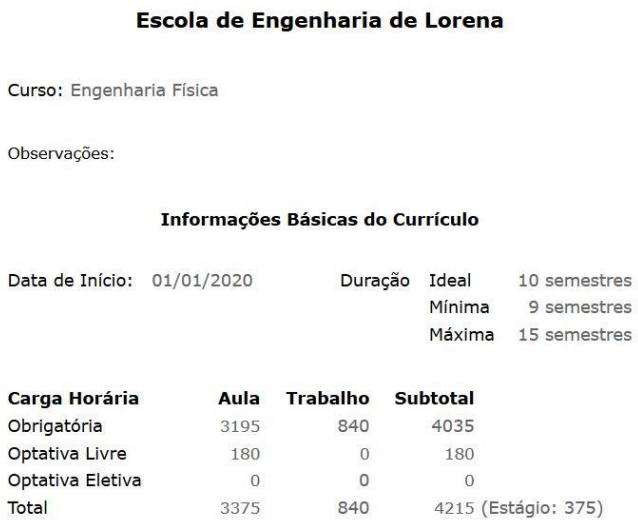
Por fim, para obter as informações da *tag* “ ”, deve-se encontrar a mesma dentro de cada uma das ocorrências da *tag* “<td>”, o que é feito usando a linha de código a seguir:

```
for dado in cell.findAll("div"):
```

Depois disso, usando as cláusulas *if* necessárias é possível obter todos os dados principais de cada matéria, incluindo dados dispostos de maneira diferente na página, como o período ideal da matéria, que está apenas no cabeçalho de cada tabela de período ideal, e os requisitos de cada matéria, que estão nas linhas abaixo da matéria à qual estes se referenciam. Cada um desses dados também tinha sua respectiva lista devido a se encontrarem em uma estrutura diferente na página.

Como este trabalho aborda todos cursos oferecidos na unidade da USP de Lorena, é necessário também incluir a informação de qual curso essa grade curricular representa pois cursos diferentes têm a mesma matéria em sua grade curricular porém com requisitos diferentes. Portanto, para tornar único o registro de cada matéria de cada curso, é necessário obter também as informações de cada um dos cursos. Essa informação se encontra na seguinte tabela, que tem o mesmo formato para cada um dos cursos:

Figura 6 - Informações específicas do curso de engenharia física



Escola de Engenharia de Lorena

Curso: Engenharia Física

Observações:

Informações Básicas do Currículo

Data de Início: 01/01/2020 Duração Ideal 10 semestres
Mínima 9 semestres
Máxima 15 semestres

Carga Horária	Aula	Trabalho	Subtotal
Obrigatória	3195	840	4035
Optativa Livre	180	0	180
Optativa Eletiva	0	0	0
Total	3375	840	4215 (Estágio: 375)

Fonte: Jupiterweb

Como a tabela da Figura 6 é uma outra tabela em relação às tabelas que contêm dados das matérias, além de ter um formato diferente, deve-se fazer uma busca da *tag* “<table>” desta tabela em específico, que é feita utilizando a linha de código a seguir:

```
html_links_table = html_all.find_all("table",{"width":"400","align":"cen
ter"})
```

Nesse caso, como a tabela tinha um visual diferente, os atributos da *tag* “<table>” também se diferenciam em relação às tabelas de períodos ideais.

É interessante mostrar a parte seguinte do código, pois essa tabela não contem muitas informações irrelevantes no código HTML e, por esse motivo, o código é bem sucinto e ilustrativo para a demonstração da maneira na qual o código navega pelas *tags*:

```
Cursos_Dados = []
    for table in html_links_table:
        for row in table.findAll("tr"):
            for cell in row.findAll("td"):
                for span in cell.findAll("span"):
                    if 'Engenharia' in str(span.string):
                        Cursos_Dados.append(str(span.string).strip())
                    if 'semestre' in str(span.string):
                        Cursos_Dados.append(str(span.string).strip())

                for font in cell.findAll("font",color='#666666', face=Tr
ue):
                    if str(font.string).strip().isnumeric():
```

```
                    Cursos_Dados.append(str(font.string).strip())
```

No código acima a hierarquia das *tags* fica bem clara: cada *loop for* representa a iteração das ocorrências de uma *tag* dentro de uma lista de ocorrência dessa mesma *tag*. O primeiro *loop for* procura todas as ocorrências da *tag* “<table>”. O segundo *loop for* procura todas ocorrências da *tag* “<tr>” em cada ocorrência da *tag* “<table>” e assim em diante. Portanto, a sequência de *tags* a serem identificadas é: “<table>”, “<tr>”, “<td>”, “” e “<font>”, sendo cada uma dessas *tags* contida na *tag* anterior, exceto “” e “<font>”, que se encontram ambas no *loop for* da *tag* “<td>” e, portanto, se encontram ambas dentro da mesma.

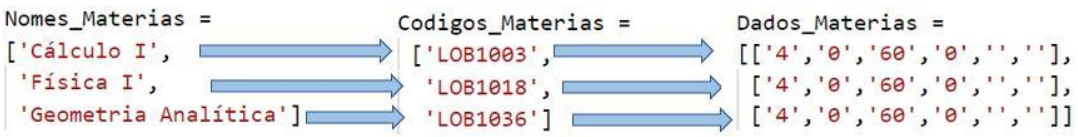
Finalmente, a parte final do código de *scraping* merece atenção. A linha de código abaixo define o resultado que a função “ScrapeHtml()” retorna:

```
    return zip(Codigos_Materias,Nomes_Materias,Cursos,Requisitos_Completos[1
:],Dados_Materias,Obrigatorias_Optativas,Periodos_Ideais,Links_Materias)
```

A função “zip()” une os primeiros elementos de cada um das listas especificadas como argumentos, depois une os segundos elementos de cada uma das listas e assim em diante, em um só objeto “zip”, que pode posteriormente ser “descompactado” e guardado em novas variáveis. Essa função nativa do Python é útil para relacionar cada um dos elementos de várias listas com seus respectivos em um só objeto. Nesse caso, o primeiro elemento da lista “Codigos_Materias” é relacionado ao primeiro elemento de cada uma das outras listas especificadas no argumento da função, unindo todas as informações obtidas ao longo do código para cada uma das matérias em uma só variável.

Exemplificando: as listas a seguir contêm os primeiros 3 elementos de três das listas que foram usadas como parâmetros da função “zip()”:

Figura 7 - Explicação fo funcionamento da função “zip()”



Fonte: O autor

A função “zip()” pega o primeiro elemento de cada uma dessas listas e “compacta” em um só elemento. Posteriormente faz a mesma coisa com os segundos elementos de cada uma das listas e assim por diante, retornando um objeto “zip” que pode ser descompactado posteriormente. No caso desse trabalho, essa função centraliza em um só objeto todas as listas com os dados da página criadas usando o *webscraper*.

Esse objeto “zip” é posteriormente “descompactado” e usado para gerar uma tabela relacional com todos os dados de cada matéria.

Com todos dados obtidos, é necessário que sejam feitas algumas manipulações com os dados para que estes sejam convertidos em scripts de SQL e inseridos na base de dados. Para isso foi criada a função “DataToDF()”, que tem como objetivo transformar o objeto “zip” retornado pela função “ScrapeHtml()” em um *Dataframe*, objeto que pode ser manipulado usando a biblioteca *pandas* do Python.

Primeiramente o objeto “zip” é descompactado em variáveis que representam cada uma das colunas de dados das tabelas de períodos ideais utilizando a linha de codigo abaixo:

```
NomeCurso, DuracaoIdeal, DuracaoMin, DuracaoMax, CHAulaObrigatoria, CHTrabalhoObrigatoria, CHSubtotalObrigatoria, CHAulaOptLivre, CHTrabalhoOptLivre, CHSubtotalOptLivre, CHAulaOptEletiva, CHTrabalhoOptEletiva, CHSubtotalOptEletiva, CHAulaTotal, CHTrabalhoTotal, CHTotal = ScrapeHtmlCursos(url)
```

```
df = pd.DataFrame.from_records(Dados, columns=['Codigo_Materia','Nome_Materia','Curso','Tipo_Disciplina','CA','CT','CH','CE','CP','ATPA','Link_Materia','Requisitos'])
```

A linha de código acima converte uma lista de listas em um *DataFrame*, sendo este uma estrutura de dados bidimensional, ou seja, tem a forma de uma matriz M_{ixj} com i linhas e j colunas. Por uma lista de listas entende-se uma lista cujo cada elemento é uma lista com os dados de cada uma das matérias, tomando assim uma forma matricial. Nesse caso, todos os primeiros elementos de cada uma das lista é equivalente a mesma informação, sendo ela o código da matéria. Todos segundos elementos são os nomes das matérias e assim por diante. Essa conversão para *DataFrame* acontece com o objetivo de usar a biblioteca Pandas para manipular informações de maneira mais eficiente.

Depois de algumas manipulações necessárias para preparar os dados de requisitos para inserção nas tabelas do banco de dados, o *Dataframe* com todas as informações obtidas é dividido em três *Dataframes* diferentes. Isso é necessário para que o conceito de normalização de base de dados explicado anteriormente seja obedecido. Essa divisão é feita nas linhas de código a seguir:

```
Table_Cursos = pd.DataFrame.from_records(Dados_Cursos, columns=['NomeCurso','DuracaoIdeal', 'DuracaoMin', 'DuracaoMax', 'CHAulaObrigatoria', 'CHTrabalhoObrigatoria', 'CHSubtotalObrigatoria', 'CHAulaOptLivre', 'CHTrabalhoOptLivre', 'CHSubtotalOptLivre', 'CHAulaOptEletiva', 'CHTrabalhoOptEletiva', 'CHSubtotalOptEletiva', 'CHAulaTotal', 'CHTrabalhoTotal', 'CHTotal','LinkCurso'])
Table_Materias = df[df.columns[:-1]]
Table_Requisitos = df[['Codigo_Materia','Curso','Requisitos']]
```

Os dados foram divididos em três tabelas: Table_Cursos, Table_Materias e Table_Requisitos. Para obedecer o conceito de normalização de bases de dados explicado anteriormente, dados que são dependentes apenas entre si ou que têm relação de um para N devem ser separados em tabelas diferentes.

Nesse conjunto de dados, a chave primária é a combinação da coluna “Codigo_Materia” e a coluna “Curso”. Existem dados que são dependentes apenas de colunas não chave. Por exemplo: existem dados que são referentes apenas à coluna “Curso” e não têm relação nenhuma com a coluna “Codigo_Materia” (parte da chave primária), como a duração ideal do curso, total de créditos do curso, entre outras. Essas informações que tem relação apenas com a coluna “Curso” devem ser separadas em uma tabela que contem apenas informações relacionadas à essa coluna. Portanto, é criado o *Dataframe* “Table_Cursos”, que tem a coluna “Curso” como chave primária e o restante das colunas que dependem apenas da coluna “Curso”.

O *Dataframe* “Table_Requisitos” também foi criado para satisfazer as regras de normalização pois existem matérias que têm mais de um requisito, portanto a relação entre a coluna “Requisitos” e a chave primária dos dados (combinação entre “Codigo_Materias” e “Curso”) é um relação um para N.

Foi feita a escolha de fazer certas manipulações usando scripts de SQL ao invés de Python devido a maior familiaridade do autor com SQL. Porém é importante salientar que as manipulações seguintes poderiam ser feitas usando tanto Python, quanto SQL.

Por motivos de praticidade, o código abaixo converte os *Dataframes* de cada tabela em arquivos CSV (Comma Separated Values), que podem ser abertos em forma de planilha por softwares como o Microsoft Excel. Essa conversão é feita com o código abaixo:

```
Table_Cursos.to_csv(StorePath + 'Table_Cursos.csv', sep=';',encoding='utf-8-sig')
Table_Materias.to_csv(StorePath + 'Table_Materias.csv', sep=';',encoding='utf-8-sig')
Table_Requisitos.to_csv(StorePath + 'Table_Requisitos.csv', sep=';',encoding='utf-8-sig')
```

Finalmente, a função “DataToDF()” retorna os três *Dataframes* mencionados acima com a seguinte liha de código:

```
return Table_Materias,Table_Requisitos,Table_Cursos
```

Com os dados preparados para a inserção na base de dados, inicia-se agora o processo de criação da base de dados e suas tabelas. Para este propósito é usada a biblioteca de Python SQLite, que permite ao usuário criar bases de dados sem necessidade de um servidor de maneira simples e eficiente.

Com as linhas de código abaixo, a base de dados “GRADES_CURRICULARES.db” é criada, caso não exista. Caso exista, é estabelecida a conexão com a base de dados:

```
SQLite_connection = sqlite3.connect("GRADES_CURRICULARES.db")
cursor = SQLite_connection.cursor()
```

Um objeto chamado cursor também deve ser criado, pois é este que permite a execução de comandos de SQL na base utilizando Python.

À partir de agora o mesmo processo irá se repetir para a criação e população dos dados em todas as tabelas do banco, portanto o processo será descrito apenas para uma das tabelas e deve ser replicado para as demais. O código completo também pode ser encontrado no Apêndice A do trabalho.

O conceito utilizado nas linhas de código abaixo será replicado sempre que deseja-se executar uma ação usando código fonte de SQL. Primeiramente define-se uma variável que contem uma string representando o código exato de SQL que deseja-se executar e, posteriormente, o método “.execute()” é usado para executar o comando definido na variável anterior.

Nesse caso, como de praxe na utilização de base de dados, é necessário checar se a tabela que deseja-se criar já não existe no banco e, caso exista, é necessário apagá-la antes de se criar uma nova. Portanto, nas linhas de código abaixo, é verificado se uma tabela com nome “MATERIAS” existe na base, se existir, a mesma é apagada:

```
table_check_MATERIAS = '''DROP TABLE IF EXISTS [MATERIAS];'''
cursor.execute(table_check_MATERIAS)
```

As linhas de Código a seguir executam o comando para criar a tabela “MATERIAS” na base de dados:

```
create_table_MATERIAS = '''
CREATE TABLE MATERIAS (
id INT IDENTITY(1,1),
CodigoMateria NVARCHAR(50) NOT NULL,
NomeMateria NVARCHAR(100)  NOT NULL,
Curso NVARCHAR(50) NOT NULL,
TipoDisciplina NVARCHAR(100),
CreditosAula INT,
CreditosTrabalho INT,
CargaHorariaTotal INT,
CE INT,
CP INT,
ATPA INT,
LinksMaterias NVARCHAR(2000)
);
'''
cursor.execute(create_table_MATERIAS)
```

Com as tabelas criadas, executando a função “DataToDF()”, que transforma os dados obtidos nas páginas web em *Dataframes* descrita anteriormente, são obtidas as informações no formato de *Dataframes*:

```
DF_TabelaMaterias, DF_TabelaRequisitos, DF_TabelaCursos = DataToDF("")
```

A sintaxe de inserção de dados em tabelas usando SQL é a seguinte:

```
INSERT INTO NomeTabela (NomeColuna1, NomeColuna2, NomeColuna3) VALUES
(value1, value2, value3), (value1, value2, value3), ...
```

A linha de código a seguir utiliza as funções nativas de Python “str()” e “tuple()” para reproduzir a sintaxe da segunda linha do exemplo acima (parte após “VALUES”), para que, combinada com a string definida abaixo na variável “insert_MATERIAS”, reproduza a sintaxe do comando “INSERT INTO” demonstrado acima:

```
records_MATERIAS = [str(tuple(x)) for x in DF_TabelaMaterias.values]
```

Para cada linha de valores do *dataframe* “DF_TabelaMaterias”, será criada uma tupla (tipo de variável de Python que é definida utilizando parênteses) com os valores dessa linha, que será convertida em tipo *string* utilizando a função “str()”, para que possa ser concatenada à variável “insert_MATERIAS”. Uma lista com as tuplas de todos os dados desse *dataframe*, será armazenada na variável “records_MATERIAS”. Exemplificando essa tranformação:

Dados *dataframe*:

```
LOB1003;Cálculo I;Engenharia Ambiental;
LOB1018;Física I;Engenharia Ambiental;
```

Dados variável records_MATERIAS:

```
[(LOB1003,Cálculo I,Engenharia Ambiental),
(LOB1018,Física I,Engenharia Ambiental)]
```

```
insert_MATERIAS = '''
INSERT INTO [Materias](
[CodigoMateria],
[NomeMateria],
[Curso],
[TipoDisciplina],
[CreditosAula],
[CreditosTrabalho],
[CargaHorariaTotal],
[CE],
[CP],
[ATPA],
[LinksMaterias]
)
VALUES
'''
```

Posteriormente é utilizado o método “.join()”, que concatena todos os elementos de uma lista separando-os pela *string* definida antes do método, que nesse caso é uma vírgula, para simular a sintaxe do comando “INSERT INTO”. Além disso, a linha de código abaixo também concatena a string da variável “insert_MATERIAS” e o comando é executado:

```
cursor.execute(insert_MATERIAS+",".join(records_MATERIAS))
```

Finalmente, com todas as tabelas populadas, é criado um *dump* da base de dados, ou seja, todo o código em SQL necessário para criar e popular a base de dados é escrito em um arquivo .sql para que a base de dados possa ser recriada apenas rodando esse *script*. Isso é feito executando as linhas de script abaixo, que criam o arquivo “GRADES_CURRICULARES_dump.sql” com o *dump* da base de dados utilizando o método “iterdump()”, que gera um objeto que pode ser iterado, linha por linha, e escrito no arquivo de *dump*:

```
with open('GRADES_CURRICULARES_dump.sql', 'w') as f:
    for line in SQLite_connection.iterdump():
        f.write('%s\n' % line)
```

5 Conclusão

A plataforma virtual utilizada pela Universidade de São Paulo contem uma quantidade enorme de informações úteis para os graduandos, porém de maneira mais dispersa e, portanto, menos clara. Por esse motivo, a criação de uma base de dados contendo apenas as informações mais relevantes de maneira clara e centralizada auxiliaria na consulta e análise dessas informações.

Utilizando uma técnica chamada de *webscraping* foi possível criar um código em Python, fazendo uso de sua versatilidade, para obter praticamente todas as informações relevantes referentes a todas as disciplinas das grades curriculares dos cursos disponibilizados na Escola de Engenharia de Lorena da Universidade de São Paulo (EEL-USP). Estas informações foram centralizadas em uma base de dados relacional utilizando SQL, que contem três tabelas principais com dados sobre cada um dos cursos, suas disciplinas e requisitos.

O código criado, além de disponibilizar o arquivo *dump* da base de dados, ou seja, um arquivo com todo o código em SQL necessário para a reconstrução da base de dados com os dados obtidos, também gera arquivos em csv, que podem ser manipulados utilizando, entre outros, o Microsoft Excel, possibilitando para aqueles que não tem conhecimento de base de dados e SQL o aproveitamento das informações. Além disso, a base pode ser aproveitada para desenvolvimentos futuros, como um ponto de partida para a atribuição de carga horária entre docentes, abertura de turmas e planejamento de grades horárias.

6 Referências

Ávila, G. (2000). Cantor e a Teoria de Conjuntos. *Revista do Professor de Matemática*. Fonte: Dia a Dia Educação.

Ben-Gan, I. (2012). *Microsoft SQL Server 2012 T-SQL Fundamentals.* Sebastopol: O'Reilly Media, Inc.

Chamberlin, D. (2012). Early History of SQL. *IEEE Annals of the History of Computing*, 78-82.

Codd, E. F. (24 de June de 2009). Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks. *SIGMOD Record*.

Feitosa, H. d., & Paulovich, L. (2005). *Um Prelúdio à Lógica.* São Paulo: Editora UNESP.

Johnson, P. (1972). *A History of Set Theory.* Prindle: Weber & Schmidt.

Lutz, M. (2001). *Programming Python.* Sebastopol: O'Reilly & Associates, Inc.

Oppel, A. (2004). *Databases Demystified.* San Francisco: McGraw-Hill Osborne Media.

Stack Overflow Trends. (2020). Fonte: Stack Overflow: https://insights.stackoverflow.com/trends?tags=python%2Cjavascript%2Cjava%2Cc%23%2Cphp%2Cc%2B%2B&utm_source=so-owned&utm_medium=blog&utm_campaign=gen-blog&utm_content=blog-link&utm_term=incredible-growth-python

Tuya, J., Suárez-Cabal, M. J., & Riva, C. d. (2010). Full Predicate Coverage for Testing SQL Database Queries. *Journal of Software: Testing, Verification and Reliability*.

7 APÊNDICES

7.1 APÊNDICE A – Código Fonte

```
import bs4, requests, re

def ScrapeHtml(ScrapeLink):


    #Baixa html inteiro do link provido
    html = requests.get(ScrapeLink)
    html.raise_for_status()

    #Variável com todo o código html da página incial
    html_all = bs4.BeautifulSoup(html.text, 'html.parser')


    Obrigatorias_Optativas = [] #Lista com obrigatoriedade de todas materias
    Nomes_Materias = [] #Lista com nomes de todas as materias
    Periodos_Ideais = [] #Lista com periodo ideal de todas materias
    Links_Materias = [] #Lista com links de todas materias
    Requisitos = [] #Lista com todas as materias que são requisitos
    Requisito = [] #Lista com requisitos para cada materia
    Tipos_Requisitos = [] #Lista com dados do tipo de requisito para cada mate
ria requisito
    Todos_Tipos_Requisitos = [] #Lista com dados de tipo de requisitos de toda
s materias
    Cursos = [] #Lista com o curso de todas materias
    Codigos_Materias = [] #Lista de codigos das materias
    Dados_Materias = [] #Lista dos dados adicionais de todas materias
    Dados_Materia = [] #Lista dos dados adicionais de cada materias


    html_links_table = html_all.find_all("table",{"width":"400","border":"0",
"align":"center"})

    Curso = ''
    for table_links in html_links_table:
        html_links_span = table_links.find_all("span")
        for span_links in html_links_span:
            if "Enge" in str(span_links.string):
                Curso = span_links.string.strip()

    html_aux = str(html_all).split('''<table border="0" cellpadding="0" cellsp
acing="2" width="100%">''',1)[1]
    html_to_scrape = bs4.BeautifulSoup(html_aux, 'html.parser')

    #Procura todas tags 'a' que tem um atributo de class= link_gray
```

```
    #html_links_table = html_all.find_all("table",{"width":"100%","cellspacing
":"2", "cellpadding":"0"})

    for row in html_to_scrape.findAll("tr"):

        if "<b>" in str(row): #Procura a tag que contem a informação se materi
a é obrigatoria ou não
            Obrigatoria_Optativa = " ".join(row.b.string.strip().split())

        cells = row.findAll("td")

        for cell in cells:

            if cell.a is None and cell.div is None and cell.span is None and c
ell.b is None: #Filtra as celulas que contem apenas nome de materia
                if len(cell.string) > 1:
                    Nomes_Materias.append(cell.string.strip())

            if "txt_arial_8pt_black" in str(cell):
                if cell.span.string is not None and len(cell.span.string) > 5:
 #Filtra celulas que contem apenas periodo ideal
                    Periodo_Ideal = " ".join(cell.span.string.strip().split())

            if cell.a is not None and cell.a.has_attr("href"): #Filtra apenas
o link de cada materia
                Links_Materias.append(cell.a["href"])

            if ('LO' in str(cell.string) or '88' in str(cell.string)) and len(
cell.string.strip()) == 7:
                Requisitos.append(Requisito) #Primeiro loop appenda uma vazia,
 logo tenho que pegar a lista inteira exceto o primeiro item
                Requisito = [] #Zera lista aqui pois toda vez que encontra mat
eria nova, os requisitos começam do 0
                Todos_Tipos_Requisitos.append(Tipos_Requisitos) #Primeiro loop
 appenda uma vazia, logo tenho que pegar a lista inteira exceto o primeiro ite
m
                Tipos_Requisitos = [] #Zera lista aqui pois toda vez que encon
tra materia nova, os requisitos começam do 0
                Obrigatorias_Optativas.append(Obrigatoria_Optativa)
                Periodos_Ideais.append(Periodo_Ideal)
                Codigos_Materias.append(cell.string.strip())
                Cursos.append(Curso)

            if 'LO' in str(cell.string) and len(cell.string.strip()) > 7:
                Requisito.append(" ".join(cell.string.replace("\r\n","").repla
ce("\xa0","").strip().split())) #adiciona no requisito apenas as strings que n
ão sao apenas codigos de materias
```

```
        if len(Dados_Materia) > 0: #Algumas listas vazias sao retornadas, filt
ra apenas as que tem dado
            Dados_Materias.append(Dados_Materia)

        Dados_Materia = [] #Zera lista pois encontrou a proxiam materia
        for cell in cells:

            for dado in cell.findAll("div"): #Procura apenas os dados de cada
materia

                if dado.string is None:
                    Dados_Materia.append("") #Alguns retornos são None, tem qu
e ser tratados pois nao aceitam o .strip()
                elif 'Requisito' not in dado.string and 'dica' not in dado.str
ing: #Pega apenas os dados nao referentes a requisitos
                    Dados_Materia.append(str(dado.string).strip())

                elif dado.string is not None and ('Requisito' in dado.string o
r 'Indicação' in dado.string): #Pega apenas os dados referentes a requisitos
                    Tipos_Requisitos.append(" ".join(dado.string.replace("\r\n
","").replace("\xa0","").strip().split()))


    Requisitos_Completos = [] #Lista que junta cada requisito com seu respecti
vo tipo
    for requisito, tipo_requisito in zip(Requisitos,Todos_Tipos_Requisitos):
        Requisitos_Completos_Por_Materia = []
        for i,j in zip(requisito, tipo_requisito):
            Requisitos_Completos_Por_Materia.append([i,j])
        Requisitos_Completos.append(Requisitos_Completos_Por_Materia)

    return zip(Codigos_Materias,Nomes_Materias,Cursos,Requisitos_Completos[1:]
,Dados_Materias,Obrigatorias_Optativas,Periodos_Ideais,Links_Materias)



def ScrapeHtmlCursos(ScrapeLink):

    #Baixa html inteiro do link provido
    html = requests.get(ScrapeLink)
    html.raise_for_status()

    #Variável com todo o código html da página incial
    html_all = bs4.BeautifulSoup(html.text, 'html.parser')

    html_links_table = html_all.find_all("table",{"width":"400", "align":"cent
er"})

    Cursos_Dados = []
```

```
    for table in html_links_table:
        for row in table.findAll("tr"):
            for cell in row.findAll("td"):
                for span in cell.findAll("span"):
                    if 'Engenharia' in str(span.string):
                        Cursos_Dados.append(str(span.string).strip())
                        NomeCurso = str(span.string).strip()
                    if 'semestre' in str(span.string):
                        Cursos_Dados.append(str(span.string).strip())

                for font in cell.findAll("font",color='#666666', face=True):
                    if str(font.string).strip().isnumeric():
                        Cursos_Dados.append(str(font.string).strip())

    print('Data curso %s has been scraped.' % NomeCurso)

    return Cursos_Dados



from TCC_CodeV3 import ScrapeHtml,ScrapeHtmlCursos
import pandas as pd
import bs4, requests

def GetHTMLCursos():

    url = 'https://uspdigital.usp.br/jupiterweb/jupCursoLista?codcg=88&tipo=N'

    radical = 'https://uspdigital.usp.br/jupiterweb/'

    #Baixa html inteiro do link provido
    html = requests.get(url)
    html.raise_for_status()

    #Variável com todo o código html da página incial
    html_all = bs4.BeautifulSoup(html.text, 'html.parser')

    Grades_Curriculares = []
    for a in html_all.findAll("a", href=True):
        if 'Grade' in str(a["href"]):
            Grades_Curriculares.append(radical+a["href"])

    return Grades_Curriculares



def DataToDF(StorePath):
```

```
    links = GetHTMLCursos()
    Dados_Cursos = []
    Dados = []
    Requisitos_dict = {}

    for url in links:

        NomeCurso, DuracaoIdeal, DuracaoMin, DuracaoMax, CHAulaObrigatoria, CH
TrabalhoObrigatoria, CHSubtotalObrigatoria, CHAulaOptLivre, CHTrabalhoOptLivre
, CHSubtotalOptLivre, CHAulaOptEletiva, CHTrabalhoOptEletiva, CHSubtotalOptEle
tiva, CHAulaTotal, CHTrabalhoTotal, CHTotal = ScrapeHtmlCursos(url)
        Dados_Cursos.append([NomeCurso,DuracaoIdeal, DuracaoMin, DuracaoMax, C
HAulaObrigatoria, CHTrabalhoObrigatoria, CHSubtotalObrigatoria, CHAulaOptLivre
, CHTrabalhoOptLivre, CHSubtotalOptLivre, CHAulaOptEletiva, CHTrabalhoOptEleti
va, CHSubtotalOptEletiva, CHAulaTotal, CHTrabalhoTotal, CHTotal, url])

        for Codigo_Materia,Nome_Materia,Curso,Requisitos,Dados_Materia,Tipo_Di
sciplina,Periodo_Ideal,Link_Materia in ScrapeHtml(url):
            Linha = []
            Requisitos_aux = []

            for requisito in Requisitos:
                for item in requisito:
                    Requisitos_aux.append(item)

            for i in range(0,len(Requisitos_aux),2):
                Requisitos_dict = {Requisitos_aux[i]: Requisitos_aux[i + 1] fo
r i in range(0, len(Requisitos_aux), 2)}

            CA = Dados_Materia[0]
            CT = Dados_Materia[1]
            CH = Dados_Materia[2]
            CE = Dados_Materia[3]
            CP = Dados_Materia[4]
            ATPA = Dados_Materia[5]

            Linha = [Codigo_Materia,Nome_Materia,Curso,Tipo_Disciplina,CA,CT,C
H,CE,CP,ATPA,Periodo_Ideal,Link_Materia,Requisitos_dict]
            Dados.append(Linha)


    Table_Cursos = pd.DataFrame.from_records(Dados_Cursos, columns=['NomeCurso
','DuracaoIdeal', 'DuracaoMin', 'DuracaoMax', 'CHAulaObrigatoria', 'CHTrabalho
Obrigatoria', 'CHSubtotalObrigatoria', 'CHAulaOptLivre', 'CHTrabalhoOptLivre',
 'CHSubtotalOptLivre', 'CHAulaOptEletiva', 'CHTrabalhoOptEletiva', 'CHSubtotal
OptEletiva', 'CHAulaTotal', 'CHTrabalhoTotal', 'CHTotal','LinkCurso'])
```

```
    df = pd.DataFrame.from_records(Dados, columns=['Codigo_Materia','Nome_Mate
ria','Curso','Tipo_Disciplina','CA','CT','CH','CE','CP','ATPA','Periodo_Ideal'
,'Link_Materia','Requisitos'])
    df['Codigo_Materia'] = df['Codigo_Materia'].astype(str)
    df['Requisitos'] = df['Requisitos'].astype(str)
    df['Requisitos'] = df['Requisitos'].str.replace("'","").str.replace("{",""
).str.replace("}","")
    Table_Materias = df[df.columns[:-1]]
    print(Table_Materias)
    Table_Requisitos = df[['Codigo_Materia','Curso','Requisitos']]

    if len(StorePath) > 0:
        if StorePath[-1:] != "\\":
            StorePath = StorePath + "\\"

    Table_Cursos.to_csv(StorePath + 'Table_Cursos.csv', sep=';',encoding='utf-
8-sig')
    Table_Materias.to_csv(StorePath + 'Table_Materias.csv', sep=';',encoding='
utf-8-sig')
    Table_Requisitos.to_csv(StorePath + 'Table_Requisitos.csv', sep=';',encodi
ng='utf-8-sig')

    print('Data has been read.')

    return Table_Materias,Table_Requisitos,Table_Cursos


import sqlite3
from TCC_read_dataV3 import DataToDF
import sqlalchemy as sal
import pandas as pd
import urllib
import os

#Conectar à base, se não existir a base é criada
SQLite_connection = sqlite3.connect("GRADES_CURRICULARES.db")

#Cria um cursor que possibilita rodar queries usando python
cursor = SQLite_connection.cursor()
print("The connection to the database is estabilished.")

print(u"current directory: %s" % os.getcwd())

select_query = "SELECT sqlite_version();"

#Executa uma query
cursor.execute(select_query)

#Fecthall pega o resultado da query
```

```
record = cursor.fetchall()
print("The SQLite version is: ", record)

table_check_MATERIAS = '''DROP TABLE IF EXISTS [MATERIAS];'''

create_table_MATERIAS = '''
CREATE TABLE MATERIAS (
id INT IDENTITY(1,1),
CodigoMateria NVARCHAR(50) NOT NULL,
NomeMateria NVARCHAR(100)  NOT NULL,
Curso NVARCHAR(50) NOT NULL,
TipoDisciplina NVARCHAR(100),
CreditosAula INT,
CreditosTrabalho INT,
CargaHorariaTotal INT,
CE INT,
CP INT,
ATPA INT,
PeriodoIdeal NVARCHAR(50),
LinksMaterias NVARCHAR(2000)
);
'''

table_check_REQUISITOS_RAW = '''DROP TABLE IF EXISTS [REQUISITOS_RAW];'''

create_table_REQUISITOS_RAW = '''
CREATE TABLE [REQUISITOS_RAW] (
id INT IDENTITY(1,1),
CodigoMateria NVARCHAR(50),
Curso NVARCHAR(50),
Requisitos NVARCHAR(1000)
);
'''

create_table_REQUISITOS = '''
DROP TABLE IF EXISTS [REQUISITOS];

CREATE TABLE REQUISITOS (
id INT IDENTITY(1,1),
CodigoMateria NVARCHAR(50) NOT NULL,
Requisito NVARCHAR(100)  NOT NULL,
TipoRequisito NVARCHAR(50)
);
'''


#Codigo para inserção de dados na tabela
insert_MATERIAS = '''
INSERT INTO [Materias](
```

```
[CodigoMateria],
[NomeMateria],
[Curso],
[TipoDisciplina],
[CreditosAula],
[CreditosTrabalho],
[CargaHorariaTotal],
[CE],
[CP],
[ATPA],
[PeriodoIdeal],
[LinksMaterias]
)
VALUES
'''

#Pega os dois dataframes criados
DF_TabelaMaterias, DF_TabelaRequisitos, DF_TabelaCursos = DataToDF("")

#Cria uma sequencia de tuplas contendo os valores de cada linha, para assemelh
ar comando INSERT VALUES do SQL
records_MATERIAS = [str(tuple(x)) for x in DF_TabelaMaterias.values]

cursor.execute(table_check_MATERIAS)
cursor.execute(create_table_MATERIAS)
cursor.execute(insert_MATERIAS+",".join(records_MATERIAS))

select_query = "SELECT * FROM [MATERIAS]"

cursor.execute(select_query)
result = cursor.fetchall()


insert_REQUISITOS_RAW = '''
INSERT INTO [REQUISITOS_RAW](
[CodigoMateria],
[Curso],
[Requisitos]
)
VALUES
'''

records_REQUISITOS_RAW = [str(tuple(x)) for x in DF_TabelaRequisitos.values]

cursor.execute(table_check_REQUISITOS_RAW)
cursor.execute(create_table_REQUISITOS_RAW)
cursor.execute(insert_REQUISITOS_RAW+",".join(records_REQUISITOS_RAW))

table_check_CURSOS_RAW = '''DROP TABLE IF EXISTS [CURSOS_RAW];'''
```

```
create_table_CURSOS_RAW = '''
CREATE TABLE CURSOS_RAW (
id INT IDENTITY(1,1),
NomeCurso NVARCHAR(50) NOT NULL,
DuracaoIdeal NVARCHAR(50),
DuracaoMin NVARCHAR(50),
DuracaoMax NVARCHAR(50),
CHAulaObrigatoria INT,
CHTrabalhoObrigatoria INT,
CHSubtotalObrigatoria INT,
CHAulaOptLivre INT,
CHTrabalhoOptLivre INT,
CHSubtotalOptLivre INT,
CHAulaOptEletiva INT,
CHTrabalhoOptEletiva INT,
CHSubtotalOptEletiva INT,
CHAulaTotal INT,
CHTrabalhoTotal INT,
CHTotal INT,
LinkCurso NVARCHAR(2000)
);
'''


insert_CURSOS_RAW = '''
INSERT INTO [CURSOS_RAW](
[NomeCurso],
[DuracaoIdeal],
[DuracaoMin],
[DuracaoMax],
[CHAulaObrigatoria],
[CHTrabalhoObrigatoria],
[CHSubtotalObrigatoria],
[CHAulaOptLivre],
[CHTrabalhoOptLivre],
[CHSubtotalOptLivre],
[CHAulaOptEletiva],
[CHTrabalhoOptEletiva],
[CHSubtotalOptEletiva],
[CHAulaTotal],
[CHTrabalhoTotal],
[CHTotal],
[LinkCurso]
)
VALUES
'''
```

```
#Cria uma sequencia de tuplas contendo os valores de cada linha, para assemelh
ar comando INSERT VALUES do SQL
records_CURSOS_RAW = [str(tuple(x)) for x in DF_TabelaCursos.values]

cursor.execute(table_check_CURSOS_RAW)
cursor.execute(create_table_CURSOS_RAW)
cursor.execute(insert_CURSOS_RAW+",".join(records_CURSOS_RAW))


table_check_REQUISITOS = '''DROP TABLE IF EXISTS [REQUISITOS];'''

create_table_REQUISITOS = '''
CREATE TABLE REQUISITOS (
CodigoMateria NVARCHAR(50),
Curso NVARCHAR(50),
Requisito NVARCHAR(100),
TipoRequisito NVARCHAR (100)
);
'''


#Codigo processo criado para separas os requisitos e seus respectivos tipos pa
ra cada materia e inserir na tabela Requisitos
insert_REQUISITOS_DEPRECATED = '''
INSERT INTO [REQUISITOS] (
[CodigoMateria],
[Curso],
[Requisito],
[TipoRequisito]
)
SELECT [CodigoMateria],[Curso],
[Requisito] = CASE WHEN CHARINDEX(':',VALUE) > 0 THEN LTRIM(LEFT(VALUE,CHARIND
EX(':',VALUE)-1)) ELSE VALUE END,
[TipoRequisito] = CASE WHEN CHARINDEX(':',VALUE) > 0 THEN LTRIM(RIGHT(VALUE,LE
N(VALUE) - CHARINDEX(':',VALUE))) ELSE VALUE END
FROM [REQUISITOS_RAW]
CROSS APPLY STRING_SPLIT([Requisitos],',')
'''

insert_REQUISITOS_2 = '''
WITH RECURSIVE split(s, last, rest) AS (
  --
VALUES('', '', 'LOB1004 - Cálculo II: Requisito fraco, LOB1036 - Geometria Ana
lítica: Requisito fraco')
  SELECT '','', [Requisitos] FROM REQUISITOS_RAW
  UNION ALL
  SELECT s || substr(rest, 1, 1),
         substr(rest, 1, 1),
         substr(rest, 2)
```

```
  FROM split
  WHERE rest <> ''
)
SELECT REPLACE(s,',','')
FROM split
WHERE rest = ''
   OR last = ',';
'''

insert_REQUISITOS = '''
INSERT INTO [REQUISITOS] (
[CodigoMateria],
[Curso],
[Requisito],
[TipoRequisito]
)
WITH split(CodigoMateria, Curso, word, str) AS (
    SELECT [CodigoMateria],[Curso],'', [Requisitos] ||','
    FROM REQUISITOS_RAW
    UNION ALL SELECT [CodigoMateria], [Curso],
    substr(str, 0, instr(str, ',')),
    substr(str, instr(str, ',')+1)
    FROM split WHERE str!=''
) SELECT [CodigoMateria], [Curso],
CASE WHEN INSTR(word,':') > 0 THEN LTRIM(SUBSTR(word,1,INSTR(word,':')-
1)) ELSE word END AS 'Requisito',
CASE WHEN INSTR(word,':') > 0 THEN SUBSTR(word,INSTR(word,':')+1,LENGTH(word)-
(INSTR(word,':'))) ELSE word END AS 'TipoRequisito'
FROM split WHERE word!='';
'''


cursor.execute(table_check_REQUISITOS)
cursor.execute(create_table_REQUISITOS)
cursor.execute(insert_REQUISITOS)


table_check_CURSOS = '''DROP TABLE IF EXISTS [CURSOS];'''

create_table_CURSOS = '''
CREATE TABLE CURSOS (
id INT IDENTITY(1,1),
NomeCurso NVARCHAR(50) NOT NULL,
DuracaoIdeal NVARCHAR(50),
DuracaoMin NVARCHAR(50),
DuracaoMax NVARCHAR(50),
CHAulaObrigatoria INT,
CHTrabalhoObrigatoria INT,
CHSubtotalObrigatoria INT,
```

```
CHAulaOptLivre INT,
CHTrabalhoOptLivre INT,
CHSubtotalOptLivre INT,
CHAulaOptEletiva INT,
CHTrabalhoOptEletiva INT,
CHSubtotalOptEletiva INT,
CHAulaTotal INT,
CHTrabalhoTotal INT,
CHTotal INT,
LinkCurso NVARCHAR(2000)
);
'''

insert_CURSOS = '''
INSERT INTO [CURSOS](
[NomeCurso],
[DuracaoIdeal],
[DuracaoMin],
[DuracaoMax],
[CHAulaObrigatoria],
[CHTrabalhoObrigatoria],
[CHSubtotalObrigatoria],
[CHAulaOptLivre],
[CHTrabalhoOptLivre],
[CHSubtotalOptLivre],
[CHAulaOptEletiva],
[CHTrabalhoOptEletiva],
[CHSubtotalOptEletiva],
[CHAulaTotal],
[CHTrabalhoTotal],
[CHTotal],
[LinkCurso]
)
SELECT CASE WHEN [NomeCurso] = 'Engenharia Química' AND [DuracaoIdeal] = '10 s
emestres' THEN 'Engenharia Química (Diurno)'
WHEN [NomeCurso] = 'Engenharia Química' AND [DuracaoIdeal] = '12 semestres' TH
EN 'Engenharia Química (Noturno)' ELSE [NomeCurso] END,
[DuracaoIdeal],
[DuracaoMin],
[DuracaoMax],
[CHAulaObrigatoria],
[CHTrabalhoObrigatoria],
[CHSubtotalObrigatoria],
[CHAulaOptLivre],
[CHTrabalhoOptLivre],
[CHSubtotalOptLivre],
[CHAulaOptEletiva],
[CHTrabalhoOptEletiva],
[CHSubtotalOptEletiva],
```

```
[CHAulaTotal],
[CHTrabalhoTotal],
[CHTotal],
[LinkCurso]
FROM [Cursos_RAW]
'''

cursor.execute(table_check_CURSOS)
cursor.execute(create_table_CURSOS)
cursor.execute(insert_CURSOS)

table = pd.read_sql("SELECT * FROM [CURSOS]", SQLite_connection)
table.to_csv('output.csv')

#print(cursor.fetchall())


#Fecha conexão com o banco
cursor.close()

#Cria arquivo com o dump da base de dados
with open('GRADES_CURRICULARES_dump.sql', 'w') as f:
    for line in SQLite_connection.iterdump():
        f.write('%s\n' % line)

print('GRADES_CURRICULARES_dump.sql')

SQLite_connection.close()
```